



# **Introduction to Module Development**

***Release 2.0***

**Acquia**

May 29, 2013

# CONTENTS

|          |  |           |
|----------|--|-----------|
| <b>1</b> | <b>Before we start</b>   | <b>1</b>  |
| 1.1      | Something to keep in mind . . . . .  | 1         |
| 1.2      | Install before we start . . . . .  | 1         |
| 1.3      | Schedule . . . . .   | 3         |
| <b>2</b> | <b>Create your first module</b>  | <b>4</b>  |
| 2.1      | Drupal as a framework . . . . .  | 4         |
| 2.2      | Step 1: Create a module . . . . .  | 6         |
| 2.3      | Step 2: Implement your first hook . . . . .                                  | 7         |
| 2.4      | Step 3: Menu System - Defining menu callbacks . . . . .                      | 8         |
| 2.5      | Step 4: Form Alter - Change Forms in Drupal . . . . .                        | 10        |
| 2.6      | Summary . . . . .  | 11        |
| <b>3</b> | <b>Adding more modules</b>   | <b>12</b> |
| 3.1      | The Red Button Module - Working with nodes . . . . .                         | 12        |
| 3.2      | Efficient coding . . . . .   | 15        |
| 3.3      | Introducing Mailfish . . . . .   | 16        |
| <b>4</b> | <b>Adding a Signup Tab</b>   | <b>22</b> |
| 4.1      | Create a Subscribe Tab Using hook_menu() . . . . .                           | 22        |
| 4.2      | Notes . . . . .  | 23        |
| <b>5</b> | <b>Form API: Creating the Mailfish Signup Form</b>                           | <b>24</b> |
| 5.1      | Form System: Drupal Form API and Form Creation . . . . .                     | 24        |
| 5.2      | Exercise: Using drupal_get_form as a callback . . . . .                      | 27        |
| 5.3      | Form Validation and Submission . . . . .                                     | 27        |
| <b>6</b> | <b>Adding an Install File</b>  | <b>31</b> |
| 6.1      | Exercise: Setup the database schema for mailfish subscription data . . . . . | 31        |
| <b>7</b> | <b>Database Integration</b>  | <b>34</b> |
| 7.1      | Adding Mailfish data to the database . . . . .                               | 34        |
| <b>8</b> | <b>Mailfish Permissions</b>  | <b>37</b> |
| 8.1      | The Permissions System . . . . .   | 37        |
| <b>9</b> | <b>Block system and theme system</b>   | <b>40</b> |
| 9.1      | Adding a Mailfish subscription block . . . . .                               | 40        |
| 9.2      | Theming the Mailfish Subscription Block . . . . .                            | 42        |

|  |           |
|--|-----------|
| <b>10 Adding Administrative Settings</b>                               | <b>46</b> |
| 10.1 Creating administrative settings . . . . .                        | 46        |
| 10.2 Drupal Variables . . . . .  | 48        |
| 10.3 Menu Callback Files (.admin.inc, .pages.inc) . . . . .            | 49        |
| <b>11 Reporting Results</b>  | <b>50</b> |
| 11.1 Exercise: Reporting results . . . . .                             | 50        |
| <b>12 Adding a Mailfish Setting Per Node</b>                           | <b>54</b> |
| 12.1 Exercise: Add a per-node setting with hook_form_alter() . . . . . | 54        |
| 12.2 Exercise: Add mailfish_enabled table to mailfish schema . . . . . | 56        |
| 12.3 Challenge exercise: Implement hook_update_N . . . . .             | 57        |
| <b>13 Database Integration</b>   | <b>59</b> |
| 13.1 Database Abstraction Layer . . . . .                              | 59        |
| 13.2 Exercise: Defining Our Mailfish Database Queries . . . . .        | 59        |
| <b>14 Node operations</b>  | <b>62</b> |
| 14.1 Exercise: Adding node hooks to mailfish.module . . . . .          | 62        |
| 14.2 Exercise: Display signup form conditionally . . . . .             | 64        |
| 14.3 Exercise: Add a signup form to the node page . . . . .            | 65        |
| <b>15 Testing your module: simpletest</b>                              | <b>67</b> |
| 15.1 Simpletest . . . . .  | 67        |
| <b>16 Appendix I: Drupal coding standards and conventions</b>          | <b>70</b> |
| 16.1 Using The Coder Module . . . . .                                  | 71        |
| <b>17 Appendix II: Module development conventions</b>                  | <b>73</b> |
| 17.1 Server Requirements . . . . .                                     | 74        |
| 17.2 Drupal site speed check . . . . .                                 | 74        |
| 17.3 Tools to improve performance . . . . .                            | 76        |
| 17.4 Check Front End Performance . . . . .                             | 77        |
| 17.5 Using Devel to locate slow queries . . . . .                      | 77        |

# BEFORE WE START

## 1.1 Something to keep in mind

Before we start writing our own custom code on top of Drupal, let's consider this maxim: *The best programmers are lazy programmers*. Drupal, more than many other systems, holds that 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.

## 1.2 Install before we start

### 1.2.1 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. Refer to the documentation for installation advice <https://docs.acquia.com/dev-desktop>

You will need 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>

### 1.2.2 Sample code

Download the sample code at <http://bit.ly/modulecode>

Please remember to TYPE out all exercises and code. Do not copy and paste. Use the sample code for reference only.

### 1.2.3 Drupal Tools

Install these modules, which are useful during 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.

**Admin menu module** [http://drupal.org/project/admin\\_menu](http://drupal.org/project/admin_menu) Quick access across administration options such as clearing the cache. Make sure you disable the core Toolbar module when you enable Admin menu.

### 1.2.4 Drush

Drush <http://drupal.org/project/drush> is not a module, but this tool provides command-line scripts to speed up development tasks. With Drush, you can download/enable/disable/uninstall, any modules/themes/translations and more.

For example, you can quickly install and enable modules with Drush.

1. First, browse to the directory of your current installation. If you have a multi-site set up, as with Acquia Dev Desktop, you must go to the directory where your selected site is located. For example  
`/path/to/my/drupal/install/sites/sitename.localhost/`

```
cd /path/to/my/drupal/install
```

2. Type the command to download a module, in this case Views.

```
drush dl views -y
```

3. Type the command to enable a module.

```
drush en views -y
```

#### Installation tips

Installation instructions available here: <http://drupal.org/project/drush>

Drush comes installed with Acquia's DevDesktop on Mac, and there is a handy Windows installer. When you install on a Windows environment, do not install it anywhere with a space in the file path. For example, installing Drush under C:\Program Files\drush is a bad idea. Try C:\drush instead.

#### Useful Drush commands

Refer to the documentation for more tips on using Drush <http://drupal.org/documentation/modules/drush>

| Command name        | Description                |
|---------------------|----------------------------|
| drush status        | Check if Drush is working  |
| drush cc all        | Clears all caches          |
| drush dl modulename | Download a selected module |
| drush en modulename | Enable a selected module   |

## 1.2.5 Resources

These sites are going to be useful throughout the course:

**Drupal API: [api.drupal.org](http://api.drupal.org)** The official programming reference for all Drupal versions.

**DrupalContrib.org** API documentation for contributed modules

**PHP.net/docs.php** Official PHP Manual is available online

**Drupal.stackexchange.com** Drupal Answers is a question and answer site for Drupal developers and administrators. It's 100% free, no registration required.

## 1.3 Schedule

The schedule is based on a 9-4 schedule.

### 1.3.1 Day 1

Take a 15 mins break mid-morning and mid-afternoon.

**9:00-9:30** Introductions

**9:30-12:00** Work on your first modules. Chapters 1-3

**12:00-1:00** Lunch

**1:00-3:45** Develop the Mailfish module. Chapters 4-8

**3:45-4:00** Review the main concepts.

### 1.3.2 Day 2

**9:00-9:15** Re-cap and set plan for the day

**9:15-12:00** Develop the Mailfish module further. Block/theme, admin settings and reports. Chapters 9-11

**12:00-1:00** Lunch

**1:00-3:30** Develop the Mailfish module, per-node settings, database integration, node operations and testing. Chapters 12-15

**3:30-4:00** Review of best practices in the Appendix, practice and questions.

# CREATE YOUR FIRST MODULE

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 What are hooks?

Drupal’s 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: `hook_menu()`

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:

1. An example from jQuery

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

2. An example from WordPress

- Taken from [http://codex.wordpress.org/Plugin\\_API](http://codex.wordpress.org/Plugin_API)

```
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'));
```

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.

```
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);
    }
}
```

### More example hooks

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.



|                  |   |
|------------------|---|
| hook_node_update | Call when a node is about to be updated                               |
| hook_menu        | Add custom paths to Drupal's menu system                              |
| hook_form_alter  | Alter to output of forms before they're rendered on a page            |
| hook_cron        | Called when cron is run   |
| hook_init        | Called at beginning of bootstrap process on each page view            |
| hook_block_info  | Tell Drupal about your custom blocks, loaded on the block config page |

## 2.2 Step 1: Create a module

### 1. Choose a descriptive name

- The first step in creating a new module is choosing a descriptive name. In this case, we are just going to call our module “My Module”.

### 2. Create the sites/all/modules/**custom** directory

---

**Tip:** Keep your module directory 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. Under site/all/modules create /custom for your own modules, and /contrib for contributed modules.

---

### 3. 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/custom and name it **mymodule**.

### 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 mymodule.
- Create the following blank file – sites/all/modules/custom/mymodule/**mymodule.info**

### 5. Open up the mymodule.info file, and add the following information to it:

```
name = My Module
description = My first Drupal module
core = 7.x
```

### 6. 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/custom/mymodule/**mymodule.module**

#### 7. Enable the module

- Go to the Modules Administration page and enable the module.
- 

**Tip:** Clear your cache

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.

1. Go to *Configuration* → *Development* → *Performance*
2. Click on Clear all caches.

It is worth noting that many developers install the Admin menu module ([http://drupal.org/project/admin\\_menu](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.

---

## 2.3 Step 2: Implement your first hook

Now we'll add our first hook: `hook_init()`.

1. Make sure that the Devel module is enabled
  - We'll be using the Devel module <http://drupal.org/project/devel> throughout the course to help us develop Drupal modules. Make sure it's enabled on your site. This module provides a function called `dpm()` that we'll use in this exercise.
  - The Drupal print message command - `dpm()` is provided by the Devel module to output variables for development debugging. This command also allows you to inspect array and objects. <http://api.drupal.org/api/devel/devel.module/function/dpm/7>
2. Implement `hook_init()`. This hook runs whenever a page request runs without a page cache.
  - To implement `hook_init()`, define a function called `mymodule_init()` in your `mymodule.module` file.
  - For any hook you use in your module, ALWAYS remove the word 'hook', and replace it with the name of your module.
  - Your code should look like this:

```
/**
 * Implement hook_init().
 */

function mymodule_init() {
}
```

---

**Tip:** Look up the API

Whenever you implement a new hook, look up the documentation page on <http://api.drupal.org>. For this hook, see [http://api.drupal.org/api/function/hook\\_init/7](http://api.drupal.org/api/function/hook_init/7)

### 3. Clear the cache

- Every time that you add a new hook in Drupal, you need to clear the cache. Go to *Configuration* → *Development* → *Performance* to clear the cache.

### 4. Test that the hook implementation is working

- In your implementation of `hook_init()`, add a `dpm()` call. Just pass a string to the `dpm()` function so that you can tell whether the module is working.

```
/**
 * Implementation of hook_init().
 */

function mymodule_init() {
  dpm('My Module's hook is running!');
}
```

---

**Tip:** Comments

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.

---

## 2.4 Step 3: Menu System - Defining menu callbacks

First we'll learn about Drupal's menu system, then we'll add our own callback.

### 2.4.1 About Drupal's menu system

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.

When you want your module to do something when a visitor visits a particular path (i.e. display a form), you need to create a page callback. This involves two steps:

1. Use `hook_menu()` to tell Drupal about the path.
2. Define a page callback function that will run when that page is requested and return the output of the page.

#### Anatomy of a call back

Open the API documentation and locate `hook_menu()`. Refer to [http://api.drupal.org/api/function/hook\\_menu/7](http://api.drupal.org/api/function/hook_menu/7)

This hook needs to return an array of menu items where the key is the path and the value is an array that defines the page (and menu item) associated with that path. For the moment, we'll add a single page, and create an array that defines the following:

**title** This will be the title of the page.

**description** The description of the menu item.

**page callback** We set the page callback to be 'mymodule\_callback'. This will be the name of the function that we'll create in the next step.

**access arguments** For the access arguments, we create an array of permissions that are required to view the page. Our module doesn't have its own permissions, so we'll use the 'administer site configuration' permission. Anyone with that permission will be able to view this page.

In this step, we want to take control of a particular path so that our module can print out information for administrators. To do this we need to add an entry to Drupal's menu routing system so that we may add a menu item for our new module at '/mymodule'. 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.

## 2.4.2 Exercise: Create your first page callback

In the code we'll add next, we are defining our new menu item using an array to describe its various elements to Drupal. This code says, "When a site admin goes to /mymodule, call the function mymodule\_callback." Drupal expects that function to return content which will make up the middle of the page a.k.a. the "content" region.

### 1. Open the API documentation

- More information on hook\_menu() can be found in Drupal's API docs.
- Refer to [http://api.drupal.org/api/function/hook\\_menu/7](http://api.drupal.org/api/function/hook_menu/7)

### 2. Implement hook\_menu() - Create your first page callback

- This hook needs to return an array of menu items where the key is the path and the value is an array that defines the page (and menu item) associated with that path.
- Course code should look like this:

```
/**
 * Implementation of hook_init().
 */

function mymodule_menu() {
  $items = array();
  $items['mymodule'] = array(
    'title' => 'My Page',
    'description' => 'Landing Page for My Module',
    'page callback' => 'mymodule_callback',
    'access arguments' => array('administer site configuration'),
  );
  return $items;
}
```

### 3. Add a page callback function

- Next we create a page callback function (using the function name we used in the last step). Our page callback function should just return some placeholder text at this point.

```
function mymodule_callback() {
  return 'My first page callback';
}
```

### 4. Clear the cache and test

- You've added code or changed the files in your module. You have to clear your cache to see the results.
- Go to /mymodule and see the output.

## 2.5 Step 4: Form Alter - Change Forms in Drupal

First we'll learn about `hook_form_alter()` and then implement it in your module.

### 2.5.1 About `hook_form_alter()`

To alter other modules' forms you implement `hook_form_alter()`.

This hook allows you 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.

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.

### 2.5.2 Exercise: Change the Comment Form

Let's say someone wants to change the label on the submit button in the comment form. We can alter the form output using `hook_form_alter()`.

#### 1. Implement `hook_form_alter()`

- Look up `hook_form_alter()` on [api.drupal.org](http://api.drupal.org). Which parameters should your function take?
- Add an implementation of `hook_form_alter()` to `mymodule.module`.

```
/**
 * Implementation of hook_form_alter().
 */

function mymodule_form_alter(&$form, $form_state, $form_id) {
  // add code here
}
```

#### 2. Alter just the comment form

- Add a check to see if the `$form_id == 'comment_node_article_form'`. This will ensure that only the comment form on article nodes are changed. We'll use a more advanced technique for targeting a specific form later on in the course.

```
if ($form_id == 'comment_node_article_form') {  
  // alter this form  
}
```

### 3. Inspect the form array

- Run a dpm() on the \$form array. Load a page where the comment form appears. The dpm should output the contents of \$form, in a format which you can click on to inspect.
- Find the submit button element in the array. This is what you'll need to alter in your function.

### 4. Change the form submit button text

- Change the submit button so that its label is 'Comment' instead of 'Save'.

```
/**  
 * Implementation of hook_form_alter().  
 */  
  
function mymodule_form_alter(&$form, $form_state, $form_id) {  
  if ($form_id == 'comment_node_article_form') {  
    $form['actions']['submit']['#value'] = t('Comment');  
  }  
}
```

### 5. Clear cache and test your module. Where should you go to see your changes?

## 2.6 Summary

You learned the essentials of adding a new module, using Drupal's API as a reference and implementing hooks in Drupal.

# ADDING MORE MODULES

In this chapter we'll add two more modules. In the first module, Red Button, you'll see how we will work with nodes. You'll repeat the same steps as you did in the previous chapter. In the second module, Mailfish, you will start a module which we'll develop throughout the rest of the course with increasing complexity. We'll reiterate some of the coding standards in Drupal.

## 3.1 The Red Button Module - Working with nodes

In this module, we'll start to work with nodes. You'll also get to practice the steps you learned in your first module.

### 3.1.1 About Renderable arrays

Drupal's "Renderable arrays" are covered in more detail later. If you have experience with theming, you have come across them. 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>`. See the documentation on Render arrays in Drupal 7: <http://drupal.org/node/930760>

`$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 Drupal developers, but an unavoidable part of Drupal development.

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

## Exercise: The Red Button module

# Test Red Button

View

Edit

Devel

Submitted by [admin](#) on Wed, 06/08/2011 - 10:55

This is a test!

**Delete this node**

Nodes do not have a direct deletion link like comments do in Drupal, we're going to change with our Red Button module! 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 click the edit tab first and then click delete, and then confirm deletion. 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.

Repeat the steps that we did to set up mymodule in the last chapter.

## 3.1.2 Step 1: Create the module

1. Choose a descriptive name
  - In this case, we are going to call our module "The Red Button."
2. Create the module folder
  - Create a new folder in sites/all/modules/custom and name it redbutton.
3. Create the .info file
  - Create the redbutton.info file in sites/all/modules/custom/redbutton.
4. Edit the .info file. 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
```
5. Create the .module file
  - Create the module file at sites/all/modules/custom/redbutton/**redbutton.module**

## 3.1.3 Step 2: Implement hook\_node\_view()

1. The first hook we will implement is hook\_node\_view(). In this exercise, we don't need to create a new page, we're just going to add some HTML markup to the existing nodes in Drupal.
2. Look up the documentation on [api.drupal.org](http://api.drupal.org/api/search/7/hook_node_view) to find out more about this hook and which arguments to use in your implementation: [http://api.drupal.org/api/search/7/hook\\_node\\_view](http://api.drupal.org/api/search/7/hook_node_view).



3. Open up `redbutton.module` and add our implementation of `hook_node_view()`. For now, just add a `dpm()` call to this function so that we know whether or not it's being called.

```
/**
 * Implementation of hook_node_view()
 */
function redbutton_node_view($node, $view_mode) {
  dpm('You are viewing a node');
}
```

### 3.1.4 Step 3: Enable the module and test it

1. Go to the Modules Administration page and enable the module.
2. Test your module by viewing a node page. Check whether your `dpm` statement is working.

### 3.1.5 Step 4: Add some text to the node page

1. In `hook_node_view()`, you have access to a renderable array (`$node->content`).
2. Replace your `dpm` with some code that adds additional text to the node object. This will be rendered as HTML on the page when you view the node. For now, just add some placeholder text:

```
/**
 * Implementation of hook_node_view()
 */
function redbutton_node_view($node, $view_mode) {
  $node->content['redbutton'] = array();
  $node->content['redbutton']['#markup'] = 'This is a node';
}
```

### 3.1.6 Step 5: Add a link

1. Replace the placeholder text with a link to delete the node. First, figure out what the path to that page should be. Place the link in HTML directly.
2. Next, use a variable in the link path to make sure that the correct node is deleted.
3. Test your module. Does your new delete link work?

### 3.1.7 Step 6: Use the `l()` function

Drupal includes a function for creating links. Rather than using HTML for our link, use the `l()` function to generate the link to the delete page.

---

**Tip:** The `l()` function

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.

---

Your implementation should look something like this:

```
function redbutton_node_view($node, $view_mode) {
  $link_text = t('Delete this node');
  $link_options = array(
    'attributes' => array('style' => 'color:#ff0000'),
  );
  $link_markup = l($link_text, "node/$node->nid/delete", $link_options);
  $node->content['redbutton'] = array();
  $node->content['redbutton']['#markup'] = $link_markup;
}
```

### 3.1.8 Step 7: Test your module

Return to another node and test your module.

### 3.1.9 Step 8: Challenge Exercise

Let's say we only want the link to appear on article nodes. How would you change the code to achieve this?

## 3.2 Efficient coding

Best practices for coding in Drupal.

### 3.2.1 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.

#### Doxygen comments

As you saw earlier 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>

#### 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,  
);
```

## 3.3 Introducing Mailfish

Next we will focus on creating a more sophisticated module called Mailfish.

### 3.3.1 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'.

# Test Event

View

Edit

Subscribe

Devel...

Email address \*

suzanne@evolvingwel

Join our mailing list




































Sign Up








## Database Integration

The email addresses users submit will be stored to the database.

127.0.0.1:33066 ▶ module ▶ mailfish "Stores the email address, timestamp, node id and user id..."

Browse Structure SQL Search Insert Export Import Operations Empty Drop







| Field                            | Type        | Collation       | Attributes | Null | Default | Extra          | Action  |
|----------------------------------|-------------|-----------------|------------|------|---------|----------------|---|
| <input type="checkbox"/> id      | int(10)     |                 | UNSIGNED   | No   | None    | AUTO_INCREMENT |                      |
| <input type="checkbox"/> uid     | int(11)     |                 |            | No   | 0       |                |                      |
| <input type="checkbox"/> nid     | int(11)     |                 |            | No   | 0       |                |               |
| <input type="checkbox"/> mail    | varchar(64) | utf8_general_ci |            | Yes  |         |                |        |
| <input type="checkbox"/> created | int(11)     |                 |            | No   | 0       |                |        |

Check All / Uncheck All With selected:       

Print view Relation view Propose table structure

Add 1 field(s) At End of Table At Beginning of Table After id Go

**Indexes:**

| Action  | Keyname   | Type  | Unique | Packed | Field | Cardinality | Collation | Null | Comment |
|---|-----------|-------|--------|--------|-------|-------------|-----------|------|---------|
|   | PRIMARY   | BTREE | Yes    | No     | id    | 0           | A         |      |         |
|   | node      | BTREE | No     | No     | nid   | 0           | A         |      |         |
|   | node_user | BTREE | No     | No     | nid   | 0           | A         |      |         |
|   |           |       |        |        | uid   | 0           | A         |      |         |

## 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.

**Sign up for *Acquia Training***

---

**Email address**

Join our mailing list

**Sign Up**

### Add a Reporting Page

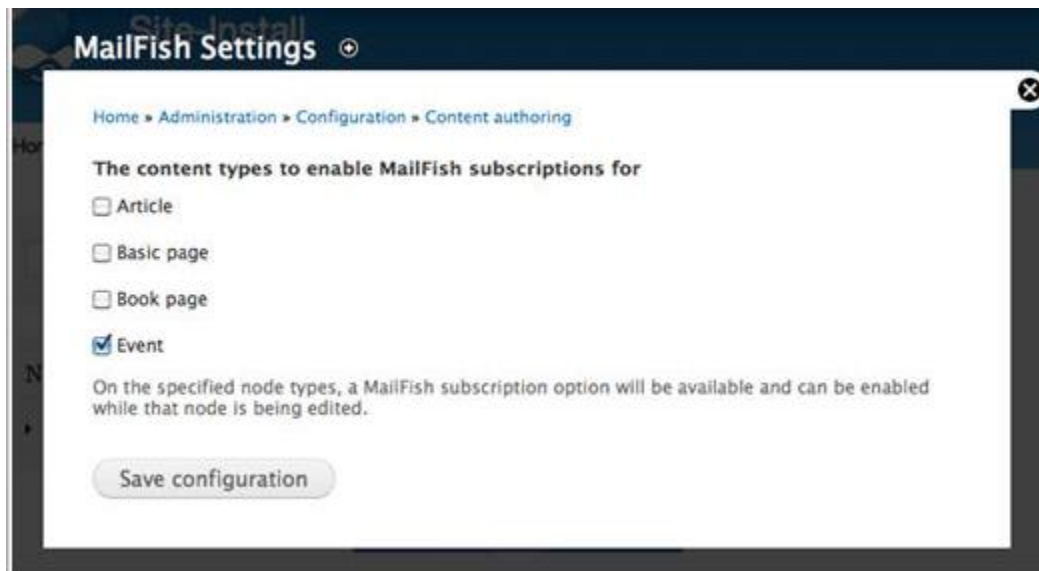
We will create a reporting page where administrators can see the email addresses submitted for each node.

| USER  | NODE       | EMAIL             | CREATED               |
|-------|------------|-------------------|-----------------------|
| admin | Test Event | admin@example.com | May 27, 2011 12:46 PM |

Thank you for joining the mailing list for Test Event. You have been added as admin@example.com.

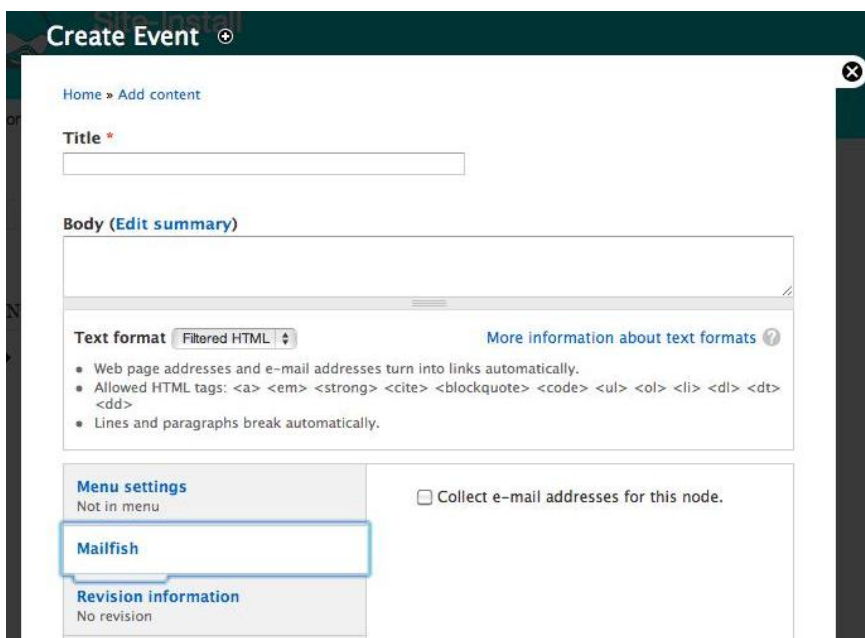
### Enable settings per-content type

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 per node

Administrators will also be able to turn on or off mailfish signups for an individual node.



## 3.3.2 Exercise: Create the Mailfish module

### Step 1: Create the module

1. Choose a descriptive name
  - In this case, we are going to call our module “MailFish”
2. Create the module folder

- Create a new folder in sites/all/modules/custom and name it mailfish.

### 3. Create the .info file

- Create the file mailfish.info and place in folder: sites/all/modules/custom/mailfish/**mailfish.info**

### 4. Edit 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
```

## 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 |
|--------------------------|------------------------|---------|---|------------|
| <input type="checkbox"/> | <b>Mailfish module</b> | 0.2-dev | Example module that allows users to subscribe to nodes.   |            |
| <input type="checkbox"/> | <b>Another module</b>  | 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 2: Start mailfish.module

1. Create the mailfish.module file at sites/all/modules/custom/mailfish/mailfish.module

2. 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 page or block.
 */
```

3. 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 page 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 3: Enable the module

In the Modules Administration list, Mailfish will be listed under a package, with a version number. Compare this with the other modules we have added.

#### ▼ MAILFISH

| ENABLED                             | NAME            | VERSION | DESCRIPTION   | OPERATIONS |
|-------------------------------------|-----------------|---------|---|------------|
| <input checked="" type="checkbox"/> | Mailfish module | 0.2-dev | Example module that allows users to subscribe to nodes. |            |

#### ▼ OTHER

| ENABLED                             | NAME           | VERSION | DESCRIPTION                              | OPERATIONS |
|-------------------------------------|----------------|---------|--|------------|
| <input checked="" type="checkbox"/> | The Red Button |         | Adds a delete link to every article node |            |



# ADDING A SIGNUP TAB

The Mailfish module allows users to signup to a node. Where are we going to put the signup form? We're going to create something called a 'Menu Local Task', a new tab on each node, alongside 'View' and 'Edit'.

The Devel module does something similar, adding a 'Devel' tab to each node. Open up the Devel module code and see if you can tell how it does this. Look at its implementation of `hook_menu()`.

## 4.1 Create a Subscribe Tab Using `hook_menu()`

In our first example module (mymodule), we created a page using `hook_menu`. You can also use `hook_menu()` to add a page whose path takes an argument. This is how node pages are generated, using the node ID as the argument. In this exercise, we'll use `hook_menu()` to add a subscribe tab to each node page.

### 4.1.1 Exercise: Add a subscribe tab to each node

#### Step 1: Add our `hook_menu`

What does the path need to be for our signup pages? We'll use the path 'node/%/signup' in our module. The '%' is a placeholder for the node ID argument.

Start by implementing `hook_menu()` for the mailfish module using this path. For now, use the permission 'access content' as the access argument.

```
function mailfish_menu() {
  $items = array();
  $items['node/%/subscribe'] = array(
    'title' => 'Subscribe',
    'description' => 'Subscribe to this node',
    'page callback' => 'mailfish_subscribe',
    'access arguments' => array('access content'),
  );
  return $items;
}
```

#### Step 2: Create a page callback

Create a page callback function. For now, our page callback should return a placeholder text. Test by going to any path that matches our path pattern (i.e. 'node/1/subscribe'). Do you see the placeholder text? Did you clear the cache?

The callback function for the signup page should look something like this:

```
function mailfish_subscribe() {
    return 'Placeholder for the subscribe page.';
}
```

### Step 3: Setup the subscribe page as a tab

We want the subscribe page to show up as a tab on each node. In order to make this appear, we need to specify a ‘type’ in the array that defines our menu item. Set the type to `MENU_LOCAL_TASK` to make this work.

Your `hook_menu()` implementation should look something like this:

```
function mailfish_menu() {
    $items = array();
    $items['node/%/subscribe'] = array(
        'title' => 'Subscribe',
        'description' => 'Subscribe to this node',
        'page callback' => 'mailfish_subscribe',
        'access arguments' => array('access content'),
        'type' => MENU_LOCAL_TASK,
    );
    return $items;
}
```

Your nodes should now have a subscribe tab like this:

## Test Event

---

Placeholder text for the subscribe page

## 4.2 Notes

### Translation of text in the page callback

Notice that the text returned by the page callback was passed through a function called `t()`. What do you think this function does?

### Translation of the title and description

Notice the title and description parameters in `hook_menu` are automatically parsed through `t()`, unlike all other strings that do require that call.

# FORM API: CREATING THE MAILFISH SIGNUP FORM

## 5.1 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.

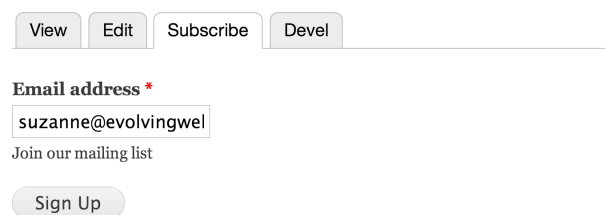
Each form element’s properties or attributes are listed items in the array corresponding to the element that they describe. In some cases form elements can be nested inside other form elements and so attributes are always prefixed with the ‘#’ symbol so that Drupal may distinguish between a form element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

Form API Reference: [http://api.drupal.org/api/drupal/developer!topics!forms\\_api\\_reference.html/7](http://api.drupal.org/api/drupal/developer!topics!forms_api_reference.html/7).

### 5.1.1 Exercise: Building The Email Submission Form

The first thing we need to do is build a form to collect our email address submissions. This form will be printed out on the ‘Subscribe’ tab that we created in the last chapter. It will look like this:

#### Test Event



#### *Form IDs:*

Every form in Drupal has an ID. The ID is a string which happens to be the name of the function which will be responsible for defining the structure of the form.

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.

### Step 1: Create the function to define our form

First, choose a form ID for our new form (let's say `mailfish_email_form`) and create a function in `mailfish.module` that defines this form. The name of the function will be the form ID. In this function, we need to create an array of form elements, which we can call `$form`. The `$form` array will be made up of arrays, each of which define a form element. Our function needs to return `$form`.

```
function mailfish_email_form($form, $form_state, $nid = 0) {  
  $form = array();  
  // TODO: Define the form fields as elements of $form  
  return $form;  
}
```

### Step 2: Add the email field

Which form elements do we need to create?

Let's create the email form element first. What type of form element is this going to be? Use the '#type' property to indicate this. You'll also need to specify a '#title' and '#description' property. Use the Form API Reference as a guide: [http://api.drupal.org/api/drupal/developer!topics!forms\\_api\\_reference.html/7](http://api.drupal.org/api/drupal/developer!topics!forms_api_reference.html/7).

### Step 3: Add the form to our subscribe tab

Now that we have our form roughly defined, we need to get it to show up on the subscribe tab. We can use the function `drupal_get_form()` to process our form.

In your page callback for the subscribe tab, replace the placeholder text with a call to `drupal_get_form()`. You'll need to pass the form ID to `drupal_get_form()` to tell Drupal which form to process and render. Test that the form is showing up on all the nodes.

Once the form is showing up, you'll notice some things that we need to fix or improve on. We'll address that in the following steps.

### Step 4: Make the email field required

Use the '#required' property to make the email field required.

### Step 5: Add a default value for the email address

If users are already logged into the site, we can use the email address as a default value. We can access the current logged in user's account by declaring:

```
global $user;
```

Then, use the `dpm()` function to inspect `$user` to see which values we have access to. Find out how to access the user's email address. Set the `$default_value` property to be the user's email address.

### Step 6: Add a submit button

We also need to include a submit button. For the submit button, you need to set a `#type` and `#value` property. Use the Form API Reference as a guide

## Step 7: Make the form contextual

In the form, we also need to keep track of the node ID for the node that the user is subscribing to. First, you'll need to add the node ID to the `mailfish_email_form()` function as an extra parameter.

What type of field should we use to keep track of this value in the form? If you're not sure, check the [Form API Reference](#).

Your `mailfish_email_form()` function should look something like this:

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

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

## Step 8: Pass the node ID to the form

Now that the signup form function accepts a node ID parameter, how do we get that information to the function?

When implementing `hook_menu()`, you can add a 'page arguments' property for each menu item that you define. 'Page arguments' is an array of items that will be passed as arguments to the page callback function.

In the `mailfish_menu()` function, add a 'page arguments' property to the subscribe tab menu item. Set the value of this element to be `array(1)`. This tells the callback function to use the path segment at position 1 as the argument. In our case, it will pass the node ID from the path to the page callback. This is how we can handle arguments in paths defined by a Drupal module.

In your `mailfish_subscribe()` function, add `$nid` as a parameter, and then pass this value as an argument to `drupal_get_form()`.

Your `hook_menu()` implementation should now look something like this:

```
function mailfish_menu() {
  $items = array();
  $items['node/%/subscribe'] = array(
    'title' => 'Subscribe',
    'description' => 'Subscribe to this node',
    'page callback' => 'mailfish_subscribe',
```

```

    'page arguments' => array(1),
    'access arguments' => array('access content'),
    'type' => MENU_LOCAL_TASK,
  );
  return $items;
}

```

Your callback function should look something like this:

```

function mailfish_subscribe($nid) {
  return drupal_get_form('mailfish_email_form', $nid);
}

```

### Page arguments

Page arguments are used extensively in almost every module. Refer to the API documentation for `hook_menu` to see the possibilities. Go to <http://api.drupal.org> and search for `hook_menu`.

### drupal\_get\_form

You're probably wondering what `drupal_get_form($form_id)` does. `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.

## 5.2 Exercise: 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:

Instead of calling the page callback we defined (`mailfish_subscribe`) which then in turn calls `drupal_get_form`, we can make `drupal_get_form` the page callback. We still need to tell `drupal_get_form` which `form_id` to generate. This is done by supplying the form ID as a page argument.

Your menu item array for the subscribe page should look like this:

```

$items['node/%/subscribe'] = array(
  'title' => 'Subscribe',
  'description' => 'Subscribe to this node',
  'page callback' => 'drupal_get_form',
  'page arguments' => array('mailfish_email_form', 1),
  'access arguments' => array('access content'),
  'type' => MENU_LOCAL_TASK,
);

```

The end result is that when someone navigates to `'node/[nid]/subscribe'`, the function `drupal_get_form` is called like this:

```
drupal_get_form('mailfish_email_form', $nid);
```

And the result of that function will be returned and displayed on the page.

## 5.3 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. We'll also create a submission handler to process the form values after they've been validated and tell the user that they've successfully signed up.

### 5.3.1 Exercise: Build the Validation Handler

If you try submitting the form right now, you'll see that you can enter any text into the form, and it will validate. We want the form to reject email addresses that are invalid and return an error like this:

#### Step 1: Create a validation handler

Create a validation handler function for the form. 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*. The validation handler takes `$form` and `&$form_state` as parameters.

#### Step 2: Add email validation

In the validation function, you have access to all the form values in the `$form_state['values']` array. How can you figure out which form values are available to you? *Hint:* use the Devel module.

Check whether the email address entered into the form is valid using Drupal's `valid_email_address()` function. [http://api.drupal.org/api/drupal/includes-common.inc/function/valid\\_email\\_address/7](http://api.drupal.org/api/drupal/includes-common.inc/function/valid_email_address/7).

#### Step 3: Set form errors

Use the function `form_set_error()` to set errors and stop submission of the form if the email address isn't valid. This function takes the name of the form element and the error message as arguments. Be sure to pass the error message through the `t` function.

Test the form. What happens if the email isn't valid?

#### Challenge Exercise: Translating strings with variables

Add the user's email address to the error message.

Since we're passing the message through the `t` function, how would we include the email and node title since these are variables? *Hint:* <http://api.drupal.org/api/drupal/includes%21bootstrap.inc/function/t/7>

Your validation handler should look something like this:

```
function mailfish_email_form_validate($form, &$form_state) {
  $email = $form_state['values']['email'];
  if (!valid_email_address($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);
  }
}

```

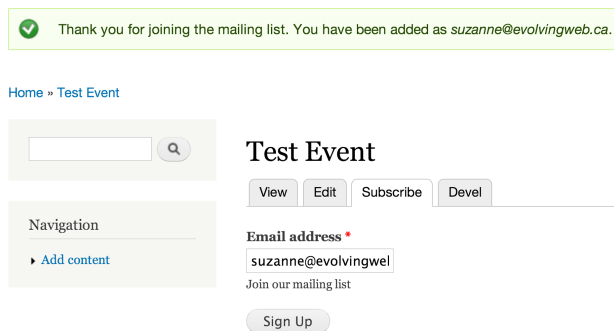
## What's Missing?

Once we have the database integration done and signups are being logged to the database, we'll also want to check to ensure that the user has not already subscribed to the node before validating the form.

### 5.3.2 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.

The submit handler will send a message to the user that they have successfully signed up for the node.



#### Step 1: Add a submit handler function

Add a submission handler function to `mailfish.module`.

#### Step 2: Tell the user they've subscribed

Add a `drupal_set_message()` call to tell the user that they've signed up for the node. This function takes a message as a parameter and adds it to the top of the next page that gets loaded. Make sure that you pass the message through the `t` function.

Test that your submit handler gets called when you fill in the form.

#### Step 3: Add a stub signup function

Create a `mailfish_signup()` function, which will add mailfish signup data to the database. For now, this will just pretend to sign up the user. We'll add the database integration in the next two chapters. The function should accept an email address and node ID as parameters. You can put a `dpm()` call `mailfish_signup()` to test that it's being called.

Your stub signup function should look something like this:



```
function mailfish_signup($email, $nid) {  
  dpm('Pretending to signup for this node');  
}
```

Add a call to the `mailfish_signup()` function in your submission handler.

Your submission handler should look something like this:

```
<?php  
  
/**  
 * Submission handler for mailfish_email_form.  
 */  
function mailfish_email_form_submit($form, &$form_state) {  
  // Get the title of the node to use in the successful signup message.  
  $nid = isset($form_state['values']['nid']) ? $form_state['values']['nid'] : 0;  
  if ($nid && is_numeric($nid)) {  
    // Signup the user.  
    mailfish_signup($form_state['values']['email'], $nid);  
  
    // Provide the user with a translated confirmation message.  
    drupal_set_message(t('Thank you for joining the mailing list. You  
have been added as %email.', array('%email' => $form_state['values']['email'])));  
  }  
}
```

# ADDING AN INSTALL FILE

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](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](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](http://api.drupal.org/api/function/hook_uninstall/7)



## 6.1 Exercise: Setup the database schema for mailfish subscription data

Before we can add signup data to the database, we need to create the table where that data will live. In this exercise, we'll create an install file that sets up our database schema.

127.0.0.1:33066 ▶ module ▶ mailfish "Stores the email address, timestamp, node id and user id..."







| Field                            | Type        | Collation       | Attributes | Null | Default | Extra          | Action  |
|----------------------------------|-------------|-----------------|------------|------|---------|----------------|---|
| <input type="checkbox"/> id      | int(10)     |                 | UNSIGNED   | No   | None    | AUTO_INCREMENT |      |
| <input type="checkbox"/> uid     | int(11)     |                 |            | No   | 0       |                |      |
| <input type="checkbox"/> nid     | int(11)     |                 |            | No   | 0       |                |      |
| <input type="checkbox"/> mail    | varchar(64) | utf8_general_ci |            | Yes  |         |                |      |
| <input type="checkbox"/> created | int(11)     |                 |            | No   | 0       |                |      |

Check All / Uncheck All With selected:     

Print view  Relation view  Propose table structure

Add 1 field(s) At End of Table At Beginning of Table After id Go

Indexes:

| Action  | Keyname   | Type  | Unique | Packed | Field | Cardinality | Collation | Null | Comment |
|---|-----------|-------|--------|--------|-------|-------------|-----------|------|---------|
|   | PRIMARY   | BTREE | Yes    | No     | id    | 0           | A         |      |         |
|   | node      | BTREE | No     | No     | nid   | 0           | A         |      |         |
|   | node_user | BTREE | No     | No     | nid   | 0           | A         |      |         |
|   |           |       |        |        | uid   | 0           | A         |      |         |

### 6.1.1 Step 1: Create mailfish.install

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

### 6.1.2 Step 2: Define the Mailfish schema

In the mailfish.install file, we need to implement hook\_schema to define the database for the mailfish subscriptions. The schema is going to define a table called 'mailfish'.

Which fields do you think we need to keep track of in the database?

- The ID, which will serve as the primary key for the data (type 'serial')
- The user ID to keep track of which user created the subscription (type 'int')
- The node ID, to track which node the user signed up to (type 'int')
- The email address (type 'varchar')
- The created date, to track the timestamp when the user subscribed (type 'int')

These fields are added to a serialized array for the mailfish table.

Your hook\_schema() implementation should look something like this:

```
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'),
),
);
return $schema;
}
```

### 6.1.3 Step 4: Reinstall the module

Our new schema will only be loaded when the module is first installed. Because the module is already installed, we need to uninstall and reinstall. 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.

# DATABASE INTEGRATION

## 7.1 Adding Mailfish data to the database

Now that the schema has been installed, and the mailfish table exists in the database, the next step is to add data to that table when a user signs up to a node.

We need to complete the mailfish\_signup function that we created earlier.

### 7.1.1 Exercise: Add mailfish signups to the database

#### Step 1: Create an array of signup data

Go back to the stub mailfish\_signup function we created in the Form API chapter. We'll create a new array, let's call it \$value, which will include all the values that we need to add to the database for the current signup. Each element in our array will correspond to a database column. The keys in the array should match up to the names of the fields in our schema, so refer back to the fields we added to the mailfish table in mailfish.install.

How can you get the values to add for each signup record?

- The \$email and \$nid values are already available to us in the mailfish\_signup() function.
- For the 'created' field, we can use a PHP function to get the current timestamp? Search <http://www.php.net/manual> to figure out which function to use.
- You can derive the user ID from the current logged in user. To get access the current user, you need to declare 'global \$user'. Then, use the dpm() function to inspect \$user and figure out how to access to user's ID.

#### Step 2: Call drupal\_write\_record() to add signups to the database

Drupal includes a database abstraction layer with useful functions for getting data in and out of the database. We'll use a function called drupal\_write\_record() to add signups to the mailfish table.

The function drupal\_write\_record() can be used to add records to the database when they match the schema exactly. We just need to pass it the name of the table ('mailfish') and the array of values to add.

In the mailfish\_signup() function, use drupal\_write\_record() to add the signup values to the mailfish table.

#### Step 3: Test!

Try out the signup form and check the database using PHP MyAdmin to make sure that records are being added properly.

Your code should look something like this:

```
<?php
/**
 * Store a mailfish email signup.
 */
function mailfish_signup($email, $nid) {
  global $user;

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

  drupal_write_record('mailfish', $value);
}
```

## 7.1.2 Exercise: Add check for previous signups in validation handler

Now that we have mailfish signups being added to the database, we can improve our validation functionality. Instead of blindly adding new signups to our mailfish table, we should check whether the email address has already been added to the list of signups.

### Step 1: Use the db\_query() function

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

In your validation handler, add a call to the db\_query function to determine whether an email address is already signed up to a node. Assign the returned value to a variable. In the next steps, we'll add arguments to this function to define the query.

### Step 2: Form your SELECT query

The first thing we pass to db\_query is a SELECT statement. We need to tell the database:

- which field to return (the email address)
- from which database table (mailfish)
- **what the conditions are for the query (that the nid and email address match** the values in our signup form)

Our SELECT query will look like this:

```
SELECT mail FROM {mailfish} WHERE nid = :nid AND mail = :mail
```

Even if you're familiar with MySQL queries, you'll need to be aware of some conventions to follow when forming queries for the db\_query() function:

- Table names within db\_query() functions are enclosed within curly brackets. For our query, mailfish is our table {mailfish}. This allows optional table name prefixing to work.
- 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. For our query, the identifiers are :nid and :mail.

**Step 3: Pass the db\_query() function an array of placeholders and values**

Our next step is to pass an associative array to db\_query(), which indicates the values for our placeholders. In the array, the keys are the placeholder identifiers and the values are the values from our form. In our db\_query call, we assign values to the placeholders :nid and :mail as follows:

```
array('nid' => $nid, 'mail' => $email)
```

**Step 4: Use the fetchField() method**

To obtain a result from our database SELECT query we append ->fetchField() to the end of the db\_query() function.

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

**Step 4: Add a form\_set\_error() call if the email is signed up**

If the email is already signed up, return an error message. Which function should we use to set an error?

Your updated validation handler should look something like this:

```
function mailfish_email_form_validate($form, &$form_state) {

    $email = $form_state['values']['email'];
    if (!valid_email_address($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 = $form_state['values']['nid'];
    if (!valid_email_address($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);
        // 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)));
        }
    }
}
```

# MAILFISH PERMISSIONS

## 8.1 The Permissions System

So far, anyone can sign up using the mailfish module who has permission to ‘access content’. To refine our module and provide more granular control over who can subscribe, we’ll need to add some custom permissions.

Drupal’s role-based permissions system is typically used to determine who can access 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 the `user_access()` function.

### 8.1.1 Permissions

We will want to add permissions 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](http://api.drupal.org/api/function/hook_permission/7)

#### Exercise: Define mailfish permissions with `hook_permission`

We need to define some custom permissions for the mailfish module.

| PERMISSION                    | ANONYMOUS<br>USER                   | AUTHENTICATED<br>USER               | ADMINISTRATOR                       |
|-------------------------------|-------------------------------------|-------------------------------------|-------------------------------------|
| <b>MailFish</b>               |                                     |                                     |                                     |
| View Mailfish subscriptions   | <input type="checkbox"/>            | <input type="checkbox"/>            | <input checked="" type="checkbox"/> |
| Create Mailfish subscriptions | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> |
| Manage Mailfish subscriptions | <input type="checkbox"/>            | <input type="checkbox"/>            | <input checked="" type="checkbox"/> |
| Administer Mailfish settings  | <input type="checkbox"/>            | <input type="checkbox"/>            | <input checked="" type="checkbox"/> |

#### Step 1: Implement `hook_permission()`

Open up the `mailfish.module` file if it’s not already open, and add an implementation of `hook_permission()`. This is a hook we can use to tell Drupal about the permissions for our module.

See [http://api.drupal.org/api/drupal/modules!system!system.api.php/function/hook\\_permission/7](http://api.drupal.org/api/drupal/modules!system!system.api.php/function/hook_permission/7).



## Step 2: Define Mailfish permissions

Your implementation of `hook_permission()` should return an array of permissions where each key is the name of the permission (in lower case, with spaces) and the value is an array defining that permission's title.

For now, we need a permission to allow users to create a mailfish subscription (i.e. signup for a node). We should also include permissions for subscribing using mailfish, managing subscriptions, and administering mailfish settings. We'll use these permissions later on. Here are the permissions we need to define:

- administer mailfish settings
- view mailfish subscriptions
- create mailfish subscriptions
- manage mailfish subscriptions

Your implementation of `hook_permission()` should look something like this:

```
<?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 subscriptions'),
    ),
    'administer mailfish settings' => array(
      'title' => t('Administer Mailfish settings'),
    )
  );
  return $perm;
}
```

## Step 3: Adjust access arguments for our menu item

In your implementation of `hook_menu()`, we use 'access content' as the access argument for the subscribe tab. Replace this with our custom permission: 'create mailfish subscriptions'.

```
$items['node/%/subscribe'] = array(
  'title' => 'Subscribe',
  'description' => 'Subscribe to this node',
  'page callback' => 'drupal_get_form',
  'page arguments' => array('mailfish_email_form', 1),
  'access arguments' => array('create mailfish subscriptions'),
  'type' => MENU_LOCAL_TASK,
);
```

**Step 4: Assign permissions.**

Administrators will now be able to assign these permissions to different roles on the permissions settings page. Clear the cache and go to the permissions page to test it out.

# BLOCK SYSTEM AND THEME SYSTEM

## 9.1 Adding a Mailfish subscription block

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>.

### 9.1.1 Exercise: Creating a Mailfish Subscription Block

We want to create a block for our subscription form. The block will look like this:



#### Step 1: Implement hook\_block\_info()

Our first step is to implement hook\_block\_info() - in the mailfish.module file. Hook\_block\_info() declares a block or set of blocks for the Mailfish module. This hook is triggered when you load the blocks configuration page.

The hook\_block\_info() implementation needs to return an array of blocks, where each key in the array is the block identifier (in this case, let's call our block 'mailfish\_email'). Each value is an array defining a block. For the mailfish\_email block, we just need to pass the array array(info => 'Mailfish signup block'). This tells Drupal that the administrative label of our block should be 'Mailfish signup block'. Go to [http://api.drupal.org/api/function/hook\\_block\\_info/7](http://api.drupal.org/api/function/hook_block_info/7) for more information about hook\_block\_info().

```
<?php
/**
 * Implements hook_block_info().
 */
function mailfish_block_info() {
  $blocks = array();
```

```

$blocks['mailfish_subscribe'] = array(
  'info' => t('Mailfish Signup Form'),
);
return $blocks;
}

```

## Step 2: Implement hook\_block\_view()

In order to render the block and display it to users, we add an implementation of `hook_block_view()` - [http://api.drupal.org/api/function/hook\\_block\\_view/7](http://api.drupal.org/api/function/hook_block_view/7).

In this function, we need to tell Drupal more about our block, defining the title that gets displayed to users (called the subject) and content. For now, we'll just add placeholder text to our block.

The `hook_block_view()` implementation will look something like this:

```

function mailfish_block_view($delta) {
  $block = array();
  switch ($delta) {
    case 'mailfish_subscribe':
      $block['subject'] = t('Sign up for this node');
      $block['content'] = t('Subscription form');
      break;
  }
  return $block;
}

```

## Step 3: Configure the block placement

Go to *Structure* → *Blocks* configuration page. Place the block in the sidebar of your site. View your site to test that your block is appearing.

## Step 4: Add the signup form to the block

Replace the placeholder text with a call to `drupal_get_form` to render our signup form in the block. The form takes a node ID as an argument. For now, just hardcode the node ID. Test that the form is appearing correctly.

```

function mailfish_block_view($delta) {
  $block = array();
  switch ($delta) {
    case 'mailfish_email':
      $block['subject'] = t('Sign up for this node');
      $block['content'] = drupal_get_form('mailfish_email_form');
      break;
  }
  return $block;
}

```

## Step 5: Make the block signup contextual

In order to make the signup form contextual (i.e. signup the user to the correct node), we need to get the node ID for the current page and pass it to `drupal_get_form()`.

We can access the segments of the system path to the current page using the `arg()` function. For example, use `arg(0)` to get the first segment of the path (i.e. 'node') and `arg(1)` to get the second segment (i.e. the node ID).

We can check whether we're looking at a node page, and then retrieve the node ID using the following code:

```
if (arg(0) == 'node' && is_numeric(arg(1))) {
  $nid = arg(1);
}
```

Once you have the `$nid` value, pass it to `drupal_get_form()`.

Your finished `hook_block_view()` should look something like this:

```
function mailfish_block_view($delta) {
  $block = array();
  switch ($delta) {
    case 'mailfish_subscribe':
      if (arg(0) == 'node' && is_numeric(arg(1))) {
        $nid = arg(1);
        $form = drupal_get_form('mailfish_email_form', $nid);
        $block = array(
          'subject' => "Mailfish Subscription",
          'content' => theme('mailfish_block', array('rendered_form' => drupal_render($form))),
        );
      }
      break;
    }
  return $block;
}
```

## 9.2 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()` and `hook_block_view()`. This 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 function and template 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

### 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](http://api.drupal.org/api/function/theme_username/7)

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

## 9.2.1 Exercise: Theming the Mailfish Subscription Block

### Step 1: Implement hook\_theme()

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

Open up the `mailfish.module` file and implement `hook_theme()`. Our function should return an array of themable elements. Each element is defined by an array, in which we need to specify the name of the template to use when rendering the element and any variables that the template needs to print out. In our case, we need to define a `'rendered_form'` variable.

Your `hook_theme()` implementation should look something like this:

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

### Step 2: Create 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.

The file should live at `sites/all/modules/mailfish/mailfish-block.tpl.php`.

In the mailfish template file, you will have access to the `$rendered_form` variable, which you should print out. Add some extra markup so that you know whether or not the template is being used.

Your `mailfish-block.tpl.php` file should look something like this:

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

### Step 3: Call the theme function

Next, we need to tell the block system to use that template to theme our particular block. In `mailfish_block_view`, we're going to pass the rendered form to the theme function. This will send it through Drupal's theming system and eventually to your template.

```
<?php
$form = drupal_get_form('mailfish_email_form', $nid);
$block = array(
  'subject' => "Mailfish Subscription",
  'content' => theme('mailfish_block', array('rendered_form' => drupal_render($form))),
);
```

“`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.

### Step 4: Clear the theme registry

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.

### Step 5: Test

Test that your template is being used when the Mailfish subscription form is rendered.

#### 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](http://drupal.org/project/admin_menu)

Enable the following modules from that package:

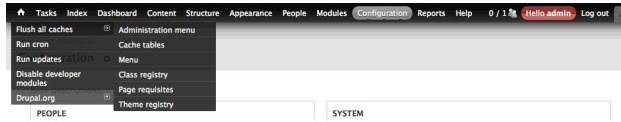
- Administration Development tools

- 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.



# ADDING ADMINISTRATIVE SETTINGS

## 10.1 Creating administrative settings

So far, we've set up a subscription form for every node on our site.

To make our module more flexible, we'll need to set up configuration settings so that administrators can choose to enable the subscription functionality for certain content types, and even on a node-by-node basis.

We'll start by providing an administrative settings form to allow administrators to choose which content types to enable mailfish for.

### 10.1.1 Exercise: Create a mailfish settings form

#### Step 1: Add a mailfish settings page

In your `hook_menu()` implementation, add a new menu item for a mailfish settings form. The path to this page should be `'admin/config/content/mailfish'`. Add a placeholder callback function to return a placeholder text. Test that the new admin settings page is working.

#### Step 2: Restrict permission to the settings page

Restrict access to this page to users who have the `'administer mailfish settings'` permission.

#### Step 3: Define the settings form

Add a settings form, with form ID `'mailfish_admin_settings_form'`. This should define a form with a single form element `'mailfish_types'`. This will be a set of checkboxes indicating which content types subscriptions are available for.

The checkboxes form element uses an `'#options'` property to specify a list of checkbox labels/values. You'll need to pass an array to this property.

For now, just use some placeholder checkboxes:

```
'#options' => array('option1', 'options2');
```

### Step 4: Use `system_settings_form()` to handle form submission

Rather than returning just the `$form` array, pass this array through `system_settings_form()` first. This will take care of the submit button and submit handler for us. We won't need to create validation and submission handlers.

### Step 5: Add the form to the settings page

Replace your placeholder callback function with a `drupal_get_form()` call and pass the form ID as a page argument. Now test that the form is appearing on your settings page.

### Step 6: Use Drupal's list of content types

Rather than creating a placeholder list of options for our checkboxes, replace it with a call to `node_type_get_names()`, which returns a list of content types. Test how this works in your form.

### Step 7: Set the `mailfish_types` default value

We want the admin settings form to pre-populate the Mailfish types checkbox with a the Mailfish types we've already chosen. We'll add a `#default_value` property, and populate it by calling `'variable_get'`, which will get the `'mailfish_types'` variable from the database. The `mailfish_types` value will be stored to the database by the `system_settings_form()` function.

```
'#default_value' => variable_get('mailfish_types', array()),
```

Test that the form is now storing the content types that you've selected.

Your function should look something like this:

```
<?php
/**
 * Defines the Mailfish admin settings form.
 */
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);
}
```

### Step 8: Use the `mailfish_types` setting

Conditionally show the subscription form if the node is in the list of `mailfish_types`.

Where will you need to change your code to make this work? Which function can you use to check this setting?

Add a check for the node type in `mailfish_block_view()` like this:

```
function mailfish_block_view($delta) {
  $block = array();
  switch ($delta) {
    case 'mailfish_subscribe':
      if (arg(0) == 'node' && is_numeric(arg(1))) {
        $nid = arg(1);
        $node = node_load($nid);
        $types = variable_get('mailfish_types', array());
        if (!empty($types[$node->type])) {
          $form = drupal_get_form('mailfish_email_form', $nid);
          $block = array(
            'subject' => "Mailfish Subscription",
            'content' => theme('mailfish_block', array('rendered_form' => drupal_render($form))),
          );
        }
      }
      break;
  }
  return $block;
}
```

### Step 9: Implement 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() {
  variable_del('mailfish_types');
}
```

## 10.2 Drupal Variables

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](http://api.drupal.org/variable_get)
- *variable\_set()* - [http://api.drupal.org/variable\\_get](http://api.drupal.org/variable_get)
- *variable\_del()* - [http://api.drupal.org/variable\\_get](http://api.drupal.org/variable_get)

## 10.3 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.

### 10.3.1 Exercise: Create the mailfish.admin.inc file

#### Step 1: Start mailfish.admin.inc

Create a mailfish.admin.inc file and place it in the mailfish module folder.

#### Step 2: Move our admin form function to that file

Open up the mailfish.admin.inc file and move the mailfish\_admin\_settings\_form function to it.

#### Step 3: Modify the hook\_menu implementation

In our hook\_menu() implementation, we need to add a new 'file' property to our menu item indicating where the page callback is located.

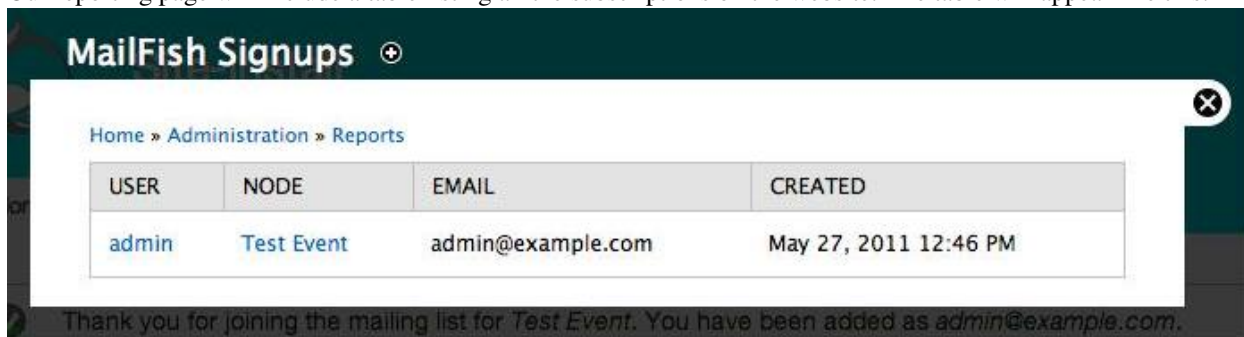
```
<?php
$items['admin/config/content/mailfish'] = array(
  'title' => 'Mailfish Settings',
  'description' => 'Administer Mailfish Settings.',
  'page callback' => 'drupal_get_form',
  'page arguments' => array('mailfish_admin_settings_form'),
  'access arguments' => array('manage mailfish settings'),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

# REPORTING RESULTS

We also want to provide site administrators with a list of all the node email subscriptions on their site. First, we need to add a Mailfish reports page to the site using our `hook_menu()` implementation. Then we'll pull all the relevant subscription information from the database and display it on the reports page at `admin/reports/mailfish`. To do this, we'll add a callback function to our `mailfish.admin.inc` file.

## 11.1 Exercise: Reporting results

Our reporting page will include a table listing all the subscriptions on the website. The table will appear like this:



| USER  | NODE       | EMAIL             | CREATED               |
|-------|------------|-------------------|-----------------------|
| admin | Test Event | admin@example.com | May 27, 2011 12:46 PM |

Thank you for joining the mailing list for *Test Event*. You have been added as *admin@example.com*.

### 11.1.1 Step 1: Add a reporting page

Modify our implementation of `hook_menu()` to add a reporting page at `'admin/reports/mailfish'`. This page should only be accessible to users with permission to `'view mailfish subscriptions'`.

The menu item in your `hook_menu()` implementation should look like this:

```
<?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',
  );
```

### 11.1.2 Step 2: Create a page callback

We'll need to create a new page callback to render the table of subscription data. In which file should this callback appear? Create a page callback function called `mailfish_signups()` and test that it's working.

### 11.1.3 Step 3: Pull out the subscription data from the database

In the page callback, we need to get the records from the mailfish table out of the database. We do this by creating a query object and then running the query on the database and generating an array of results.

Here's the code we'll use to generate our `$results` array:

```
// Dynamically load the schema for the mailfish table
$fields = drupal_get_schema('mailfish');

// Instantiate a query object by using the db_select wrapper
$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'))
  ->orderBy('m.created', $direction = 'ASC')
  ->execute()
  ->fetchAll();
```

There's a lot going on in this function. Here's a breakdown:

1. Get the array of `$fields` that we need from our mailfish table schema
2. Create a `$query` object using the `db_select()` function to grab the whole whole mailfish table
3. Do a join on the node table to also retrieve node information for each node referenced in the mailfish table
4. Load the fields we need by running the `fields()` method on our query object
5. Run the `orderBy()` method to sort the fields by the 'created' date
6. Use the `execute()` method to run the query against the database
7. Use the `fetchAll()` method to return an array of objects, one for each result

At this point, you'll want to use the `dpm()` function to inspect `$results` to see what data we have available to us.

### 11.1.4 Step 4: Create an array of `$results` to include in the table

Next, we need to create an array called `$rows` that we can use to create a table. This is a keyless array in which each element is an array that defines a row in our table. For each row, we need to specify:

- The username of the user who created the subscription (if they have an account)
- The node that they subscribed to
- The user's email address
- The date on which they subscribed

We'll use the data in the `$results` array to generate `$rows`.

```
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),
  );
}
```

### 11.1.5 Step 5: Render the results as an HTML table

Now we have our results in an array that we can pass to the theme function. We also need to pass an array called `$header` to the theme function. This array includes the table column names (which will be rendered in HTML as the th elements). Each item in the array is a table header label. Use the same order that we did for the data in `$results` above.

The theme function takes the name of the theme hook as the first argument. In our case, this is `'table'`. The second argument is an array of variables defining how the element should be rendered. The structure of this array will vary depending on what you'd like to render (i.e. a list, an image, etc.) See <http://api.drupal.org/api/function/theme/7>.

Our call to the theme function should look something like this:

```
theme('table', array('header' => $header, 'rows' => $rows));
```

Your completed page callback function should look something like this:

```
<?php

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

  // Dynamically load the schema for this table
  $fields = drupal_get_schema('mailfish');

  // Instantiate a query object by using the db_select wrapper
  $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'))
    ->orderBy('m.created', $direction = 'ASC')
    ->execute();
```

```
->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;
```

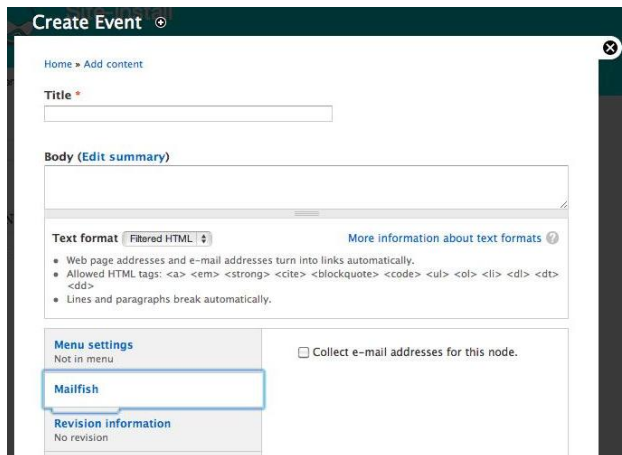
### 11.1.6 Challenge Exercise: Displaying results per node

How would you display a table of results per node? How would you add a new menu item for each node? How would you modify the code above to be node-specific?



# ADDING A MAILFISH SETTING PER NODE

## 12.1 Exercise: Add a per-node setting with hook\_form\_alter()



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. To add this setting, we will implement `hook_form_alter()` - [http://api.drupal.org/api/drupal/modules-system-system.api.php/function/hook\\_form\\_alter/7](http://api.drupal.org/api/drupal/modules-system-system.api.php/function/hook_form_alter/7)

Most `hook_form_alter` implementations 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](http://api.drupal.org/api/function/hook_form_FORM_ID_alter/7)

### 12.1.1 Step 1: Implement hook\_form\_alter()

Add an implementation of `hook_form_alter()` to `mailfish.module`. This function takes three parameters:

**&\$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. `hook_form_alter()` will alter all forms on your Drupal site. To target a single form, we need to figure out the form ID.

***\$form\_id*** The name that identifies form being altered

In the `hook_form_alter()` implementation, use the `dpm()` function to print the `$form_id` to the page. Load the node edit form to determine the form ID.

```
function mailfish_form_alter(&$form, &$form_state, $form_id) {  
  dpm($form_id);  
}
```

### 12.1.2 Step 2: Implement `hook_form_BASE_FORM_ID_alter()`

Now that you know what the form ID is, replace the `hook_form_alter()` implementation with an implementation of `hook_form_BASE_FORM_ID_alter()` where you replace the `BASE_FORM_ID` with the form ID you found in the previous step.

### 12.1.3 Step 3: Add a form element for `mailfish_enabled`

Add a new checkbox form element called 'mailfish\_enabled'. Populate the `#default_value` for the form element with `$node->mailfish_enabled`. This is a value that we'll store for each node.

Clear cache and test the form. Does the new checkbox appear?

Since we haven't set up a way of storing `mailfish_enabled` in the database, our form value for this setting won't get saved yet.

### 12.1.4 Step 4: Add a fieldset for our form element

Rather than mixing our setting with all the other form elements on the page, create a fieldset form element called 'mailfish'. Then, modify our checkbox so that it gets added to the fieldset.

When building forms in Drupal, you can put fields inside of a fieldset by adding the form elements as children of the fieldset array. Here we are adding the checkbox field inside the fieldset.

### 12.1.5 Step 5: Add the fieldset to the 'vertical tabs'

The vertical tabs are the tabs at the bottom of the node edit form, with the 'Menu settings', 'Publishing options', etc. Inspect the form array to figure out how to add our fieldset to this group.

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

### 12.1.6 Step 6: Only add the setting to certain nodes

Do a check on our 'mailfish\_types' variable so that this checkbox is only added for nodes if the content type is in the list of mailfish types.

### 12.1.7 Step 7: Check for permissions

Add a permissions check to only show the setting if the user has permission to ‘administer mailfish settings’.

Your code should look something like this:

```
<?php
/**
 * Implements hook_form_BASE_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,
    );
  }
}
```

## 12.2 Exercise: Add mailfish\_enabled table to mailfish schema

Now that we have a form to collect the mailfish setting for each node, we need to make a new table to store this information in the database.

### 12.2.1 Step 1: Update the install file

Open up the mailfish.install file. Add a table called mailfish\_enabled with a single field of type ‘int’ to store the node ID.

```
$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'),
);

```

### 12.2.2 Step 2: Reinstall the Mailfish module

Tell Drupal about our new database table by uninstalling and re-installing the Mailfish module. Alternatively, do the next challenge exercise to add the table using an update hook.

### 12.2.3 Step 3: Test

Use PHP MyAdmin or the command line to test that the new mailfish\_enabled table is appearing in the database.

## 12.3 Challenge exercise: Implement hook\_update\_N

### 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](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.

Rather than uninstalling and reinstalling the mailfish module to install the new database tables, we implement hook\_update\_N and then run update.php.

### 12.3.1 Step 1: Implement hook\_update\_N()

Implement your update function in mailfish.install. By convention, you'll want to use 7001 as the update number, so your function will be called mailfish\_update\_7001. If you make a subsequent database update, this could be implemented by creating a function called mailfish\_update\_7002.

### 12.3.2 Step 2: Create the new table programmatically

In your hook implementation, use db\_create\_table() to add the new table to our database. This function takes two parameters:

**name** The name of the new table (a string)

**table** An array defining our new table

When our update runs, Drupal will add the new table for us. See [http://api.drupal.org/api/drupal/includes!database!database.inc/function/db\\_create\\_table/7](http://api.drupal.org/api/drupal/includes!database!database.inc/function/db_create_table/7) for more details about db\_create\_table().

Your update hook implementation should look something like this:

```
<?php
function mailfish_update_7001() {
  $mailfish_enabled_table = 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'),
  );
  db_create_table('mailfish_enabled', $mailfish_enabled_table);
}
```

### 12.3.3 Step 3: Run update.php

Run update.php by going to '/update.php' in the browser. Alternatively, from the command line you can call drush updatedb.

# DATABASE INTEGRATION

Now, we're going to learn about some of the other database API functions you need to use to update the mailfish\_enabled table in the database.

## 13.1 Database Abstraction Layer

So far, we've learned about the db\_query() function when checking for previous signups and creating our reporting page. Other Drupal database functions include:

| database functions | Notes   |
|--------------------|---|
| db_insert()        | Used to run INSERT queries to the active database.<br><a href="http://api.drupal.org/api/function/db_insert/7">http://api.drupal.org/api/function/db_insert/7</a> |
| db_update()        | Used to run UPDATE queries to the active database.<br><a href="http://api.drupal.org/api/function/db_update/7">http://api.drupal.org/api/function/db_update/7</a> |
| db_delete()        | Used to run DELETE queries to the active database.<br><a href="http://api.drupal.org/api/function/db_delete/7">http://api.drupal.org/api/function/db_delete/7</a> |

## 13.2 Exercise: Defining Our Mailfish Database Queries

We are now going to define the database wrapper functions to manage the mailfish\_enabled setting per node in the database. These functions will be called when we create, edit, and delete nodes to keep this setting up-to-date.

### 13.2.1 Step 1: Add a mailfish\_get\_node\_enabled() function

Open mailfish.module if it's not already open, and create a mailfish\_get\_node\_enabled() function. This will take the \$nid as a parameter and return TRUE or FALSE based on whether or not the setting is enabled for the given node.

Use the db\_query() function to wrap our database query. This returns an iterable object. We'll call the fetchField() method on this object to pull out the \$nid field.

```
<?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;
}

```

### 13.2.2 Step 2: Add a mailfish\_set\_node\_enabled() function

Now we'll create a mailfish\_set\_node\_enabled() function to set mailfish\_enabled in the database for a given node. The function will take \$nid as a parameter, and doesn't need to return anything.

Use the db\_insert() function to add the \$nid to the mailfish\_enabled table in the database. This function returns an object. We can run the fields() method on the object to specify which fields to insert, and the execute() method which we use to actually insert the row in the database.

Your setter function should look something like this:

```

<?php
/**
 * 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($nid) & ! mailfish_get_node_enabled($nid)) {
        db_insert('mailfish_enabled')
            ->fields(array('nid' => $nid))
            ->execute();
    }
}

```

### 13.2.3 Step 3: Add a mailfish\_delete\_node\_enabled() function

When subscriptions get disabled for a certain node, or a node gets deleted, we need to have a way to remove rows from the mailfish\_enabled table.

Add a new mailfish\_delete\_node\_enabled() function, which takes \$nid as a parameter.

Use the db\_delete() function to delete a row from the mailfish\_enabled table. Use the condition() method to select only the nodes that match our \$nid, and the execute() function to run our database change.

Your function should look something like this:

```

<?php
 * 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)) {  
    db_delete('mailfish_enabled')  
        ->condition('nid', $nid)  
        ->execute();  
}  
}
```

### 13.2.4 Why use database wrapper functions?

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.



# 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](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.

## 14.1 Exercise: Adding node hooks to mailfish.module

In order for us to run the database wrapper functions that we created in the last chapter, we need to hook into various node operations in Drupal. When a node is created, updated, deleted, or loaded, we need to run the appropriate database wrapper functions to keep the `mailfish_enabled` setting up-to-date for each node. For example, when a node is deleted, we need to make sure to delete the corresponding row in the `mailfish_enabled` table if it exists.

### 14.1.1 Step 1: Implement `hook_node_load()`

The `node_load()` function gets called whenever the node object is used. We need add mailfish data to the node object when it gets loaded (so that we can use `$node->mailfish_enabled` in our code when we load the `$node` object).

[http://api.drupal.org/api/function/hook\\_node\\_load/7](http://api.drupal.org/api/function/hook_node_load/7)

When implementing `hook_node_load()`, we have the `$nodes` parameter available to us. This is because the `node_load` function can load more than one node at a time.

In our function, we need to iterate through `$nodes` and set `$node->mailfish_enabled = mailfish_get_node_enabled($node-nid)` for each one.

### 14.1.2 Step 2: Implement `hook_node_insert()`

When a node is being created for the first time, we need to add a `mailfish_enabled` record if mailfish is enabled for that node.

[http://api.drupal.org/api/function/hook\\_node\\_insert/7](http://api.drupal.org/api/function/hook_node_insert/7)

In our function, we have the `$node` parameter available to us (the node object). Since we've already implemented `hook_node_load()`, what data about the node can we retrieve from `$node`?

Which database helper function do we need to call if mailfish\_enabled is set?

### 14.1.3 Step 3: Implement hook\_node\_update()

When a node is being updated (i.e. edited after its initial creation) we can delete the existing mailfish\_enabled record, and add a new one if it's set for this node.

[http://api.drupal.org/api/function/hook\\_node\\_update/7](http://api.drupal.org/api/function/hook_node_update/7)

In our function, we have the \$node parameter available to us, and we can check whether mailfish\_enabled is already set using `$node->mailfish_enabled`.

Which database helper functions should we use to keep the mailfish\_enabled setting up-to-date?

### 14.1.4 Step 4: Implement hook\_node\_delete()

When a node gets deleted, we need to make sure that the mailfish\_enabled record for that node is deleted (if it exists).

[http://api.drupal.org/api/function/hook\\_node\\_delete/7](http://api.drupal.org/api/function/hook_node_delete/7)

Your node operations functions should look something like this:

```
<?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);
}
```

## 14.2 Exercise: Display signup form conditionally

Now that our mailfish\_enabled table is kept up-to-date, we can use it to determine whether or not to show the ‘subscribe’ tab, or the signup block.

### 14.2.1 Step 1: Add a condition to our block display

Start by adding a condition to our hook\_block\_view() implementation to test whether mailfish is enabled for the current node.

*Hint:* Use the mailfish\_get\_node\_enabled() function.

### 14.2.2 Step 2: Modify the ‘Subscribe’ menu local task

For the subscribe tab, it’s a bit more complicated. We want to display the tab only if the current node has mailfish enabled. We’ll need to implement something called an ‘access callback’ to provide the logic for whether or not to display the tab. By default, the user\_access function is used as the access callback. This checks that the user accessing the page has the appropriate permissions.

In the hook\_menu() implementation, add an ‘access callback’. Then, add the node ID to the ‘access arguments’ (just like we did for the page arguments).

```
<?php
  $items['node/%/subscribe'] = array(
    'title' => 'Subscribe',
    'description' => 'Subscribe to this node',
    'page callback' => 'drupal_get_form',
    'page arguments' => array('mailfish_email_form', 1),
    'access callback' => 'mailfish_access_subscribe_tab',
    'access arguments' => array('create mailfish subscriptions', 1),
    'type' => MENU_LOCAL_TASK,
  );
```

### 14.2.3 Step 3: Add an access callback for the subscribe tab

Our access callback takes two parameters, the permission that the user needs to see the signup form, and the node ID. In the callback, we need to do a check whether the user has permission to add subscriptions, as well as a check whether the node is Mailfish enabled. The function should return TRUE or FALSE.

The access callback should look like this:

```
<?php
/*
 * Access callback for subscribe tab
 */
function mailfish_access_subscribe_tab($permission, $nid) {
  return mailfish_get_node_enabled($nid) && user_access($permission);
}
```

## 14.3 Exercise: Add a signup form to the node page

Another node hook that we can use is `hook_node_view()`. This is triggered when the node is rendered on the page to the user.

In this exercise, we'll implement `hook_node_view()` to display the signup form as part of the node content. It will look like this:

### Test Event

Submitted by [suzanne](#) on Fri, 02/15/2013 - 20:16

We are having an event on Friday. Enter your email to RSVP.

Email address \*

Join our mailing list

### 14.3.1 Step 1: Implement `hook_node_view()`

Add an implementation of `hook_node_view()` to `mailfish.module`. Which parameters does this function take? See [http://api.drupal.org/api/function/hook\\_node\\_view/7](http://api.drupal.org/api/function/hook_node_view/7).

### 14.3.2 Step 2: Add a placeholder text to each node

In `hook_node_view()`, you have access to a renderable array (`$node->content`) that will be rendered on the node page as HTML.

Add your own custom renderable array to the node output like this:

```
$node->content['mailfish'] = array();
$node->content['mailfish']['#markup'] = 'Hello World';
```

The `#markup` property takes a string, and this string will be added to the content of the node.

### 14.3.3 Step 3: Move the placeholder text to the bottom of the node

You can specify a `#weight` property in the renderable array to specify the order in which the element should appear (the lower the weight, the higher it appears on the page).

```
$node->content['mailfish']['#weight'] = 100;
```

### 14.3.4 Step 3: Replace the placeholder text with the form

Now that we see how to add content to the node, try replacing the placeholder text with the signup form. Since the `drupal_get_form()` returns a renderable array, you can add its results directly to `$node->content['mailfish']`;

### 14.3.5 Step 4: Check whether or not to display the form

Add a permissions check and a check to see if mailfish is enabled for the current node before adding the form.

Your `hook_node_view()` implementation should look like this:

```
<?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'] = drupal_get_form('mailfish_email_form', $node->nid);
    $node->content['mailfish']['#weight'] = 100;
  }
}
```

### 14.3.6 Challenge Exercise: Add an administrative setting for the signup block location

Since the signup form is already available in a block and on the 'signup' tab, it's overkill to add it to the content of the node as well.

Add an administrative setting to determine where the signup form will appear on the website (you can add this to the existing admin form).

How would you need to change the code we've written to respect this setting?

# TESTING YOUR MODULE: SIMPLETEST

## 15.1 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.

### 15.1.1 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(
        'administer 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 present');

    // 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 .  
}  
}
```

### **Running tests**

1. Disable Mailfish for the “Basic page” content type.
2. Go to *Configuration* → *Development* → *Performance* and clear all caches.
3. Enable the Testing module on admin/modules.
4. Go to *Configuration* → *Development* → *Testing* (admin/config/development/testing).
5. Select the option for “Mailfish”.
6. Click *Run tests*.



# 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 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()  
}
```

### 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)));
?>
```

## 16.1 Using The Coder Module

The Coder module enables you to review the code of other modules, including your own. It is an excellent 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 *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
```

# 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 17.1: 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.  |

## 17.1 Server Requirements

Custom server requirements can be handled with `hook_requirements`: [http://api.drupal.org/api/function/hook\\_requirements/7](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 public-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.

## 17.2 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.

### 17.2.1 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>

## 17.2.2 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 under *Configuration → Development → Performance*

## 17.2.3 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.

1. To do this: Go to *Configuration → Development → Performance*
2. Under Bandwidth optimization:
  - Select Aggregate and compress CSS files.
  - Select Aggregate JavaScript files.
3. Save configuration.

## 17.2.4 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.

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

## 17.2.5 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.

1. To do this: Go to *Configuration → Development → Performance*
2. Under Caching select both:
  - Cache pages for anonymous users
  - Cache blocks.

- 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.
3. Minimum cache lifetime. - Set to not lower than 5 mins. This sets how long a the cached pages should be saved for until they are rendered again.
  4. Expiration of cached pages. Set it to no lower than 5 mins. This setting tells visitors, such as search engines or browsers, how soon to check for a new version how X soon. This reduces visits to your server.
  5. Save configuration.

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>

## 17.3 Tools to improve performance

**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>

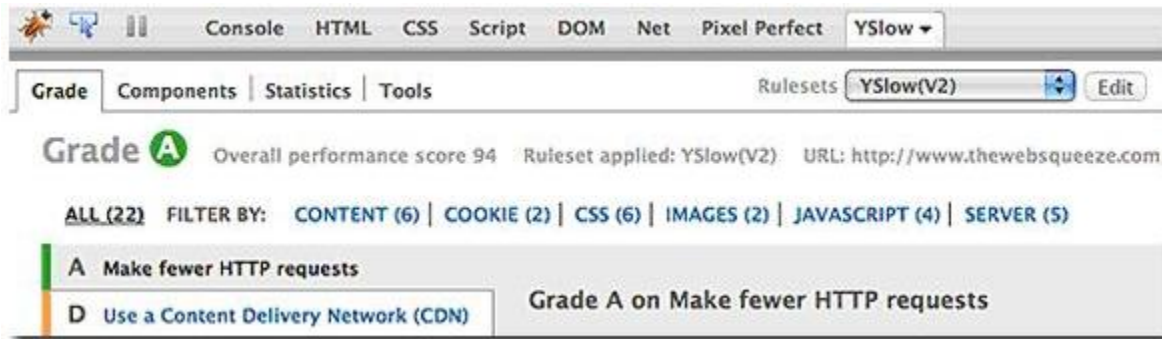
**Boost:** Currently the Drupal 7 version of the Boost module is under development.

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.

## 17.4 Check Front End Performance

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



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](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 suggestions 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/>

## 17.5 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 the table name, 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 duration.

**Slow query highlighting:**

Enter an integer in milliseconds. Any query which takes longer than this many milliseconds will be highlighted in red.

☐ **Store executed queries**

Store statistics about executed queries. See the devel\_x tables.

**Sampling Interval:**

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

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

**API Site:**


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

☒ **Display page timer**

Display page execution time in the query log box.

☒ **Display memory usage**

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

☐ **Display redirection page**

When a module executes `drupal_goto()`, the query log and other developer information is lost. Enabling this will display a page before continuing to the destination page.

☐ **Display form element keys and weights**

Form element names are needed for performing theming or altering a form. Their weights determine the order of the form items.

**Crumo display:**

- ☒ default
- ☐ blue
- ☐ green
- ☐ orange
- ☐ schablon.com
- ☐ disabled

Select a skin for your debug messages or select *disabled* to display object and array output in standard HTML.

☐ **Rebuild the theme registry on every page load**

While 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                  |
| 0.01   | 1 | taxonomy_vocabulary_load  | SELECT v.* 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>