Module developer's guide

Module developer's guide

Table of Contents

Preface ... vi
1. Introduction to Drupal modules ... 1
2. Drupal's menu building mechanism ... 2
3. Drupal's node building mechanism ... 6
    How Drupal handles access ... 9
4. Drupal's page serving mechanism ... 10
5. Creating modules - a tutorial ... 15
    Getting started ... 15
    Letting Drupal know about the new function ... 15
    Telling Drupal about your module ... 16
    Telling Drupal who can use your module ... 17
    Announce we have block content ... 17
    Generate content for a block ... 18
    Installing, enabling and testing the module ... 21
    Create a module configuration (settings) page ... 21
    Adding menu links and creating page content ... 23
    Adding a 'more' link and showing all entries ... 24
    Conclusion ... 24
6. Updating your modules ... 26
    Converting 3.0 modules to 4.0 ... 26
    Converting 4.0 modules to 4.1 ... 26
        Required changes ... 26
        Optional changes ... 27
    Converting 4.1 modules to 4.2 ... 27
    Converting 4.2 modules to 4.3 ... 29
        Creating modules for version 4.3.1 ... 30
        How to build up a _help hook ... 39
        How to convert a _system hook ... 40
        How to convert an _auth_help hook ... 41
    Converting 4.3 modules to 4.4 ... 42
        Menu system ... 42
        Theme system ... 44
        Node system ... 44
        Filter system ... 45
        Hook changes ... 46
        Emitting links ... 47
        Status and error messages ... 47
    Converting 4.4 modules to 4.5 ... 47
        Menu system ... 47
        Path changes ... 48
        Node changes ... 48
        Filtering changes ... 49
        Check_output() changes ... 50
        Filter hook ... 50
        Filter tips ... 51
        Other changes ... 51
    Converting 4.5 modules to 4.6 ... 52
    Converting 4.6 modules to HEAD ... 55
7. Join forces ... 57
8. Reference ... 58
    'Status' field values for nodes and comments ... 58
    Values of 'comment' field in node table ... 58
9. Module how-to's ... 59

How to write a node module 59
How to write database independent code 59
How to write efficient database JOINs 59
How to connect to multiple databases within Drupal 60
How to write themable modules 61

Preface

Developer documentation can be found at http://drupaldocs.org/ and in the remainder of the Drupal developer's guide below.

- drupaldocs.org documents the Drupal APIs [http://drupaldocs.org/api] and presents an overview of Drupal's building blocks [http://drupaldocs.org/api/head] along with handy examples [http://drupaldocs.org/api/head].
- The Drupal developer guide provides guidlines as how to upgrade your modules (API changes) along with development tips/tutorials.

Chapter 1. Introduction to Drupal modules

When developing Drupal it became clear that we wanted to have a system which is as modular as possible. A modular design will provide flexibility, adaptability, and continuity which in turn allows people to customize the site to their needs and likings.

A Drupal module is simply a file containing a set of routines written in PHP. When used, the module code executes entirely within the context of the site. Hence it can use all the functions and access all variables and structures of the main engine. In fact, a module is not any different from a regular PHP file: it is more of a notion that automatically leads to good design principles and a good development model. Modularity better suits the open-source development model, because otherwise you can't easily have people working in parallel without risk of interference.

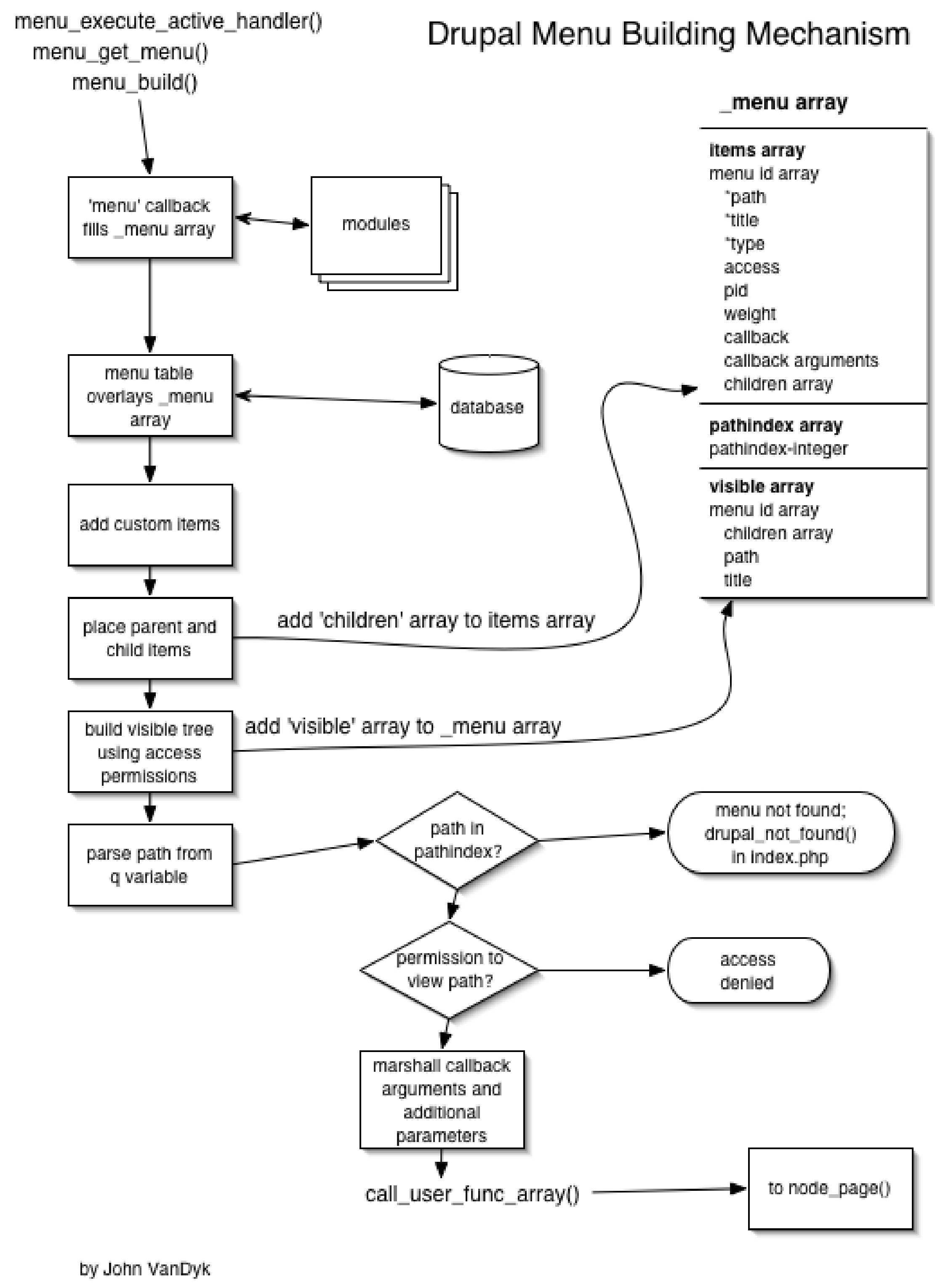
The idea is to be able to run random code at given places in the engine. This random code should then be able to do whatever needed to enhance the functionality. The places where code can be executed are called "hooks" and are defined by a fixed interface.

In places where hooks are made available, the engine calls each module's exported functions. This is done by iterating through the modules directory where all modules must reside. Say your module is named foo (i.e. modules/foo.module) and if there was a hook called bar, the engine will call foo_bar() if this was exported by your module.

See also the overview of module hooks [http://drupaldocs.org/api/head/group/hooks], which is generated from the Drupal source code.

Chapter 2. Drupal's menu building mechanism

(Note: this is an analysis of the menu building mechanism in pre-4.5 CVS as of August 2004. It does not include menu caching.)


menu_execute_active_handler()
menu_get_menu()
menu_build()
Drupal Menu Building Mechanism
_menu array
items array
menu id array
*path
*title
*type
access
pid
weight
callback
callback arguments
children array
pathindex array
pathindex-integer
visible array
menu id array
children array
path
title
'menu' callback fills _menu array
modules
menu table overlays _menu array
database
add custom items
place parent and child items
add 'children' array to items array
build visible tree using access permissions
add 'visible' array to _menu array
parse path from q variable
path in pathindex?
menu not found; drupal_not_found() in index.php
permission to view path?
access denied
marshall callback arguments and additional parameters
call_user_func_array()
to node_page()
by John VanDyk

This continues our examination of how Drupal serves pages. We are looking specifically at how the menu system works and is built, from a technical perspective. See the excellent overview in the menu system documentation [http://drupaldocs.org/api/head/group/menu].

We begin in index.php, where menu_execute_active_handler() has been called. Diving in from menu_execute_active_handler(), we immediately set the $menu variable by calling menu_get_menu(). The latter function declares the global $_menu array (note the underline, it means a 'super global', which is a predefined array [%20http://www.php.net/manual/en/language.variables.predefined.php] in PHP lore) and calls _menu_build() to fill the array, then returns $_menu. Although menu_get_menu() initializes the $_menu array, the _menu_build() function actually reinitializes the $_menu array. Then it sets up two main arrays within $_menu: the items array and the path index array.

The items array is an array keyed to integers. Each entry contains the following fields:

Required fields		
path	string	the partial URL to the page for this menu item
title	string	the title that this menu item will have in the menu
type	integer	a constant denoting the menu item type (see comments in menu.inc)
Optional fields		
access	boolean	
pid	integer	
weight	integer	
callback	string	name of the function to be called if this menu item is selected
callback arguments	array	

The $menu_item_list array is normalized by making sure each array entry has a path, type and weight entry. As each entry is examined, the path index array of the $_menu array is checked to see if the path of this menu item exists. If an equivalent path is already there in the path index array, it is blasted away. The path index of this menu item is then added as a key with the value being the menu id. In the items array of the $_menu array, the menu id is used as the key and the entire array entry is the value.

Note: the $temp_mid and $mid variables seem to do the same thing. Why, syntactically, cannot only one be used?

The path index array contained 76 items when serving out a simple node with only the default modules enabled.

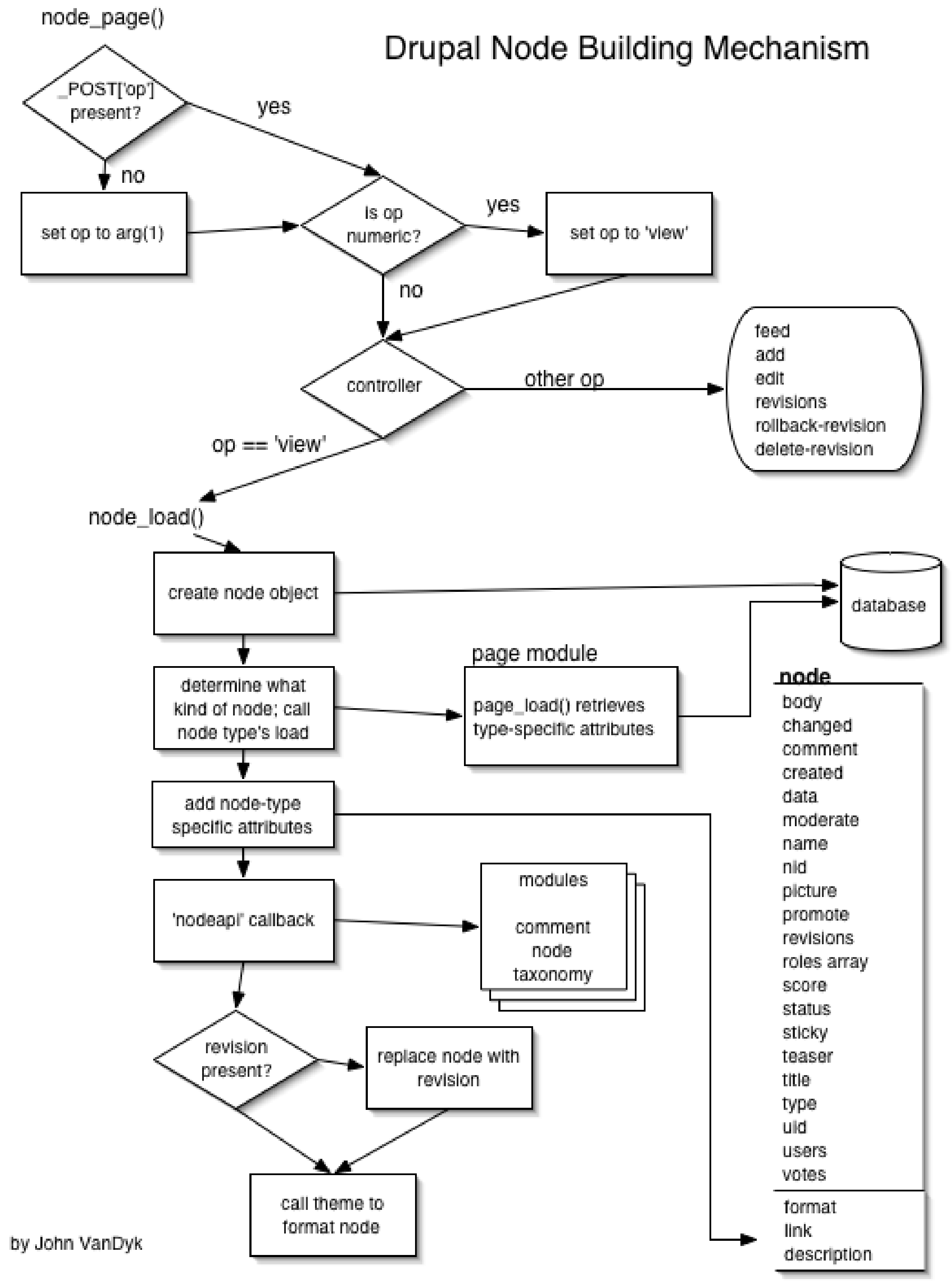
Next the menu table from the database is fetched and its contents are used to move the position of existing menu items from their current menu ids to the menu ids saved in the database. The comments says "reassigning menu IDs as needed." This is probably to detect if the user has customized the menu entries using the menu module. The path index array entries generated from the database can be recognized because their values are strings, whereas up til now the values in the path index array have been integers.

Now I get sort of lost. It looks like the code is looking at paths to determine which menu items are children of other menu items. Then _menu_build_visible_tree is a recursive function that builds a third subarray inside $_menu, to go along with items and path index. It is called visible and takes into account the access attribute and whether or not the item is hidden in order to filter the items array. As an anonymous user, all items but the Navigation menu item are filtered out. See also the comments in menu.inc for menu_get_menu(). In fact, read all the comments in menu.inc!

Now the path is parsed out from the q parameter of the URL. Since node/1 is present in the path index, we successfully found a menu item. It points to menu item -44 in our case, to be precise, but there must be a bug in the Zend IDE because it shows item -44 as null. Anyway, the menu item entry is checked for callback arguments (there are none) and for additional parameters (also none), and execution is passed off to node_page() through the call_user_func_array function.

Chapter 3. Drupal's node building mechanism

(This walkthrough done on pre-4.5 CVS code in August 2004.)



The node_page controller checks for a $_POST['op'] entry and, failing that, sets $op to arg(1) which in this case is the '1' in node/1. A numeric $op is set to arg(2) if arg(2) exists, but in this case it doesn't ('1' is the end of the URL, remember?) so the $op is hardcoded to 'view'. Thus, we succeed in the 'view' case of the switch statement, and are shunted over to node_load(). The function node_load() takes two arguments, $conditions (an array with nid set to desired node id -- other conditions can be defined to further restrict the upcoming database query) for which we use arg(1), and $revision, for which we use

_GET['revision']. The 'revision' key of the _GET array is unset so we need to make brief stop at error_handler because of an undefined index error. That doesn't stop us, though, and we continue pell-mell into node_load using the default $revision of -1 (that is, the current revision). The actual query that ends up being run is

SELECT n.*, u.uid, u.name, u.picture, u.data FROM node n INNER JOIN users u on u.uid WHERE n = '1'

We get back a joined row from the database as an object. The data field from the users table is serialized, so it must be unserialized. This data field contains the user's roles. How does this relate to the user_roles table? Note that the comment "// Unserialize the revisions and user data fields" should be moved up before the call to drupal_unpack().

We now have a complete node that looks like the following:

Attribute	**Value**
body	This is a test node body
changed	1089859653
comment	2
created	1089857673
data	a:1:{s:5... (serialized data)
moderate	0
name	admin
nid	1
picture	''
promote	1
revisions	''
roles	array containing one key-value pair, 0 = '2'
score	0
status	1
sticky	0
teaser	This is a test node body
title	Test
type	page
uid	1
users	''
votes	0

All of the above are strings except the roles array.

So now we have a node loaded from the database. It's time to notify the appropriate module that this has happened. We do this via the node_invoke($node, 'load') call. The module called via this callback may return an array of key-value pairs, which will be added to the node above.

The node_invoke() function asks node_get_module_name() to determine the name of the module that corresponds with the node's type. In this case, the node type is a page, so the page.module is the one we'll call, and the specific name of the function we'll call is page_load(). If the name of the node type has a hyphen in it, the left part is used. E.g., if the node type is page-foo, the page module is used.

The page_load() function turns out to be really simple. It just retrieves the format, link and description columns from the page table. The 'format' column specifies whether we're dealing with a HTML or PHP page. The 'link' and 'description' fields are used to generate a link to the newly created page, however, those will be deprecated with the improved menu system. To that extend, the core themes no longer use this information (unlike some older themes in the contributions repository). We return to node_load(), where the format, link and description key-value pairs are added to the node's definition.

Now it's time to call the node_invoke_nodeapi() function to allow other modules to do their thing. We check each module for a function that begins with the module's name and ends with _nodeapi(). We hit paydirt with the comment module, which has a function called comment_nodeapi(&$node, $op, arg = 0). Note that the node is passed in by reference so that any changes made by the module will be reflected in the actual node object we built. The $op argument is 'load', in this case. However, this doesn't match any of comment_nodeapi()'s symbols in its controller ('settings', 'fields', 'form admin', 'validate' and 'delete' match). So nothing happens.

Our second hit is node_nodeapi(&$node, $op, $arg = 0) in the node.module itself. Again, no symbols are matched in the controller so we just return.

We'll try again with taxonomy_nodeapi(&$node, $op, $arg = 0). Again, no symbols match; the taxonomy module is concerned only with inserts, updates and deletes, not loads.

Note that any of these modules could have done anything to the node if they had wished.

Next, the node is replaced with the appropriate revision of the node, if present as an attribute of $node. It is odd that this occurs here, as all the work that may have been done by modules is summarily blown away if a revision other than the default revision is found.

Finally, back in node_page(), we're ready to get down to business and actually produce some output. This is done with the statement

print theme('page', node_show($node, arg(3)), $node->title);

And what that statement calls is complex enough to again warrant another commentary. (Not yet done.)

How Drupal handles access

I believe this page should explain how user_access table works. 1.- Drupal checks if the user has access to that module, if he does ... 2.- The he checks the user _access page where gid is the role, view should be 1 and realm should be "all". If there is no access given in that table, he will not give the access to the user.

I believe there is not enough documentation on how to use node access, and hopefully this page will have more information as people contribute.

Chapter 4. Drupal's page serving mechanism

This is a commentary on the process Drupal [http://www.drupal.org/] goes through when serving a page. For convenience, we will choose the following URL, which asks Drupal to display the first node for us. (A node is a thing, usually a web page.)

```
http://127.0.0.1/~vandyk/drupal/?q=node/1
```

A visual companion to this narration can be found here [http://www.lo.redjupiter.com/gems/iowa/drupalwalk.png]; you may want to print it out and follow along. Before we start, let's dissect the URL. I'm running on an OS X machine, so the site I'm serving lives at /Users/vandyk/Sites/. The drupal directory contains a checkout of the latest Drupal CVS [http://drupal.org/book/view/320] tree. It looks like this:

```
CHANGELOG.txt
cron.php
CVS/
database/
favicon.ico
includes/
index.php
INSTALL.txt
LICENSE.txt
MAINTAINERS.txt
misc/
modules/
phpinfo.php
scripts/
themes/
tiptoe.txt
update.php
xmlrpc.php
```

So the URL above will be be requesting the root directory `/` of the Drupal site. Apache translates that into `index.php`. One variable/value pair is passed along with the request: the variable 'q' is set to the value 'node/1'.

So, let's pick up the show with the execution of index.php [http://drupaldocs.org/api/head/file/index.php], which looks very simple and is only a few lines long.

Let's take a broad look at what happens during the execution of `index.php`. First, the `includes/bootstrap.inc` file is included, bringing in all the functions that are necessary to get Drupal's machinery up and running. There's a call to `drupal_page_header()`, which starts a timer, sets up caching, and notifies interested modules that the request is beginning. Next, the `includes/common.inc` file is included, giving access to a wide variety of utility functions such as path formatting functions, form generation and validation, etc. The call to `fix_gpc_magic()` is there to check on the status of PHP "magic quotes" and to ensure that all escaped quotes enter Drupal's database consistently. Drupal then builds its navigation menu and sets the variable `$status` to the result of that operation. In the switch statement, Drupal checks for cases in which a Not Found or Access Denied message needs to be generated, and finally a call to `drupal_page_footer()`, which notifies all interested modules that the request is ending. Drupal closes up shop and the page is served. Simple, eh?

Let's delve a little more deeply into the process outlined above.

The first line of `index.php` includes the `includes/bootstrap.inc` file, but it also executes code towards the end of `bootstrap.inc`. First, it destroys any previous variable named `$conf`. Next, it calls `conf_init()`. This function allows Drupal to use site-specific configuration files, if it finds them. The name of the site-specific configuration file is based on the hostname of the server, as reported by PHP. `conf_init` returns the name of the site-specific configuration file; if no site-specific configuration file is found, sets the variable `$config` equal to the string `$confdir/default`. Next, it includes the named configuration file. Thus, in the default case it will include `sites/default/settings.php`. The code in `conf_init()` would be easier to understand if the variable `$file` were instead called `$potential_filename`. Likewise `$conf_filename` would be a better choice than `$config`.

The selected configuration file (normally `/sites/default/settings.php`) is now parsed, setting the `$db_url variable`, the optional `$db_prefix variable`, the `$base_url` for the website, and the `$languages` array (default is `"en"=>"english"`).

The `database.inc` file is now parsed, with the primary goal of initializing a connection to the database. If MySQL is being used, the `database.mysql.inc` files is brought in. Although the global variables `$db_prefix`, `$db_type`, and `$db_url` are set, the most useful result of parsing `database.inc` is a global variable called `$active_db` which contains the database connection handle.

Now that the database connection is set up, it's time to start a session by including the `includes/session.inc` file. Oddly, in this include file the executable code is located at the top of the file instead of the bottom. What the code does is to tell PHP to use Drupal's own session storage functions (located in this file) instead of the default PHP session code. A call to PHP's `session_start()` function thus calls Drupal's `sess_open()` and `sess_read()` functions. The `sess_read()` function creates a global `$user object` and sets the `$user->roles` array appropriately. Since I am running as an anonymous user, the `$user->roles` array contains one entry, `1->"anonymous user"`.

We have a database connection, a session has been set up...now it's time to get things set up for modules. The `includes/module.inc` file is included but no actual code is executed.

The last thing `bootstrap.inc` does is to set up the global variable `$conf`, an array of configuration options. It does this by calling the `variable_init()` function. If a per-site configuration file exists and has already populated the `$conf` variable, this populated array is passed in to `variable_init()`. Otherwise, the `$conf` variable is null and an empty array is passed in. In both cases, a populated array of name-value pairs is returned and assigned to the global `$conf` variable, where it will live for the duration of this request. It should be noted that name-value pairs in the per-site configuration file have precedence over name-value pairs retrieved from the "variable" table by `variable_init()`.

We're done with `bootstrap.inc`! Now it's time to go back to `index.php` and call `drupal_page_header()`. This function has two responsibilities. First, it starts a timer if `$conf['dev_timer']` is set; that is, if you are keeping track of page execution times. Second, if caching has been enabled it retrieves the cached page, calls `module_invoke_all()` for the 'init' and 'exit' hooks, and exits. If caching is not enabled or the page is not being served to an anonymous user (or several other special cases, like when feedback needs to be sent to a user), it simply exits and returns control to `index.php`.

Back at `index.php`, we find an include statement for `common.inc`. This file is chock-full of miscellaneous utility goodness, all kept in one file for performance reasons. But in addition to putting all these utility functions into our namespace, `common.inc` includes some files on its own. They include `theme.inc`, for theme support; `pager.inc` for paging through large datasets (it has nothing to do with calling your pager); and `menu.inc`. In `menu.inc`, many constants are defined that are used later by the menu system.

The next inclusion that `common.inc` makes is `xmlrpc.inc`, with all sorts of functions for dealing with XML-RPC calls. Although one would expect a quick check of whether or not this request is actually an XML-RPC call, no such check is done here. Instead, over 30 variable assignments are made, apparently so that if this request turns to actually be an XML-RPC call, they will be ready. An `xmlrpc_init()` function instead may help performance here?

A small `tablesort.inc` file is included as well, containing functions that help behind the scenes with sortable tables. Given the paucity of code here, a performance boost could be gained by moving these into `common.inc` itself.

The last include done by `common.inc` is `file.inc`, which contains common file handling functions. The constants `FILE_DOWNLOADS_PUBLIC = 1` and `FILE_DOWNLOADS_PRIVATE = 2` are set here, as well as the `FILE_SEPARATOR`, which is `\\` for Windows machines and `/` for all others.

Finally, with includes finished, common.inc sets PHP's error handler to the `error_handler()` function in the `common.inc` file. This error handler creates a watchdog entry to record the error and, if any error reporting is enabled via the `error_reporting` directive in PHP's configuration file `(php.ini<code>), it prints the error message to the screen. Drupal's <code>error_handler()` does not use the last parameter `$variables`, which is an array that points to the active symbol table at the point the error occurred. The comment "`// set error handler:`" at the end of common.inc is redundant, as it is readily apparent what the function call to `set_error_handler()` does.

The `Content-Type` header is now sent to the browser as a hard coded string: "`Content-Type: text/html; charset=utf-8`".

If you remember that the URL we are serving ends with `/~vandyk/drupal/?q=node/1`, you'll note that the variable `q` has been set. Drupal now parses this out and checks for any path aliasing for the value of `q`. If the value of `q` is a path alias, Drupal replaces the value of `q` with the actual path that the value of `q` is aliased to. This sleight-of-hand happens before any modules see the value of `q`. Cool.

Module initialization now happens via the `module_init()<code> function. This function runs <code>require_once()<code> on the <code>admin`, `filter`, `system`, `user` and `watchdog` modules. The filter module defines `FILTER_HTML*` and `FILTER_STYLE*` constants while being included. Next, other modules are `include_once`'d via `module_list()`. In order to be loaded, a module must (1) be enabled (that is, the status column of the "system" database table must be set to 1), and (2) Drupal's throttle mechanism must determine whether or not the module is eligible for exclusion when load is high. First, it determines whether the module is eligible by looking at the throttle column of the "system" database table; then, if the module is eligible, it looks at `$conf["throttle_level"]` to see whether the load is high enough to exclude the module. Once all modules have been `include_once`'d and their names added to the `$list` local array, the array is sorted by module name and returned. The returned `$list` is discarded because the `module_list()` invocation is not part of an assignment (e.g., it is simply `module_list()` and not `$module_list = module_list()`). The strategy here is to keep the module list inside a static variable called `$list` inside the `module_list()` function. The next time `module_list()` is called, it will simply return its static variable `$list` rather than rebuilding the whole array. We see that as we follow the final objective of `module_init()`; that is, to send all modules the "init" callback.

To see how the callbacks work let's step through the init callback for the first module. First `module_invoke_all()` is called and passed the string enumerating which callback is to be called. This string could be anything; it is simply a symbol that call modules have agreed to abide by, by convention. In this case it is the string "init".

The `module_invoke_all()` function now steps through the list of modules it got from calling `module_list()`. The first one is "`admin`", so it calls `module_invoke("admin","init")`. The `module_invoke()` function simply puts the two together to get the name of the function it will call. In this case the name of the function to call is "`admin_init()`". If a function by this name exists, the function is called and the returned result, if any, ends up in an array called `$return` which is returned after all modules have been invoked. The lesson learned here is that if you are writing a module and intend to return a value from a callback, you must return it as an array. [Jonathan Chaffer: *Each "hook" (our word for what you call a callback) defines its own return type. See the full list of hooks available to module developers [http://drupaldocs.org/api/head/group/hooks], with documentation* *about what they are expected to return.*]

Back to `common.inc`. There is a check for suspicious input data. To find out whether or not the user has permission to bypass this check, `user_access()` is called. This retrieves the user's permissions and stashes them in a static variable called `$perm`. Whether or not a user has permission for a given action is determined by a simple substring search for the name of the permission (e.g., "bypass input data check") within the `$perm` string. Our `$perm` string, as an anonymous user, is currently "0access content, ". Why the 0 at the beginning of the string? Because `$perm` is initialized to 0 by `user_access()`.

The actual check for suspicious input data is carried out by `valid_input_data()` which lives in `common.inc`. It simply goes through an array it's been handed (in this case the `$_REQUEST` array) and checks all keys and values for the following "evil" strings: javascript, expression, alert, dynsrc, datasrc, data, lowsrc, applet, script, object, style, embed, form, blink, meta, html, frame, iframe, layer, ilayer, head, frameset, xml. If any of these are matched watchdog records a warning and Drupal dies (in the PHP sense). I wondered why both the keys and values of the `$_REQUEST` array are examined. This seems very time-consuming. Also, would it die if my URL ended with "`/?xml=true`" or "`/?format=xml`"?

The next step in `common.inc`'s executable code is a call to `locale_init()` to set up locale data. If the user is not an anonymous user and has a language preference set up, the two-character language key is returned; otherwise, the key of the single-entry global array `$language` is returned. In our case, that's "en".

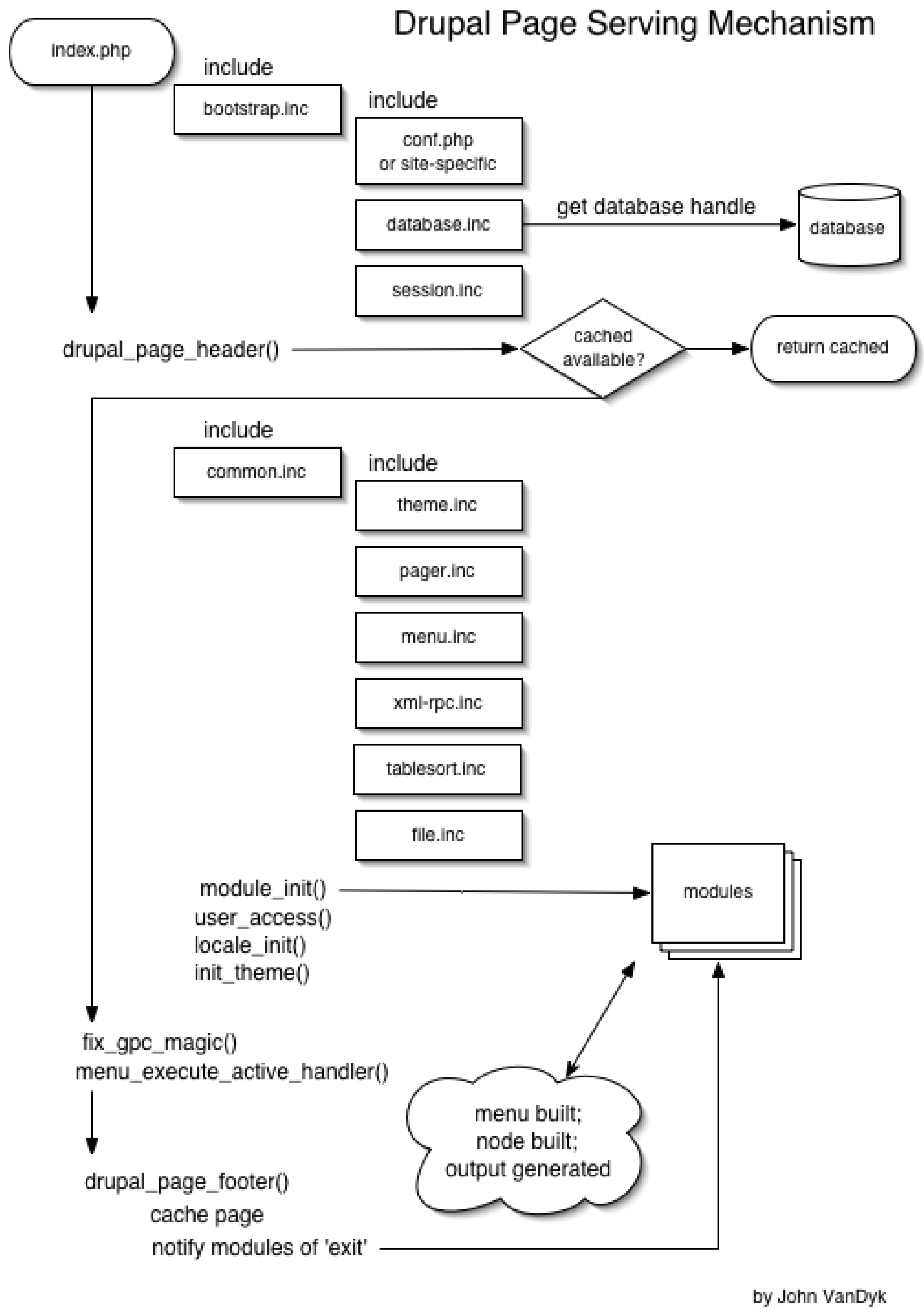
The last gasp of `common.inc` is to call `init_theme()`. You'd think that for consistency this would be called `theme_init()` (of course, that would be a namespace clash with a callback of the same name). This finds out which themes are available, which the user has selected, and then `include_once`'s the chosen theme. If the user's selected theme is not available, the value at `$conf["theme_default"]` is used. In our case, we are an anonymous user with no theme selected, so the default xtemplate theme is used. Thus, the file `themes/xtemplate/xtemplate.theme` is `include_once`'d. The inclusion of `xtemplate.theme` calls `include_once("themes/xtemplate/xtemplate.inc")`, and creates a new object called xtemplate as a global variable. Inside this object is an xtemplate object called "`template`" with lots of attributes. Then there is a nonfunctional line where `SetNullBlock` is called. A comment indicates that someone is aware that this doesn't work.

Now we're back to `index.php`! A call to `fix_gpc_magic()` is in order. The "gpc" stands for Get, Post, Cookie: the three places that unescaped quotes may be found. If deemed necessary by the status of the boolean `magic_quotes_gpc` directive in PHP's configuration file (`php.ini`), slashes will be stripped from `$_GET`, `$_POST`, `$_COOKIE`, and `$_REQUEST` arrays. It seems odd that the function is not called `fix_gpc_magic_quotes`, since it is the "magic quotes" that are being fixed, not the magic. In my distribution of PHP, the `magic_quotes_gpc` directive is set to "Off", so slashes do not need to be stripped.

The next step is to set up menus. This step is crucial. The menu system doesn't just handle displaying menus to the user, but also determines what function will be handed the responsibility of displaying the page. The "`q`" variable (we usually call the Drupal path) is matched against the available menu items to find the appropriate callback to use. Much more information on this topic is available in the menu system documentation for developers [http://drupaldocs.org/api/head/group/menu]. We jump to `menu_execute_active_handler()` in `menu.inc`. This sets up a `$_menu` array consisting of items, local tasks, path index, and visible arrays. Then the system realizes that we're not going to be building any menus for an anonymous user and bows out. The real meat of the node creation and formatting happens here, but is complex enough for a separate commentary. Back in `index.php`, the switch statement doesn't match either case and we approach the last call in the file, to `drupal_page_footer` in `common.inc`. This takes care of caching the page we've built if caching is enabled (it's not) and calls `module_invoke_all()` with the "exit" callback symbol.

Although you may think we're done, PHP's session handler still needs to tidy up. It calls `sess_write()` in `session.inc` to update the session database table, then `sess_close()` which simply returns 1.

We're done.



Chapter 5. Creating modules - a tutorial

This tutorial describes how to create a module for Drupal 4.5.*. It is an update to the tutorial for ???. Please see comments there, also.

A module is a collection of functions that link into Drupal, providing additional functionality to your Drupal installation. After reading this tutorial, you will be able to create a basic block module and use it as a template for more advanced modules and node modules.

This tutorial will not necessarily prepare you to write modules for release into the wild. It does not cover caching, nor does it elaborate on permissions or security issues. Use this tutorial as a starting point, and review other modules and the ??? and ???for more information.

This tutorial assumes the following about you:

- Basic PHP knowledge, including syntax and the concept of PHP objects
- Basic understanding of database tables, fields, records and SQL statements
- A working Drupal installation
- Drupal administration access
- Webserver access

This tutorial does not assume you have any knowledge about the inner workings of a Drupal module. This tutorial will not help you write modules for versions of Drupal earlier than 4.5.

Getting started

To focus this tutorial, we'll start by creating a block module that lists links to content such as blog entries or forum discussions that were created one week ago. The full tutorial will teach us how to create block content, write links, and retrieve information from Drupal nodes.

Start your module by creating a PHP file and save it as 'onthisdate.module' in the modules directory of your Drupal installation.

```
<?php
?>
```

As per the ???, use the longhand <?php tag, and not <? to enclose your PHP code.

All functions in your module are named {modulename}_{hook}, where "hook" is a well defined function name. Drupal will call these functions to get specific data, so having these well defined names means Drupal knows where to look.

Letting Drupal know about the new function

As mentioned above, the function we just wrote isn't a 'hook': it's not a Drupal recognized name. We need to tell Drupal how to access the function when displaying a page. We do this with the menu() hook. The menu() hook defines the association between a URL and the function that creates the content for that url. The hook also does permission checking, if desired.

```
<?phpfunctiononthisdate_menu() {
  $items=
array();
  $items[]
= array('path'=>'onthisdate',

'title'=>t('on this
date'),

'callback'=>'_onthisdate_all',

'access'=>user_access('access content'),

'type'=>MENU_CALLBACK);
  return$items;
}?>
```

Basically, we're saying if the user goes to "onthisdate" (either via ?q=onthisdate or http://.../onthisdate), the content generated by onthisdate_all will be displayed. The title of the page will be "on this date". The type MENU_CALLBACK Drupal to not display the link in the user's menu, just use this function when the URL is accessed. Use MENU_LOCAL_TASK if you want the user to see the link in the side navigation block.

Navigate to /onthisdate (or ?q=onthisdate) and see what you get.

Telling Drupal about your module

The first function we'll write will tell Drupal information about your module: its name and description. The hook name for this function is 'help', so start with the onthisdate_help function:

```
<?phpfunctiononthisdate_help($section='')
{
}?>
```

The $section variable provides context for the help: where in Drupal or the module are we looking for help. The recommended way to process this variable is with a switch statement. You'll see this code pattern in other modules.

```
<?php/**
* Display help and module information
* @param section which section of the site we're displaying help
* @return help text for section
*/functiononthisdate_help($section='') {
  $output='';
  switch ($section) {
    case"admin/modules#description":
      $output=t("Displays links to nodes created on
this date");
      break;
  }
  return$output;
}// function
onthisdate_help?>
```

You will eventually want to add other cases to this switch statement to provide real help messages to the user. In particular, output for "admin/help#onthisdate" will display on the main help page accessed by the admin/help URL for this module (/admin/help or ?q=admin/help).

Telling Drupal who can use your module

The next function to write is the permissions function. The permissions function doesn't grant permission, it just specifies what permissions are available for this module. Access based on these permissions is defined later in the {module}_access function below. At this point, we'll give permission to anyone who can access site content or administrate the module:

```
<?php/**
* Valid permissions for this module
* @return array An array of valid permissions for the onthisdate module
*/functiononthisdate_perm() {
  return array('access
content');
}// function
onthisdate_perm()?>
```

Conversely, if you are going to write a module that needs to have finer control over the permissions, and you're going to do permission control, you should expand this permission set. You can do this by adding strings to the array that is returned. For example:

```
<?phpfunctiononthisdate_perm() {
return array('access content','access onthisdate','administer
onthisdate');
}// function
onthisdate_perm?>
```

For this tutorial, start with the first one. We'll later move to the second version.

You'll need to adjust who has permission to view your module on the administer » accounts » permissions page. We'll use the user_access function to check access permissions later (whoa, so many "laters!").

Your permission strings must be unique within your module. If they are not, the permissions page will list the same permission multiple times. They should also contain your module name, to avoid name space conflicts with other modules.

Announce we have block content

There are several types of modules: block modules and node modules are two. Block modules create abbreviated content that is typically (but not always, and not required to be) displayed along the left or right side of a page. Node modules generate full page content (such as blog, forum, or book pages).

We'll create a block content to start, and later discuss node content. A module can generate content for blocks and also for a full page (the blogs module is a good example of this). The hook for a block module is appropriately called "block", so let's start our next function:

```
<?php/**
* Generate HTML for the onthisdate block
* @param op the operation from the URL
* @param delta offset
* @returns block HTML
*/functiononthisdate_block($op='list',$delta=0) {
}// end function
onthisdate_block?>
```

The block function takes two parameters: the operation and the offset, or delta. We'll just worry about the operation at this point. In particular, we care about the specific case where the block is being listed in the blocks page. In all other situations, we'll display the block content.

When the module will be listed on the blocks page, the $op parameter's value will be 'list':

```
<?php/**
* Generate HTML for the onthisdate block
* @param op the operation from the URL
* @param delta offset
* @returns block HTML
*/functiononthisdate_block($op='list',$delta=0) {
  // listing of blocks, such as on the
admin/block page
  if ($op=="list") {
     $block[0]["info"] =t('On This
Date');
     return$block;
  } else {
     // our block
content
  }
}// end onthisdate_block?>
```

Generate content for a block

Now, we need to generate the 'onthisdate' content for the block. Here we'll demonstrate a basic way to access the database.

Our goal is to get a list of content (stored as "nodes" in the database) created a week ago. Specifically, we want the content created between midnight and 11:59pm on the day one week ago. When a node is first created, the time of creation is stored in the database. We'll use this database field to find our data.

First, we need to calculate the time (in seconds since epoch start, see ht-tp://www.php.net/manual/en/function.time.php for more information on time format) for midnight a week ago, and 11:59pm a week ago. This part of the code is Drupal independent, see the PHP website (http://php.net/) for more details.

```
<?php/**
* Generate HTML for the onthisdate block
* @param op the operation from the URL
* @param delta offset
* @returns block HTML
*/functiononthisdate_block($op='list',$delta=0) {
  // listing of blocks, such as on the
admin/block page
  if ($op=="list") {
    $block[0]["info"]
=t('On This Date');
    return$block;
  } else {
    // our block content
    // Get today's date
    $today=getdate();
    // calculate midnight one
week ago
    $start_time=mktime(0,0,0,

$today['mon'], ($today['mday'] -7),$today['year']);
    // we want items that
occur only on the day in question, so
    // calculate 1 day
    $end_time=$start_time+86400;
    // 60 * 60 * 24 = 86400
seconds in a day
    ...
  }
}?>
```

The next step is the SQL statement that will retrieve the content we'd like to display from the database. We're selecting content from the node table, which is the central table for Drupal content. We'll get all sorts of content type with this query: blog entries, forum posts, etc. For this tutorial, this is okay. For a real module, you would adjust the SQL statement to select specific types of content (by adding the 'type' column and a WHERE clause checking the 'type' column).

Note: the table name is enclosed in curly braces: `{node}`. This is necessary so that your module will support database table name prefixes. You can find more information on the Drupal website by reading the ??? page in the Drupal handbook.

```
<?php
$query="SELECT nid,
title, created FROM ".
         "{node} WHERE created >= '".$start_time.
         "' AND created <= '".$end_time."'";?>
```

Drupal uses database helper functions to perform database queries. This means that, for the most part, you can write your database SQL statement and not worry about the backend connections.

We'll use db_query() to get the records (i.e. the database rows) that match our SQL query, and db_fetch_object() to look at the individual records:

```
<?php
  // get the links
  $queryResult=  db_query($query);
  // content variable that will be
returned for display
  $block_content='';
  while ($links=db_fetch_object($queryResult)) {
    $block_content.='<a href="'.url('node/'.$links->nid)
.'">'.

$links->title.'</a><br />';
  }
  // check to see if there was any
content before setting up
  //  the block
  if ($block_content=='') {
    /* No content from a week
ago.  If we return nothing, the block
     * doesn't show, which is what we want. */
    return;
  }
  // set up the block
  $block['subject']
='On This Date';
  $block['content']
=$block_content;
  return$block;
}?>
```

Notice the actual URL is enclosed in the l() function. `l` generates <a href="link"> links, adjust the URL to the installation's URL configuration of either clean URLS: ??? or ???

Also, we return an array that has 'subject' and 'content' elements. This is what Drupal expects from a block function. If you do not include both of these, the block will not render properly.

You may also notice the bad coding practice of combining content with layout. If you are writing a module for others to use, you will want to provide an easy way for others (in particular, non-programmers) to adjust the content's layout. An easy way to do this is to include a class attribute in your link, or surround the HTML with a <div> tag with a module specific CSS class and not necessarily include the <br /> at the end of the link. Let's ignore this for now, but be aware of this issue when writing modules that others will use.

Putting it all together, our block function at this point looks like this:

```
<?phpfunctiononthisdate_block($op='list',$delta=0) {
  // listing of blocks, such as on the
admin/block page
  if ($op=="list") {
    $block[0]["info"]
=t("On This Date");
    return$block;
  } else {
  // our block content
    // content variable that will be returned for
display
    $block_content='';
    // Get today's date
    $today=getdate();
    // calculate midnight one
week ago
    $start_time=mktime(0,0,0,$today['mon'],
                                (
$today['mday'] -7),$today['year']);
    // we want items that
occur only on the day in question, so
    //calculate 1 day
    $end_time=$start_time+86400;
    // 60 * 60 * 24 = 86400
seconds in a day
    $query="SELECT nid, title, created FROM
".

"{node} WHERE created >= '".$start_time.

"' AND created <= '".$end_time."'";
    // get the links
    $queryResult=  db_query($query);
    while ($links=db_fetch_object($queryResult)) {
      $block_content.='<a href="'.url('node/'.$links->nid).'">'.

$links->title.'</a><br />';
    }
    // check to see if there
was any content before setting up the block
    if ($block_content=='') {
      // no content
from a week ago, return nothing.
      return;
    }
    // set up the block
    $block['subject']
='On This Date';
    $block['content']
=$block_content;
```

```
    return$block;
  }
}?>
```

Installing, enabling and testing the module

At this point, you can install your module and it'll work. Let's do that, and see where we need to improve the module.

To install the module, you'll need to copy your onthisdate.module file to the modules directory of your Drupal installation. The file must be installed in this directory or a subdirectory of the modules directory, and must have the .module name extension.

Log in as your site administrator, and navigate to the modules administration page to get an alphabetical list of modules. In the menus: administer » modules, or via URL:

```
http://.../admin/modules
```

or

```
http://.../?q=admin/modules
```

When you scroll down, you'll see the onthisdate module listed with the description next to it.

Enable the module by selecting the checkbox and save your configuration.

Because the module is a blocks module, we'll need to also enable it in the blocks administration menu and specify a location for it to display. Node modules may or may not need further configuration depending on the module. Any module can have settings, which affect the functionality/display of a module. We'll discuss settings later. For now, navigate to the blocks administration page: `admin/block` or administer » blocks in the menus.

Enable the module by selecting the enabled checkbox for the 'On This Date' block and save your blocks. Be sure to adjust the location (left/right) if you are using a theme that limits where blocks are displayed.

Now, head to another page, say, select the modules menu. In some themes, the blocks are displayed after the page has rendered the content, and you won't see the change until you go to new page.

If you have content that was created a week ago, the block will display with links to the content. If you don't have content, you'll need to fake some data. You can do this by creating a blog, forum topic or book page, and adjust the "Authored on:" date to be a week ago.

Alternately, if your site has been around for a while, you may have a lot of content created on the day one week ago, and you'll see a large number of links in the block.

Create a module configuration (settings) page

Now that we have a working module, we'd like to make it better. If we have a site that has been around for a while, content from a week ago might not be as interesting as content from a year ago. Similarly, if we have a busy site, we might not want to display all the links to content created last week. So, let's create a configuration page for the administrator to adjust this information.

A module's configuration is set up with the 'settings' hook. We would like only administrators to be able to access this page, so we'll do our first permissions check of the module here:

```
<?php/**
* Module configuration settings
* @return settings HTML or deny access
*/functiononthisdate_settings() {
  // only administrators can access this
module
  if (!user_access("admin onthisdate")) {
    returnmessage_access();
  }
}?>
```

If you want to tie your modules permissions to the permissions of another module, you can use that module's permission string. The "access content" permission is a good one to check if the user can view the content on your site:

```
<?php
  ...
  // check the user has content
access
  if (!user_access("access content")) {
    returnmessage_access();
  }
  ...?>
```

We'd like to configure how many links display in the block, so we'll create a form for the administrator to set the number of links:

```
<?phpfunctiononthisdate_settings()
{
  // only administrators can access this
module
  if (!user_access("admin onthisdate")) {
    returnmessage_access();
  }
  $output.=form_textfield(t("Maximum number of
links"),"onthisdate_maxdisp",

variable_get("onthisdate_maxdisp","3"),2,2,

t("The maximum number
of links to display in the block."));  return$output;}?>
```

This function uses several powerful Drupal form handling features. We don't need to worry about creating an HTML text field or the form, as Drupal will do so for us. We use `variable_get` to retrieve the value of the system configuration variable "onthisdate_maxdisp", which has a default value of 3. We use the form_textfield function to create the form and a text box of size 2, accepting a maximum length of 2 characters. We also use the translate function of t(). There are other form functions that will automatically create the HTML form elements for use. For now, we'll just use the form_textfield function.

Of course, we'll need to use the configuration value in our SQL SELECT, so we'll need to adjust our query statement in the onthisdate_block function:

```
<?php
  $limitnum=variable_get("onthisdate_maxdisp",3);
  $query="SELECT nid, title, created FROM
".

"{node} WHERE created >= '".$start_time.
```

```
"' AND created <= '".$end_time."' LIMIT ".$limitnum;?>
```

You can test the settings page by editing the number of links displayed and noticing the block content adjusts accordingly.

Navigate to the settings page: admin/modules/onthisdate or administer » configuration » modules » onthisdate. Adjust the number of links and save the configuration. Notice the number of links in the block adjusts accordingly.

Note: We don't have any validation with this input. If you enter "c" in the maximum number of links, you'll break the block.

Adding menu links and creating page content

So far we have our working block and a settings page. The block displays a maximum number of links. However, there may be more links than the maximum we show. So, let's create a page that lists all the content that was created a week ago.

```
<?phpfunctiononthisdate_all()
{}?>
```

We're going to use much of the code from the block function. We'll write this ExtremeProgramming style, and duplicate the code. If we need to use it in a third place, we'll refactor it into a separate function. For now, copy the code to the new function _onthisdate_all(). Contrary to all our other functions, 'all', in this case, is not a Drupal hook. In our code, we can prefix this function with an underscore to help us remember this isn't a hook call. We'll discuss below.

```
<?phpfunction_onthisdate_all() {
  // content variable that will be
returned for display
  $page_content='';
  // Get today's date
  $today=getdate();
  // calculate midnight one week
ago
  $start_time=mktime(0,0,0,

$today['mon'], ($today['mday'] -7),$today['year']);
  // we want items that occur only on
the day in question,
  // so calculate 1 day
  $end_time=$start_time+86400;
  // 60 * 60 * 24 = 86400 seconds in a
day
  // NOTE!  No LIMIT clause here!  We want to
show all the code
  $query="SELECT nid, title, created FROM
".

"{node} WHERE created >= '".$start_time.

"' AND created <= '".$end_time."'";
  // get the links
  $queryResult=  db_query($query);
  while ($links=db_fetch_object($queryResult)) {
    $page_content.='<a
href="'.url('node/'.$links->nid).'">'.

$links->title.'</a><br />';
```

```
  }
  ...
}?>
```

We have the page content at this point, but we want to do a little more with it than just return it. When creating pages, we need to send the page content to the theme for proper rendering. We use this with the theme() function. Themes control the look of a site. As noted above, we're including layout in the code. This is bad, and should be avoided. It is, however, the topic of another tutorial, so for now, we'll include the formatting in our content:

```
<?phpprinttheme("page",$content_string);?>
```

The rest of our function checks to see if there is content and lets the user know. This is preferable to showing an empty or blank page, which may confuse the user.

Note that we are responsible for outputting the page content with the 'print theme()' syntax.

```
<?phpfunction_onthisdate_all() {
  ...
  // check to see if there was any
content before
  // setting up the block
  if ($page_content=='') {
    // no content from a week
ago, let the user know
    printtheme("page",

"No events occurred on this site on this date in history.");
    return;
  }
  printtheme("page",$page_content);
}?>
```

Adding a 'more' link and showing all entries

Because we have our function that creates a page with all the content created a week ago, we can link to it from the block with a "more" link.

Add these lines just before that $block['subject'] line, adding this to the $block_content variable before saving it to the $block['content'] variable:

```
<?php// add a more link to our
page that displays all the links
  $block_content.=
         "<div class=\"more-link\">".
         l(t("more"),"onthisdate", array("title"=>t("More
events on this day.")))
         ."</div>";?>
```

This will add the more link.

Conclusion

We now have a working module. It created a block and a page. You should now have enough to get started writing your own modules. We recommend you start with a block module of your own and move onto a node module. Alternately, you can write a filter or theme.

As is, this tutorial's module isn't very useful. However, with a few enhancements, it can be entertaining. Try modifying the select query statement to select only nodes of type 'blog' and see what you get. Alternately, you could get only a particular user's content for a specific week. Instead of using the block function, consider expanding the menu and page functions, adding menus to specific entries or dates, or using the menu callback arguments to adjust what year you look at the content from.

If you start writing modules for others to use, you'll want to provide more details in your code. Comments in the code are incredibly valuable for other developers and users in understanding what's going on in your module. You'll also want to expand the help function, providing better help for the user. Follow the Drupal [Coding standards], especially if you're going to add your module to the project.

Two topics very important in module development are writing themeable pages and writing translatable content. Please check the ??? for more details on these two subjects.

Chapter 6. Updating your modules

As Drupal develops with each release it becomes necessary to update modules to take advantage of new features and stay functional with Drupal's API.

Converting 3.0 modules to 4.0

Converting modules from version 3.0 to version 4.0 standards requires rewriting the `form()` function, as follows:

Drupal 3.0:

```
function form($action, $form, $method = "post", $options = 0)

// Example

global $REQUEST_URI;
$form = form_hidden("nid", $nid);
print form($REQUEST_URI, $form);
```

Drupal 4.0:

```
function form($form, $method = "post", $action = 0, $options = 0)

// Example

$form = form_hidden("nid", $nid);
print form($form);
```

Converting 4.0 modules to 4.1

Drupal 4.1 changed the block hook function and taxonomy API. To convert a version 4.0 module to 4.1, the following changes must be made. First, the `*_block()` function must be re-written. Next, calls to `taxonomy_get_tree()` must be re-written to supply the parameters required by the new function. Finally, you may wish to take advantage of new functions added to the taxonomy API.

Required changes

Modified block hook: **Drupal 4.0:**

```
function *_block() {
  $blocks[0]["info"] = "First block info";
  $blocks[0]["subject"] = "First block subject";
  $blocks[0]["content"] = "First block content";

  $blocks[1]["info"] = "Second block info";
  $blocks[1]["subject"] = "Second block subject";
  $blocks[1]["content"] = "Second block content";

  // return array of blocks
  return $blocks;
  }
}
```

Drupal 4.1:

```
function *_block($op = "list", $delta = 0) {
  if ($op == "list") {
    $blocks[0]["info"] = "First block info";
    $blocks[1]["info"] = "Second block info";
    return $blocks; // return array of block infos
  }
  else {
    switch($delta) {
      case 0:
        $block["subject"] = "First block subject";
        $block["content"] = "First block content";
        return $block;
      case 1:
        $block["subject"] = "Second block subject";
        $block["content"] = "Second block content";
        return $block;
    }
  }
}
```

Modified taxonomy API:

Changes: in function taxonomy_get_tree()

- there is no longer a "parent" property; rather "parents" is an array
- the result tree is now returned instead of being passed by reference

Drupal 4.0:

```
function taxonomy_get_tree($vocabulary_id, &$tree, $parent = 0,
```

Drupal 4.1:

```
$tree = taxonomy_get_tree($vocabulary_id, $parents = 0, $depth =
```

Optional changes

- Take advantage of new taxonomy functions `taxonomy_get_vocabulary_by_name($name)` and `taxonomy_get_term_by_name($name)`
- Take advantage of pager functions
- Move hardcoded markup from modules to themes, using theme_invoke [http://drupal.org/node/view/608]

Converting 4.1 modules to 4.2

Some points posted by Axel [http://lists.drupal.org/pipermail/drupal-devel/2003-February/022183.html] on drupal-devel [http://drupal.org/node.php?id=322] on migrating 4.1.0 modules to CVS [updated and added to by ax]:

- the big "clean URL [http://lists.drupal.org/pipermail/drupal-devel/2003-January/020973.html]" patch: Over the weekend, [dries] bit the bullet and converted every single URL in Drupal's code. meaning we'll [can] have clean URLs like http://foo.com/archive/2003/01/06, http://foo.com/user/42, http://foo.com/blog, and so on.. meaning, for the code:
  - `drupal_url(array("mod" => "search", "op" => "bla"), "module"[, $anchor = ""])` became`url("search/bla")`, with the first url part being the module, the second (typically) being the operation ($op); more arguments are handled differently per module convention.
  - `l("view node", array("op" => "view", "id" => $nid), "node"[, $anchor = "", $attributes = array()])` became`l("view node", "node/view/$nid"[,$attributes = array(), $query = NULL])`
  - similar,`lm()`, which meant "module link" and used to be `module.php?mod=bla&op=blub...`, is now `l("title", "bla/blub/...")`; and`la()`, which meant "admin link" and used to be `admin.php?mod=bla&op=blub...`, is now `l("title", "admin/bla/blub/..."`
  - After fixing those functions, you'll need to edit your _page() function and possibly others so that they get their arguments using the arg() function (see includes/common.inc. These arguments used to be globals called "mod", "op", "id" etc. now these same arguments must be accessed as arg(1), arg(3), for example.
- `$theme->function()` became `theme("function")`. see [drupal-devel] renaming 2 functions [http://lists.drupal.org/pipermail/drupal-devel/2003-February/021824.html], [drupal-devel] theme("function") vs $theme->function() [http://lists.drupal.org/pipermail/drupal-devel/2003-February/021927.html] and [drupal-devel] [CVS] theme() [http://lists.drupal.org/pipermail/drupal-devel/2003-February/021936.html]
- `&lt;module&gt;_conf_options()` became `&lt;module&gt;_settings()` - see [drupal-devel] renaming 2 functions [http://lists.drupal.org/pipermail/drupal-devel/2003-February/021824.html]. note that doesn't get an extra menu entry, but is accessed via "site configuration > modules > modules settings"
- the administration pages got changed quite a lot to use a "database driven link system" and become more logical/intuitive - see [drupal-devel] X-mas commit: administration pages [http://lists.drupal.org/pipermail/drupal-devel/2002-December/020726.html]. this first try resulted in poor performance and a not-so-good api, so it got refactored - see [PATCH] menus [http://lists.drupal.org/pipermail/drupal-devel/2003-February/022134.html]. this, as of time ax is writing this, isn't really satisfying, neither (you cannot build arbitrary menu-trees, some forms don't work (taxonomy > add term), ...), so it probably will change again. and i won't write more about this here.

  well, this: you use `menu()` to add entries to the admin menu. `menu("admin/node/nodes/0", "new or updated posts", "node_admin", "help", 0);` adds a menu entry "new or updated posts" 1 level below "post overview" (admin/node/nodes) and 2 level below "node management" (admin/node) (ie. at the 3. level), with a weight of 0 in the 3. level, with a line "help" below the main heading. for the callback ("node_admin") ... ask dries or zbynek

  one more note, though: you do not add `&lt;module&gt;_settings()` to the menu (they automatically go to "site configuration > modules > module settings" - you only add

`&lt;module&gt;_admin...()` ... things.

- [from comment_is_new function lost [http://drupal.org/node/view/4230#6570]]

```
-  comment_is_new($comment)
+  node_is_new($comment->nid, $comment->timestamp)
```

please add / update / correct!

Converting 4.2 modules to 4.3

Database table prefix

On 2003 Jul 10, Dries committed [http://drupal.org/node/view/805] Slavica's table prefix patch which allows for a configurable "prefix to each drupal mysql table to easily share one database for multiply applications on server with only one database allowed." This patch requires all table names in SQL-queries to be enclosed in {curly brackets}, eg.

```
-  db_query("DELETE FROM book WHERE nid = %d", $node->nid);
+  db_query("DELETE FROM {book} WHERE nid = %d", $node->nid
```

so that the table prefix can be dynamically prepended to the table name. See the original feature request [http://drupal.org/node/view/805] and the corresponding discussion at the mailing list [http://lists.drupal.org/pipermail/drupal-devel/2003-July/027145.html] for details.

New help system

From Michael Frankowski [http://lists.drupal.org/archives/drupal-devel/2003-10/msg00519.html] message:

There is a block of text placed at the top of each admin page by the admin_page function. After 4.3.0 is out the door the function`menu_get_active_help()` should probably be renamed/ moved into the help module and be attached -- somehow -- to every `_page` hook (probably in the node module) so that we can use this system through out Drupal but for now, there is a block of text displayed at the top of every admin page. This is the active help block. (context sensitive help?)

If the URL of the admin page matches a URL in a `_help` hook then the text from that `_help` hook is displayed on the top of the admin page. If there is no match, the block it not displayed. Because Drupal matches URLs in order to stick "other" stuff in the `_help` hook we have taken to sticking descriptors after a "#" sign. So far, the following descriptors are recognised:

Descriptor	**Function**
admin/system/modules#name	The name of a module (unused, but there)
admin/system/modules#description	The description found on the admin/system/modules page.
admin/help#modulename	The module's help text, displayed on the admin/help page and through themodule's individual help link.
user/help#modulename	The help for a distrbuted authorization module

In the future we will probably recognise #block for the text needed in a block displayed by the help system.

Creating modules for version 4.3.1

This tutorial describes how to create a module for Drupal-CVS (i.e. Drupal version > 4.3.1). A module is a collection of functions that link into Drupal, providing additional functionality to your Drupal installation. After reading this tutorial, you will be able to create a basic block module and use it as a template for more advanced modules and node modules.

This tutorial will not necessarily prepare you to write modules for release into the wild. It does not cover caching, nor does it elaborate on permissions or security issues. Use this tutorial as a starting point, and review other modules and the [Drupal handbook] and [Coding standards] for more information.

This tutorial assumes the following about you:

- Basic PHP knowledge, including syntax and the concept of PHP objects
- Basic understanding of database tables, fields, records and SQL statements
- A working Drupal installation
- Drupal administration access
- Webserver access

This tutorial does not assume you have any knowledge about the inner workings of a Drupal module. This tutorial will not help you write modules for Drupal 4.3.1 or before.

Getting Started

To focus this tutorial, we'll start by creating a block module that lists links to content such as blog entries or forum discussions that were created one week ago. The full tutorial will teach us how to create block content, write links, and retrieve information from Drupal nodes.

Start your module by creating a PHP file and save it as 'onthisdate.module'.

```
<?php
?>
```

As per the [Coding standards], use the longhand <?php tag, and not <? to enclose your PHP code.

All functions in your module are named {modulename}_{hook}, where "hook" is a well defined function name. Drupal will call these functions to get specific data, so having these well defined names means Drupal knows where to look.

Telling Drupal about your module

The first function we'll write will tell Drupal information about your module: its name and description. The hook name for this function is 'help', so start with the onthisdate_help function:

```
<?phpfunctiononthisdate_help($section)
```

```
{
}?>
```

The $section variable provides context for the help: where in Drupal or the module are we looking for help. The recommended way to process this variable is with a switch statement. You'll see this code pattern in other modules.

```
<?php/* Commented out until bug
fixed */
/*
function onthisdate_help($section) {
  switch($section) {
    case "admin/system/modules#name":
      $output = "onthisdate";
      break;
    case "admin/system/modules#description":
      $output = "Display a list of nodes that
were created a week ago.";
      break;
    default:
      $output = "onthisdate";
      break;
  }
  return $output;
}
*/?>
```

You will eventually want to add other cases to this switch statement to provide real help messages to the user. In particular, output for "admin/help#onthisdate" will display on the main help page accessed by the admin/help URL for this module (/admin/help or ?q=admin/help).

Note: This function is commented out in the above code. This is on purpose, as the current version of Drupal CVS won't display the module name, and won't enable it properly when installed. Until this bug is fixed, comment out your help function, or your module may not work.

Telling Drupal who can use your module

The next function to write is the permissions function. Here, you can tell Drupal who can access your module. At this point, give permission to anyone who can access site content or administrate the module.

```
<?phpfunctiononthisdate_perm() {
  return array("administer
onthisdate");
}?>
```

If you are going to write a module that needs to have finer control over the permissions, and you're going to do permission control, you may want to define a new permission set. You can do this by adding strings to the array that is returned:

```
<?phpfunctiononthisdate_perm() {
  return array("access
onthisdate","administer onthisdate");
}?>
```

You'll need to adjust who has permission to view your module on the administer » accounts » permissions page. We'll use the user_access function to check access permissions ???.

Be sure your permission strings must be unique to your module. If they are not, the permissions page will list the same permission multiple times.

Announce we have block content

There are several types of modules: block modules and node modules are two. Block modules create abbreviated content that is typically (but not always, and not required to be) displayed along the left or right side of a page. Node modules generate full page content (such as blog, forum, or book pages).

We'll create a block content to start, and later discuss node content. A module can generate content for blocks and also for a full page (the blogs module is a good example of this). The hook for a block module is appropriately called "block", so let's start our next function:

```
<?phpfunctiononthisdate_block($op='list',$delta=0) {
}?>
```

The block function takes two parameters: the operation and the offset, or delta. We'll just worry about the operation at this point. In particular, we care about the specific case where the block is being listed in the blocks page. In all other situations, we'll display the block content.

```
<?phpfunctiononthisdate_block($op='list',$delta=0) {
  // listing of blocks, such as on the
admin/system/block page
  if ($op=="list") {
    $block[0]["info"]
=t("On This Date");
    return$block;
  } else {
  // our block content
  }
}?>
```

Generate content for a block

Now, we need to generate the 'onthisdate' content for the block. In here, we'll demonstrate a basic way to access the database.

Our goal is to get a list of content (stored as "nodes" in the database) created a week ago. Specifically, we want the content created between midnight and 11:59pm on the day one week ago. When a node is first created, the time of creation is stored in the database. We'll use this database field to find our data.

First, we need to calculate the time (in seconds since epoch start, seeht-tp://www.php.net/manual/en/function.time.php for more information on time format) for midnight a week ago, and 11:59pm a week ago. This part of the code is Drupal independent, see the PHP website (http://php.net/) for more details.

```
<?phpfunctiononthisdate_block($op='list',$delta=0) {
  // listing of blocks, such as on the
admin/system/block page
  if ($op=="list") {
    $block[0]["info"]
=t("On This Date");
    return$block;
  } else {
  // our block content
    // Get today's date
    $today=getdate();
    // calculate midnight one
week ago
    $start_time=mktime(0,0,0,

$today['mon'], ($today['mday'] -7),$today['year']);
    // we want items that
occur only on the day in question, so calculate 1 day
    $end_time=$start_time+86400;  // 60 * 60 * 24 = 86400
seconds in a day
    ...
  }
}?>
```

The next step is the SQL statement that will retrieve the content we'd like to display from the database. We're selecting content from the node table, which is the central table for Drupal content. We'll get all sorts of content type with this query: blog entries, forum posts, etc. For this tutorial, this is okay. For a real module, you would adjust the SQL statement to select specific types of content (by adding the 'type' column and a WHERE clause checking the 'type' column).

Note: the table name is enclosed in curly braces: `{node}`. This is necessary so that your module will support database table name prefixes. You can find more information on the Drupal website by reading the [Table Prefix (and sharing tables across instances)] page in the Drupal handbook.

```
<?php
  $query="SELECT nid, title, created FROM ".

"{node} WHERE created >= '".$start_time.

"' AND created <= '".$end_time."'";?>
```

Drupal uses database helper functions to perform database queries. This means that, for the most part, you can write your database SQL statement and not worry about the backend connections.

We'll use db_query() to get the records (i.e. the database rows) that match our SQL query, and db_fetch_object() to look at the individual records:

```
<?php
  // get the links
  $queryResult=  db_query($query);
  // content variable that will be
returned for display
  $block_content='';
  while ($links=db_fetch_object($queryResult)) {
    $block_content.='<a href="'.url('node/view/'.$links->nid)
.'">'.

$links->title.'</a><br />';
  }
  // check to see if there was any
content before setting up the block
  if ($block_content=='') {
    /* No content from a week
ago.  If we return nothing, the block
     * doesn't show, which is what we want. */
    return;
  }
  // set up the block
  $block['subject']
='On This Date';
  $block['content']
=$block_content;
```

```
  return$block;
}?>
```

Notice the actual URL is enclosed in the url() function. This adjusts the URL to the installations URL configuration of either clean URLS:http://sitename/node/view/2 or http://sitename/?q=node/view/2

Also, we return an array that has 'subject' and 'content' elements. This is what Drupal expects from a block function. If you do not include both of these, the block will not render properly.

You may also notice the bad coding practice of combining content with layout. If you are writing a module for others to use, you will want to provide an easy way for others (in particular, non-programmers) to adjust the content's layout. An easy way to do this is to include a class attribute in your link, and not necessarily include the <br /> at the end of the link. Let's ignore this for now, but be aware of this issue when writing modules that others will use.

Putting it all together, our block function looks like this:

```
<?phpfunctiononthisdate_block($op='list',$delta=0) {
  // listing of blocks, such as on the
admin/system/block page
  if ($op=="list") {
    $block[0]["info"]
=t("On This Date");
    return$block;
  } else {
  // our block content
    // content variable that will be returned for
display
    $block_content='';
    // Get today's date
    $today=getdate();
    // calculate midnight one
week ago
    $start_time=mktime(0,0,0,

$today['mon'], ($today['mday'] -7),$today['year']);
    // we want items that
occur only on the day in question, so calculate 1 day
    $end_time=$start_time+86400;  // 60 * 60 * 24 = 86400
seconds in a day
    $query="SELECT nid, title, created FROM
".

"{node} WHERE created >= '".$start_time.

"' AND created <= '".$end_time."'";
    // get the links
    $queryResult=  db_query($query);
    while ($links=db_fetch_object($queryResult)) {
      $block_content.='<a href="'.url('node/view/'.$links->nid).'">'.

$links->title.'</a><br />';
    }
    // check to see if there
was any content before setting up the block
    if ($block_content=='') {
      // no content
from a week ago, return nothing.
      return;
    }
    // set up the block
    $block['subject']
='On This Date';
    $block['content']
=$block_content;
    return$block;
  }
}?>
```

Installing, enabling and testing the module

At this point, you can install your module and it'll work. Let's do that, and see where we need to improve the module.

To install the module, you'll need to copy your onthisdate.module file to the modules directory of your Drupal installation. The file must be installed in this directory or a subdirectory of the modules directory, and must have the .module name extension.

Log in as your site administrator, and navigate to the modules administration page to get an alphabetical list of modules. In the menus: administer » configuration » modules, or via URL:

```
http://.../admin/system/modules
```

or

```
http://.../?q=admin/system/modules
```

Note: You'll see one of three things for the 'onthisdate' module at this point:

- You'll see the 'onthisdate' module name and no description
- You'll see no module name, but the 'onthisdate' description
- You'll see both the module name and the description

Which of these three choices you see is dependent on the state of the CVS tree, your installation and the help function in your module. If you have a description and no module name, and this bothers you, comment out the help function for the moment. You'll then have the module name, but no description. For this tutorial, either is okay, as you will just enable the module, and won't use the help system.

Enable the module by selecting the checkbox and save your configuration.

Because the module is a blocks module, we'll need to also enable it in the blocks administration menu and specify a location for it to display. Navigate to the blocks administration page: admin/system/block or administer » configuration » blocks in the menus.

Enable the module by selecting the enabled checkbox for the 'On This Date' block and save your blocks. Be sure to adjust the location (left/right) if you are using a theme that limits where blocks are displayed.

Now, head to another page, say select the module. In some themes, the blocks are displayed after the page has rendered the content, and you won't see the change until you go to new page.

If you have content that was created a week ago, the block will display with links to the content. If you don't have content, you'll need to fake some data. You can do this by creating a blog, forum topic or book page, and adjust the "Authored on:" date to be a week ago.

Alternately, if your site has been around for a while, you may have a lot of content created on the day one week ago, and you'll see a large number of links in the block.

Create a module configuration (settings) page

Now that we have a working module, we'd like to make it better. If we have a site that has been around for a while, content from a week ago might not be as interesting as content from a year ago. Similarly, if we have a busy site, we might not want to display all the links to content created last week. So, let's create a configuration page for the administrator to adjust this information.

The configuration page uses the 'settings' hook. We would like only administrators to be able to access this page, so we'll do our first permissions check of the module here:

```
<?phpfunctiononthisdate_settings()
{
  // only administrators can access this
module
  if (!user_access("admin onthisdate")) {
    returnmessage_access();
  }
}?>
```

If you want to tie your modules permissions to the permissions of another module, you can use that module's permission string. The "access content" permission is a good one to check if the user can view the content on your site:

```
<?php
  ...
  // check the user has content
access
  if (!user_access("access content")) {
    returnmessage_access();
  }
  ...?>
```

We'd like to configure how many links display in the block, so we'll create a form for the administrator to set the number of links:

```
<?phpfunctiononthisdate_settings()
{
  // only administrators can access this
module
  if (!user_access("admin onthisdate")) {
    returnmessage_access();
  }
  $output.=form_textfield(t("Maximum number of
links"),"onthisdate_maxdisp",

variable_get("onthisdate_maxdisp","3"),2,2,

t("The maximum number
of links to display in the block."));
  return$output;
}?>
```

This function uses several powerful Drupal form handling features. We don't need to worry about creating an HTML text field or the form, as Drupal will do so for us. We use `variable_get` to retrieve the value of the system configuration variable "onthisdate_maxdisp", which has a default value of 3. We use the form_textfield function to create the form and a text box of size 2, accepting a maximum length of 2 characters. We also use the translate function of t(). There are other form functions that will automatically create the HTML form elements for use. For now, we'll just use the form_textfield function.

Of course, we'll need to use the configuration value in our SQL SELECT, so we'll need to adjust our query statement in the onthisdate_block function:

```
<?php
  $limitnum=variable_get("onthisdate_maxdisp",3);
  $query="SELECT nid, title, created FROM
".

"{node} WHERE created >= '".$start_time.

"' AND created <= '".$end_time."' LIMIT ".$limitnum;?>
```

You can test the settings page by editing the number of links displayed and noticing the block content adjusts accordingly.

Navigate to the settings page: admin/system/modules/onthisdate or administer » configuration » modules » onthisdate. Adjust the number of links and save the configuration. Notice the number of links in the block adjusts accordingly.

Note: We don't have any validation with this input. If you enter "c" in the maximum number of links, you'll break the block.

Adding menu links and creating page content

So far we have our working block and a settings page. The block displays a maximum number of links. However, there may be more links than the maximum we show. So, let's create a page that lists all the content that was created a week ago.

```
<?phpfunctiononthisdate_all() {
}?>
```

We're going to use much of the code from the block function. We'll write this ExtremeProgramming style, and duplicate the code. If we need to use it in a third place, we'll refactor it into a separate function. For now, copy the code to the new function onthisdate_all(). Contrary to all our other functions, 'all', in this case, is not a Drupal hook. We'll discuss below.

```
<?phpfunctiononthisdate_all() {
  // content variable that will be
returned for display
  $page_content='';
  // Get today's date
  $today=getdate();
  // calculate midnight one week
ago
  $start_time=mktime(0,0,0,

$today['mon'], ($today['mday'] -7),$today['year']);
  // we want items that occur only on
the day in question, so calculate 1 day
  $end_time=$start_time+86400;  // 60 * 60 * 24 = 86400
seconds in a day
  // NOTE!  No LIMIT clause here!  We want to
show all the code
  $query="SELECT nid, title, created FROM
".

"{node} WHERE created >= '".$start_time.
"' AND created <= '".$end_time."'";
  // get the links
  $queryResult=  db_query($query);
  while ($links=db_fetch_object($queryResult)) {
    $page_content.='<a
href="'.url('node/view/'.$links->nid).'">'.

$links->title.'</a><br />';
  }
  ...
}?>
```

We have the page content at this point, but we want to do a little more with it than just return it. When creating pages, we need to send the page content to the theme for proper rendering. We use this with the theme() function. Themes control the look of a site. As noted above, we're including layout in the code. This is bad, and should be avoided. It is, however, the topic of another tutorial, so for now, we'll include the formatting in our content:

```
<?php
    printtheme("page",$content_string);?>
```

The rest of our function checks to see if there is content and lets the user know. This is preferable to showing an empty or blank page, which may confuse the user.

Note that we are responsible for outputting the page content with the 'print theme()' syntax. This is a change from previous 4.3.x themes.

```
<?phpfunctiononthisdate_all() {
  ...
  // check to see if there was any
content before setting up the block
  if ($page_content=='') {
    // no content from a week
ago, let the user know
    printtheme("page",

"No events occurred on this site on this date in history.");
    return;
  }
  printtheme("page",$page_content);
}?>
```

Letting Drupal know about the new function

As mentioned above, the function we just wrote isn't a 'hook': it's not a Drupal recognized name. We need to tell Drupal how to access the function when displaying a page. We do this with the _link hook and the menu() function:

```
<?phpfunctiononthisdate_link($type,$node=0) {
}?>
```

There are many different types, but we're going to use only 'system' in this tutorial.

```
<?phpfunctiononthisdate_link($type,$node=0) {
  if (($type=="system")) {
    // URL, page title, func
called for page content, arg, 1 = don't disp menu
```

```
    menu("onthisdate",t("On This Date"),"onthisdate_all",1,1);
  }
}?>
```

Basically, we're saying if the user goes to "onthisdate" (either via ?q=onthisdate or http://.../onthisdate), the content generated by onthisdate_all will be displayed. The title of the page will be "On This Date". The final "1" in the arguments tells Drupal to not display the link in the user's menu. Make this "0" if you want the user to see the link in the side navigation block.

Navigate to /onthisdate (or ?q=onthisdate) and see what you get.

Adding a more link and showing all entries

Because we have our function that creates a page with all the content created a week ago, we can link to it from the block with a "more" link.

Add these lines just before that $block['subject'] line, adding this to the $block_content variable before saving it to the $block['content'] variable:

```
<?php
  // add a more link to our page that
displays all the links
   $block_content.="<div
class=\"more-link\">".l(t("more"),"onthisdate", array("title"=>t("More
events on this day."))) ."</div>";?>
```

This will add the more link.

Conclusion

We now have a working module. It created a block and a page. You should now have enough to get started writing your own modules. We recommend you start with a block module of your own and move onto a node module. Alternately, you can write a filter or theme.

As is, this tutorial's module isn't very useful. However, with a few enhancements, it can be entertaining. Try modifying the select query statement to select only nodes of type 'blog' and see what you get. Alternately, you could get only a particular user's content for a specific week. Instead of using the block function, consider expanding the menu and page functions, adding menus to specific entries or dates, or using the menu callback arguments to adjust what year you look at the content from.

If you start writing modules for others to use, you'll want to provide more details in your code. Comments in the code are incredibly valuable for other developers and users in understanding what's going on in your module. You'll also want to expand the help function, providing better help for the user. Follow the Drupal [Coding standards], especially if you're going to add your module to the project.

Two topics very important in module development are writing themeable pages and writing translatable content. Please check the [Drupal Handbook] for more details on these two subject.

How to build up a _help hook

The following template can be used to build a `_help` hook.

```
<?phpfunction
<modulename>_help($section){
  $output="";
  switch ($section) {
  }
  return$output;
}?>
```

In the template replace modulename with the name of your module.

If you want to add help text to the overall administrative section. (admin/help) stick this inside the switch:

```
<?php
    case'admin/help#<modulename>':
      $output=t('The text you want
displayed');
      break;?>
```

If you also want this same text displayed for an individual help link in your menu area. You have this kind of tree:

```
  + Administration
  |
  -> Your area
  |  |
  |  -> Your configuration
  |  -> help
  |
  -> Overall admin help.
```

Change the function line to this:

```
<?phpfunction
<modulename>_help($section='admin/help#<modlename>')
{?>
```

Now that you have the template started place a case statement in for any URL you want a "context sesitive" help message in the admin section. An example, you have a page that individually configures your module, it is at admin/system/modules/, you want to add some text to the top help area.

```
<?phpcase'admin/system/modules/<modulename>':$output=t('Your new help text');
      break;?>
```

How to convert a _system hook

There are three things that can appear in a _system hook:

Field	Function
$field == "name"	The module name
$field == "description"	The description placed in the module list
$field == "admin-help"	The help text placed at the TOP of this module's individual configurationarea.

Take the text for each one and move it into the _help hook. Replace the `$system[<name>]` that is normally at the front of each one with $output, now place a "break;" after the line and a `case '<name>':` before it where name is one of the following:

- If `$system` is `$system["name"]` then the case is `case 'admin/system/modules#name'`
- If $system is $system["description"] then case is `case 'admin/system/modules#description'`
- If `$system` is `$system["admin-help"]` then the case is `case 'admin/system/modules/<modulename>'`

Now remove the `_system` function and you are done.

An example:

```
<?phpfunctionexample_system($field){
  $system["description"] =t("This is my example _system hook to
convert for
the help system I have spent a lot of time with.");
  $system["admin-help"] =t("Can you believe that I would actually
write an
indivdual setup page on an EXAMPLE module??");
  return$system[$field];
}?>


<?phpfunctionexample_help($section)
{
  $output="";
  switch ($section) {
    case'admin/system/modules#example':
      $output=t("This is my example _system hook to
convert for the help
system I have spent a lot of time with.");
      break;
    case'admin/system/modules/example':
      $output=t("Can you believe that I would actually
write an indivdual
setup page on an EXAMPLE module??");
      break;
  }
  return$output;
}?>
```

How to convert an _auth_help hook

Okay, you have written your Distributed Authorization module, and given us a great help text for it and I had to go and ruin it all by changing the help system. What a terrible thing for me to do. How do you convert it?

It is not that hard. There are two places you have to deal with:

1. The text inside the _auth_help hook needs to be moved inside the _help hook under the section `user/help#<modulename>` and
2. You have to change the _page hook, which normally displays that help text, to find your text in a new location by changing the function call `<modulename>_auth_help()` to `<modulename>_help("user/help#<modulename>")`.

See, it is not THAT terrible.

An example:

```
<?phpfunctionexampleda_page() {
  theme("header");
  theme("box","Example DA",exampleda_auth_help());
  theme("footer");
}
functionexampleda_auth_help() {
  $site=variable_get("site_name","this web site");
  $html_output="
  <p>This is my example Distributed Auth help. Using this
example you cannot login to <i>%s</i> because it has no _auth
hook.&</p>
<p><u>BUT</u> you should still use Drupal since it is a
<b>GREAT</b> CMS and is only getting better.</p>
<p>To learn about about Drupal you can <a
href=\"www.drupal.org\">visit the
site</a></p>";
  returnsprintf(t($html_output),$site);
}?>
```

```
<?phpfunctionexampleda_page() {
  theme("header");
  theme("box","Example DA",exampleda_help('user/help#exampleda'));
  theme("footer");
}
functionexampleda_help($section)
{
  $output="";
  switch ($section) {
    case'user/help#exampleda':
      $site=variable_get("site_name","this web site");
      $output.="<p>This is my
example Distributed Auth help. Using this example you cannot login to %site
because it has no _auth hook.</p>";
      $output.="<p><u>BUT&</u> you should still use
Drupal
since it is a <b>GREAT</b> CMS and is only getting
better.</p>";
      $output.="<p>To learn about
about Drupal you can
visit %drupal.</p>";
      $output=t($output,
array("%site"=>"<i>$site</i>","%drupal"=>"<a
href=\"www.drupal.org\">visit the site</a>"));
      break;
  }
  return$output}?>
```

Converting 4.3 modules to 4.4

Since Drupal 4.3, major changes have been made to the theme, menu, and node systems. Most themes and modules will require some changes.

Menu system

The Drupal menu system has been extended to drive all pages, not just administrative pages. This is continuing the work done for Drupal 4.3, which integrated the administrative menu with the user menu. We now have consistency between administrative and "normal" pages; when you learn to create one, you know how to create the other.

The flow of page generation now proceeds as follows:

1. The `_link` hook in all modules is called, so that modules can use `menu()` to add items to the menu. For example, a module could define:

   ```
   <?phpfunctionexample_link($type)
   {
     if ($type=="system") {
       menu("example",t("example"),"example_page");
       menu("example/foo",t("foo"),"example_foo");
     }
   }?>
   ```

2. The menu system examines the current URL, and finds the "best fit" for the URL in the menu. For example, if the current URL is `example/foo/bar/12`, the above `menu()` calls would cause `example_foo("bar", 12)` to get invoked.
3. The callback may set the title or breadcrumb trail if the defaults are not satisfactory (more on this later).
4. The callback is responsible for printing the requested page. This will usually involve preparing the content, and then printing the return value of `theme("page")`. For example:

   ```
   <?phpfunctionexample_foo($theString,$theNumber) {
     $output=$theString." - ".$theNumber;
     printtheme("page",$output);
   }?>
   ```

The following points should be considered when upgrading modules to use the new menu system:

- The `_page` hook is obsolete. Pages will not be shown unless they are declared with a `menu()` call as discussed above. To convert former `_page` hooks to the new system as simply as possible, just declare that function as a "catchall" callback:

  ```
  <?php
    menu("example",t("example"),"example_page",0,MENU_HIDE);?>
  ```

  The trailing MENU_HIDE argument in this call makes the menu item hidden, so the callback functions but the module does not clutter the user menu.
- Old administrative callbacks returned their content. In the new system, administrative and normal callbacks alike are responsible for printing the entire page.
- The title of the page is printed by the theme system, so page content does not need to be wrapped in a `theme("box")` to get a title printed. If the default title is not satisfactory, it can be changed by calling `drupal_set_title($title)` before `theme("page")` gets called, or by passing the title to `theme("page")` as a parameter.
- The breadcrumb trail is also printed by the theme. If the default one needs to be overridden (to present things like forum hierarchies), this can be done by calling `drupal_set_breadcrumb($breadcrumb)` before `theme("page")` gets called, or by

passing the breadcrumb to `theme("page")` as a parameter. `$breadcrumb` should be a list of links beginning with "Home" and proceeding up to, but not including, the current page.

Theme system

For full information on theme system changes, see converting 4.3 themes to CVS [http://drupal.org/node/view/4475]. The following points are directly relevant to module development:

- All theme functions now return their output instead of printing them to the user. Old `theme()` usage:

  ```
  <?php
  theme("box",$title,$output);?>
  ```

  New usage:

  ```
  <?phpprinttheme("box",$title,$output);?>
  ```

  Modules that define their own theme functions should also return their output.

- The naming of theme functions defined by modules has been standardized to `theme_&lt;module&gt;_&lt;name&gt;`. When using a theme function there is no need to include the theme_ part, as `theme()` will do this automatically. Example:

  ```
  <?phpfunctiontheme_example_list($list)
  {
    returnimplode('<br />',$list);
  }
  printtheme('example_list', array(1,2,3));?>
  ```

  Theme functions must always be called using `theme()` to allow for the active theme to modify the output if necessary.

- The `theme("header")` and `theme("footer")` functions are not available anymore. Module developers should use the `theme("page")` function which wraps the content in the site theme. The full syntax of this function is

  ```
  <?php
  theme("page",$output,$title,$breadcrumb);?>
  ```

  where `$title` and `$breadcrumb` will override any values set before for these properties.

Node system

The node system has been upgraded to allow a single module to define more than one type of node. This will allow some of the more convoluted code in, for example, project.module to be tidied up.

- The `_node()` hook has been deprecated. In its place, modules that define nodes should use `_node_name()` and `_help()`.
- The `_node_name()` function should return a translated string containing the human-readable name of the node type.
- The `_help()` function, when called with parameter `"node/add#modulename"`, should return a translated string containing the description of the node type.

- Modules wishing to use the new ability to define multiple node types should see the Doxygen documentation [http://drupal.org/doxygen/drupal/] for `hook_node_name()` [http://drupal.org/doxygen/drupal/group__hooks.html#ga30] and `hook_node_types()` [http://drupal.org/doxygen/drupal/group__hooks.html#ga31].

Filter system

- The various filter hooks ('filter', 'conf_filters') have been merged into one 'filter' hook. A module that provides filtering functionality should implement:

  ```
  <?phpfunctionexample_filter($op,$text="")
  {
    switch ($op) {
      case"name":
        returnt("Name of
  the filter");
      case"prepare":
        // Do
  preparing on $text
        return$text;
      case"process":
        // Do
  processing on $text
        return$text;
      case"settings":
        // Generate
  $output of settings
        return$output;
    }
  }?>
  ```

  - "name" is new, and should return a friendly name for the filter.
  - "prepare" is also new. This is an extra step that is performed before the default HTML processing, if HTML tags are allowed. It is meant to give filters the chance to escape HTML-like data before it can get stripped. This means, to convert meaningful HTML characters like < and > into entities such as &lt; and &gt;.

    Common examples include filtering pieces of PHP code, mathematical formulas, etc. It is not allowed to do anything other than escaping in the "prepare" step.

    If your filter currently performs such a step in the main "process" step, it should be moved into "prepare" instead. If you don't need any escaping, your filter should simply return $text without processing in this case.
  - "process" is the equivalent of the old "filter" hook. Normal filtering is performed here, and the changed $text is returned.
  - "settings" is the equivalent of the old "conf_filters" hook. If your filter provides configurable options, you should return them here (using the standard form_* functions).
- The filter handling code has been moved to a new required `filter.module`, and thus most of the filter function names changed, although none of those should have been called from modules. The `check_output()` function is still available with the same functionality.
- Node filtering is optimized with the `node_prepare()` function now, which only runs the body through the filters if the node view page is displayed. Otherwise, only the teaser is filtered.

- The `_compose_tips` hook (defined by the contrib `compose_tips.module`) is not supported anymore, but more advanced functionality exists in the core. You can emit extensive compose tips related to the filter you define via the `_help` hook with the `'filter#long-tip'` section identifier. The `compose_tips` URL is thus changed to `filter/tips`. The `form_allowed_tags_text()` function is replaced with `filter_tips_short()`, which now supports short tips to be placed under textareas. Any module can inject short tips about the filter defined via the `_help` hook, with the `'filter#short-tip'` section identifier.

Hook changes

Other than those mentioned above, the following hooks have changed:

- The `_view` hook has been changed to return its content rather than printing it. It also has an extra parameter, `$page`, that indicates whether the node is being viewed as a standalone page or as part of a larger context. This is important because nodes may change the breadcrumb trail if they are being viewed as a page. Old usage:

  ```
  <?phpfunctionexample_view($node,$main=0)
  {
    if ($main) {
      theme("node",$node,$main);
    }
    else {
      $breadcrumb[] =l(t("Home"),"");
      $breadcrumb[] =l(t("foo"),"foo");
      $node->body=theme("breadcrumb",$breadcrumb) ."<br />".$node->body;
      theme("node",$node,$main);
    }
  }?>
  ```

  New usage:

  ```
  <?phpfunctionexample_view($node,$main=0,$page=0) {
    if ($main) {
      returntheme("node",$node,$main,$page);
    }
    else {
      if ($page) {
        $breadcrumb[] =l(t("Home"),"");
        $breadcrumb[] =l(t("foo"),"foo");
        drupal_set_breadcrumb($breadcrumb);
      }
      returntheme("node",$node,$main,$page);
    }
  }?>
  ```

- The `_form` hook used by node modules no longer takes 3 arguments. The second argument `$help`, typically used to print submission guidelines, has been removed. Instead, the help should be emitted using the module's `_help` hook. For examples, check the story, forum or blog module.

- The `_search` hook was changed to not only return the result set array, but a two element array with the result group title and the result set array. This provides more precise control over result group titles.

- The `_head` hook is eliminated and replaced with the `drupal_set_html_head()` and `drupal_get_html_head()` functions. You can add JavaScript code or CSS to the HTML head part with the `drupal_set_html_head()` function instead.

- See also the description of the `_compose_tips` hook changes below.

Emitting links

- The functions `url()` and `l()` take a new `$fragment` parameter. Calls to `url()` or `l()` that have '#' in the `$url` parameter need to be updated. If you don't update such calls, Drupal's path aliasing won't work for URLs with # in them.
- Drupal now emits relative URLS instead of absolute URLs. Contributed modules must be updated whenere an absolute url is required. For example:
-
  - Any module that outputs an RSS feed without using `node_feed()` should be updated. Note: this is discouraged. please use `node_feed()` instead. Also modules using `node_feed()` should provide an absolute link in the 'link' key, if any.
  - Any module which send email should be updated so that links in the email have absolute urls instead of relative urls. You do this using a parameter in your call to `l()` or `url()`

Status and error messages

- Modules that use `theme('error', ...)` to print error messages should be updated to use `drupal_set_message(..., 'error')` unless used to print an error message below a form item.

  ```
  <?php
  drupal_set_message(t('failed
  to update X','error'));  // set the second
  parameter to 'error'?>
  ```

- Modules that print status messages directly to the screen using `status()` should be updated to use `drupal_set_message()`. The `status()` function has been removed.

  ```
  <?php
  drupal_set_message(t('updated
  X'));?>
  ```

Converting 4.4 modules to 4.5

Menu system

The Drupal menu system got a complete rewrite. The new features include:

- The administrator may now customize the menu to reorder, remove, and add items.
- Menu items may be classified as "local tasks," which will by default be displayed as tabs on the page content.
- The menu API is much more consistent with the rest of Drupal's API.

The menu() function is no more. In its place, we have hook_menu(). The old hook_link() remains, but will no longer be called with the "system" argument. The hook reference in the Doxygen documentation details all the specifics [http://drupal.org/doxygen/drupal/group__hooks.html#ga15] of this new hook. In short, rather than making many calls to menu() in your hook_link() implementation, you will implement hook_menu() to return an array of the menu items you define.

As an example, the old pattern:

```
<?phpfunctionblog_link($type,$node=0,$main) {
  global$user;
  if ($type=='system') {
    menu('node/add/blog',t('blog entry'),user_access('maintain personal
blog') ?MENU_FALLTHROUGH:MENU_DENIED,0);
    menu('blog',t('blogs'),user_access('access content') ?'blog_page':MENU_DENIED,
    menu('blog/'.$user->uid,t('my blog'),MENU_FALLTHROUGH,1,MENU_SHOW,MENU_LOCKED)
    menu('blog/feed',t('RSS feed'),user_access('access content') ?'blog_feed':MENU
  }
}?>
```

```
<?phpfunctionblog_menu($may_cache) {
  global$user;
  $items=
array();
  if ($may_cache) {
    $items[] = array('path'=>'node/add/blog','title'=>t('blog entry'),
      'access'=>user_access('maintain personal blog'));
    $items[] = array('path'=>'blog','title'=>t('blogs'),
      'callback'=>'blog_page',
      'access'=>user_access('access content'),
      'type'=>MENU_SUGGESTED_ITEM);
    $items[] = array('path'=>'blog/'.$user->uid,'title'=>t('my blog'),
      'access'=>user_access('maintain personal blog'),
      'type'=>MENU_DYNAMIC_ITEM);
    $items[] = array('path'=>'blog/feed','title'=>t('RSS feed'),
      'callback'=>'blog_feed',
      'access'=>user_access('access content'),
      'type'=>MENU_CALLBACK);
  }
  return$items;
}?>
```

Drupal now distinguishes between 404 (Not Found) pages and 403 (Forbidden) pages. To accommodate this, modules should abandon the practice of not declaring menu items when access is denied to them. Instead, they should set the "access" attribute of their newly-declared menu item to FALSE. This will have the effect of the menu item being hidden, and also preventing the callback from being invoked by typing in the URL. Modules may also want to take advantage of the drupal_access_denied() function, which prints a 403 page (the analogue of drupal_not_found(), which prints a 404).

Path changes

Some internal URL paths have changed; check the links printed by your code. Most significant is that paths of the form "node/view/52" are now "node/52" instead, while "node/edit/52" becomes "node/52/edit".

Node changes

- The database field `static` has been renamed to `sticky`.

- Error handling of forms (such as node editing forms) is now done using form_set_error(). It simplifies the forms and validation code; however, it does change the node API slightly:
    - The _validate hook and the _nodeapi('validate') hook of the node API no longer take an "error" parameter, and should no longer return an error array. To set an error, call form_set_error().
    - Node modules' hook_form() implementations no longer take an "error" parameter and should not worry about displaying errors. The same applies to hook_nodeapi('form_post') and hook_nodeapi('form_pre').
    - All of the form_ family of functions can take a parameter that marks the field as required in a standard way. Use this instead of adding that information to the field description.
- In order to allow modules such as book.module to inject HTML elements into the view of nodes safely, hook_nodeapi() was extended to respond to the 'view' operation. This operation needs to be invoked after the filtering of the node, so hook_view() was changed slightly [http://drupal.org/doxygen/drupal/group__hooks.html#ga38] to no longer require a return value. Instead of calling theme('node', $node) and returning the result as before, the hook can just modify $node as it sees fit (including running $node->body and $node->teaser through the filters, as before), and the calling code will take care of sending the result to the theme. Most modules will just work under the new semantics, as the return value from the hook is just discarded, but the $node parameter is now required to be passed by reference (this was common but optional before).
- We have node-level access control now! This means that node modules need to make very small changes to their hook_access() implementations. The check for $node->status should be removed; the node module takes care of this check. A value should only be returned from this hook if the node module needs to override whatever access is granted by the node_access table. See the hook API for details [http://drupal.org/doxygen/drupal/group__hooks.html#ga2].

  Node listing queries need to be changed as well, so that they properly check for whether the user has access to the node before listing it. Queries of the form

  ```
  <?php
    db_query('SELECT n.nid, n.title FROM {node} n WHERE n.status = 1 AND foo');
  ?>
  ```

  become

  ```
  <?php
    db_query(
       'SELECT n.nid, n.title FROM {node} n '.node_access_join_sql()
       .' WHERE n.status = 1 AND '.node_access_where_sql()
       .' AND foo');
  ?>
  ```

  See node access rights [http://drupal.org/doxygen/drupal/group__node__access.html] in the Doxygen reference.

Filtering changes

This change affects non-filter modules as well! Please read on even if your module does not filter.

The filter system was changed to support multiple input formats. Each input format houses an entire filter configuration: which filters to use, in what order and with what settings. The filter system now supports multiple filters per module as well.

Check_output() changes

Because of the multiple input formats, a module which implements content has to take care of managing the format with each item. If your module uses the node system and passes content through `check_output()`, then you need to do two things:

- Pass $node->format as the second parameter to `check_output()` whenever you use it.
- Add a filter format selector to `hook_form` using a snippet like:

  ```
  <?php
  $output.=filter_form('format',$node->format);?>
  ```

The node system will automatically save/load the format value for you.

If your module provides content outside of the node system, you can decide if you want to support multiple input formats or not. If you don't, the default format will always be used. However, if your module accepts input through the browser, it is strongly advised to support input formats!

To do this, you must:

- Provide a selector for input formats on your forms, using filter_form() [http://www.drupaldocs.org/filter_form].
- Validate the chosen input format on submission, using filter_access() [http://www.drupaldocs.org/filter_access].
- Store the format ID with each content item (the format ID is a number).
- Pass the format ID to check_output().

Check the API documentation for these functions for more information on how to use them.

Filter hook

The _filter hook was changed significantly. It's best to start with the following framework:

```
<?phpfunctionhook_filter($op,$delta=0,$format=
-1,$text='') {
  switch ($op) {
    case'list':
      return array(0=>t('Filter
name'));
    case'description':
      returnt("Short
description of the filter's actions.");/*
    case 'no cache':
      return true;
*/
    case'prepare':
      $text= ...
      return$text;
    case'process':
      $text= ...
      return$text;
    case'settings':
```

```
      $output= ...;
      return$output;
    default:
      return$text;
  }
}?>
```

However, you should now include the $format parameter in the variable names for filter settings. If your filter has a setting "myfilter_something", it should be changed to "myfilter_something_$format". This allows the setting to be set separately for each input format. To check if it works correctly, add your filter to two different input formats and give each instance different settings. Verify that each input format retains its own settings.

Unlike before, the 'settings' operation should only be used to return actually useful settings, because there is now a separate overview of all enabled filters. A filter does not need its own on/off toggle. If a filter has no configurable settings, it should return nothing for the settings, rather than a message like we did before.

Finally, the filter system now includes caching. If your filter's output is dynamic and should not be cached, uncomment the 'no cache' snippet. Only do this when absolutely necessary, because this turns off caching for any input format your filter is used in. Beware of the filter cache when developing your module: it is advised to uncomment 'no cache' while developing, but be sure to remove it again if it's not needed.

Filter tips

Filter tips are now output through the format selector. Modules no longer need to call filter_tips_short() to display them.

A module's filter tips are returned through the filter_tips hook:

```
<?phpfunctionhook_filter_tips($delta,$format,$long=false)
{
  if ($long) {
    returnt("Long
tip");
  }
  else {
    returnt("Short
tip");
  }
}?>
```

Other changes

In addition to the above mentioned changes:

- hook_user() was changed to allow multiple pages of user profile information. The new syntax [http://drupal.org/doxygen/drupal/core_8php.html#a23] of the hook is given in the API reference. Pay particular attention to the "categories", "form", and "view" operations.
- When processing a form submission, you should use drupal_goto() to redirect to the result if the submission was accepted. This prevents a double post when people refresh their browser right after submitting. Messages set with drupal_set_message() will be saved across the redirect. If a submission was rejected, you should not use drupal_goto(), but simply print out the form along with error messages.

Converting 4.5 modules to 4.6

Block system

Every block now has a configuration page to control block-specific options. Modules which have configurations for their blocks should move those into hook_block() [http://drupaldocs.org/hook_block].

The only required changes to modules implementing hook_block() is to be careful about what is returned. Do not return anything if $op is not 'list' or 'view'. Once this change is made, modules will still be compatible with Drupal 4.5.

If a specific block has configuration options, implement the additional $op options in your module. The implementation of 'configure' should return a string containing the configuration form for the block with the appropriate $delta. 'save' will have an additional $edit argument, which will contain the submitted form data for saving.

Search system

The search system got a significant overhaul.

Node indexing now uses the node's processed and filtered output, which means that any custom node fields will automatically be included in the index, as long as they are visible to normal users who view the node. Modules that implement `hook_search()` and `hook_update_index()` just to have extra node fields indexed no longer need to do this.

If you wish to have additional information indexed that is not visible in the node display at node/id, then you can do so using nodeapi('update index'). If you want to add extra information to the node results, use nodeapi('search result').

However, the standard search is still limited to a keyword search. Modules that implement custom, specific search forms (like project.module) can still do so. Custom search forms that do not use hook_search() should be located/moved to a local task under the /search page.

If you are unsure of what you need to do, please refer to the complete search documentation [http://drupaldocs.org/api/head/group/search].

Module paths

The function `module_get_path` was renamed to `drupal_get_path` which now returns the path for all themes, theme engines and modules. Because of this abstraction you must pass an additional parameter identifying the type of item for which the path is requested. The following example compares retrieving the path to image module between Drupal 4.5 and 4.6.

```
<?php// Drupal
4.5:$path=module_get_path('image');// Drupal
4.6:$path=drupal_get_path('module','image');?>
```

All instances of `module_get_path` should be renamed to `drupal_get_path`.

Database backend

The function `check_query` was renamed to `db_escape_string` and now has a database specific implementation. All instances of `check_query` should be renamed to `db_escape_string`.

Theme system

The function theme_page() [http://drupaldocs.org/theme_page] no longer takes `$title` or `$breadcrumb` arguments. Set page titles using hook_menu() [http://drupaldocs.org/hook_menu] or, if the title must be dynamically determined, use drupal_set_title(). Set breadcrumb trails first using hook_menu(), which can be overridden with menu_set_location() [http://drupaldocs.org/menu_set_location] and drupal_set_breadcrumb() [http://drupaldocs.org/drupal_set_breadcrumb].

Watchdog messages

The watchdog() function now takes a severity attribute, so `watchdog($type, $message, $link);` becomes `watchdog($type, $message, $severity, $link);`. Specify a severity in case you are reporting a warning or error. Possible severity constants are: `WATCHDOG_NOTICE`, `WATCHDOG_WARNING` and `WATCHDOG_ERROR`. Also make sure that you provide the type as a literal string, so translation extraction can pick it up.

If you are unsure of which severity to use, remember these rules:

- If the problem is caused by a definite fault and should be fixed as soon as possible, use an error message.
- If the problem could point to a fault, but could also be harmless, use a warning message. This type should also be used whenever the problem could be caused by a remote server (example: ping timeout, failed to aggregate a feed, etc).
- Normal messages should be notices.

Node markers

If you have a module calling `theme('mark')`, note that it is now possible to have different markers for different states of a node. The supported states are `MARK_NEW`, `MARK_UPDATED` and `MARK_READ`. You can get the marker state from `node_mark()`, which replaces the `node_new()` function available in previous Drupal versions.

Control over destination page after form processing

Occasionally a module might want to specify where a user should go after he submits a form. This is now possible by passing a querystring parameter `&destination=<path>`. For example, editing of nodes and comments from within the Admin pages now returns the user to those pages after he is done. For example usage, search drupal_get_destination() which can be found in path.module, node.module, comment.module, and user.module

Confirmation messages

Confirmations for dangerous actions should now be presented with the `theme('confirm')` function for consistency. Check the function's documentation [http://drupaldocs.org/theme_confirm] or look at some of the core modules for examples.

Note that this is a themable function which should be invoked through `theme('confirm')` and not `theme_confirm()`.

Inter module calls

New features are available -- it's not necessary to use them. Now you can really (and should) use module_invoke to call a function from another module. For example, `taxonomy_get_tree` should be called by `module_invoke('taxonomy', 'get_tree')` If you need to loop through the implementations of a hook, please check the new module_implements [http://drupaldocs.org/module_implements] function.

Node queries

If you have a module which retrieves a list of nodes by issuing its own database query, then the following applies.

The functions node_access_join_sql() and node_access_where_sql() should not be used any more but the SELECT-queries should be wrapped in a db_rewrite_sql() call.

If you have used DISTINCT(nid) -- because of node_access_join_sql() -- you no longer need it, replace it simply with n.nid. If you have SELECT *, please replace it with SELECT n.nid, n.* -- and always make sure that n.nid field comes first in the SELECT statement -- this way the db_rewrite_sql() function can rewrite the query to use DISTINCT(nid) should there be a need for it. If the n.nid field is not first, the query will fail when node access modules are enabled. Also, at the moment db_rewrite_sql can not handle AS -- either leave it out or lowercase it.

Always use table name before the field names, especially before nid because other tables may be JOINed during the rewrite process.

Example:

```
<?php// Drupal
4.5:$nodes=db_query_range('SELECT DISTINCT(n.nid) FROM {node} n
'.node_access_join_sql()
.' WHERE '.node_access_where_sql()
.' AND n.promote = 1 AND n.status = 1 ORDER BY
n.created DESC',0,15);// Drupal 4.6:$nodes=db_query_range(db_rewrite_sql('SELECT n
n.status = 1 ORDER BY n.created DESC'),0,15);?>
```

If you are not using the node table, then you shall pass the table name from which you SELECTing the nodes. For example

```
<?php
$result=db_query(db_rewrite_sql("SELECT f.nid, f.* from {files} f WHERE filepath =
'%s'",'f'),$file);?>
```

note the 'f' parameter of db_rewrite_sql().

Avoid USING because there could be JOINs before it, which will break the USING clause.

Text output

Drupal's text output was audited and several escaping bugs were found. For more info, see the check_plain patch [http://drupal.org/node/18817].

You need to pay attention that all user-submitted plain-text in your module is escaped using check_plain() when you output it into HTML. No escaping should be done on data that is going into the database: only escape when outputting to HTML.

Check_plain() replaces drupal_specialchars() and check_form(), so if you are using any of those two, you should use check_plain() instead.

You should also wrap user-submitted text in messages with `theme('placeholder', $text).` For example for "created term %term".

Pay attention in particular to node and comment titles as their behaviour has been changed. They are now stored as plain-text, like other single-line fields in Drupal and should be escaped when output. However, the function l() now takes plain-text by default instead of HTML, which means that whenever $node->title is used as the caption for a link, it will automatically be escaped. When outputting titles literally, you still have to escape them yourself.

URLs also require attention, as the URL functions (url, request_uri, referer_uri, etc) were changed to output 'real' URLs rather than HTML-escaped URLs. When putting any of them inside an HTML tag attribute (e.g. <a href="...">), you need to pass it through check_url() first. When putting an URL into HTML outside of a tag or attribute, you can use check_url() or check_plain(), it doesn't matter. Don't use check_url() in situations where a real URL is expected (e.g. the HTTP "Location: ..." header).

The best test is to submit forms with HTML tags in the plain-text/single-line fields (e.g. "<u>test</u>"). If the underline tag is not interpreted, but displayed literally, your module is escaping the text correctly.

Nothing has changed for filtered/rich text, which still uses check_output() like before.

Converting 4.6 modules to HEAD

Taxonomy API change

In order to provide more meaningful messages to the user, you are now required to provide your own when using the taxonomy APIs to create or modify terms and vocabularies. This applies to `taxonomy_save_vocabulary()` and `taxonomy_save_term()`.

A status message is returned, which can be either `SAVED_NEW`, `SAVED_UPDATED` or `SAVED_DELETED`.

This snippet shows you an example of handling this:

```
<?php// Drupal
4.6taxonomy_save_vocabulary($edit);// Drupal
4.7switch (taxonomy_save_vocabulary($edit))
{
  caseSAVED_NEW:
    drupal_set_message(t('Created
new vocabulary %name.', array('%name'=>theme('placeholder',$edit['name']))));
    break;
  caseSAVED_UPDATED:
    drupal_set_message(t('Updated
vocabulary %name.', array('%name'=>theme('placeholder',$edit['name']))));
    break;
  caseSAVED_DELETED:
    drupal_set_message(t('Deleted
vocabulary %name.', array('%name'=>theme('placeholder',$deleted_name))));
    break;
}?>
```

Table API change

Themes tables sometimes were called with arguments set to NULL or an empty string to indicate that there were either no rows or no header. This has now to be an empty array.

Change

```
<?php
theme('table','',$rows);?>
```

```
<?php
theme('table', array(),$rows);?>
```

Check Output change

Due to a security vulnerability discovered earlier in the filter system, we have tightened security around the `check_output()` function. The format passed to `check_output()` is now checked for access by default. If you don't want this check, pass FALSE for the third parameter, `$check`.

```
<?phpfunctioncheck_output($text,$format=FILTER_FORMAT_DEFAULT,$check=TRUE) {?>
```

Note that if you disable the check by passing FALSE, you need to make sure the `$format` value has been checked by `filter_access()` before. `filter_access()` checks the permissions of the current user, so it should be checked on submission, not on output.

Chapter 7. Join forces

Too often new modules are contributed that do nothing new, only do it in a different way. We are then stuck with two modules that offer nearly similar functionality, but both do not do it well enough. This leads to confusion, clutter and a lot of innefficiency.

So please consider the following guidelines or ideas:

- Develop and use one central API. do not introduce any new .incs, .modules or other files with APIS, if there are modules that have these already.
- Consult other developers of modules in your domain when you plan to add features, or plan to add a module. try to agree on features, to avoid overlapping. Nothing is more confusing for a user when he has, for example a Spam Queue for comments, and a completely different one for trackbacks, which does not respect the options you set for comments. Even worse, but certainly not unheard of, is that module Foo breaks module Bar, because they want to do the same, or want to use the same database tables.
- Do not try to duplicate functionality because "you do not really like how its done there" That only adds clutter. Rather improve the existing one, then introduce yet another-half-witted-module.

Note that they are not rules or laws. But that respecting them will most often help you and the community better. For only then will we be able to "stand on the shoulders of Giants" as they say in Open Source Land. If you keep reinventing wheels, you will be stuck with lots of incompatible and half finished wheels, in the end. When you use existing wheels and build a car on top of them, you will be able to get somewhere, one day.

Chapter 8. Reference

This section is intended as a handy reference, collecting things which you may need to look up as you code to Drupal.

'Status' field values for nodes and comments

Just documenting the **status** field for the following tables

NODES

- 0: not published
- 1: published

COMMENTS

- 0: published
- 1: not published
- 2: deleted (no longer exists in Drupal 4.5 and above)

Values of 'comment' field in node table

Here are the values of the 'comment' field in the node table:

- 0 = comments cannot be added to this node and published comments will not display
- 1 = comments cannot be added to this node, but published comments will display
- 2 = new comments can be added and published comments will display

Chapter 9. Module how-to's

This section collects various 'How-to' articles of interest to module writers and hackers.

How to write a node module

This information is superseded by the Doxygen documentation. In particular, its example node module [http://drupaldocs.org/api/head/file/contributions/docs/developer/examples/node_example.module] is a good tutorial.

How to write database independent code

In order to ensure that your module works with all compatible database servers (currently Postgres and MySQL), you'll need to remember a few points.

- When you need to LIMIT your result set to certain number of records, you should use the db_query_range() function instead of db_query(). The syntax of the two functions is the same, with the addition of two required parameters at the end of db_query_range(). Those parameters are $from and then $count. Usually, $from is 0 and $count is the maximum number of records you want returned.
- If possible, provide SQL setup scripts for each supported database platform. The differences between each platform are slight - we hope documentation on these differences will be forthcoming.
- You should test any complex queries for ANSI compatibility using this tool by Mimer [http://developer.mimer.se/validator/index.htm]
- If you are developing on MySQL, use it's ANSI compatibility mode [http://www.mysql.com/doc/en/ANSI_mode.html]
- If you can install all database servers in your environment, it is helpful to create shell databases in each and then run sample queries in each platform's query dispatch tool. Once your query succeeds in all tools, congratulate yourself.
- Don't use `''` when you mean `NULL`
- Avoid table and field names that might be reserved words on any platform.
- Don't use auto-increment or SERIAL fields. Instead, use an integer field and leverage Drupal's own sequencing wrapper: `db_next_id(<tablename_fieldname>)`

How to write efficient database JOINs

This page is based on an e-mail posted by Craig Courtney on 6/21/2003 to the drupal-devel mailing list: http://drupal.org/node/view/322.

There are 3 kinds of join: INNER, LEFT OUTER, and RIGHT OUTER. Each requires an ON clause to let the RDBMS know what fields to use joining the tables. For each join there are two tables: the left table and the right table. The syntax is as follows:

{left table} {INNER | LEFT | RIGHT} JOIN {right table} ON {join criteria}

An INNER JOIN returns only those rows from the left table having a matching row in the right table based on the join criteria.

A LEFT JOIN returns ALL rows from the left table even if no matching rows where found in the right table. Any values selected out of the right table will be null for those rows where no matching row is found in the right table.

A RIGHT JOIN works exactly the same as a left join but reversing the direction. So it would return all rows in the right table regardless of matching rows in the left table.

It is recommended that you not use right joins, as a query can always be rewritten to use left joins which tend to be more portable and easier to read.

With all of the joins, if there are multiple rows in one table that match one row in the other table, that row will get returned many times.

For example: Table A tid, name 1, 'Linux' 2, 'Debian'

Table B fid, tid, message 1, 1, 'Very Cool' 2, 1, 'What an example'

Query 1: SELECT a.name, b.message FROM a INNER JOIN b ON a.tid = b.tid Result 1: Linux, Very Cool Linux, What an example

Query 2: SELECT a.name, b.message FROM a LEFT JOIN b ON a.tid = b.tid Result 2: Linux, Very Cool Linux, What an example Debian, <null>

Hope that helps in reading some of the queries.

How to connect to multiple databases within Drupal

Drupal can connect to different databases with elegance and ease!

First define the database connections Drupal can use by editing the `$db_url` string in the Drupal configuration file (settings.php for 4.6 and above, otherwise conf.php). By default only a single connection is defined

```
<?php
$db_url='mysql://drupal:drupal@localhost/drupal';?>
```

To allow multiple database connections, convert `$db_url` to an array.

```
<?php
$db_url['default'] ='mysql://drupal:drupal@localhost/drupal';$db_url['mydb']
='mysql://user:pwd@localhost/anotherdb';$db_url['db3']
='mysql://user:pwd@localhost/yetanotherdb';?>
```

Note that database storing your Drupal installation should be keyed as the `default` connection.

To query a different database, simply set it as active by referencing the key name.

```
<?php
db_set_active('mydb');db_query('SELECT * FROM other_db');//Switch back to the defa
connection when finished.db_set_active('default');?>
```

Make sure to always switch back to the default connection so Drupal can cleanly finish the request life-cycle and write to its system tables.

How to write themable modules

Note: this page describes Drupal's theming from the code side of things.

Drupal's theme system is very powerful. You can accommodate rather major changes in overall appearance and significant structural changes. Moreover, you control all aspects of your drupal site in terms of colors, mark-up, layout and even the position of most blocks (or boxes). You can leave blocks out, move them from right to left, up and down until it fits your needs.

At the basis of this are Drupal's theme functions. Each theme function takes a particular piece of data and outputs it as HTML. The default theme functions are all named `theme_something()` or `theme_module_something()`, thus allowing any module to add themeable parts to the default set provided by Drupal. Some of the basic theme functions include: `theme_error()` and `theme_table()` which as their name suggest return HTML code for an error message and a table respectively. Theme functions defined by modules include `theme_forum_display()` and `theme_node_list()`.

Custom themes can implement their own version of these theme functions by defining `mytheme_something()` (if the theme is named `mytheme`). For example, functions named: `mytheme_error()`, `mytheme_table()`, `mytheme_forum_display()`, `mytheme_node_list()`, etc. corresponding to the default theme functions described above.

Drupal invokes these functions indirectly using the `theme()` function. For example:

```
<?php
$node=node_load(array('nid'=>$nid));$output.=theme("node",$node);?>
```

```
theme_node($node)
```

```
mytheme_node()
```

```
mytheme_node($node)
```

This simple and straight-forward approach has proven to be both flexible and fast.

However, because direct PHP theming is not ideal for everyone, we have implemented mechanisms on top of this: so-called template engines can act as intermediaries between Drupal and the template/theme. The template engine will override the `theme_functions()` and stick the appropriate content into user defined (X)HTML templates. This way, no PHP knowledge is required and a lot of the complexity is hidden away. More information about this can be found in the Theme developer's guide [http://drupal.org/node/509], specifically the Theming overview [http://drupal.org/node/11774].