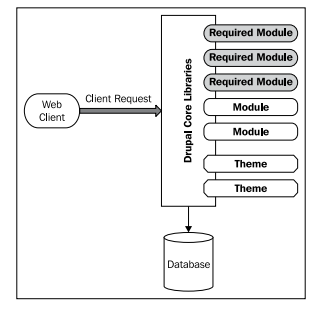
HOW TO DRUPAL WORKS

* A user enters the URL http://example.com/node/123 in a web browser and hits Enter.
* The browser contacts the web server at example.com and requests the resource at /node/123
* The web server recognizes that the request must be handled by PHP, and starts up (or contacts) a PHP environment to handle the request
* PHP executes Drupal's index.php file, handing it the path /node/123
* The Drupal core undergoes a bootstrap process, initializing resources, and then uses the menu system to find out how to handle /node/123
* The node system responds to the request for /node/123 by loading the node (piece of content) with the ID 123. This data is typically loaded from a database
* The theme system takes the node information and applies formatting and style information to it, essentially transforming data into chunks of HTML with associated CSS
* The Drupal core completes all processing (giving many other subsystems the opportunity to handle various bits of data processing) and then returns the data to the client
* The client web browser transforms the HTML and CSS into a visual presentation for the user, running any necessary JavaScript along the way
* The user views the document



User hit the url, the request comes to the server and server check the php, executes the index.php its handle the path or pass it to the Drupal core libraries

DRUPAL CORE LIBRARY

Drupal core is the foundational layer of Drupal which ships with a handful of core libraries along with over a dozen core modules. The index file loads the libraries and then initializes Drupal, a process called bootstrapping.  
  
Once Drupal has loaded core libraries and initialized the database, it loads the enabled modules and themes, and then it begins a systematic, step-by-step process of handling the request. This process is what I call the lifecycle of a request. It works as follows.

DRUPAL HOOKS

After the initialization of the modules It scans the loaded modules to see if any of them implement the function hook\_init(). To implement a hook in Drupal is to declare a function that follows the hook naming pattern.

Through this hook\_init() hook, Drupal provides modules the ability to initialize themselves or their own resources right at the beginning of the request. Once all of these modules have been initialized, Drupal moves on to the next step. As it progresses through the request, it calls hook after hook, giving modules the opportunity to do their thing. Finally, it makes one last call for modules that implement hook\_exit(), and then it returns any remaining data to the client and terminates the request.

Drupal core modules

Drupal provides several core modules. These modules cannot be disabled, as their capabilities are integral to the standard functioning of Drupal. Just like other modules (including the ones we will be writing), core modules function by implementing Drupal hooks.

DATABASE

It is used only for store the data and perform such functions like insert, update or delete nothing role in the drupal of the database.

The theme system

The final component from our initial architectural diagram is the theme system. Drupal separates the look-and-feel components of the system from the rest of Drupal, and provides a programmatic way for theming data. The system for handling this is collectively called the theme system

A theme is a structured bundle of code (like a module) that provides tools for transforming raw data into formatted output. Sites use at least one theme to apply a consistent and custom look-and-feel to all of the pages on the site

a module might declare a rough layout for a component, but Drupal provides the structure for a theme developer to later modify the theme (not the module) to re-layout that component in a different way.

Drupal's major subsystems

1. Theme
2. Menu
3. Nodes
4. Files
5. Users
6. Comments
7. Fields and entities
8. Forms Api
9. Installation Profiles
10. Simple Test
11. Blocks

CREATING FIRST MODULE

We are create module folder “First” inside the /sites/default/modules, then created 2 files inside that folder is first.info and another is first.module.

In first.info declared the name of the module, description and package, files[] = first.module

In first.module declare the code of the php and functionality

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

function first\_help($path, $arg) {

if ($path == 'admin/help#first') {

return t('A demonstration module.');

} }

name: first\_help(). The name follows the hook pattern. If the hook is named hook\_help(), then to implement it, we replace the word hook with the name of the module. Thus, to implement hook\_help(), we simply declare a function in our first module named first\_help().

A hook\_help() implementation takes two arguments: $path: The help system URI path $arg: The arguments used when accessing this URL

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

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

----------------------------------------------------------------------------------------------------

Drupal's Theme Layer

Drupal's theming system is the Appearance page, which lists all of the themes installed on your website. When you choose a theme from the Appearance admin page, you are applying a specific graphic design to your website's data and functionality. However, the applied theme is in reality only a small part of the entire theming layer.

API hooks, Drupal also has theme hooks. A theme hook is simply the name of a particular way to markup some data. For example, passing data to the item\_list theme hook will result in different markup then passing data to the links theme hook. However, while normally every module's hook function will be called when Drupal invokes an API hook, only one theme hook implementation will be invoked when Drupal invokes a theme hook.

Advantages of Data granular

There are two advantages to this granular system.

First, since each module is responsible for theming its own data, it can either create a very specialized theme hook for its data or it can re-use an existing theme hook. Re-using a theme hook ensures a consistent set of markup for similar data structures while still allowing customized CSS classes.

Having a fine-grained theming system means that a theme, if it chooses to, can literally rewrite all of the markup for its own design purposes. As module developers we need to be keenly aware of the themer's desire to have granular theming overrides.

Theme functions

For a module developer, the easiest type of implementation to understand is a theme function. Theme functions just need to follow a few simple rules in order for them to work properly. First, the name of the theme function follows the pattern: theme\_[theme hook name] Since the theme hook name is used directly in the theme function's name, theme hook names have the same constraints on naming as regular PHP function names; the only valid characters in theme hook names are alphanumeric characters and underscores. So if a module has created an example\_format theme hook, it would implement it with theme function named theme\_example\_format().

Theming a Module

First Draft

/\*\* \* Implements hook\_block\_info(). \*/

function single\_blog\_block\_info() {

$blocks = array();

// The array key defines the $delta parameter used in all

// other block hooks.

$blocks['recent'] = array(

// The name of the block on the blocks administration page.

'info' => t('Recent blog posts'),

);

return $blocks;

}

/\*\*

\* Implements hook\_block\_view().

\* First draft!

\*

\* @pararm $delta

\* The name of the requested block.

\*/

function single\_blog\_block\_view($delta = '') {

// Create an empty block.

$block = array(

'subject' => '',

'content' => '',

);

// Check which block is being requested.

if ($delta == 'recent') {

// Set the block title.

$block['subject'] = t('Recent blog posts');

// Check if the user can access content.

if (user\_access('access content')) {

// Retrieve the most recent nodes.

$result = single\_blog\_list(SINGLE\_BLOG\_LIST\_COUNT);

// Create links for each blog entry.

$items = array();

foreach ($result as $node) {

$items[] = array(

'data' => l($node->title, 'node/' . $node->nid),

'class' => array('node-' . $node->nid),

);

}

if (!empty($items)) {

// Theme the list of blog entries.

$block['content'] = theme('item\_list', array(

'items' => $items));

}

}

}

return $block;

}

//second Draft

/\*\*

\* Implements hook\_block\_view().

\*

\* Second draft!

\*

\* @pararm $delta

\* The name of the requested block.

\*/

function single\_blog\_block\_view($delta = '') {

// Create an empty block.

$block = array(

'subject' => '',

'content' => '',

);

// Check which block is being requested.

if ($delta == 'recent') {

// Set the block title.

$block['subject'] = t('Recent blog posts');

// Check if the user can access content.

if (user\_access('access content')) {

// Retrieve the most recent nodes.

$result = single\_blog\_list(SINGLE\_BLOG\_LIST\_COUNT);

// Create links for each blog entry.

$items = array();

foreach ($result as $node) {

$items[] = array(

'data' => array(

'#type' => 'link',

'#title' => $node->title,

'#href' => 'node/' . $node->nid,

),

'class' => array('node-' . $node->nid),

);

}

if (!empty($items)) {

// Theme the list of blog entries.

$block['content']['list'] = array(

'#theme' => 'item\_list\_\_single\_blog',

'#items' => $items,

'#pre\_render' => array('single\_blog\_item\_list\_child\_render'),);

// Add a link to the full list of blog entries.

$block['content']['more'] = array(

'#theme' => 'more\_link',

'#url' => 'blog',

'#title' => t('Read the latest blog entries.'),

);

// Add a CSS file to style the block.

$block['content']['#attached']['css'][] = drupal\_get\_path('module', 'single\_blog') . '/single-blog.css';

}

}

}

return $block;

}

Steps to build a default theme implementation

create a single-blog-block-item.tpl.php to render the contents of each item in our list of blog posts.

Preprocess function

Template files

The first part of any template file should be a large docblock explaining all the variables available to themers, including convenience variables, not just the ones printed inside our template.

The $classes variable is created by template\_processs() and its corresponding $classes\_array variable is created by template\_preprocess(). If we want to add additional classes to the $classes string, we should append an array element to the $classes\_array variable during our preprocess function and it will automatically be added to the $classes string before reaching the template file. The string passed to the t() function, by !username includes the !username token to give context to translators when trying to translate "by"; see the t().

ADMIN INTERFACE

The User Warn module

First always need to create .info, for the module development

And then create .modules file and inside this we developed functionality inside it

The Drupal menu system

Drupal's menu system is deceptively named. The name implies that it is responsible for the navigation of your site, and while this is true it does a great deal more. At its core, the menu system is responsible for mapping Drupal paths to the functions that generate the contents of the requested page. The menu system is also responsible for controlling access to Drupal pages, acting as one of the central gatekeepers of Drupal security. Drupal module developers can map paths to Drupal functions by implementing hook\_menu(), which adds paths to the menu system, assigns them access rules, and optionally creates navigational elements for them.

Defining a page callback with hook\_menu

For this module we need to implement two pages

1. Configuration
2. A Tab In The User Profile Area

These will each require their own hook\_menu() implementation

Drupal hook implementations, hook\_menu() returns a structured associative array with information about the menu items being defined.

First defines the module configuration page, and the second one defines the user tab where administrators can go to send the actual e-mail.

If you are creating a menu item for site administration it must begin with admin. This places it into Drupal's administrative interface and applies the admin theme defined by the site settings.

FORM API

Managing persistent data

Persistent variables are stored in a database table, keyed by a unique name provided by the module that implements them. Persistent variables are saved using variable\_set(), and retrieved using variable\_get().

These variables can be any type of data that a developer needs, be it a simple string, an associative array, or a PHP object. The Drupal API for setting/ getting them takes care of all the serialization/unserialization that is necessary behind the scenes.

variable\_get() can also provide a default value, which is useful for situations where you need a variable which has not already been set, for instance, after a module is installed for the first time.

SUBMITTING data

Form submit functions take two arguments. $form is the original Form API array for the submitted form, and $form\_state is an associative array containing information specific to this submission. In particular, $form\_state['values'] contains all the submitted form values keyed on their name properties. In general, $form\_state['values'] is the only thing you will need to worry about in validate and submit functions

What are tokens?

A token is a small piece of text that can be placed into a piece of text via the use of a placeholder. When the text is passed through the function token\_replace(), then the tokens are replaced with the appropriate information. Tokens allow users to include data that could change in text blocks, without having to go back and change it everywhere they're referenced.

Working with Content

Those separate object types were merged into a single super-system known as "entities". Nodes, users, comments, and several other types of data objects are now particular instances of the generic Entity data object concept. That allows all types of data to have the same, or at least very similar, API and workflow, avoiding duplicate code and reducing the number of moving parts developers need to keep track of. Most importantly, it allows us to attach Fields, discrete structured pieces of information, to any type of entity rather than just to nodes.

BUNDLE: A bundle is a sub-type of an entity that can be configured separately. Node types are an example of a bundle. Not all entity types have bundles. Users, for instance, do not have separate bundles.

The Schema API allows database-agnostic definition and manipulation of the tables in Drupal's SQL database.

The most important hook in the artwork.install file is hook\_schema(), which defines the database tables this module provides.

Declaring our entity

A controller is a loader object for an entity. All entity types must have a controller, but many can use the default. Different controllers may require additional keys on an entity definition.

Once again, our primary means of communicating with Drupal is through large structured arrays that define all the information we need. In this case, our $return array has a single entry, artwork. The string artwork, as the top-most key, will serve as the "machine name" of this entity, which is how it will be referred to in code. The label key specifies what name should be shown to the user. The base table, revision table, and object keys entries tell the entity system about how our artwork is going to be stored, and are used by the default controller:

The entity declaration

In our case, we defined the bundle to use the type property so our artwork types have a property called type that contains the machine name of the bundle. A property called name for the human-friendly name of the bundle is a standard convention but not strictly required, as is a human-readable description property.

Drupal\_static:

There is one other important detail here, and that is the drupal\_static() function. That function acts as a central collector for static PHP variables, that is, those that are not technically global but should persist between calls to a function. They are quite commonly used as a lightweight cache to avoid re-processing or refetching the same data within the same page request, but that can in some cases lead to weird side effects when the data being cached changes mid-request, such as when writing unit tests.

The drupal\_static() function acts as a central collector for such static variables. By putting all such static variables in one place and giving the variable a name that matches our function (that's what the \_\_FUNCTION\_\_ PHP constant means), we allow systems that need to forcibly reset static caches without having a separate $reset parameter for every part of the system.

Creating Fields

Why we need fields

1. We want to conceptually treat that piece of data as a single chunk with its own meaning rather than as a series of chunks that together we know has meaning.

2. We have complex data but want to have multiple instances of that data on a single entity.

3. We want to present a unified custom interface to users while editing that data, especially if it is multi-value

4. We want to display the data to the user in a custom format.

How Field API works

Field type: this is strictly speaking, just the content definition. It defines the name of the field and what its inner data structure is, but not how to save it or how to display it.

Field: this is a particular configuration of a field type. Field instance: this is the combination of a particular field with a bundle or subclass of an entity type

Widget: this is a form element that exposes the field to a content editor. It could use simple text fields or be something as complex as an interactive Flash-based tool.

Formatter: this is a piece of code that formats a field for display on screen. Typically it just wraps Drupal's theme system to do so.

we define how or where the data gets stored. That is handled by a field storage engine, which can be configured separately per field. By default all fields use a common storage engine that saves fields to Drupal's database. That's good enough for our needs, so we won't go into field storage engines in depth.

Drupal Permissions and Security

Drupal's roles and permissions concepts

Using user\_access() to assert permissions

Using hook\_permission()

Access control with hook\_menu()

Common errors in defining permissions

Declaring your own access functions

Securing sensitive actions with permissions

Responding when access is denied

Enabling permissions programmatically

Permissions, security, and Drupal forms

Security considerations for AJAX processing

User access:

The function is quite elementary, taking only two arguments: user\_access($string, $account = NULL) Here, $string is the machine readable name of the permission, and $account is an optional copy of a $user object, as returned by the function user\_load().

user\_access() function defaults to using the current $user object, that is, the user currently making the page request. The $user object is stored in a global variable, and so it can be accessed any time a specific $account is not specified.

Note:

The use of the $account object instead of the $user object is a standard practice of Drupal, and a good coding practice. In Drupal, the $user object is a global value, and it would be a mistake to pass it (sometimes by reference!) when we only mean to extract information from it. Instead, lookup functions like hook\_user\_view() always act on a copy called $account. This pattern occurs frequently in Drupal core.

Hook\_permission

hook\_permission() was known as hook\_perm(). The change was made for clarity in the code, as part of a general semantic cleanup of Drupal core. (I wrote the patch, in fact.) hook\_permission() also includes a number of usability improvements, which altered the format of the function's return value. These changes are substantial enough for even experienced Drupal developers to explore each element of the new hook

The purpose of hook\_permission() is to define and return an array that contains all the necessary information about your module's permissions. This includes the simple strings that can be passed to user\_access(), plus a human-readable name for the permission and an optional description.

Defining your module's permissions

Writing hook\_permission()

Our permission array is quite direct, and has three possible parts:

1. The machine-readable name of the permission. This element will be used by the module code to check user\_access(). For our example, we use the string "view content creation permissions". By convention, this string must be in English and all lower case. It need not be a complete sentence.

2. The human-readable label for the permission. This may be the same as the machine-readable string, but should be formatted with an initial capital. You may capitalize words as needed. Unlike the machine-readable name, this string must be wrapped in t() so that the output may be translated. We will use "View the content creation options for registered users" because anonymous users do not have an account page to view.

3. An optional description of the permission. This string should be a complete sentence, and wrapped in t(). Drupal's user interface guidelines encourage you to use this element if the permission needs special clarification, especially if permission is being given to untrustworthy users who could pose a security risk. For our example, we will clearly state: "Displays the content types that a user may create on the user's profile page." When we put these three parts together, our hook looks like the following code: /\*\* \* Implement hook\_permission(). \*/

function example\_permission() {

return array(

'view content creation permissions' => array(

'title' => t('View the content creation options for registered users'),

'description' => t('Displays the content types that a user may create on the user\'s profile page.'),

),

);

}

Enabling permissions programmatically

Drupal user roles and permissions are handled through configurations in the user interface. However, there may be use cases where your module needs to set or modify permissions. There is even a module called Secure Permissions (http://drupal.org/project/secure\_permissions) which disables the UI for editing roles and permissions and forces all settings to be defined in code.

module creates a page callback that should be visible by 'authenticated' but not 'anonymous' users. To activate this feature when the module is enabled, you can use hook\_enable() as follows: function example\_enable() { $permissions = array('view example page'); user\_role\_change\_permissions(DRUPAL\_AUTHENTICATED\_USER, $permissions); } This function goes into your module's .install file.

e. When the module is enabled, Drupal will add the view example page permission to the authenticated user role. You can (and normally should) do the reverse when the module is disabled: function example\_disable() { $permissions = array('view example page'); $roles = user\_roles(); // Since permissions can be set per role, remove our permission from // each role. foreach ($roles as $rid => $name) { user\_role\_revoke\_permissions($rid, $permissions); } } It is also possible to add/remove multiple permissions at the same time