Introduction to Module Development

Release 1.0

Acquia

October 22, 2012

CONTENTS

CHAPTER

ONE

BEFORE WE START

The best programmers are lazy programmers. Drupal, more than many other systems, holds that maxim to be true. Why do lazy programmers have an advantage?

Drupal is a software framework, but it's a community first. There are over ten thousand existing modules to choose from and thousands more blog posts and online resources to copy code from. Add to that a thriving community on IRC channels, forums and commercial support companies like Acquia and there are many ways to find solutions. The last thing you should resort to is writing custom code.

Custom code is expensive to maintain, risky to a project and creates a dependency on the people who wrote it. So while it is necessary to write some code, on the vast majority of large Drupal sites, strive to practice configuration before coding. You can accomplish a lot just by picking the right modules and configuring them to suit your needs.

1.1 Install before we start

1.1.1 Environment

You'll need a functioning Apache, MySQL and PHP (5.2+). This can be setup via Acquia's Dev Desktop in minutes. Go to *http://www.acquia.com/downloads* to get it.

A text editor or Integrated Development Environment (IDE). Use an editor or environment with syntax highlighting and autocompletion. Drupal can integrate with a number of IDEs. With Netbeans, for example can format your code according to Drupal standards. There are also a number of code templates available. Two examples:

- NetBeans - Open source - Platform independent.
  - Download: http://netbeans.org/
  - Configure NetBeans http://drupal.org/node/1019816
- Sublime Text 2 - Free evaluation. License fee for use. - Mac, Windows, Linux
  - Download: http://www.sublimetext.com/2
  - Configure Sublime Text http://drupal.org/node/1346890
- Refer to Development tools in documentation for more tips http://drupal.org/node/147789

1.1.2 Drupal Tools

Let's add a tool that will aid us in development:

- Devel module (http://drupal.org/project/devel) gives you a way to quickly debug your code.

- Coder module (http://drupal.org/project/coder) Checks your code for style errors.

CHAPTER

TWO

EXTENDING DRUPAL WITH CUSTOM CODE

Drupal is a modular, open source web content management framework that ships with basic functionality in the form of core modules. For the most part, additional functionality is added by enabling third party modules (known as contributed modules) that can be downloaded from the contributed modules section of Drupal.org - *http://drupal.org/project/modules*.

2.1 Drupal as a framework

Drupal "core" does very little out of the box. The richness of Drupal applications is largely due to modules which are added on to the core platform. In this way, Drupal is as much a "framework" as it is a product.

Technically speaking, what is a module?

A module is a directory which contains PHP code and optionally javascript or CSS files. When the administrator turns the module on via the modules page (*http://example.com/admin/modules*), this code is run on every page request.

What are some examples of modules?

- **AddThis:** Creates a widget to encourage sharing on social networks
- **Pathauto:** An automated tool to create user- and SEO-friendly URLs for every new piece of content.
- **Google Analytics:** Adds integration for Google Analytics.

2.1.1 The Page Request Process

When Drupal receives a page request, among other processes, it loads the core libraries and initializes the database. Drupal then loads any enabled modules and themes on the site, and begins a systematic process of handling the request. During this process Drupal typically checks what resources and code are needed to handle the request, whether or not the user requesting the page has access to it, and if any of the requested data can be retrieved from cache. Drupal continues through a list of similar operations until the request is complete.

From the perspective of modules, what is important to understand about this systematic process is that after each operation, Drupal offers modules the opportunity to hook into the operation and alter or handle it in a specific way. This is where hooks come into play.

2.1.2 Drupal Hooks

The hook system allows modules to reach into the request process and extend Drupal's functionality. This is not a Peter Pan reference, but rather has its origins in computer science. The idea is that you can "hook into" the program at specified points. Drupal says "Hey do any modules want to do anything when I save a new user?" And all modules which implement hook_user_presave() will get called when a user is saved. Think of it as similar to signing up to an email list with Amazon.com. Every time a new pet rock is put on the market, you'll be the first to know. In this silly case, you are a module and you are implementing hook_pet_rock_added().

Both drupal core and many contributed modules provide hooks that your module can implement in order to take a pass at modifying or augmenting Drupal's behavior. For a given hook, your module implements a function called [your_module_name]_hook_name. It is worth noting that Drupal hooks are additive - your hooks run in addition to other hooks rather than replacing hooks.

Example:

The function hook_menu($items) lets modules define which URLs they will handle.

Module name	Machine name	Function name
AddThis	addthis	addthis_menu()
Google Analytics	google_analytics	google_analytics_menu().

The "Observer Pattern"

You may have seen a similar pattern in other languages or frameworks. It is called an Observer pattern. Here are two examples:

jQuery

```
$('#mybutton').click(function() { alert('I was clicked'); });
```

Wordpress

Taken from http://codex.wordpress.org/Plugin_API

```
<?php
// Wordpess Hook Example:
class emailer {
  function send($post_ID) {
    $friends = 'bob@example.org,susie@example.org';
    mail($friends,"sally's blog updated",'I just put something on my blog:
http://blog.example.com');
    return $post_ID;
  }
}
add_action('publish_post', array('emailer', 'send'));
```

2.1.3 Example of a hook in Drupal

Here is a simple hook_node_insert to reflect the jQuery and Wordpress examples. Compare this to the examples above.

```
<?php

// Drupal Hook Example:
function mail_node_insert($node) {
  if($node->type == "blog") {
    $friends = 'bob@example.org,susie@example.org';
    mail($friends,"sally's blog updated",'I just put something on my blog:
    http://blog.example.com/node/'. $node->nid);
  }
}
```

A full list of Drupal 7 hooks can be found here - *http://api.drupal.org/api/drupal/includes!module.inc/group/hooks/7*. Take a couple of minutes to look over the hook list and get a sense for all of the ways you can hook into the system.

2.2 Your first module: The Red Button

Nodes do not have a direct deletion link like comments do in Drupal. Reviewing new nodes in teaser view and deleting inappropriate nodes can be made easier by adding a link to the delete form directly in the node links. Without this module, you would need to edit the node first and then select delete. Of course, there are better ways to do this, but this is a simple example of how you would implement a hook to modify the output on a page.

2.2.1 Exercise: Starting A New Module

Step 1: Choose a Descriptive Name

The first step in creating a new module is choosing a descriptive name. In this case, we are going to call our module "The Red Button."

Step 2: Create the sites/all/modules Directory

We will be placing The Red Button module within the sites/all/modules directory to keep our files organized. Many beginning Drupal Developers keep their custom modules directly in the modules folder under the docroot. This is a bad practice since you can no longer differentiate between "core" modules and modules you have added.

Step 3: Locate the module Folder

Next, we need to create the folder that will contain all of our module files. Create a new folder in sites/all/modules and name it redbutton.

Step 4: Create the .info File

Every Drupal module contains a .info file (pronounced dot-info). The .info file is a plain text file that gives Drupal specific information about the module. As with all module files, the .info file must be given the same name as the module, in this case redbutton. Go ahead and create the following blank file – sites/all/modules/redbutton/*redbutton.info*

Open up the redbutton.info file, and add the following information to it:

```
name = The Red Button
description = Adds a delete link to every article node.
core = 7.x
```

Warning: Clear your cache
After changing your module's files, such as the .info file, you have to clear the cache for your changes to take effect.

Step 5: Create the .module File

As with the .info file, the .module file must be given the same name as our module. Create the following blank file - sites/all/modules/redbutton/*redbutton.module*

Step 6: Adding your first hook

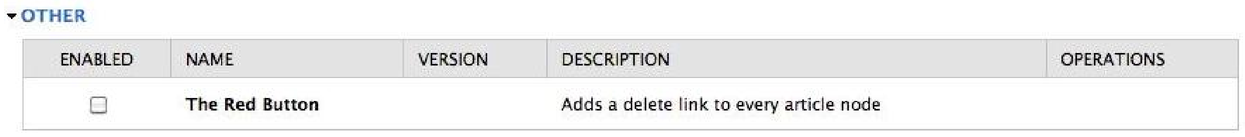
The first hook we will implement is hook_node_view() http://api.drupal.org/api/search/7/hook_node_view

```
<?php
/**
 * Implements hook_node_view().
 *
 * This hook gets called everytime a node is being rendered.
 *
 * The $node argument
 * contains the node object. By modifying $node->content, one can alter how the
 * node will be displayed.
 *
 * The $view_mode argument
 * is a string which describes how (or for what purpose) the node is being
 * displayed.
 * Most common values are "rss" (in a feed), "teaser" (short form in a listing),
 * and "full" (on its own page).
 *
 * This function does not return anything, but modifies the node object.
 */
function redbutton_node_view($node, $view_mode) {
  // Check the type of the node. We only want to show the link for articles.
  if ($node->type == 'article') {
    // The t() function is used to pass strings through Drupal's translation
engine.
    $link_text = t('Delete this node');
    // This is an array of options for the l() function (see below).
    $link_options = array(
      'attributes' => array('style' => 'color:#ff0000'),
    );
    // The l() function is used to generate HTML for a link (<a> tag).
    // The first parameter is the text of the link.
    // The second parameter is the href of the link.
    // The third parameter is an array of options. There are lots of available
options
    // see the documentation page
http://api.drupal.org/api/drupal/includes--common.inc/function/l/7.
    $link_markup = l($link_text, "node/$node->nid/delete", $link_options);
    $node->content['redbutton'] = array();
    $node->content['redbutton']['#markup'] = $link_markup;

  }
}
```

Step 7: Enable the module and test it.

Go to the Modules Administration page.

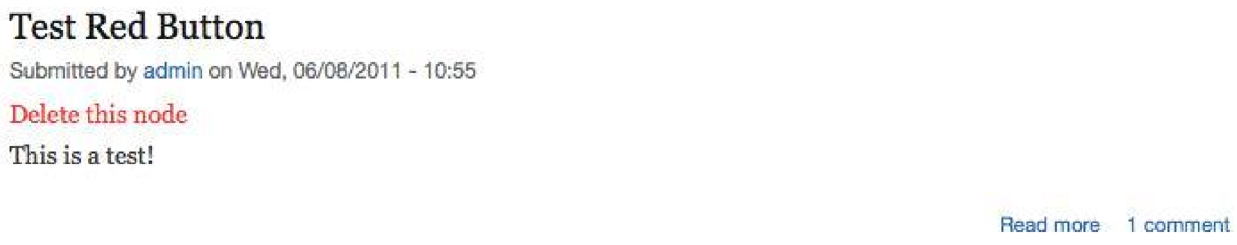


▾OTHER

ENABLED	NAME	VERSION	DESCRIPTION	OPERATIONS
☐	**The Red Button**		Adds a delete link to every article node	

Step 8: Test!

Make sure to clear your cache!



Test Red Button

Submitted by admin on Wed, 06/08/2011 - 10:55

Delete this node

This is a test!

Step 9: Breakdown

Let’s inspect some code, step by step.

```
<?php
// Red button module

/**
 * Implements hook_node_view().
 *
 * This hook gets called everytime a node is being rendered.
 * @param $node The node object being rendered.
 * @param $view_mode How the node is being viewed.
 */
```

Every hook implementation should contain this comment to make it easier for other developers to read your code and search for hook invocations.

Read about Doxygen comment formatting conventions: http://drupal.org/node/1354

```
 * @param $node The node object being rendered.
 * @param $view_mode How the node is being viewed.
```

The node object represents a single piece of content. In our example, we are only going to take action if the node type is ‘article’. This demonstrates how you add conditional logic inside of a hook.

The l() function is used to generate HTML for a link (<a> tag).

- The first parameter is the text of the link.
- The second parameter is the href of the link.
- The third parameter is an array of options. There are lots of available options.

See the documentation ‘http://api.drupal.org/api/function/l/7 ‘ to learn more.

```
$node->content['redbutton'] = array();
$node->content['redbutton']['#markup'] = $link_markup;
```

$node->content contains an array of elements which will be merged together when the final node display is being generated. This is fairly confusing for new people, but an unavoidable part of Drupal development. These "Renderable arrays" are covered in more detail later, but for now, just know that we are adding some additional output to the node display and the output we are adding is a link. The $link_markup variable is just HTML which looks something like <a href="node/1/delete">Delete this node</a>

2.3 Efficient coding

Best practices for coding in Drupal.

2.3.1 Drupal Coding Standards

The Drupal community has agreed that the Drupal codebase must adhere to a set of coding standards. This standardization makes the code more readable, and thus easier for developers to understand and edit each other's code. It is crucial that you learn these as you become involved in Drupal development.

Full reference at: http://drupal.org/coding-standards/

Also see *Appendix I: Drupal coding standards and conventions* at the end of this manual.

For now we're going to just highlight a few key points.

Line Indentation

Drupal code uses 2 spaces for indentation, with no tabs.

PHP Tags

Use opening PHP tags (<?php), but do not use closing PHP tags (?>).

Control Structures

Control structures control the flow of execution in a program and include: if, else, elseif, switch statements, for, foreach, while, and do-while. Control structures should have a single space between the control keyword and the opening parenthesis. Opening braces should be on the same line as the control keyword, whereas closing braces should have their own line.

```
if($a == $b) {
 do_this()
}
else {
 do_that()
}
```

Function Declarations

Function declarations should not contain a space between the function name and the opening parenthesis.

```
function mymodule_function($a, $b) {
  $do_something = $a + $b;
  return $do_something;
}
```

Arrays

Arrays should be formatted with spaces separating each element and assignment operator. If the array spans more than 80 characters, each element in the array should be given its own line.

```
$car['colors'] = array(
  'red' => TRUE,
  'orange' => TRUE,
  'yellow' => FALSE,
  'purple' => FALSE,
);
```

2.4 Introducing Mailfish

Next we will focus on creating a custom module.

What does this module do?

The module will allow users to enter their email address to sign up for mailing lists. The module will be called 'MailFish'.

Enable settings per-content type

MailFish will have a setting to allow administrators to enable signup functionality per-content type. For example, you may create a content type called 'Event' and set MailFish to allow signups on Event nodes.

Set MailFish capability when content is created

Each Event node would then include a form for users to submit their email addresses to be added to a list for that node. Administrators can also set for an individual node (of an enabled content type) whether or not to include the signup form.

These settings and the email addresses users submit will be stored to the database. We will create a reporting page where administrators can see the email addresses submitted for each node.

Display sign-up in a block

To explore Drupal's block and theme systems we will also use the same signup system to create an additional sitewide signup form as a block.

2.4.1 Exercise: Starting A New Module

Step 1: Choose a Descriptive Name

The first step in creating a new module is choosing a descriptive name. In this case, we are going to call our module MailFish.

Step 2: Create the module Folder

Next, we need to create the folder that will contain all of our module files. Create a new folder in sites/all/modules and name it mailfish.

Step 3: Create the .info File

Every Drupal module contains a .info file (pronounced dot-info). The .info file is a plain text file that gives Drupal specific information about the module. As with all module files, the .info file must be given the same name as the module, in this case mailfish. Go ahead and create the following file – sites/all/modules/mailfish/mailfish.info.

Open up the mailfish.info file, and add the following information to it:

```
name = MailFish
description = Example module that allows users to subscribe to nodes.
core = 7.x
package = MailFish
version = 0.2-dev
dependencies[] = block
configure = admin/config/content/mailfish
```

The required properties are name, description and core.

Breakdown

Here's a rundown of what each line means:

```
name = MailFish
```

The name of our module. This will be displayed in the module administration section of our website.

```
description = Example module that allows users to subscribe to nodes.
```

A description of our module. This will be displayed alongside the module name in the module administration section.

```
core = 7.x
```

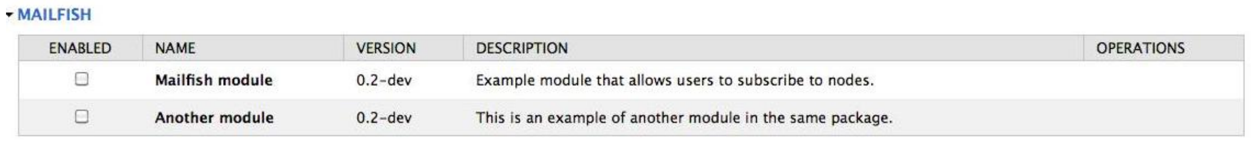
Here we specify the version of Drupal our module is compatible with – in this case, Drupal 7.

```
package = MailFish
```

This is used for grouping modules together on the modules admin page. For example, if we had two modules within the MailFish package – Module A and Module B, they would be grouped on the module admin page as follows:

MAILFISH

- Mailfish module
- Another module



▾ MAILFISH

ENABLED	NAME	VERSION	DESCRIPTION	OPERATIONS
☐	**Mailfish module**	0.2-dev	Example module that allows users to subscribe to nodes.	
☐	**Another module**	0.2-dev	This is an example of another module in the same package.	

```
version = 0.2-dev
```

Displays the version of the module on the module admin page.

```
dependencies[] = block
```

Other modules that our module depends on. The MailFish module depends on the block module, so we list this here. If our module depended on more than one module, we would repeat the above code - one line per dependency.

Finally, we specify the path of our module's main configuration page.

Step 4: Start mailfish.module

Create the mailfish.module file:

sites/all/modules/mailfish/mailfish.module

Open the mailfish.module file and add the following code to the beginning of the file:

```
<?php
/**
 * @file
 * Collect email addresses from a form within a node or block.
 */
```

This breaks down as follows:

```
<?php
```

As with any PHP file, we must start the file with an opening PHP tag. However, in Drupal, we do not use the closing PHP tag – ?> at the end of our scripts. This is due to trailing whitespace issues that closing tags can cause in files (see *http://drupal.org/coding-standards* for details).

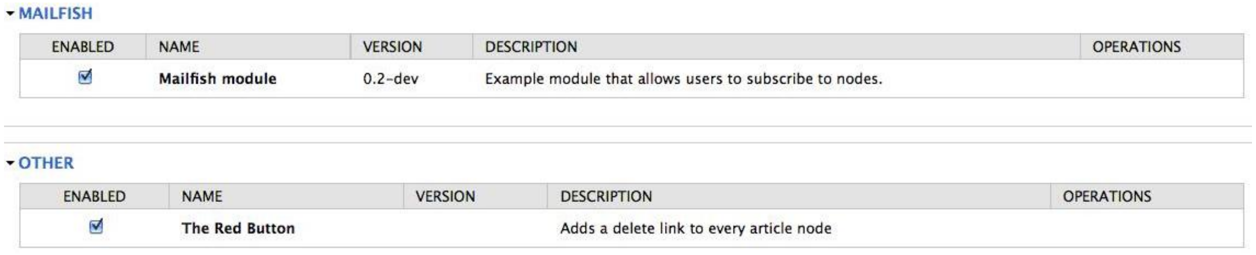
```
<?php
/**
 * @file
 * Collect email addresses from a form within a node or block.
 */
```

This is an example of a block comment. @file is a token that tells Drupal that these comments describe the file. There are strict conventions in Drupal when it comes to adding comments to your code and it is highly recommended that you adhere closely to them. Block comments used to describe a file or function begin with /**, and on each succeeding line we use a single asterisk indented with one space. */ on a line by itself ends the comments.

For more details on Drupal's coding standards it is highly recommended that you familiarize yourself with the relevant handbook page: *http://drupal.org/coding-standards*.

Step 5: Enable the module

In the Modules Administration list, MailFish will be listed under a package, with a version number. Compare this with The Red Button module.



MAILFISH

ENABLED	NAME	VERSION	DESCRIPTION	OPERATIONS
✓	**Mailfish module**	0.2-dev	Example module that allows users to subscribe to nodes.	

OTHER

ENABLED	NAME	VERSION	DESCRIPTION	OPERATIONS
✓	**The Red Button**		Adds a delete link to every article node	

CHAPTER

THREE

MENUS AND PERMISSIONS

3.1 Menu System: Defining menu callbacks

As mentioned at the beginning of this handout, modules integrate with Drupal using a system of hooks – specially named PHP functions in your .module file that allow your module to “hook” into Drupal at various stages during the page request process.

Drupal’s Menu API is both the system that creates the various menus used to navigate a site as well as the system that routes page requests to generate the appropriate page title and content for each URL (page callbacks). It also handles the access control for Drupal pages, and prevents unprivileged users from both accessing a protected page and seeing that page listed in menus.

Here, we want to take control of a particular path so that our module can provide an administration page. To do this we need to add an entry to Drupal’s menu routing system so that we may add an admin menu item for our new module at admin/config/content/mailfish. We will also want to add a page to report on the site’s subscription: to Drupal’s reports section: admin/reports/mailfish, so that administrators can view MailFish email subscriptions. In order to hook into Drupal’s menu system, we need to implement hook_menu() and to return a list of paths that our module will take responsibility for.

Open the API documentation

More information on hook_menu() can be found in Drupal’s API docs. Go to *http://api.drupal.org/api/function/hook_menu/7*

3.1.1 Exercise: Create your first page callback

```
<?php
function mailfish_menu() {
  $items = array();
  $items['admin/config/content/mailfish'] = array(
    'title' => 'MailFish Settings',
    'description' => 'Administer MailFish Settings.',
    'page callback' => 'mailfish_admin_settings_callback',
    'access arguments' => array('manage mailfish settings'),
  );
  return $items;
}

/**
 * Menu Callback; Allows admins to pick which content types should be enabled
 * for signups.
 */
```

```
function mailfish_admin_settings_callback() {
  return 'Settings form should go here.';
}
```

You can now go to admin/config/content/mailfish and see the output. But wait! It's still not showing?

Caching

Something you need to be aware of at this point is that Drupal caches anything it can to speed up page requests. This includes what modules implement what hooks and every entry in the menu system, so new or updated menu items will not be visible on your site until the cache is cleared. Being aware of this fact can save you hours of banging your head against the wall trying to figure out why your new menu item is not showing up on your site.

It is therefore important that you get into the habit of clearing the site's cache every time you add a new menu item. To do this manually, simply visit the Performance admin page – Admin>Configuration>Development>Performance, and click on Clear all caches. It is worth noting that many developers install the admin menu module (*http://drupal.org/project/admin_menu*) for a more convenient way of clearing the cache or install the drush command line utility (*http://drupal.org/project/drush*) to make this more convenient. Both are highly recommended for developers.

Menu Callback Files (.admin.inc, .pages.inc)

There are a few important conventions regarding files that should be noted at this point.

1. All hook implementations belong in the main .module file.
2. Administrative menu callbacks (and related functions) go in a .admin.inc file.
3. Menu callbacks for non-admin pages go in a .pages.inc file.

These conventions help to keep things organized, and make it easier for other developers to locate the various components of our module. However, there is also an important performance aspect to this. Menu callbacks only need to be loaded and parsed when needed. Therefore, we put them in separate files as opposed to including them in our .module file, which is loaded and ran with every web request.

3.1.2 Exercise: Define menu items with hook_menu

Step 1: Implement hook_menu

Here is the full hook_menu implementation. You can replace the code we added above with the following:

```
<?php
/**
 * Implements hook_menu().
 */
function mailfish_menu() {
  $items = array();
  $items['admin/config/content/mailfish'] = array(
    'title' => 'MailFish Settings',
    'description' => 'Administer MailFish Settings.',
    'page callback' => 'drupal_get_form',
    'page arguments' => array('mailfish_admin_settings_form'),
    'access arguments' => array('manage mailfish settings'),
    'file' => 'mailfish.admin.inc',
  );
  $items['admin/reports/mailfish'] = array(
    'title' => 'MailFish Signups',
    'description' => 'View MailFish Signups',
    'page callback' => 'mailfish_signups',
    'access arguments' => array('view mailfish subscriptions'),
    'file' => 'mailfish.admin.inc',
  );
  return $items;
}
```

The code breaks down as follows:

```
<?php
/**
 * Implements hook_menu().
 */
```

Every time you implement a hook in your code, the coding standards dictate that we use this comment format so that other developers can easily tell which Drupal hook you are implementing and so that implementations can easily be found by searching your code.

function mailfish_menu() {

You'll notice here that hook_menu() has now become mailfish_menu(). For any hook you use in your module, ALWAYS remove the word 'hook', and replace it with the name of your module.

```
<?php
function mailfish_menu() {
  $items = array();
  $items['admin/config/content/mailfish'] = array(
    'title' => 'MailFish Settings',
    'description' => 'Administer MailFish Settings.',
    'page callback' => 'drupal_get_form',
    'page arguments' => array('mailfish_admin_settings_form'),
    'access arguments' => array('manage mailfish settings'),
    'file' => 'mailfish.admin.inc',
  );
```

Notice the title parameter in hook_menu is automatically parsed through t(), unlike all other strings that do require that call.

Here we are defining our MailFish admin menu item using an array to describe its various elements to Drupal. This code says, "When a site admin goes to *http://example.com/admin/config/content/mailfish*, call the function mailfish_admin_settings_callback." Drupal expects that function to return content which will make up the middle of the page a.k.a. the "content" region. Finally, only allow users with the permission manage mailfish settings to view this menu item.

```
<?php
  $items['admin/reports/mailfish'] = array(
    'title' => 'MailFish Signups',
    'description' => 'View MailFish Signups',
    'page callback' => 'mailfish_signups',
    'access arguments' => array('view mailfish subscriptions'),
    'file' => 'mailfish.admin.inc',
  );
```

This code works in exactly the same way as the previous block only here we are defining our reporting page menu item at *http://example.com/admin/reports/mailfish*. We won't be building this page until a later chapter, but just leave it in there for now.

3.2 The Permissions System

Drupal's role-based permissions system is typically used to determine access to menu items. Many modules define their own permissions to appear at /admin/user/permissions.

Permissions are added to the system by implementing hook_permission(). Whether the current user is in a role with a given permission is determined by function user_access().

3.2.1 Permissions

We will want to add permission settings to our module so that administrators can assign Mailfish access permissions to different roles on the permission settings page - *http://example.com/admin/people/permissions*.

To do this, we will implement the permissions hook – hook_permission(), in the mailfish.module file. More information on hook_permission() can be found here - *http://api.drupal.org/api/function/hook_permission/7*

Exercise: Define new permissions with hook_permission

Open up the mailfish.module file if it's not already open, and add the following code to it:

```
<?php

/**
 * Implements hook_permission().
 */
function mailfish_permission() {
  $perm = array(
    'view mailfish subscriptions' => array(
      'title' => t('View Mailfish subscriptions'),
    ),
    'create mailfish subscriptions' => array(
      'title' => t('Create Mailfish subscriptions'),
    ),
    'manage mailfish settings' => array(
      'title' => t('Manage Mailfish settings'),
    )
  );
  return $perm;
}
```

The above code informs Drupal that there are 3 specific permissions that are used to control user access by our module:

- view mailfish subscriptions
- create mailfish subscriptions
- manage mailfish subscriptions

Administrators will now be able to assign these permissions to different roles on the permissions settings page. Clear the cache and test it at *http://example.com/admin/people/permissions*.

3.3 Menu Callback Files (.admin.inc, .pages.inc)

There are a few important conventions regarding files that should be noted at this point.

1. All hook implementations must be added to the main [modulename].module file because it is the only file that is included in every bootstrap.
2. Administrative menu callbacks (and related functions) go in a [modulename].admin.inc file.
3. Menu callbacks for non-admin pages go in a [modulename].pages.inc file (There are no non-admin pages associated with our MailFish module so we won't be using a .pages.inc file).

These conventions help to keep things organized, and make it easier for other developers to locate the various components of our module. It's important to stress that this is just a convention and that Drupal does not automatically recognize these files. This is what the file parameter in the hook_menu we created does: 'file' => 'mailfish.admin.inc'.

There is also an important performance aspect to this. Menu callbacks only need to be loaded and parsed when needed. Therefore, we put them in separate files as opposed to including them in our .module file, which is evaluated on every page request.

3.3.1 Exercise: Creating the mailfish.admin.inc file

Step 1: Starting mailfish.admin.inc

Create the following file and place it in the mailfish module folder:

sites/all/modules/mailfish/mailfish.admin.inc

Open up the mailfish.admin.inc file and add the following code to it:

```
<?php
/**
 * @file
 * Provide the admin related functions for the MailFish module.
 */
```

Nothing new here - again, we have our opening PHP tag and our comments that tell Drupal what our file is for.

CHAPTER

FOUR

FORM API(PART 1)

4.1 Form System: Drupal Form API and Form Creation

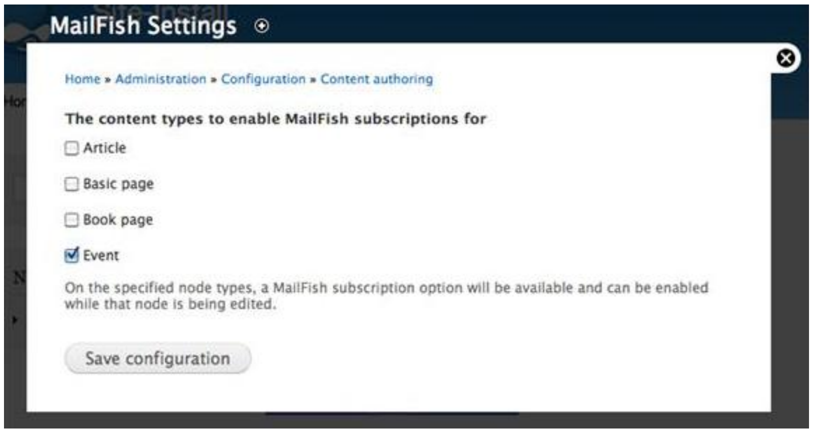
The Drupal Form API provides a framework for building forms in Drupal. Forms in Drupal are described as a nested tree structure – an array of arrays. This structure tells Drupal how the form should be rendered.

Each form element's properties or attributes are listed items in the array corresponding to the element that they describe. In some cases form elements can be nested inside other form elements and so attributes are always prefixed with the '#' symbol so that Drupal may distinguish between a form element and an element's attributes.

Form API Quick Start: *http://drupal.org/node/751826*

Form API Reference: *http://api.drupal.org/api/drupal/developer!topics!forms_api_reference.html/7*.

4.1.1 Exercise: Defining The System Settings Form in mailfish.admin.inc


MailFish Settings

Home » Administration » Configuration » Content authoring

The content types to enable MailFish subscriptions for

☐ Article

☐ Basic page

☐ Book page

☑ Event

On the specified node types, a MailFish subscription option will be available and can be enabled while that node is being edited.

Form IDs:

Every form in Drupal has an ID. The ID is a string which happens to be the name of the function which is responsible for defining the form (99% of the time).

You've already defined a menu callback for the settings form, but right now it just spits out some placeholder text. We're going to make it show the form instead.

```
<?php
/**
 * Menu Callback; Allows admins to pick which content types should be enabled for signups.
 */
```

```
function mailfish_admin_settings_callback() {
  return 'Settings form should go here.';
}
```

Let's replace the line inside that function so it looks like this:

```
<?php
/**
 * Menu Callback; Allows admins to pick which content types should be  enabled
 * for signups.
 */
function mailfish_admin_settings_callback() {
  return drupal_get_form('mailfish_admin_settings_form');
}
```

Now if you load the page, it should give you a nasty error. This is because the function mailfish_admin_settings_form doesn't exist yet.This is the Form ID for our form.So we need to define a function with the name *mailfish_admin_settings_form* which will define the form structure.

You're probably also wondering what drupal_get_form($form_id) does now. drupal_get_form is a complicated function. But at the heart of it, it expects to get a function name, it calls that function and it expects that function to return an array in a specific format that describes a form. It then does some magic to get it ready for display and returns a renderable array which can be passed to the theme. For your purposes at the moment, it is a form making factory.

Open up mailfish.admin.inc and add the following code:

```
<?php
function mailfish_admin_settings_form() {
  $form = array();
  $form['mailfish_types'] = array(
    '#title' => t('The content types to enable MailFish subscriptions for'),
    '#description' => t('On the specified node types, a MailFish subscription
option will be available and can be enabled while that node is being edited.'),
    '#type' => 'checkboxes',
    '#options' => node_type_get_names(),
    '#default_value' => variable_get('mailfish_types', array()),
  );
  return system_settings_form($form);
}
```

Save and close the mailfish.admin.inc file.

Reading the code

The code breaks down as follows

```
<?php
/**
 * Defines the MailFish admin settings form.
 */
```

More comments are always better. This describes to a developer what this form is for.

```
<?php
  $form['mailfish_types'] = array(
    '#title' => t('The content types to enable MailFish subscriptions for'),
    '#description' => t('On the specified node types, a MailFish subscription
option will be available and can be enabled while that node is being edited.'),
    '#type' => 'checkboxes',
    '#options' => node_type_get_names(),
    '#default_value' => variable_get('mailfish_types', array()),
  );
```

This section of code creates a "checkboxes" field. You can see there are several attributes to fill out. The title is the label that goes with the form, the description is helper text which usually appears under the checkboxes, options are the available checkboxes (in this case a list of all node types), the default_value is what will be checked by default. We'll cover node_type_get_names and variable_get below, so don't worry if you don't know what those are right now.

Each property name is prefixed with a '#' which tells Drupal that this element describes the form element and is not, itself, a form element. Please refer to the Form API reference listed above for a full list of the available form elements and their properties.

```
<?php
  return system_settings_form($form);
}
```

We've built our admin settings form and now we pass the form array through the Drupal function - *system_settings_form()*. This saves us time as *system_settings_form()* automatically takes care of some requirements for us, including providing a submit button and submit handler for our form.

More information on this function can be found here: *http://api.drupal.org/api/function/system_settings_form/7*. At this point, you should be able to test your form by *visiting admin/config/content/mailfish.*

API tips - Using the *node_type_get_names()* function:

http://api.drupal.org/api/function/node_type_get_names/7
This function builds a list of node types currently available on our site.
This list will then be displayed on our form as a list of checkboxes. Finally, we declare the default value to be displayed on the form when it is loaded – in this case the default value will be a list of the available node types.

4.2 Drupal Variables

In the above code under *$types*, notice that we used a special Drupal function – *variable_get()*, in order to grab the *mailfish_types* array. Drupal allows you to store and retrieve any value using the Drupal functions *variable_set()*, and *variable_get()*, respectively. The values are stored in the variables database table and are available anytime while processing a request.

This is a very convenient system for storing module configuration settings. Any variable that you store in the variables database table using the *variable_set()* function, can be removed using the *variable_del()* function. Here we have not used the *variable_set()* function directly but it is called by the form submission callback that is used by all forms passed through *system_settings_from()*.

More information on these functions can be found:

- variable_get() - *http://api.drupal.org/variable_get*
- variable_set() - *http://api.drupal.org/variable_get*
- variable_del() - *http://api.drupal.org/variable_get*

t(): Translatable

The translation function or t function (refer to the #title property in the above code) as it is more commonly referred to, facilitates string translation in Drupal. By running all text through the t function, our module can be localized into different languages via the Locale module. *http://drupal.org/handbook/modules/locale*.

4.3 Challenge: Using drupal_get_form as a callback

If a menu callback only generates a form (as we are doing), there is a more direct way of doing the same task:

```
<?php
  $items['admin/config/content/mailfish'] = array(
    'title' => 'MailFish Settings',
    'description' => 'Administer MailFish Settings.',
    'page callback' => 'drupal_get_form',
    'page arguments' => array('mailfish_admin_settings_form'),
    'access arguments' => array('manage mailfish settings'),
    'file' => 'mailfish.admin.inc',
  );
```

Instead of calling the page callback we defined (mailfish_admin_settings_callback)which then in turn calls drupal_get_form, we can make drupal_get_form the page callback. However, we still need to tell drupal_get_form what form_id to generate. This is done by supplying "page arguments" . The end result is that when someone navigates to admin/config/content/mailfish, the function drupal_get_form is called like this:

```
drupal_get_form('mailfish_admin_settings_form');
```

And the result of that function will be returned and displayed on the page. Page arguments are used extensively in almost every module. Read up the possibilities on http://api.drupal.org (search hook_menu).

Hook_form_alter(): Alter other modules' forms

Hook_form_alter() allows us to alter form arrays defined by other modules before the form is rendered. By altering forms we can add additional form elements, change or remove existing elements, and even change or add to the validation and submission handling of the form. Because all Drupal forms use the Form API, any module can alter any form. Important forms like the node creation/edit form are often altered by many modules, each one adding its own additional fields.

Most *hook_form_alter* implementation should start off with a switch to determine which form we're working with (usually based on the form's id). You can also implement hook_form_FORM_ID_alter() to target a specific form. This approach is generally the best practice. See http://api.drupal.org/api/function/hook_form_FORM_ID_alter/7

Implementations of hook_form_FORM_ID_alter() are run after the hook_form_alter() implementations have already run.

Drupal_alter and alter hooks

Alter functions are a common pattern in the Drupal API, allowing other modules to hook in and alter variables.
In addition to hook_form_alter commonly used 'alter' hooks include hook_menu_alter, hook_query_alter, and hook_node_view_alter.
'Alter' hooks are invoked by using function *drupal_alter()*. To invoke your own new alter hook, just call

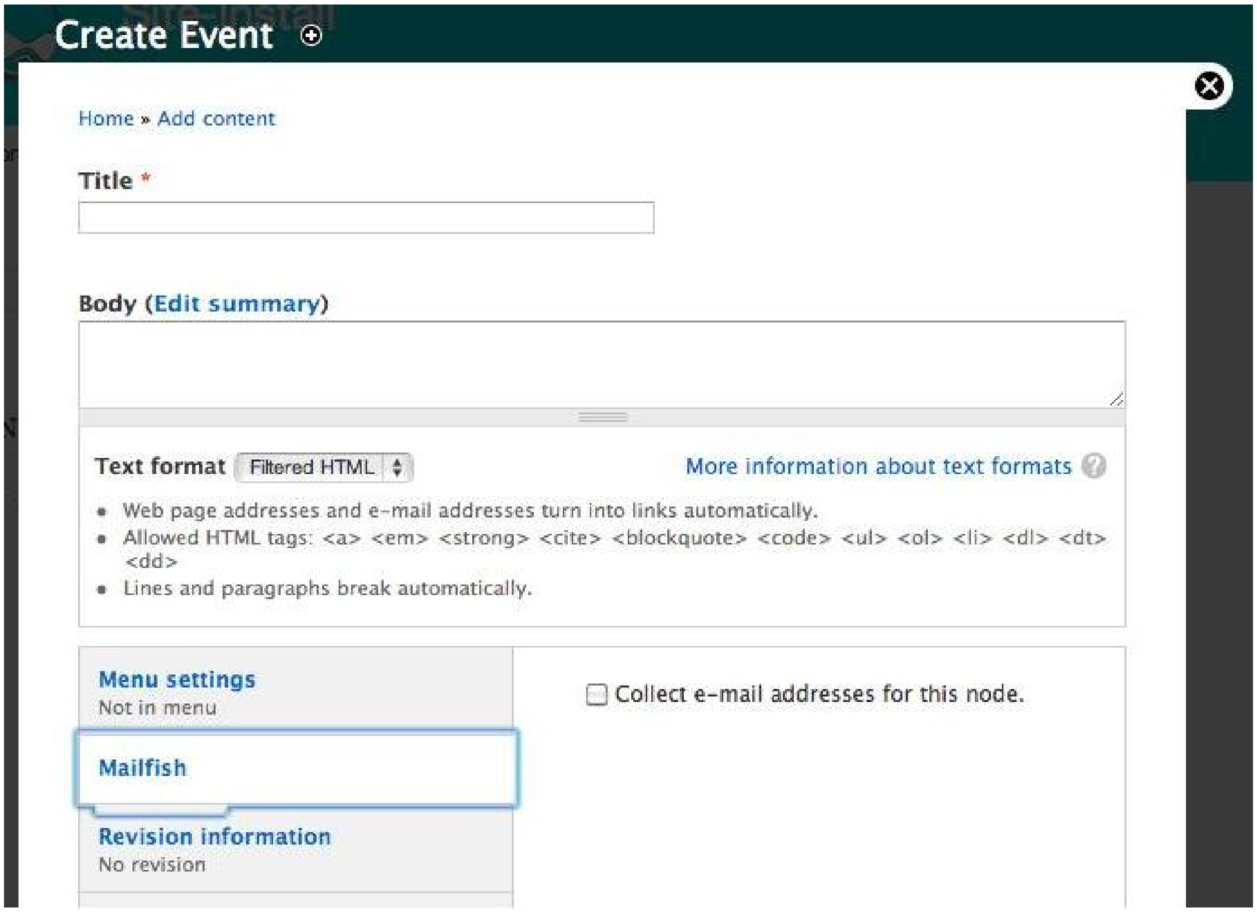
drupal_alter('mymodule_data', $data);

which will allow other modules to implement

hook_mymodule_data_alter(&$data)

to alter your data.

4.3.1 Exercise: Add a per-node setting with hook_form_alter()



Create Event

Home » Add content

Title *

Body (Edit summary)

Text format: Filtered HTML — More information about text formats

- Web page addresses and e-mail addresses turn into links automatically.
- Allowed HTML tags: <a> <em> <strong> <cite> <blockquote> <code> <ul> <ol> <li> <dl> <dt> <dd>
- Lines and paragraphs break automatically.

Menu settings — Not in menu

Mailfish

Revision information — No revision

☐ Collect e-mail addresses for this node.

We need to add a checkbox that allows admins to enable or disable per node email address collection for specific content types. This checkbox will appear on the node creation/edit form of MailFish -enabled content types. We will use an instance of hook_form_alter() - *http://api.drupal.org/api/drupal/modules–system–system.api.php/function/hook_form_alter/7* added to our mailfish.module file.

Copy the following code into mailfish.module and save the file:

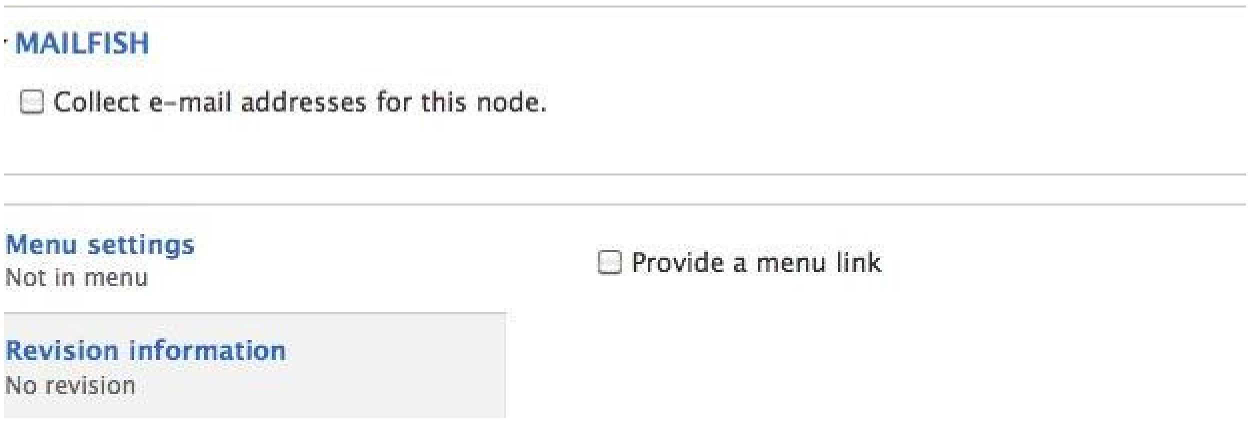
```
<?php
/**
 * Implements hook_form_FORM_ID_alter().
 *
 * Adds a checkbox to allow email address collection per node for
 * enabled content types.
 */
function mailfish_form_node_form_alter(&$form, $form_state) {
  $node = $form['#node'];
  // Perform our check to see if we should be performing an action as the very
first action.
  $types = variable_get('mailfish_types', array());
  // Check if this node type is enabled for mailfish
  // and that the user has access to the per-node settings.
  if (!empty($types[$node->type]) && user_access('manage mailfish settings')) {
    // Add a new fieldset with a checkbox for per-node mailfish setting.
    $form['mailfish'] = array(
      '#title' => t('MailFish'),
      '#type' => 'fieldset',
      '#collapsible' => TRUE,
      '#collapsed' => FALSE,
      '#group' => 'additional_settings',
    );
    $form['mailfish']['mailfish_enabled'] = array(
      '#title' => t('Collect e-mail addresses for this node.'),
      '#type' => 'checkbox',
      '#default_value' => isset($node->mailfish_enabled) ?
                            $node->mailfish_enabled : FALSE,
    );
  }
}
```

The *$form* parameter is passed by reference (hence the '&' symbol when the parameter is loaded as &$form) and therefore we have the ability to modify form definition directly without having to return anything. It is worth noting again that Drupal hooks are added to and not replaced so we must keep in mind that other modules may modify our form before or after our module has modified the form.

The *$form['mailfish']* section of the code is where our new checkbox field is defined. It is an important convention that modules namespace their additions to a form in case two modules add similar elements to the form, Drupal can never have two modules by the same name so we can safely use our module's name to identify a portion of the form.

Test the code

Clear cache and test your new form. If you enabled MailFish for Articles, test by creating a new article. It won't do anything yet, but you can see the changes. The MailFish options appear as a collapsible field at the bottom of the node create or edit form.



Modifying the code: Placing into vertical tabs settings

As a final step, we can move our new options into the vertical tabs section. In your module, add this line:

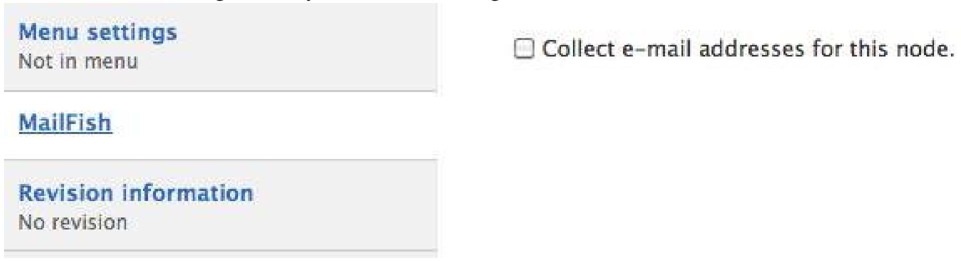
> *'#group' => 'additional_settings',*

as the last item in the form settings array:

```
<?php
    $form['mailfish'] = array(
      '#title' => t('MailFish'),
      '#type' => 'fieldset',
      '#collapsible' => TRUE,
      '#collapsed' => FALSE,
      '#group' => 'additional_settings',
```

Test again

Now test the form again Do you see the settings in the vertical tabs?



Reading the code

The code breaks down as follows:

```
<?php
/**
 * Implements hook_form_FORM_ID_alter().
 *
 * Adds a checkbox to allow email address collection per node for
 * enabled content types.
 */
```

We start with our opening PHP tag and our comments that explain what hook is being implemented, and notes about what this code does.

```
<?php
function mailfish_form_node_form_alter(&$form, $form_state) {
```

mailfish is the module name, node_form is form ID.

If someone else wanted to modify the administration settings form we created earlier (form id: mailfish_admin_settings), what would the function name be?

```
function mailfish_form_mailfish_admin_settings_alter(&$form, $form_state) {
}
```

Let's look at the two parameters this form alter takes.

&$form

This is the form structure we are modifying. It is a nested associative array of form elements and their settings, just like we defined earlier when building the admin settings form.

The ampersand "&" before the variable passes the value by reference. Without the & the function copies the value of the variable. With the & ampersand, the variable is shared. This means the function can alter the value of the variable.

$form_state

This is an array which will eventually contain the submitted values when the form has been submitted.

```
<?php
  $types = variable_get('mailfish_types', array());
  // Check if this node type is enabled for mailfish
```

Array for the types for which we enabled mailfish; for example Basic page and Article - remember, we build the form earlier for setting these.

```
<?php
  // and that the user has access to the per-node settings.
  if (!empty($types[$node->type]) && user_access('manage mailfish settings')) {
```

This condition tests if a user has the appropriate access to see this field.

```
<?php
    $form['mailfish'] = array(
      '#title' => t('MailFish'),
      '#type' => 'fieldset',
      '#collapsible' => TRUE,
      '#collapsed' => FALSE,
```

This is a group of fields (a.k.a. a fieldset).

```
<?php
    $form['mailfish']['mailfish_enabled'] = array(
      '#title' => t('Collect e-mail addresses for this node.'),
      '#type' => 'checkbox',
      '#default_value' => isset($node->mailfish_enabled) ?
```

When building forms in Drupal, you can put fields inside of a fieldset by making the forms which go inside children of that array. Here we are adding the checkbox field inside the fieldset.

```
<?php
    );
    $form['mailfish']['mailfish_enabled'] = array(
      '#title' => t('Collect e-mail addresses for this node.'),
      '#type' => 'checkbox',
      '#default_value' => isset($node->mailfish_enabled) ?
                           $node->mailfish_enabled : FALSE,
    );
      '#collapsible' => TRUE,
```

This places this setting into vertical tabs. Don't worry too much about this, it is an idiosyncrasy of this form, but just know that it adds a group on the node form.

CHAPTER

FIVE

INSTALL FILES

Install files are run the first time a module is enabled, and are used to run setup procedures as required by the module. The most common task of the .install file is to create the necessary database tables and fields for your module. Disabling and enabling a module will not cause these procedures to be run again. Disabling, uninstalling and then enabling a module will.

Term	Definition
Enable	Activate a module for use by Drupal
Disable	Process of deactivating a module
Install	Enable for the first time or after it has been uninstalled
Uninstall	Remove all traces of a module

There are three Drupal hooks commonly associated with *.install files*:

1. *hook_schema()*
   - Defines a module's database tables and fields. The schema will be created when the module is installed.
   - http://api.drupal.org/api/function/hook_schema/7
2. *hook_install()*
   - Takes care of additional setup tasks during installation.
   - http://api.drupal.org/api/function/hook_install/7
3. *hook_uninstall()*
   - Removes a module's database tables and fields should it be uninstalled.
   - http://api.drupal.org/api/function/hook_uninstall/7

5.1 Exercise: Creating The mailfish.install File

5.1.1 Step 1:

Create a new empty install file at: sites/all/modules/mailfish/mailfish.install

5.1.2 Step 2: Defining The MailFish Schema

Add the following code to *mailfish.install* and save the file:

```php
<?php
/**
 * Implements hook_schema().
 */
function mailfish_schema() {
  $schema['mailfish'] = array(
    'description' => 'Stores the email address, timestamp, node id and user id if any',
    'fields' => array(
      'id' => array(
        'description' => 'The primary identifier for the entry.',
        'type' => 'serial',
        'unsigned' => TRUE,
        'not null' => TRUE,
      ),
      'uid' => array(
        'description' => 'The {users}.uid that added this subscription.',
        'type' => 'int',
        'not null' => TRUE,
        'default' => 0,
      ),
      'nid' => array(
        'description' => 'The {node}.nid that this subscription was added on.',
        'type' => 'int',
        'not null' => TRUE,
        'default' => 0,
      ),
      'mail' => array(
        'description' => 'User\'s email address.',
        'type' => 'varchar',
        'length' => 64,
        'not null' => FALSE,
        'default' => '',
      ),
      'created' => array(
        'type' => 'int',
        'not null' => TRUE,
        'default' => 0,
        'description' => 'Timestamp for when subscription was created.',
      ),
    ),
    'primary key' => array('id'),
    'indexes' => array(
      'node' => array('nid'),
      'node_user' => array('nid', 'uid'),
    ),
  );
  $schema['mailfish_enabled'] = array(
    'description' => 'Tracks whether MailFish is enabled for a given node.',
    'fields' => array(
      'nid' => array(
        'description' => 'The {node}.nid that has MailFish enabled.',
        'type' => 'int',
        'not null' => TRUE,
        'default' => 0,
      ),
    ),
    'primary key' => array('nid'),
  );
  return $schema;
}
```

This schema describes the MailFish database table, which has 5 fields, 1 of type ‘serial’, 1 of type ‘varchar’, and the remaining 3 of type ‘int’. A second table is also defined – mailfish_enabled with 1 field of type ‘int’.

5.1.3 Step 3: Implementing hook_uninstall()

We now need to add hook_uninstall() to the mailfish.install file. This removes the mailfish_types configuration settings from the variables table in the database when the MailFish module is uninstalled. This keeps things clean by removing unnecessary data from the database when it is no longer needed.

Add the following code to mailfish.install:

```
<?php
/**
 * Implements hook_uninstall().
 */
function mailfish_uninstall() {
  drupal_uninstall_schema('mailfish');
  variable_del('mailfish_types');
}
```

Reinstalling modules

Our new hook_install will only run when the module is first installed. Because our module is already installed, we need to uninstall and reinstall it to have it run the *hook_install* function and create the schema.
To uninstall a module, first disable it, and then uninstall it on the ‘Uninstall’ tab of the module administration page. Finally re-enable the uninstalled module.

Hook_update_N()

Update functions (not needed for MailFish) are also found in *.install files*.

When updating a module, it is often necessary to make corresponding updates to the module’s database schema. This is achieved by implementing serially numbered hook_update_N() functions - *http://api.drupal.org/api/function/hook_update_N/7*.

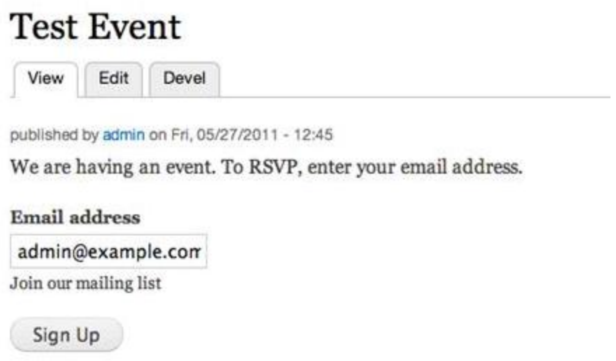
Database updates are run by visiting update.php and going through the steps. It is during this process that *hook_update_N()* gets called. The goal of this process is to bring the old database into synchronization with the expectations of the new module code. The site’s status report will warn you if there are update functions that have not yet been run.

CHAPTER

SIX

FORM API(PART 2)

6.1 Exercise: Building The Email Submission Form


Test Event

View Edit Devel

published by admin on Fri, 05/27/2011 - 12:45

We are having an event. To RSVP, enter your email address.

Email address

admin@example.com

Join our mailing list

Sign Up

The next thing we need to do is build a form to collect our email address submissions. This form is built in exactly the same way as our admin settings form, and will be displayed on nodes that are set to be MailFish enabled.

Re-open the mailfish.module file and add the following code:

```
<?php
/**
 * Implements hook_node_view().
 */
function mailfish_node_view($node, $view_mode, $langcode) {
  // If appropriate, add the mailfish email form to the node's display.
  if (!empty($node->mailfish_enabled) && user_access('create mailfish
subscription')) {
    $node->content['mailfish'] = array(
      '#markup' => drupal_render(drupal_get_form('mailfish_email_form',
$node->nid)),
      '#weight' => 100,
    );
  }
}
```

Don't worry if you don't understand what mailfish_node_view is doing. We'll be revisiting this in a later chapter and improving upon this code. All you need to know for now is that whenever a node is viewed, the signup form you see above will be shown below the content.

```
<?php
/**
 * Provide the form to add an email address.
 */
function mailfish_email_form($form, $form_state, $nid = 0) {
  global $user;
  $form['email'] = array(
    '#title' => t('Email address'),
    '#type' => 'textfield',
    '#size' => 20,
    '#description' => t('Join our mailing list'),
    '#default_value' => isset($user->mail) ? $user->mail : '',
  );

  $form['submit'] = array(
    '#type' => 'submit',
    '#value' => t('Sign Up'),
  );
  $form['nid'] = array(
    '#type' => 'hidden',
    '#value' => $nid,
  );
  return $form;
}
```

This is designing the form. Notice how we build an array structure of elements which have a *#type*, *#title* and *#default_value* or *#value* like we did with the earlier settings form checkboxes.

6.2 Form Validation and Submission

We will need to validate our form in order to ensure that the email addresses we are receiving are indeed genuine. To do this, we will build a form validation handler.

The form validation handler uses the same name as our form function *(mailfish_email_form)*, only we append *'_validate'* to the end of it so that it becomes *mailfish_email_form_validate.*

6.2.1 Exercise: Build the Validation Handler

```
<?php
/**
 * Validation handler for mailfish_email_form.
 */
function mailfish_email_form_validate($form, &$form_state) {

  $email = $form_state['values']['email'];
  if (!$email) {
    form_set_error('email', t('You must provide an email address in order to
join a mailing list.'));
  }
  elseif (!valid_email_address($email)) {
    $message = t('The address %email is not a valid email address. Please
re-enter your address.', array('%email' => $email));
    form_set_error('email', $message);
  }
  $nid = isset($form_state['values']['nid']) ? $form_state['values']['nid'] : 0;
  if (!mailfish_get_node_enabled($nid) && $nid != 0) {
    form_set_error('', t('MailFish subscriptions are not available for this
node.'));
  }

  // Do not allow multiple signups for the same node and email address.
  $previous_signup = db_query("SELECT mail FROM {mailfish} WHERE nid = :nid AND
mail = :mail", array('nid' => $nid, 'mail' => $email))->fetchField();
  if ($previous_signup) {
    form_set_error('email', t('The address %email is already subscribed to this
list.', array('%email' => $email)));
  }
}
```

WAIT! It broke!

You'll notice if you tried to submit the form, you'll get an error because the mailfish_get_node_enabled function doesn't exist yet. Just go ahead and create it as a "stub" function (i.e. a function without the code written for it yet):

```
<?php
/**
 * Determine if a node is set to display an email address form.
 *
 * @param int $nid
 *   The node id of the node in question.
 *
 * @return boolean
 */
function mailfish_get_node_enabled($nid) {
  // @TODO: This function is just a stub.
  return TRUE;
}
```

What does this code do?

This code first checks to see if an email address has been entered. If no email address has been entered, Drupal issues a warning – "You must provide an email address in order to join a mailing list", and prevents the form from being submitted. Next, the code checks to ensure that the email address entered is valid using the Drupal function – valid_email_address() - *http://api.drupal.org/api/drupal/includes–common.inc/function/valid_email_address/7*.

We also check to ensure MailFish subscriptions are available for the specific node the user is trying to subscribe to, and check to ensure that the user has not already subscribed to the list if one exists.

6.2.2 Exercise: Build The Submission Handler

In order to tell Drupal what to do with the data submitted via the email submission form, we need to build a submission handler. If the email address passes validation, then it is handed over to the submit handler for further processing. A submit handler is built in much the same way as our validation handler, only instead of appending _validate to the form name, we append _submit. Submit handlers should generally call another function to perform their duties in order to facilitate programmatic usage.

Copy the following code into mailfish.module and save the file:

```
<?php
/**
 * Submission handler for mailfish_email_form.
 */
```

```
function mailfish_email_form_submit($form, &$form_state) {
  // The sitewide signup form will not have a set $form['#node'].
  $nid = isset($form_state['values']['nid']) ? $form_state['values']['nid'] : 0;
  if ($nid) { // Comment: it might be a good idea to add an extra
is_numeric($nid) check here

    $node = node_load($nid);
  }
  // The sitewide signup form will not have a title, retrieve and use the
site's name.
  $title = isset($node) ? $node->title : variable_get('site_name', 'Drupal');

  // Signup the user.
  mailfish_signup($form_state['values']['email'], $nid);
  // Provide the user with a translated confirmation message.
  drupal_set_message(t('Thank you for joining the mailing list for %title. You
have been added as %email.', array('%title' => $title, '%email' =>
$form_state['values']['email'])));
}
```

WAIT! It broke again!

Create another "stub" function for mailfish_signup(). We'll be finish this in a couple chapters.

```
<?php
/**
 * Store a mailfish email signup.
 */
function mailfish_signup($email, $nid, $account = NULL) {
  // @TODO: Finish this function so it stores to the database.
  drupal_set_message(t('Pretending to signup %email to %nid', array('%email' =>
$email, '%nid' => $nid)));
}
```

CHAPTER

SEVEN

EXPLORING MODULES

The best way to learn how to be a great Drupal developer is to study the efforts of those who have come before you. Although not all the code will be comprehensible to you, here are a few modules which are using some of the same hooks you've seen. Try to read through them as best you can and determine how they work.

Install and enable the following modules and any necessary dependencies:

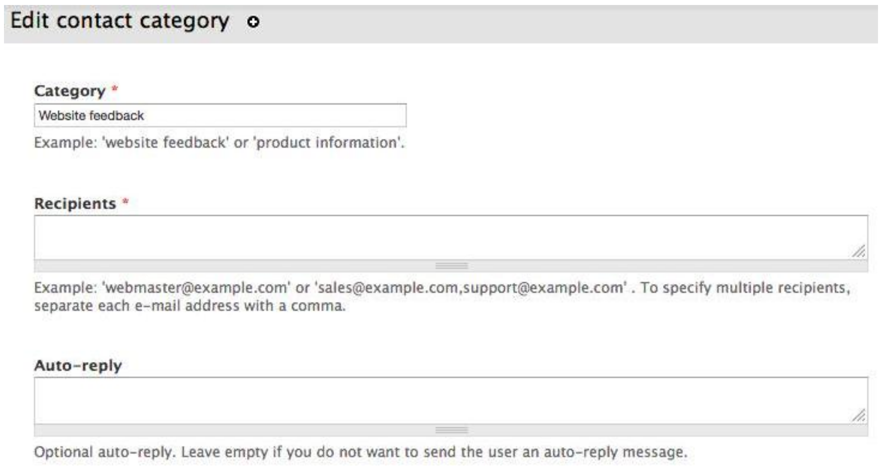
- Contact module – listed under core modules
- Page title - *http://drupal.org/project/page_title*
- Allow anonymous comments per node type -

http://drupal.org/project/comment_allow_anonymous

Contact module

Included with core.

In core, this module creates a per-site and per-user contact form. It has some complicated parts in it, but the contact form itself is pretty straight forward. Demonstrates hook_user_presave, form_alter, hook_mail, hook_permission


Edit contact category

Category *

Website feedback

Example: 'website feedback' or 'product information'.

Recipients *

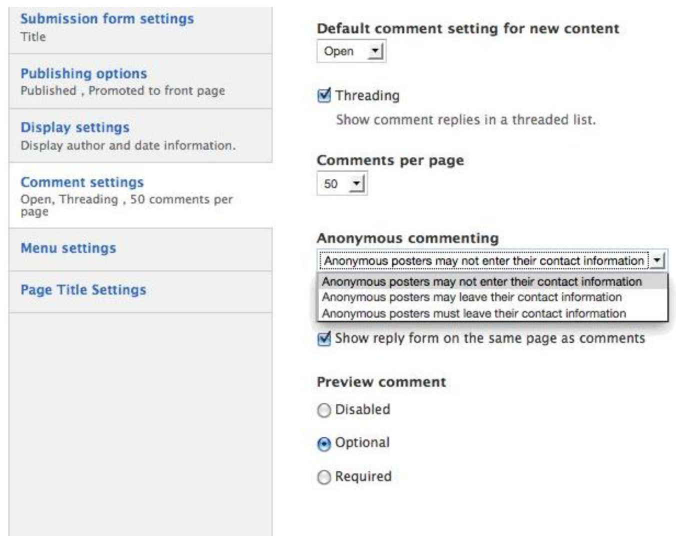
Example: 'webmaster@example.com' or 'sales@example.com,support@example.com' . To specify multiple recipients, separate each e-mail address with a comma.

Auto-reply

Optional auto-reply. Leave empty if you do not want to send the user an auto-reply message.

Allow anonymous comments per node type

http://drupal.org/project/comment_allow_anonymous Allows you to control anonymous commenting per node type via the node type edit screen once anonymous user is given the 'post comments' permission. Demonstrates form_alter, menu_alter, variable_get, permissions Compare the default comment settings for a particular content type. You cannot disable anonymous comments per-type:



Submission form settings
Title

Publishing options
Published , Promoted to front page

Display settings
Display author and date information.

Comment settings
Open, Threading , 50 comments per page

Menu settings

Page Title Settings

Default comment setting for new content
Open

✓ Threading
Show comment replies in a threaded list.

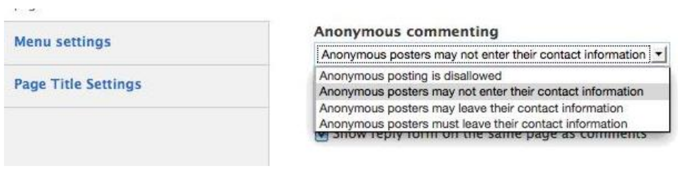
Comments per page
50

Anonymous commenting
Anonymous posters may not enter their contact information
- Anonymous posters may not enter their contact information
- Anonymous posters may leave their contact information
- Anonymous posters must leave their contact information

✓ Show reply form on the same page as comments

Preview comment
- Disabled
- [✓ Optional]
- Required

With this module you have an option to disallow anonymous comments on a particular content type.



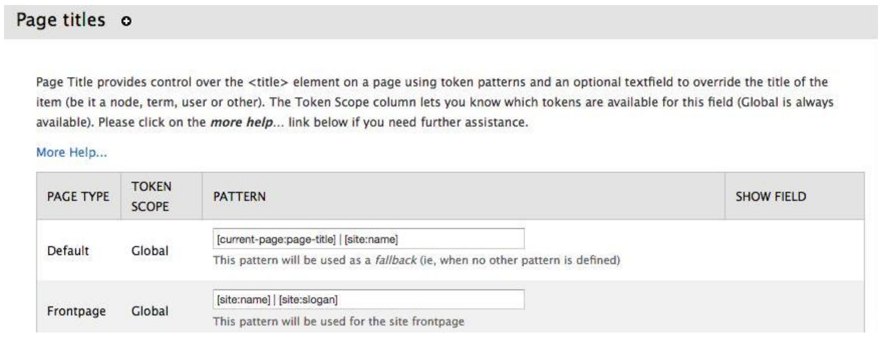
Menu settings

Page Title Settings

Anonymous commenting
Anonymous posters may not enter their contact information
- Anonymous posting is disallowed
- Anonymous posters may not enter their contact information
- Anonymous posters may leave their contact information
- Anonymous posters must leave their contact information

Page title

http://drupal.org/project/page_title This module allows more fine grained control of the title in your browser. You can specify patterns for how the title should be structured and, on content creation pages, specify the page title separately to the content's title. Demonstrates form_alter, tokens, views, settings pages



Page titles

Page Title provides control over the <title> element on a page using token patterns and an optional textfield to override the title of the item (be it a node, term, user or other). The Token Scope column lets you know which tokens are available for this field (Global is always available). Please click on the *more help*... link below if you need further assistance.

More Help...

PAGE TYPE	TOKEN SCOPE	PATTERN	SHOW FIELD
Default	Global	[current-page:page-title] | [site:name] This pattern will be used as a *fallback* (ie, when no other pattern is defined)	
Frontpage	Global	[site:name] | [site:slogan] This pattern will be used for the site frontpage	

CHAPTER

EIGHT

NODE OPERATIONS

In order for our module to be able to modify a node before it is rendered, (For example, display the email submission form on a node), we need to implement a Node Hook. Node Hooks allow us to operate on and modify a node at almost any stage of its life, starting with its creation all the way to its deletion.

For example – if we want to alter a node when it is being assembled for rendering, we would implement hook_node_view(). If we want to alter a node after it has been created, then we would implement hook_node_insert() - ‘http://api.drupal.org/api/function/hook_node_insert/7‘.

Node Hooks then, allow us to ‘hook in’ when Drupal is doing various activities with a node, enabling modules such as MailFish, to modify the node before processing continues.

8.1 Exercise: Adding node hooks to mailfish.module

Add the following code to mailfish.module and save the file:

```
<?php
/**
 * Implements hook_node_load().
 */
function mailfish_node_load($nodes, $types) {
  foreach ($nodes as $nid => $node) {
    // Add mailfish data to the node object when it is loaded.
    $node->mailfish_enabled = mailfish_get_node_enabled($node->nid);
  }
}

/**
 * Implements hook_node_insert().
 */
function mailfish_node_insert($node) {
  if ($node->mailfish_enabled) {
    // If MailFish is enabled, store the record.
    mailfish_set_node_enabled($node->nid);
  }
}

/**
 * Implements hook_node_update().
 */
function mailfish_node_update($node) {
  // Delete the old record, if one exists.
  mailfish_delete_node_enabled($node->nid);
  if ($node->mailfish_enabled) {
    // If MailFish is enabled, store the record.
    mailfish_set_node_enabled($node->nid);
  }
}

/**
 * Implements hook_node_delete().
 */
function mailfish_node_delete($node) {
  // Delete the mailfish_enabled record when the node is deleted.
  mailfish_delete_node_enabled($node->nid);
}

/**
 * Implements hook_node_view().
 */
function mailfish_node_view($node, $view_mode, $langcode) {
  // If appropriate, add the mailfish email form to the node's display.
  if (!empty($node->mailfish_enabled) && user_access('create mailfish
subscription')) {
    $node->content['mailfish'] = array(
      '#markup' => drupal_render(drupal_get_form('mailfish_email_form',
$node->nid)),
      '#weight' => 100,
    );
  }
}
```

Here we use a specific node hook for each node operation that we want to hook into. We then define what our module will do for each operation. This breaks down as follows:

- **“hook_node_load()”**
  - Add MailFish data to the node object when it is loaded.
  - *http://api.drupal.org/api/function/hook_node_load/7*“
- **“hook_node_insert()”**
  - When a node is being created for the first time – add MailFish record if mailfish is enabled for that node.
  - *http://api.drupal.org/api/function/hook_node_insert/7*
- **“hook_node_update()”**
  - When a node is being updated (i.e. edited after its initial creation) – Delete the old old MailFish record if one exists, store this record if MailFish is enabled.
  - *http://api.drupal.org/api/function/hook_node_update/7*
- **“hook_node_delete()”**
  - When deleting a node, delete the mailfish_enabled record
  - *http://api.drupal.org/api/function/hook_node_delete/7*
- **“hook_node_view()”**
  - When a user is viewing a node, check to see if the node is Mailfish enabled , and check to see if the user has permissions to create MailFish subscriptions(i.e submit their email address via the email submission form). If so then display the MailFish email submission form along with the node.

  - *http://api.drupal.org/api/function/hook_node_view/7*

CHAPTER

NINE

DATABASE SYSTEM

In the above code, you'll notice that the database queries in the various node hooks use - database wrapper functions. For example, hook_node_delete() uses the mailfish_delete_node_enabled()database wrapper function. These functions are created so that we don't have to continually repeat database queries throughout our code. Instead, we define the function once, and then call it whenever we need it.

9.1 Exercise: Defining Our MailFish Database Queries

We are now going to define the database wrapper functions used in our node hook implementations and elsewhere in our module. Open mailfish.module if it's not already open, and add the following code below the node hook implementations we added previously:

```
<?php
/**
 * Determine if a node is set to display an email address form.
 *
 * @param int $nid
 *   The node id of the node in question.
 *
 * @return boolean
 */
function mailfish_get_node_enabled($nid) {
  if (is_numeric($nid)) {
    $result = db_query("SELECT nid FROM {mailfish_enabled} WHERE nid = :nid",
array('nid' => $nid))->fetchField();
    if ($result) {
      return TRUE;
    }
  }
  return FALSE;
}

/**
 * Add an entry for a node's mailfish setting.
 *
 * @param int $nid
 *   The node id of the node in question.
 */
function mailfish_set_node_enabled($nid) {
  if (is_numeric($nid)) {
    if (!mailfish_get_node_enabled($nid)) {
      $jump = db_insert('mailfish_enabled')
```

```
        ->fields(array('nid' => $nid))
        ->execute();
    }
  }
}

/**
 * Remove an entry for a node's mailfish setting.
 *
 * @param int $nid
 *   The node id of the node in question.
 */
function mailfish_delete_node_enabled($nid) {
  if (is_numeric($nid)) {
    $vump = db_delete('mailfish_enabled')
      ->condition('nid', $nid)
      ->execute();
  }
}


/**
 * Store a mailfish email signup.
 */
function mailfish_signup($email, $nid, $account = NULL) {
  if (is_null($account)) {
    global $user;
    $account = $user;
  }

  $value = array(
    'nid' => $nid,
    'uid' => $account->uid,
    'mail' => $email,
    'created' => time(),
  );

  drupal_alter('mailfish_signup', $value);

  module_invoke_all('mailfish_signup', $value);

  $_SESSION['mailfish'] = $nid;
  drupal_write_record('mailfish', $value);
  watchdog('mailfish', 'User @uid signed up for node @nid with @email',
array('@uid' => $account->uid, '@nid' => $nid, '@email' => $email));

}
```

Our database queries are now stored away in functions that can be called whenever they are needed. This:

- Saves us time, as we don't need to write the same query over and over again.
- Keeps our code free from clutter.
- Greatly reduces the possibility of syntax errors.

9.2 Database Abstraction Layer

Database Queries

When writing basic SELECT queries in Drupal 7, we use the db_query() function - *http://api.drupal.org/api/function/db_query/7*.

The following is an example of a db_query() instance taken from our MailFish module. This example helps us to understand some Drupal specific conventions that are used when writing SQL queries:

```
$previous_signup = db_query("SELECT mail FROM {mailfish} WHERE nid = :nid AND mail = :mail", array('r
->fetchField();
```

Tables

Table names within db_query() functions are enclosed within curly brackets. In the above example, mailfish is our table {mailfish}. This allows optional table name prefixing to work.

Placeholders

When using the db_query() function, queries are written using placeholders and are defined using the convention - :identifier - where identifier is a descriptive name for the placeholder.

The placeholders in the above example are – :nid and :mail. We then assign values to our placeholders using associative arrays - where the keys are the placeholder identifiers. In the above example, we assign values to the placeholders :nid and :mail as follows – *array('nid' => $nid), array('mail' => $email).*

Results

To obtain results from our database SELECT query we simply append ->fetchField() to the end of the db_query() function like so:

```
$previous_signup = db_query("SELECT mail FROM {mailfish} WHERE nid = :nid AND mail = :mail",
array('nid' => $nid, 'mail' => $email))->fetchField();
```

Inserts, Updates, Deletes

Other Drupal database functions include:

database functions	**Notes**
db_insert()	Used to run INSERT queries to the active database. http://api.drupal.org/api/function/db_insert/7
db_update()	Used to run UPDATE queries to the active database. http://api.drupal.org/api/function/db_update/7
db_delete()	Used to run DELETE queries to the active database. http://api.drupal.org/api/function/db_delete/7

CHAPTER

TEN

BLOCK SYSTEM AND THEME SYSTEM

10.1 Block system Exercises

10.1.1 Exercise: Creating a MailFish Subscription Block

Step 1: Adding hook_block_info() to mailfish.module

We want to add a block that contains our MailFish email subscription form so that site administrators can add this to any block region within their Drupal site. Once the block has been added, it will show up on Drupal's block configuration page - *http://example.com/admin/structure/block*.

To do this, we need to implement hook_block_info() - *http://api.drupal.org/api/function/hook_block_info/7* in our mailfish.module file. Hook_block_info() declares a block or set of blocks for our module.

Add the following code to mailfish.module:

```php
<?php
/**
 * Implements hook_block_info().
 */
function mailfish_block_info() {
  $blocks = array();
  $blocks['mailfish_subscribe'] = array(
    'info' => t('MailFish Signup Form'),
    'cache' => DRUPAL_NO_CACHE,
  );
  return $blocks;
}
```

Step 2: Adding hook_block_view()

In order to render the block and display it to users, we add an instance of hook_block_view() - *http://api.drupal.org/api/function/hook_block_view/7*.

Add the following code to mailfish.module:

```php
<?php
/**
 * Implements hook_block_view().
 */
function mailfish_block_view($delta = 'mailfish_subscribe') {
  $block = array();
  switch ($delta) {
    case 'mailfish_subscribe':
      $block['subject'] = t('Sign up for %site', array('%site' =>
variable_get('site_name', 'Drupal')));
      $block['content'] =
drupal_render(drupal_get_form('mailfish_email_block_form'));
      break;
  }
  return $block;
}

/**
 * Provide the form for the block content.
 *
 * This form is the same as the node form,
 * but with a different form_id to prevent
 * conflict.
 */
function mailfish_email_block_form($form, $form_state) {
  $form = mailfish_email_form($form, $form_state);
  $form['#validate'][] = 'mailfish_email_form_validate';
  $form['#submit'][] = 'mailfish_email_form_submit';
  return $form;
}
```

This code builds our subscription block and pulls in the MailFish email submission form that we defined earlier in the Building The Email Submission Form section of this handout. Provided that the block is enabled and is assigned to a region (i.e. sidebar left, header etc.), it will be rendering and displayed to site users.

Test it

1. Clear your cache.
2. Go to *http://example.com/admin/structure/block*
3. Move your new block into a region and save the page.

10.2 Theming the MailFish Subscription Block

If you recall earlier, we added a block containing the MailFish email subscription form to the block admin page using an implementation of hook_block_info(). Hook_block_info() took care of the functional aspects of the block. However, we now want to address the aesthetic aspects and to do so, need to register a theme for our block.

Theme Registry

The theme registry is where Drupal keeps track of all theming functions and templates. Every themeable item in Drupal is themed by either a template or a function. When the theme registry is built, Drupal discovers information about each themeable item. During this process, Drupal looks for hook_theme() (used to register all themeable output in Drupal) implementations in modules to discover theme functions and template files. hook_theme() is therefore implemented in a module when we want to return an array of themeable items.

Conventions

- Modules must implement hook_theme() to declare their themeable functions and templates.
- These functions must be titled theme_function_name.
- theme('function_name', $param) is then used to call the function (for an example implementation, see the Reporting Results code in session 9).

10.2.1 Exercise: Theming the MailFish Subscription Block

What is a Theme Function?

A theme function is a PHP function within Drupal that is prefixed with **theme_**, and produces HTML for display. A good example of a theme function is theme_username().

http://api.drupal.org/api/function/theme_username/7

This function outputs a given user's username as a link on the screen.

Implementing hook_theme() in mailfish.module

We are now going to register a new theme for our MailFish subscription block called mailfish_block using hook_theme().

Open up the mailfish.module file and add the following code:

```php
<?php
/**
 * Implementation of hook_theme().
 */
function mailfish_theme() {
  $theme = array();
  $theme['mailfish_block'] = array(
    'variables' => array(
      'rendered_form' => NULL,
    ),
    'template' => 'mailfish-block',
  );
  return $theme;
}
```

Here, we have registered our new theme function – mailfish_block, and have tied it to a template mailfish-block. The full name of this template is actually mailfish-block.tpl.php, however the .tpl.php section is automatically added by Drupal and so can be omitted here. The mailfish-block.tpl.php template doesn't actually exist yet, so we are going to create it in our next and final step.

But first, we need to tell the block system to use that template to theme our particular block. In mailfish_block_view above, we used this line to output the form:

```
$block['content'] =
drupal_render(drupal_get_form('mailfish_email_block_form'));
```

Now, we're going to pass that to the "theme()" function which will send it through Drupal's theming system and eventually to your template.

```
$block['content'] = theme('mailfish_block', array('rendered_form' =>
drupal_render(drupal_get_form('mailfish_email_block_form'))));
```

The finished function should now look like this:

```
/**
 * Implements hook_block_view().
 */
function mailfish_block_view($delta = 'mailfish_subscribe') {
  $block = array();
  switch ($delta) {
    case 'mailfish_subscribe':
      $block['subject'] = t('Sign up for %site', array('%site' =>
variable_get('site_name', 'Drupal')));
```

```
      $block['content'] = theme('mailfish_block', array('rendered_form' =>
drupal_render(drupal_get_form('mailfish_email_block_form'))));
      break;
  }
  return $block;
}
```

"theme()" takes two parameters. The first is the name of the "theming function" to call. In our case, mailfish_block. But it could be "table", "node", "block", "region", "page", etc. etc. The second parameter is an array of variables which will get passed to the template file. In our case, we're passing only one variable which is "rendered_form" and it contains the HTML for the signup form.

Creating mailfish-block.tpl.php

To create our new template, we simply create a new file in our mailfish module directory and give it the same name as the template we registered using hook_theme() in the previous step. Make sure you include the .tpl.php extension.

Create the following file in the mailfish module directory:

sites/all/modules/mailfish/mailfish-block.tpl.php

open mailfish-block.tpl.php and add the following code:

```
<?php
/**
 * @file
 *   Themes the mailfish block.
 */
?>
<div id='mailfish-rocks'>
  Check it out:
  <?php print $rendered_form; ?>
</div>
```

This template will now be used to theme the MailFish subscription block, which is extremely useful should you need to add extra HTML elements such as <div> tags. Something to remember here is that every time you modify or add theme hooks, you need to clear the theme registry before your changes will be recognized. Note that you might need to clear the cache after altering a .tpl.php file.

Clearing the theme registry

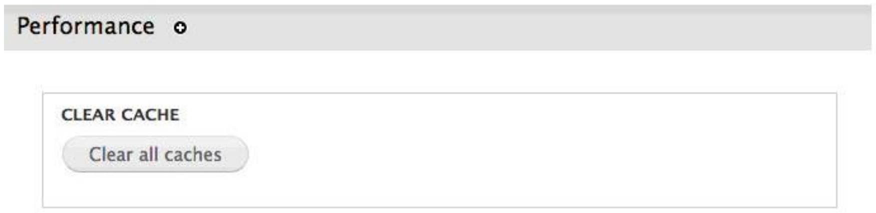
Option 1 - Drush command

You can use cache clear (cc) to Clear all caches. For example:

> *drush cc 'theme registry'*

Option 2 - Clear cache in configuration

Go to Configuration > Development > Performance. Click "Clear all caches".



Option 3 - Development module

Use the Empty cache link on the Devel block created by the Devel module. The development block must be enabled and assigned to a region in order to use this method.

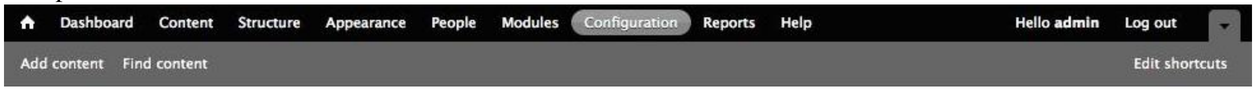
Option 4 - Install and use Administration menu

Install Administration Menu *http://drupal.org/project/admin_menu*

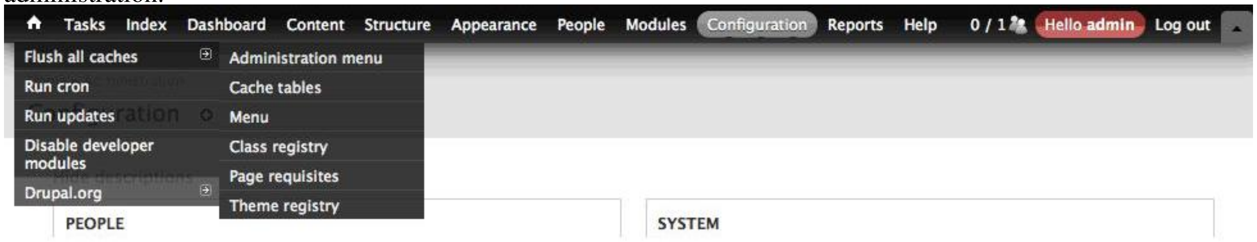
Enable the following modules from that package:

- Administration Development tools
- Administration menu
- Administration menu Toolbar style

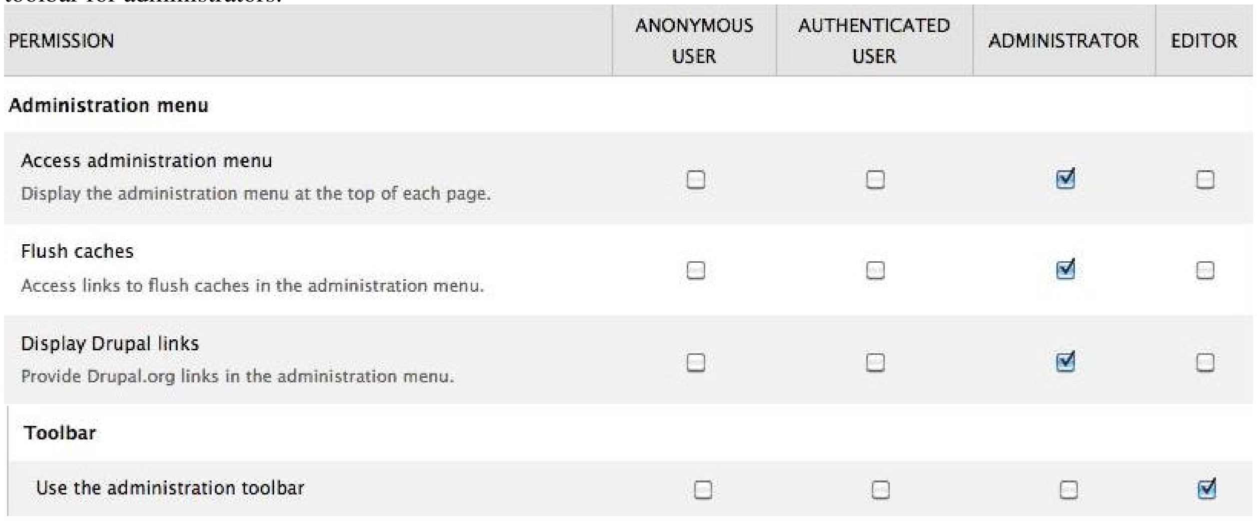
Compare the default core toolbar:



... to the Administration menu toolbar below. The Administration menu offers drop-down menus to main areas of your administration.



Manage permissions for the Administration toolbar and the default core toolbar to avoid a conflict for users. For example you could allow editors to see the core toolbar to access shortcuts, and only enable the administration menu toolbar for administrators.



PERMISSION	ANONYMOUS USER	AUTHENTICATED USER	ADMINISTRATOR	EDITOR
Administration menu				
Access administration menu Display the administration menu at the top of each page.	☐	☐	✓	☐
Flush caches Access links to flush caches in the administration menu.	☐	☐	✓	☐
Display Drupal links Provide Drupal.org links in the administration menu.	☐	☐	✓	☐
Toolbar				
Use the administration toolbar	☐	☐	☐	✓

CHAPTER

ELEVEN

REPORTING RESULTS

We also want to provide site administrators with a list of all the node email subscriptions on their site. At the beginning of the session, we built the MailFish subscriptions menu item via our hook_menu() implementation. We now want to pull all the relevant subscription information from the database and display it in the reports section of our site - http://example.com/admin/reports/mailfish. To do this, we need to add a menu callback to our mailfish.admin.inc file.

11.1 Exercise: Reporting results

```
<?php
/**
 * Menu callback.
 *
 * Displays mailfish signups.
 */
function mailfish_signups() {
  $output = '';
  $rows = array();
  $header = array(
   'User',
   'Node',
   'Email',
   'Created',
  );

  // Note - for a static query, as actually requred here, (where the SQL is
  // never going to change) the method below is recommended.
  /*
  $sql = "SELECT m.*, n.title FROM {mailfish} m
          LEFT JOIN {node} n ON n.nid = m.nid
          WHERE m.nid = :nid
          ORDER BY m.created ASC";
  $result = db_query($sql, array('nid' => $nid));
  */

  // Note - here we use the object oriented DB api for demonstrative purposes
only.

  // Dynamically load the schema for this table (which could be modified by
  // other modules using hook_schema alter).
  $fields = drupal_get_schema('mailfish');
  // Intantiate a query object by using the db_select wrapper (db_update,
  // db_insert and db_delete are also available).
  $query = db_select('mailfish', 'm');
  // Add a join on the node table.
  $table_alias = $query->innerJoin('node', 'n', 'n.nid = m.nid', array());
  // Add our desired fields to the query, loading the fields for our table
dynamically.
  $results = $query->fields('m', array_keys($fields['fields']))
    ->fields($table_alias, array('title'))
    // Add a sort to our query
    ->orderBy('m.created', $direction = 'ASC')
    // Execute our query, triggering it to run against the database.
    ->execute()
    // Return an array of stdClass objects representing our results.
    ->fetchAll();
  foreach ($results as $value) {
    $account = $value->uid ? user_load($value->uid) : '';
    $rows[] = array(
      $value->uid ? theme('username', array('account' => $account)) : '',
      $value->nid ? l($value->title, 'node/' . $value->nid) : '',
      $value->mail,
      date('F j, Y g:i A', $value->created),
    );
  }
  $output .= theme('table', array('header' => $header, 'rows' => $rows));
  return $output;
}
```

What does this code do?

Our MailFish subscription info will be displayed within a table on the MailFish reports page. The above code therefore defines the table column names - User, Node, Email, and Created, and places them in an array called $header. It then pulls all the corresponding data from the MailFish database table and stores it in an array called $rows. We then pass the $header and $rows arrays to the Drupal theme() function (*http://api.drupal.org/api/function/theme/7*), so that they will be rendered as an HTML table when viewed on the MailFish reports page *http://example.com/admin/reports/mailfish*.

CHAPTER

TWELVE

TESTING YOUR MODULE: SIMPLETEST

12.1 Simpletest

The Drupal core module Simpletest provides a framework for automated testing of other modules. Each core module and some contributed modules include tests that can be run at /admin/config/development/testing once you enable Simpletest module.

Most tests generally amount to logging in users with certain permissions, submitting forms with certain values, and checking pages for the existence or non-existence of some expected text. You can write your own tests for custom modules without too much trouble by copying and adjusting bits of the code you find in core tests.

12.1.1 Exercise: Writing a test

In this exercise we'll add a test for the mailfish per-content type setting.

Step 1: Add the test class

- Add a new file called mailfish.test beginning with a @file comment.

```
<?php
/**
 * @file
 *   Provide the automated tests for the MailFish module.
 */
```

- Because test files contain a class, they must be included in the module's .info file. Add a line to mail-fish.info:

  ```
  files[] = mailfish.test
  ```

- Define the class:

  ```
  class MailfishTestCase extends DrupalWebTestCase {
  ```

Step 2: Tell Drupal about your test

The getInfo() method is called by Drupal's testing engine to find out metadata about your test. This includes which suite it is a part of and its title and description.

```
<?php
class MailfishTestCase extends DrupalWebTestCase {

  public static function getInfo() {
    return array(
      'name' => 'Mailfish functionality',
      'description' => 'Test the mailfish settings form functionality',
      'group' => 'Mailfish',
    );
  }
```

Step 3: Define a setUp method to get call before your test runs

The setUp method is used to enable the modules needed for the test and define a user with certain permissions needed for the test. It gets called before every test in your class.

```
<?php
  /**
   * Enable modules and create users with specific permissions.
   */
  function setUp() {
    parent::setUp('mailfish');
    // Create users.
    $this->admin_user = $this->drupalCreateUser(array(
      'manage mailfish settings',
      'create page content',
    ));
  }
```

Step 4: Define a test

Now create a test, using existing tests from other modules as your guide.

In our test we will login and go to /node/add/page to check that no setting for enabling MailFish are present.

Then we will go to the MailFish settings page and enable MailFish for the page content type. After we will check /node/add/page again and confirm that the setting text is now present.

```
<?php
  /**
   * Test mailfish settings functionality.
   */
  function testMailfishSettings() {
    // Log in an admin user.
    $this->drupalLogin($this->admin_user);

    // Check that no mailfish settings appear when adding a new page.
    $this->drupalGet('node/add/page');
    $this->assertNoText(t('Collect e-mail addresses for this node.'), 'The
mailfish settings were not found.');

    // Change the settings to enable mailfish on pages.
    $edit = array('mailfish_types[page]' => TRUE);

    // $edit = array();
    $this->drupalPost('admin/config/content/mailfish', $edit, t('Save
configuration'));

    // Check that the mailfish settings appear when adding a new page.
    $this->drupalGet('node/add/page');
    $this->assertText(t('Collect e-mail addresses for this node.'), 'The
mailfish settings were not found.');
  }
}
```

Running tests

1. Disable MailFish for the “Basic page” content type.
2. Visit your site at admin/configuration/development/performance and clear all caches.
3. Enable the Testing module on admin/modules.
4. Visit your site at admin/config/development/testing.
5. Select the option for “Mailfish”.
6. Click *Run tests*.

CHAPTER

THIRTEEN

APPENDIX I: DRUPAL CODING STANDARDS AND CONVENTIONS

Coding standards

The Drupal community has agreed that the Drupal codebase adhere to a set of coding standards. This standardization makes the code more readable, and thus easier for developers to understand and edit each other.s code. This is covered in an Appendix at the end of this manual. It is crucial you learn these as you become involved in Drupal development.

A full list of Drupal.s coding standards can be found on Drupal.org - *http://drupal.org/coding-standards*. For now we.re going to just highlight a few key points.

When writing modules, it is extremely important that you follow these coding standards, especially if you intend on making your module available to the wider Drupal community. For purposes of legibility in print, the code has been indented. The reader should assume these lines are not indented.

The following is a list of must know coding standards for reference during the training:

Line Indentation

Drupal code uses 2 spaces for indentation, with no tabs.

PHP Tags

Drupal php files (i.e. dot-module files), use opening PHP tags (<?php), but do not use closing PHP tags (?>). The exception to this rule is in template files, where the closing tag is used to exit out of PHP and go back into HTML. The short opening PHP tag form .(<?) is never used in Drupal.

Control Structures

Control structures control the flow of execution in a program and include: if, else, elseif, switch statements, for, foreach, while, and do-while. Control structures should have a single space between the con-trol keyword and the opening parenthesis. Opening braces should be on the same line as the control keyword, whereas closing braces should have their own line.

```
If ($a == $b) {
  do_this()
}
```

Function Calls

Function calls should contain a space around the operator (=, >, etc.), and no space between the function name and opening parenthesis. The first parameter within the function.s parenthesis should have a space before it. The last parameter has no space after it. A comma and a space should separate additional parameters after the first parameter.

```
$variable = foo($a, $b);
```

Function Declarations

Function declarations should not contain a space between the function name and the opening parenthesis.

```
function mymodule_function($a, $b) {
  $do_something = $a + $b;
  return $do_something;
}
```

Arrays

Arrays should be formatted with spaces separating each element and assignment operator. If the array spans more than 80 characters, each element in the array should be given its own line.

```
$car[.colors.] = array(
  .red. => TRUE,
  .orange. => TRUE,
  .yellow. => FALSE,
  .purple. => FALSE,
);
```

PHP Comments

PHP comments in Drupal use the following syntax:

```
/**
 * Comments go here.
 * /
```

The leading spaces that appear before the asterisks (*) on lines after the first one are required. There is no space between a function and the comments that document it. the function should immediately follow the comments.

```
/**
  * This is a function.
  */
function mymodule_function() {
  //do something }
}
```

Strings

t() function should wrap around each string. Translates a string to the current language or to a given language.

The t() function serves two purposes. First, at run-time it translates user-visible text into the appropriate language. Second, various mechanisms that figure out what text needs to be translated work off t() – the text inside t() calls is added to the database of strings to be translated. So, to enable a fully-translatable site, it is important that all human-readable text that will be displayed on the site or sent to a user is passed through the t() function, or a related function. *http://api.drupal.org/api/function/t/7*

You can use variable substitution in your string, to put variable text such as user names or link URLs into translated text. Variable substitution looks like this:

```
<?php
$text = t("@name’s blog", array(’@name’ => format_username($account)));
?>
```

13.1 Using The Coder Module

The Coder module enables you to review the code of other modules, including your own. It is an excel-lent tool for cleaning up your code, and for learning the coding standards outlined previously.

1. Download the Coder module from the project page - http://drupal.org/project/coder, and place it in the sites/all/modules directory. Enable the module via the Module Administration Page. Go to Administer > Site building > Modules.
2. Enabling the module adds a Code Review link beside every module on the admin page. To have the Coder module review your module, click on the Code Review link beside it. This will open the code review page that contains the suggested code changes for your module. The coder module tells you the line number of the code that should be changed and the file that contains it. After making the suggested changes in the relevant module files, refresh the code review page and the suggestions should have disappeared.
3. The Coder module also contains a script that actually fixes your code formatting errors . coder_format.php. Take the following steps to run the script on your module:
   (a) Use the command line and run the following command - cd sites/all/modules.
   (b) Run the following command from the modules directory: php coder/scripts/coder_format/coder_format.php mymodule/mymodule.module
4. If the script runs successfully, coder will issue a confirmation message via the command line: mymodule/mymodule.module processed.
5. The coder_format.php script modifies the mymodule.module file, and saves the original module file as mymodule.module.coder.orig so that you have a copy of it incase something goes wrong. This file is saved in your module directory folder (i.e. sites/all/modules/mymodule) along with the mymodule.module file.
6. Using the diff command in sites/all/modules will reveal any changes the script has made:

```
diff mymodule/mymodule.module mymodule/mymodule.module.coder.orig
```

CHAPTER

FOURTEEN

APPENDIX II: MODULE DEVELOPMENT CONVENTIONS

Out-of-the-box, contributed modules will get you a long way in terms of adding functionality to your Drupal website. However, there are times when you will want to add your own specific functionality and will need to create a custom module in order to do so. If you are new to Drupal, the thought of this can be intimidating. However, with a little bit of guidance, you will soon realize that the process involved in creating such a module is actually quite straightforward.

Placement

Custom modules, like contributed modules, belong in /sites/all/modules to keep them separate from core modules. Adding modules and modifications outside of the /sites/ directory is considered .hack-ing core.. One of the cardinal rules of Drupal is .Do not hack core.. See *http://drupal.org/best-practices/do-not-hack-core*

It is a good idea to put your custom modules in a /sites/all/modules/custom or /sites/all/modules/SITE_NAME subdirectory to easily distinguish them from contributed modules.

When you set up your Drupal site, create these two directories: * /sites/all/modules/custom * /sites/all/modules/contrib

Naming

The first step in creating a new module is choosing a descriptive name. Modules have both machine and human-readable names. For instance, in one of our examples, our human-readable name will be Mailfish and the machine name will be mailfish.

Modules in Drupal follow naming conventions.

Naming convention	Bad	Good
Multiple words in modules names should be separated with underscores.	great-module	great_module
Modules names cannot begin with a number.	2nd_module	module2
All Drupal filenames (as well as function definitions and variable names) must be in lowercase.	Awesome	awesome
Custom modules should not conflict with core modules.	aggregator	custom_aggregator

Essential parts of a module

.info files

.info files contain the metadata about modules, much of which appears on the module administration page. Further information about Drupal 7 info files: *http://drupal.org/node/542202*

Table 14.1: Properties

Property	Required?	Description
name	Yes	Human-readable form of the module's name.
description	Yes	One-sentence summary of its purpose.
dependencies	No	An array of any other modules this module requires. Used when enabling modules to ensure all dependencies are met. If your module depends on a non-optional core module, it does not need to be listed as a dependency.
package	No	Places the module into a section of the Module Administration page. It will default to the Other section.
php	No	If your module requires a specific version of PHP greater than Drupal's core requirement, use the php field to require it.

14.1 Server Requirements

Custom server requirements can be handled with hook_requirements: *http://api.drupal.org/api/function/hook_requirements/7*

.module files

Every Drupal module also contains a .module file. The .module file is a PHP file that typically contains all the implementations of Drupal hooks (other than those related to installing and uninstalling) a module uses. The code in all .module files is loaded on each page load. Therefore, to reduce the load on PHP memory, it.s best practice to separate a module.s code into additional files that are loaded as needed. Many modules have .admin.inc and .pages.inc files which hold code related to their administrative and pub-lic-facing pages.

.install files

If a module needs to take actions when it is installed or uninstalled, it will have a .install file. Many modules add a new table to the database during installation and delete it, along with their other settings, when the module is uninstalled (there is a difference between enabling/disabling a module and installing/uninstalling it). Database updates required by new versions of a module also go into the .install file.

14.2 Drupal site speed check

How fast and responsive your site appears to users can be improved by reducing the payload of the pages they load. While there are many non-Drupal aspects to improving the performance of your site, we can start by optimizing and checking the configuration of Drupal. Ensure you have followed steps and best practices for a speedy Drupal site Use browser-based site optimization tools to conduct checks of front end performance Special Drupal-specific enhancements to cache and compress content. Checking queries for anything that can be improved.

In addition to disabling unused modules, larger custom modules should use includes in their hook_menu implementation like *.admin.inc and *.pages.inc so that they aren't loaded on every page.

14.2.1 Acquia Insight

Acquia Insight is a tool to assess your site. This includes a subscription to services which monitor your site's performance. There are also Drupal-specific checks. There is a free 30 day subscription trial.

http://acquia.com/insight

Read Acquia Cloud Optimization Checklist (subscription required). https://library.acquia.com/articles/acquia-cloud-optimization-checklist

14.2.2 Caching and compressing

Caching and compressing are the main methods of optimization. Yet these can cause errors and un-expected outcomes when used while developing your site. Therefore, do performance optimization as a last stage of site development. These performance suggestions do not refer to sites in development. When in development, turn off caching, and clear cached data in Administer > Performance (?q=/admin/settings/performance). Reduce HTTP requests: Enable

14.2.3 Reduce HTTP requests: Enable JavaScript and CSS optimization

By default, Drupal will have several JavaScript and CSS files loaded into the header of a template, a typical site can contain 50 or more individual CSS and Javascript files from the system, contributed modules and your theme. To reduce HTTP requests, you can combine many of the CSS files into one file; and the JavaScript files into another. This also reduces whitespace in the file.

To do this: Go to Administer > Performance (?q=/admin/settings/performance)

- Under Optimize CSS files: select Enabled
- Under Optimize JavaScript files: select Enabled
- Save Configuration

See article about changes in Drupal 7 to this system:

http://www.metaltoad.com/blog/drupal-7-taking-control-css-and-js-aggregation

14.2.4 Minify CSS

You can compress your CSS even further. Go to Configuration > Development > Performance Under Bandwidth optimization:

- Select .Aggregate and compress CSS files..
- Select .Aggregate JavaScript files..
- Save configuration.

14.2.5 Cache Views

In addition there are settings for caching of views that are set in each view. Adjusting those settings, especially for the views on the pages that are visited most often.

To do this: Go to Administer > Site building > Views (?q=/admin/build/views)

- Select the view you want to optimize.
- Under the Basic Settings column, select Caching: None.
- Scroll down and select Time-based. Either override for this display, or click Update default display.
- Edit settings for Caching options for Query results and Rendered output. Choose a setting re-lated to how often your site is updated.
- Save your changes.

14.2.6 Cache Pages and Blocks

Making sure that block caching is on if possible will also help with authenticated users. There is also an option to compress pages on the performance page which would be good to do. To do this: Go to Administer > Performance (?q=/admin/settings/performance)

- Under Block cache: Select Enabled
- Under Page compression: Select Enabled
- Save configuration.

Note that block caching is inactive when modules defining content access restrictions are enabled. (For example, modules such as Workflow which control access to content.) Not all sites can use block cache, and not all views can be cached.

Read two articles on "When and how caching can save your site".

- http://www.acquia.com/blog/when-and-how-caching-can-save-your-drupal-site
- http://www.acquia.com/blog/when-and-how-caching-can-save-your-site-part-2-authenticated-users

Read about Varnish, Memcache(d) https://docs.acquia.com/cloud/performance

14.3 Tools to improve performance

Pressflow http://pressflow.org/

Pressflow is a packaged and modified distribution of Drupal 6. Pressflow integrates performance, scalability, availability, and testing enhancements. Pressflow includes extensive documentation.

Varnish Software http://www.varnish-software.com/

While not Drupal-specific, Varnish software improves the response time of websites by caching as-sets and/or pages on your site. Varnish is supported open-source software. While you can use Varnish with any version of Drupal, Varnish is ideally suited to Pressflow, which has been modified specifically to take advantage of the functionality of Varnish.

By default, Varnish doesn't cache pages when cookies are set. Alas, out-of-the-box all versions of Drupal prior to D7 set a SESS cookie. To work around this limitation, I recommend installing the PressFlow extensions which disable the SESS cookie for anonymous sessions. Fortunately, Drupal 7 corrects the problem and doesn't set SESS cookies for anonymous page requests. Watch out for other sources of cookies. Quite a few Drupal modules and javascript-based marketing plugins (e.g., Google Analytics, Omniture, Woopra, etc.) also set cookies which cause cache misses.

New Relic http://newrelic.com/

New Relic is a performance management tool. While not Drupal-specific it provides a range of useful metrics while fine-tuning and monitoring your sites. All Acquia Network subscriptions include New Relic's Bronze service at no cost (a $900 annual value). Read more about New Relic and Drupal: *http://buytaert.net/playing-with-new-relic-on-acquia-hosting*

Modules

On Drupal.org you can filter the module listing page to display only modules related to performance and scalability. Visit this page via: *http://tinyurl.com/drupalperformance*

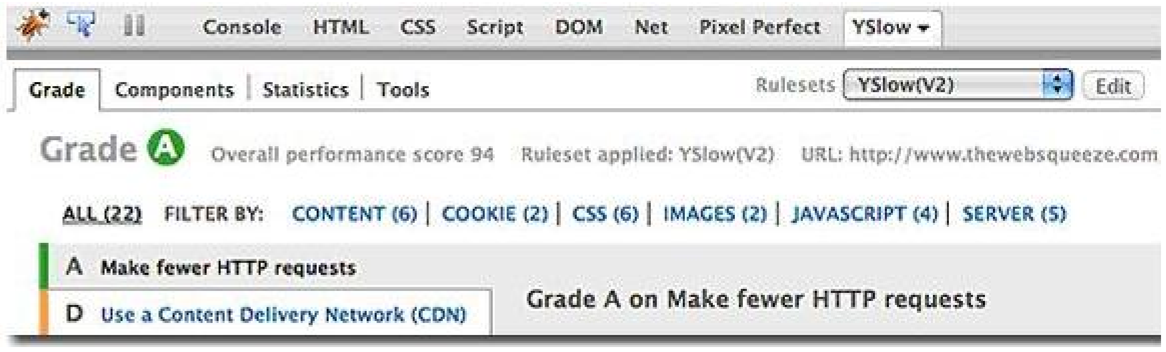
Boost: Currently the Drupal 7 version of the Boost module is under development.

Boost *http://drupal.org/project/boost* will help with the page views for anonymous users. It will make a difference if you are re-visiting pages that have been cached. It works by basically writing a static version of pages to the file system. Subsequent requests by anonymous users for the page are sent to the filesystem instead of to Drupal. It's likely a good idea if you're expecting a fair bit of anonymous traffic.

Be sure to read the accompanying README.txt and Handbook Page for installation and usage instructions. The project description warns "this is an advanced module and takes some extra effort to get it working correctly

14.4 Check Front End Performance

For slow Drupal sites, it may be useful to the front-end performance of the site.


Console
HTML
CSS
Script
DOM
Net
Pixel Perfect
YSlow
Grade
Components
Statistics
Tools
Rulesets
YSlow(V2)
Edit
Grade A
Overall performance score 94
Ruleset applied: YSlow(V2)
URL: http://www.thewebsqueeze.com
ALL (22)
FILTER BY:
CONTENT (6) | COOKIE (2) | CSS (6) | IMAGES (2) | JAVASCRIPT (4) | SERVER (5)
A Make fewer HTTP requests
D Use a Content Delivery Network (CDN)
Grade A on Make fewer HTTP requests

Browser-based site speed evaluation tools can help look at the resulting markup and output of your Drupal site to determine the .payload. of pages on your site. Are images optimized to size, or are they merely resized? How can you lower the size of the CSS, HTML and scripts to speed up the download of your pages?

Google's Page Speed and Yahoo's YSlow are two Firefox extensions which measure speed and analyze performance of your website against documented best practices.

- Google: Web Performance Best Practices http://code.google.com/speed/page-speed/docs/rules_intro.html
- Yahoo: Best Practices for Speeding Up Your Web Site http://developer.yahoo.com/performance/rules.html

Both tools have specific features not covered by the other. Both evaluate performance and give sug-gestions on how to improve speed. This can help capture a detailed overview of your site.

Tip: Reducing image size

Ensure that Imagecache is employed to control the size of images, and look for ways to improve the page size by using less images. Design using CSS and less images: *http://www.phpied.com/css-performance-ui-with-fewer-images/*

14.5 Using Devel to locate slow queries

Install and enable the Development module. http://drupal.org/project/devel

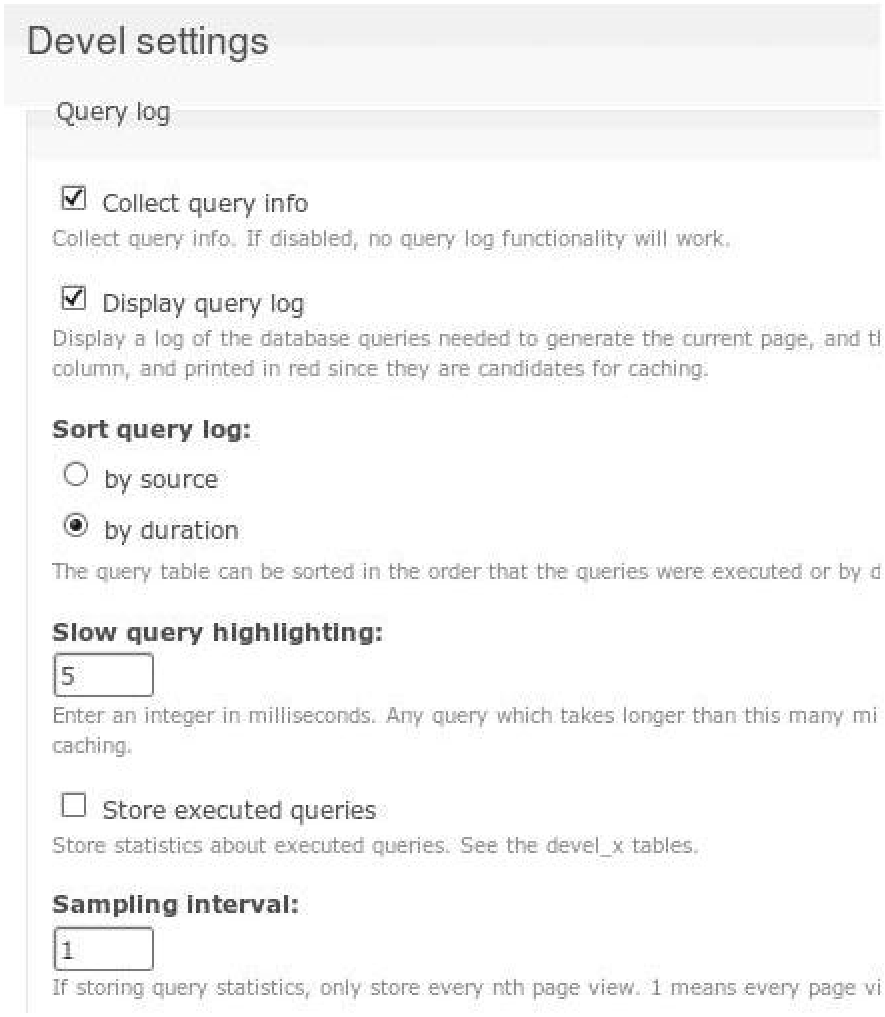
Configure Devel settings

First, select the options for Collect query info and Display query log.

- Sort query log, select by duration.
- Slow query highlighting, enter a value of 5.
- Sampling interval enter 1.

Once you've done the above, another dialog box will appear. You will need to changes some settings related to the API Site.

Select Display page timer and Display memory usage.



Devel settings

Query log

[✓] Collect query info

Collect query info. If disabled, no query log functionality will work.

[✓] Display query log

Display a log of the database queries needed to generate the current page, and tl
column, and printed in red since they are candidates for caching.

Sort query log:

( ) by source

(●) by duration

The query table can be sorted in the order that the queries were executed or by d

Slow query highlighting:

5

Enter an integer in milliseconds. Any query which takes longer than this many mi
caching.

[ ] Store executed queries

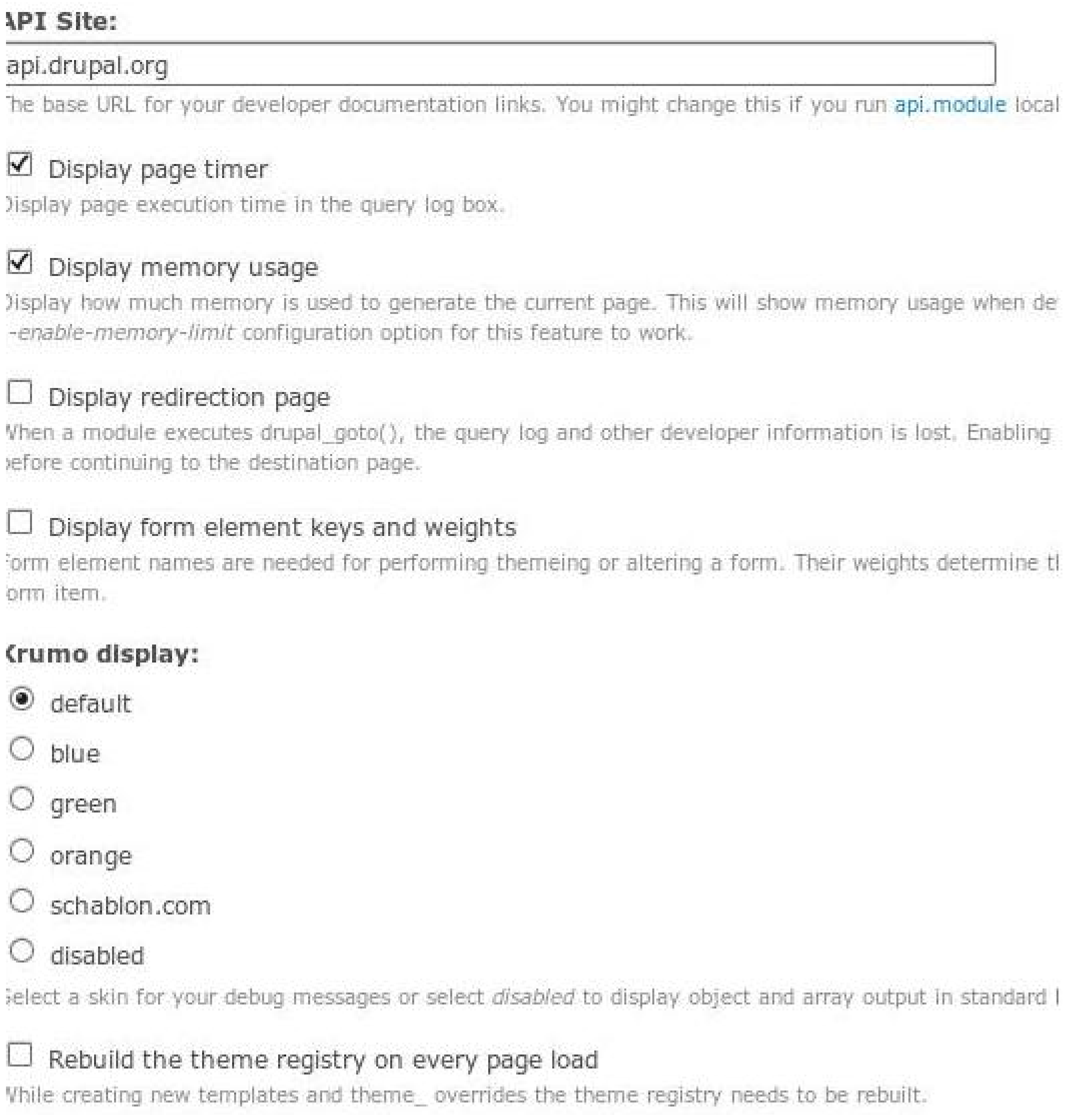
Store statistics about executed queries. See the devel_x tables.

Sampling interval:

1

If storing query statistics, only store every nth page view. 1 means every page vi

Leave the rest of the settings as they default. See the screenshot below for clarification of this.



API Site:

api.drupal.org

he base URL for your developer documentation links. You might change this if you run api.module local

[✓] Display page timer

)isplay page execution time in the query log box.

[✓] Display memory usage

)isplay how much memory is used to generate the current page. This will show memory usage when de
-*enable-memory-limit* configuration option for this feature to work.

[ ] Display redirection page

Vhen a module executes drupal_goto(), the query log and other developer information is lost. Enabling
efore continuing to the destination page.

[ ] Display form element keys and weights

orm element names are needed for performing themeing or altering a form. Their weights determine tl
orm item.

(rumo display:

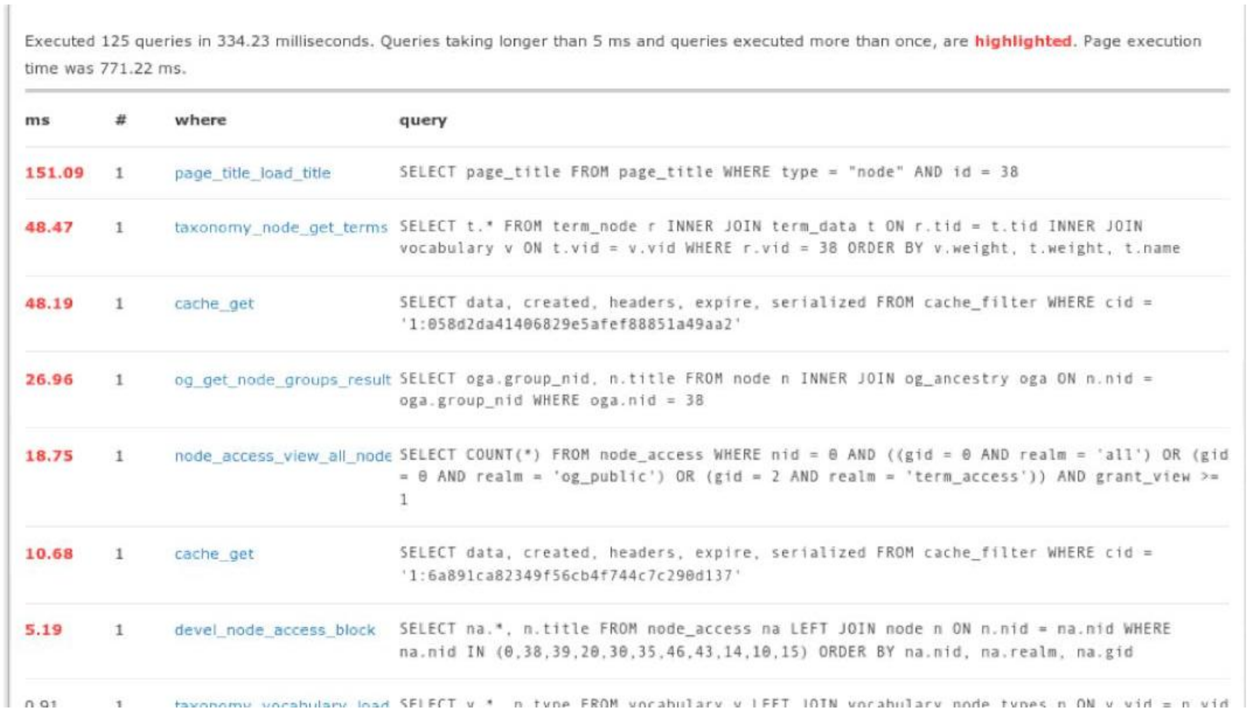
[✓ default]
blue
green
orange
schablon.com
disabled

elect a skin for your debug messages or select *disabled* to display object and array output in standard I

[ ] Rebuild the theme registry on every page load

Vhile creating new templates and theme_ overrides the theme registry needs to be rebuilt.

After you've completed the above, information will be listed at the bottom of your pages, as seen in the following screenshot:



Executed 125 queries in 334.23 milliseconds. Queries taking longer than 5 ms and queries executed more than once, are **highlighted**. Page execution time was 771.22 ms.

ms	#	where	query
151.09	1	page_title_load_title	SELECT page_title FROM page_title WHERE type = "node" AND id = 38
48.47	1	taxonomy_node_get_terms	SELECT t.* FROM term_node r INNER JOIN term_data t ON r.tid = t.tid INNER JOIN vocabulary v ON t.vid = v.vid WHERE r.vid = 38 ORDER BY v.weight, t.weight, t.name
48.19	1	cache_get	SELECT data, created, headers, expire, serialized FROM cache_filter WHERE cid = '1:058d2da41406829e5afef88851a49aa2'
26.96	1	og_get_node_groups_result	SELECT oga.group_nid, n.title FROM node n INNER JOIN og_ancestry oga ON n.nid = oga.group_nid WHERE oga.nid = 38
18.75	1	node_access_view_all_node	SELECT COUNT(*) FROM node_access WHERE nid = 0 AND ((gid = 0 AND realm = 'all') OR (gid = 0 AND realm = 'og_public') OR (gid = 2 AND realm = 'term_access')) AND grant_view >= 1
10.68	1	cache_get	SELECT data, created, headers, expire, serialized FROM cache_filter WHERE cid = '1:6a891ca82349f56cb4f744c7c290d137'
5.19	1	devel_node_access_block	SELECT na.*, n.title FROM node_access na LEFT JOIN node n ON n.nid = na.nid WHERE na.nid IN (0,38,39,20,30,35,46,43,14,10,15) ORDER BY na.nid, na.realm, na.gid
0.91	1	taxonomy_vocabulary_load	SELECT v.*, n.type FROM vocabulary v LEFT JOIN vocabulary_node_types n ON v.vid = n.vid

Using this information, you should be able to determine which queries are taking the longest to load. When you know that, you'll be able to narrow in on the culprit for the slow page loads. Besides slow queries, there might also be a bottleneck not related directly to Drupal. There are a number of ways to tune Drupal and a webserver to get optimal performance out of it.

See also:

These links include practical advice and a growing list of resources .

- Drupal caching, speed and performance http://drupal.org/node/326504
- Performance; improving the speed of Simpletest during development http://drupal.org/node/466972
- Drupal Performance Measurement & Benchmarking http://drupal.org/node/282862
- Improving Drupal's page loading performance, Wim Leers http://wimleers.com/article/improving-drupals-page-loading-performance