Drupal notes:

Contents

[1. Database : drupal712 4](#_Toc323382350)

[1.1. Database user and password drupaluser/1234 4](#_Toc323382351)

[1.2. url:http://localhost:81/drupal7 4](#_Toc323382352)

[1.3. Site admin : admin/680628 4](#_Toc323382353)

[2. Create a git repo: 4](#_Toc323382354)

[2.1. Right click the folder, create repository 4](#_Toc323382355)

[2.2. Add files to repo 4](#_Toc323382356)

[2.3. Go on to github ,create a repo 4](#_Toc323382357)

[2.4. On desktop, create a remote with the url like 4](#_Toc323382358)

[2.5. https://cleancodenz@github.com/cleancodenz/DrupalDev.git 4](#_Toc323382359)

[2.6. Push to remote 4](#_Toc323382360)

[2.7. How to get rid off of message box 4](#_Toc323382361)

[2.7.1. Go to repository, then .git directory, some time it is hidden 4](#_Toc323382362)

[2.7.2. In the core group 4](#_Toc323382363)

[2.7.3. Add 4](#_Toc323382364)

[3. Beautiful url 4](#_Toc323382365)

[3.1. Enable clean url first 4](#_Toc323382366)

[3.2. Install pathAuto module 5](#_Toc323382367)

[3.3. Install token module 5](#_Toc323382368)

[3.4. Administration:config:search and meta data:url alias:pattern tab 5](#_Toc323382369)

[3.5. For all previous contents, enable the automatic url 5](#_Toc323382370)

[4. Theme designing 5](#_Toc323382371)

[4.1. Folder is :sites/all/themes/mytheme 5](#_Toc323382372)

[4.2. .info file: sites/all/themes/mytheme/mytheme.info 5](#_Toc323382373)

[4.3. Defaults 5](#_Toc323382374)

[4.4. Regions in content 5](#_Toc323382375)

[4.4.1. Must have regions[content] = Content 5](#_Toc323382376)

[4.4.2. Output of region 5](#_Toc323382377)

[4.4.3. There are template (.tpl.php) files available for rendering individual blocks 5](#_Toc323382378)

[4.4.4. Order of regions definition 5](#_Toc323382379)

[4.4.5. Manually set content of a region using drupal\_set\_content('header', 'Welcome!') 6](#_Toc323382380)

[4.5. Color scheme, color module integration 6](#_Toc323382381)

[4.6. Clearing the cache: 6](#_Toc323382382)

[4.7. Template.php 6](#_Toc323382383)

[4.8. Add javascripts through behaviours 6](#_Toc323382384)

[5. Blue print 6](#_Toc323382385)

[5.1. Source file structure 6](#_Toc323382386)

[5.1.1. /blueprint/ css folder 6](#_Toc323382387)

[5.1.2. Main css files /blueprint/ 6](#_Toc323382388)

[6. Useful links 6](#_Toc323382389)

[6.1. Drupal 7 examples : 6](#_Toc323382390)

[6.2. Drupal 7 api documentation 6](#_Toc323382391)

[7. Key functions: 6](#_Toc323382392)

[8. Entiy Bundle field 13](#_Toc323382393)

[9. Node API 13](#_Toc323382394)

[10. Form API 14](#_Toc323382395)

[11. Structure (big picture) 14](#_Toc323382396)

[11.1. Content: 14](#_Toc323382397)

[11.1.1. Content type 14](#_Toc323382398)

[11.1.1.1. Edit 14](#_Toc323382399)

[11.1.1.1.1. There might be custom settings 14](#_Toc323382400)

[11.1.1.1.2. Submission form settings 14](#_Toc323382401)

[11.1.1.1.3. Publish options 14](#_Toc323382402)

[11.1.1.1.4. Display settings 14](#_Toc323382403)

[11.1.1.1.5. Comments Settings 14](#_Toc323382404)

[11.1.1.1.6. Menu Settings 14](#_Toc323382405)

[11.1.2. Manage Fields 15](#_Toc323382406)

[11.2. Menus 15](#_Toc323382407)

[11.2.1. Main menu 15](#_Toc323382408)

[11.2.2. Navigation menu 15](#_Toc323382409)

[11.2.3. Management menu 15](#_Toc323382410)

[11.2.4. User menu 15](#_Toc323382411)

[11.2.5. API 15](#_Toc323382412)

[11.2.5.1. Hook\_menu 15](#_Toc323382413)

[11.2.5.2. Navigation 16](#_Toc323382414)

[11.3. Block 16](#_Toc323382415)

[11.3.1. Pages: 16](#_Toc323382416)

[11.3.2. Content types 16](#_Toc323382417)

[11.3.3. Roles 17](#_Toc323382418)

[11.3.4. Users 17](#_Toc323382419)

[11.3.5. API 17](#_Toc323382420)

[11.3.5.1. Path: sites/all/modules/modulename 17](#_Toc323382421)

[11.3.5.2. Info file : sites/all/modules/modulename/modulename.info 17](#_Toc323382422)

[11.3.5.3. Module file : sites/all/modules/modulename/modulename.module 17](#_Toc323382423)

[11.3.5.4. Modulename\_help 17](#_Toc323382424)

[11.3.5.5. hook\_block\_info() 17](#_Toc323382425)

[11.3.5.6. hook\_menu() 17](#_Toc323382426)

[11.3.5.6.1. $items['admin/config/content/current\_posts'] 17](#_Toc323382427)

[11.4. Taxonomy 19](#_Toc323382428)

[12. People 19](#_Toc323382429)

[12.1. Role 19](#_Toc323382430)

[12.2. User 19](#_Toc323382431)

[12.3. Permission API 19](#_Toc323382432)

[12.3.1. hook\_permission 19](#_Toc323382433)

[13. Modules: Program Interface, APIs 20](#_Toc323382434)

[14. Hooks 20](#_Toc323382435)

# Database : drupal712

## Database user and password drupaluser/1234

## <url:http://localhost:81/drupal7>

## Site admin : admin/680628

## Useful urls:

### Useful

http://sixrevisions.com/web-development/drupal-modules-site-builds/

### Modules download

http://drupal.org/project/modules

### Examples download

http://drupal.org/project/examples

# Create a git repo:

## Right click the folder, create repository

## Add files to repo

## Go on to github ,create a repo

## On desktop, create a remote with the url like

## https://cleancodenz@github.com/cleancodenz/DrupalDev.git

## Push to remote

## How to get rid off of message box

Git unlock index or continue

### Go to repository, then .git directory, some time it is hidden

### In the core group

### Add

Following :

autocrlf = input   
safecrlf = false

# Beautiful url

## Enable clean url first

It might be turned off for apache, in wamp, apache, apache modules – enable rewrite\_module

Still the clean url test will fail if the install is not using a top level domain name but a subdirectory then RewriteBase should be changed to the subdirectory name

RewriteBase /mysite

In my case /mysite is /drupal7 as my install url is <http://localhost:81/drupal7/>

## Install pathAuto module

## Install token module

## Administration:config:search and meta data:url alias:pattern tab

## For all previous contents, enable the automatic url

# Theme designing

## Folder is :sites/all/themes/mytheme

## .info file: sites/all/themes/mytheme/mytheme.info

## Defaults

These defaults apply as a group. In other words, overriding a region with regions[sub\_header] = Sub-header will omit the rest of the default regions. To gain them back, they must be redefined. This also applies to stylesheets. Even though it's not technically in a group, defining another stylesheet will prevent "style.css" from being included unless it is redefined

## Regions in content

### Must have regions[content] = Content

### Output of region

Internal names are converted into region variables inside the "page.tpl.php" template automatically. In the above example, the [header] region will output all the blocks assigned to it through the $header variable in Drupal 6, or $page['header'] in Drupal 7.

Name of region will appear on blocks management page

### There are template (.tpl.php) files available for rendering individual blocks

### Order of regions definition

The order in which the regions are defined will be reflected in the block configuration table

### Manually set content of a region using drupal\_set\_content('header', 'Welcome!')

## Color scheme, color module integration

## Clearing the cache:

admin/config/development/performance-Clear all cache to reflect the new theme changes

Some time the debug does not work or render arrays are not updating with the code change, it might be because of caching.

## Template.php

Preprocessors for generating variables before they are merged with the mark up in side .tpl.php files

The only files that are required in the theme's root is the .info file and template.php. This is cool because you can put all your .tpl.php files in a templates/ directory and Drupal will still discover them

## Add javascripts through behaviours

# Blue print

## Source file structure

### /blueprint/ css folder

### Main css files /blueprint/

Screen.css, ie.css, print.css,

/blueprint/plugin and /blueprint/src/ will be used by main css files

# Useful links

## Drupal 7 examples :

http://api.drupal.org/api/examples/7

## Drupal 7 api documentation

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

# Key functions:

* 1. Node\_view(), /modules/node/node.module

$node->

vid 6

uid 1

title First listing

log

status 1

comment 2

promote 1

sticky 0

nid 6

type cleancode\_realty

language und

created 1331669629

changed 1331669629

tnid 0

translate 0

revision\_timestamp 1331669629

revision\_uid 1

body Array [1]

und Array [1]

0 Array [5]

value This is first listing

summary

format filtered\_html

safe\_value <p>This is first listing</p>\n

safe\_summary

cleancode\_realty\_color Array [1]

cleancode\_realty\_quantity Array [1]

cleancode\_realty\_image Array [1]

und Array [1]

0 Array [13]

fid 2

alt

title

width 1024

height 768

uid 1

filename Lighthouse.jpg

uri public://Lighthouse.jpg

filemime image/jpeg

filesize 561276

status 1

timestamp 1331669629

rdf\_mapping Array [0]

rdf\_mapping Array [9]

cid 0

last\_comment\_timestamp 1331669629

last\_comment\_name <Uninitialized>

last\_comment\_uid 1

comment\_count 0

name admin

picture 0

data b:0;

* 1. node\_build\_content($node, $view\_mode = 'full', $langcode = NULL) /modules/node/node.module
     1. get node base , node\_content
     2. run node\_content\_view, on the previous base

after the fields prepare view, $node->content is

content Array [6]

body Array [16]

#theme field

#weight 1

#title Body Example Description

#access true

#label\_display hidden

#view\_mode cleancode\_realty\_list

#language und

#field\_name body

#field\_type text\_with\_summary

#field\_translatable 0

#entity\_type node

#bundle cleancode\_realty

#object stdClass

#items Array [1]

0 Array [5]

value This is first listing

summary

format filtered\_html

safe\_value <p>This is first listing</p>\n

safe\_summary

#formatter text\_summary\_or\_trimmed

0 Array [1]

cleancode\_realty\_color Array [16]

#theme field

#weight 2

#title The colors available for this object.

#access true

#label\_display hidden

#view\_mode cleancode\_realty\_list

#language und

#field\_name cleancode\_realty\_color

#field\_type text

#field\_translatable 0

#entity\_type node

#bundle cleancode\_realty

#object stdClass

#items Array [1]

0 Array [3]

value ccc

format <Uninitialized>

safe\_value ccc

#formatter cleancode\_realty\_colors

0 Array [2]

cleancode\_realty\_image Array [16]

#theme field

#weight 4

#title Upload an image:

#access true

#label\_display hidden

#view\_mode cleancode\_realty\_list

#language und

#field\_name cleancode\_realty\_image

#field\_type image

#field\_translatable 0

#entity\_type node

#bundle cleancode\_realty

#object stdClass

#items Array [1]

0 Array [13]

#formatter image

0 Array [4]

#pre\_render Array [1]

#entity\_type node

#bundle cleancode\_realty

After links added to $node->content

links Array [5]

#theme links\_\_node

#pre\_render Array [1]

0 drupal\_pre\_render\_links

#attributes Array [1]

class Array [2]

node Array [3]

#theme links\_\_node\_\_node

#links Array [0]

#attributes Array [1]

comment Array [3]

#theme links\_\_node\_\_comment

#links Array [1]

comment\_forbidden Array [2]

title <a href="/drupal7/user/login?destination=node/6%23comment-form">Log in</a> or <a href="/drupal7/user/register?destination=node/6%23comment-form">register</a> to post comments

html true

#attributes Array [1]

class Array [2]

0 links

1 inline

If there is custom view, finally the format of build which was content before

$build Array [11]

body Array [16]

cleancode\_realty\_color Array [16]

cleancode\_realty\_image Array [16]

#pre\_render Array [1]

#entity\_type node

#bundle cleancode\_realty

links Array [5]

#theme node

#node stdClass

#view\_mode cleancode\_realty\_list

#language en

* + 1. field\_attach\_prepare\_view($entity\_type, $entities, $view\_mode, $langcode = NULL)

Field structure

entity\_types Array [1]

translatable 0

settings Array [0]

storage Array [5]

foreign keys Array [1]

indexes Array [1]

id 2

field\_name body

type text\_with\_summary

module text

active 1

locked 0

cardinality 1

deleted 0

columns Array [3]

bundles Array [1]

Instance structure

$instance Array [13]

label Body Example Description

widget Array [4]

type text\_textarea\_with\_summary

settings Array [2]

rows 20

summary\_rows 5

weight 0

module text

settings Array [3]

display\_summary true

text\_processing 1

user\_register\_form false

display Array [3]

default Array [5]

label hidden

type text\_default

settings Array [0]

module text

weight 0

teaser Array [5]

label hidden

type text\_summary\_or\_trimmed

settings Array [1]

module text

weight 0

cleancode\_realty\_list Array [5]

label hidden

type text\_summary\_or\_trimmed

settings Array [1]

module text

weight 1

required false

description

length 0

default\_value <Uninitialized>

id 18

field\_id 2

field\_name body

entity\_type node

bundle cleancode\_realty

deleted 0

* + 1. s
    2. ddd
  1. The render array is

$renderable\_array Array [2]

explanation Array [1]

node\_list Array [2]

0 Array [12]

body Array [16]

cleancode\_realty\_color Array [16]

cleancode\_realty\_image Array [16]

#pre\_render Array [1]

#entity\_type node

#bundle cleancode\_realty

links Array [5]

#theme node

#node stdClass

#view\_mode cleancode\_realty\_list

#language en

#contextual\_links Array [1]

1 Array [12]

* 1. drupal\_deliver\_html\_page($page\_callback\_result), /includes/common.inc
     1. drupal\_add\_http\_header('Content-Type', 'text/html; charset=utf-8')
     2. drupal\_add\_http\_header('Content-Language', $language->language);
     3. drupal\_render\_page($page\_callback\_result);
     4. drupal\_page\_footer();
  2. drupal\_render\_page($page) , /includes/common.inc
     1. drupal\_set\_page\_content($page);

in the end $page will be like

$page Array [10]

#show\_messages true

#theme page

#theme\_wrappers Array [1]

0 html

#type page

help Array [4]

system\_help Array [4]

#sorted true

#theme\_wrappers Array [1]

#region help

content Array [4]

system\_main Array [5]

#sorted true

#theme\_wrappers Array [1]

#region content

sidebar\_first Array [5]

system\_navigation Array [6]

user\_login Array [27]

#sorted true

#theme\_wrappers Array [1]

#region sidebar\_first

footer\_firstcolumn Array [4]

current\_posts\_current\_posts Array [6]

#sorted true

#theme\_wrappers Array [1]

#region footer\_firstcolumn

footer Array [4]

system\_powered-by Array [5]

#sorted true

#theme\_wrappers Array [1]

#region footer

page\_top Array [3]

toolbar Array [3]

#theme\_wrappers Array [1]

#region page\_top

* + 1. drupal\_render($page)
    2. ddd
  1. dddd

# Entiy Bundle field

In an example of content type ‘article’

Entity-> node

Bundle-> article (subtype, or content type, or node type)

Fields -> fields attached in the bundle

# Form API

* 1. Through form constructore function $form and $formstate

$form[‘itemname’]= array(

'#type' => 'item',

'#title' => t('A form with nothing but a textfield')

)

Each of $form field will create an item on the form

$form['submit'] = array(

'#type' => 'submit',

'#value' => 'Submit',

);

A submit button

# Rendering

Renders HTML given a structured array tree.

Renderable arrays have two kinds of key/value pairs: properties and children. Properties have keys starting with '#' and their values influence how the array will be rendered. Children are all elements whose keys do not start with a '#'. Their values should be renderable arrays themselves, which will be rendered during the rendering of the parent array. The markup provided by the children is typically inserted into the markup generated by the parent array.

## Property:’#Property’

### #type

The Element type. If this array is an element, this will cause the default element properties to be loaded, so in many ways this is shorthand for a set of predefined properties which will have been arranged through hook\_element\_info().

As in the past, every Drupal "element" (see hook\_element\_info(), which was hook\_elements() in Drupal 6) is a type. So anything that core exposes as an element or that an installed module exposes is available. Looking through system\_element\_info() we see a pile of predefined #types, including page, form, html\_tag, value, markup, link, fieldset and many more. By convention, the #-properties used by these #types are documented with the respective theme function. So you can find out the properties used by #type => 'html\_tag' elements by checking out the documentation for theme\_html\_tag(). You can also create types and properties on the fly. It's the Wild West out there.

#### Form

#### Page

#### Ajax

By default, we don't want Ajax commands being rendered in the context of an HTML page, so we don't provide defaults for #theme or #theme\_wrappers. However, modules can set these properties (for example, to provide an HTML debugging page that displays rather than executes Ajax commands).

#### Html\_tag

#### Styles

#### Submit

#### Button

#### Image\_button

#### Textfield

#### Machine\_name

#### Password

#### Password\_confirm

#### Textarea

#### Radios

#### Radio

#### Checkboxes

#### Checkbox

#### Select

#### Weight

#### Date

#### File

#### Tableselect

#### Item

Form structure

#### Hidden

#### Value

#### Markup

#### Link

#### Fieldset

#### Vertical\_tabs

#### Container

#### Actions

#### Token

### #markup

The simplest property, this simply provides a markup string for #type => 'markup'

### #prefix/#suffix

A string to be prefixed or suffixed to the element being rendered

### #pre\_render

An array of functions which may alter the actual render array before it is rendered. They can rearrange, remove parts, set #printed = TRUE to prevent further rendering, etc.

### #post\_render

An array of functions which may operate on the rendered HTML after rendering. A #post\_render function receives both the rendered HTML and the render array from which it was rendered, and can use those to change the rendered HTML (it could add to it, etc.). This is in many ways the same as #theme\_wrappers except that the theming subsystem is not used.

### #theme

A single theme hook which will take full responsibility for rendering this array element, including its children. It has predetermined knowledge of the structure of the element. Note: #theme in Drupal 7 and #theme in Drupal 6 are not really related. If you just stop thinking about Drupal 6 here, you will have an easier time.

Basically, the '#theme' = 'function\_name' calls theme\_function\_name(), and other array values of the form '#var\_name' = $value in the same array are passed as arguments to the theme function.

There is a list of all the default theme hooks at

<http://api.drupal.org/api/drupal/modules--system--theme.api.php/group/themeable/7>

Stopped here

<https://drupal.org/node/930760>

<http://www.vkareh.net/blog/render-arrays>

### #theme\_wrappers

An array of theme hooks which will get the chance to add to the rendering after children have been rendered and placed into #children. This is typically used to add HTML wrappers around rendered children, and is commonly used when the children are being rendered recursively using their own theming information. It is rare to use it with #theme.

### #cache

Mark the array as cacheable and determine its expiration time, etc. Once the given render array has been rendered, it will not be rendered again until the cache expires. Caching uses standard Drupal cache\_get() and cache\_set() techniques. This is an array of

'keys' => an array of keys which will be concatenated to form the cache key.

'bin' => the name of the cache bin to be used (as in 'cache' or 'cache\_page', etc.

'expire' => a Unix timestamp indicating the expiration time of the cache.

'granularity' => a bitmask indicating the cache type. This should be DRUPAL\_CACHE\_PER\_PAGE, DRUPAL\_CACHE\_PER\_ROLE, or DRUPAL\_CACHE\_PER\_USER

Note that items marked with #cache will not be expired until cron runs, regardless of the expiration time used.

## Children:

The key of it does not start from #, the value of it is a render array too, it forms kind of hierarchical structure for elements

HTML generation for a renderable array, and the treatment of any children, is controlled by two properties containing theme functions, #theme and #theme\_wrappers.

## HTML generation:

### #theme

#theme is the theme function called first. If it is set and the element has any children, it is the responsibility of the theme function to render these children. For elements that are not allowed to have any children, e.g. buttons or textfields, the theme function can be used to render the element itself. If #theme is not present and the element has children, they are rendered and concatenated into a string by [drupal\_render\_children](http://api.drupal.org/api/drupal/includes%21common.inc/function/drupal_render_children/7)().

### #theme\_wrappers

The #theme\_wrappers property contains an array of theme functions which will be called, in order, after #theme has run. These can be used to add further markup around the rendered children; e.g., fieldsets add the required markup for a fieldset around their rendered child elements. All wrapper theme functions have to include the element's #children property in their output, as it contains the output of the previous theme functions and the rendered children.

### #cache

# Theming

A well-made Drupal module allows all elements of its presentation to be overridden by the theme of the site on which it is used. In order for this theme layer to be usable, a module must be written to take advantage of it.

## Basics: theme()

Normally the render array has #theme in there, then you also need hook\_theme as theme registry and theme implementation to be driven by the render array.

theme.inc: theme($hook, $variables = array())

<http://api.drupal.org/api/drupal/includes!theme.inc/function/theme/7>

The first argument to this function is the name of the theme hook. For instance, to theme a table, the theme hook name is 'table'. By default, this theme hook could be implemented by a function called 'theme\_table' or a template file called 'table.tpl.php', but [hook\_theme](http://api.drupal.org/api/drupal/modules%21system%21system.api.php/function/hook_theme/7)() can override the default function or template name.

The theme implementation knows what the variables structure is and export an html string to the caller, this is to allow the hook.tpl.php to have the same effect which is to result in an html string.

### $hook

The name of the theme hook to call. If the name contains a double-underscore ('\_\_') and there isn't an implementation for the full name, the part before the '\_\_' is checked. This allows a fallback to a more generic implementation. For example, if theme('links\_\_node', ...) is called, but there is no implementation of that theme hook, then the 'links' implementation is used. This process is iterative, so if theme('links\_\_contextual\_\_node', ...) is called, theme() checks for the following implementations, and uses the first one that exists:

links\_\_contextual\_\_node

links\_\_contextual

links

This allows themes to create specific theme implementations for named objects and contexts of otherwise generic theme hooks. The $hook parameter may also be an array, in which case the first theme hook that has an implementation is used. This allows for the code that calls theme() to explicitly specify the fallback order in a situation where using the '\_\_' convention is not desired or is insufficient.

### $variables

#### #variables as a render array

From a renderable array(with #theme or #theme\_wrappers keys in the array ), $variables will be retrieved from its other properties (properties other than #theme or#theme\_wrappers ) by mapping them to variables expected by the theme hook implementations.

First it looks ‘variables’ key in theme registry

If found match #$name key with variable $name in theme registry

If not found looks for 'render element' in theme registry, if found, pass $variables into $variables[the value of 'render element’ in theme registry] , basically it just inserts a new level of final array structure $variables .

#### #variables passed through

Merge $variables with theme registry ‘variables’ and 'render element’, if no value default values will be set .

## Hook\_theme:

In order to utilize a theme hook, your module first has to register that this exists

<?php  
function forum\_theme() {  
  return array(  
    'forums' => array(  
      'template' => 'forums',  
      'variables' => array('forums' => NULL, 'topics' => NULL, 'parents' => NULL, 'tid' => NULL, 'sortby' => NULL, 'forum\_per\_page' => NULL),  
    ),  
//...  
  );  
}  
?>

This registration tells us that a theme hook named forums is implemented. The default implementation is a template. Because there are different kinds of template engines, this registration does not include the extension used by the engine, though Drupal core only supports PHPTemplate templates for modules. These template files have the extension '.tpl.php'.

It also tells us that the forums theme function takes 6 variables (or arguments), and they all default to NULL. (All arguments must be given defaults as we have no way to assure that a theme('forums', ...) call will provide the proper information. If in doubt, make the default NULL). These arguments are translated into the named variables for the template. When calling this theme hook, an author might write:

<?php  
  $output = theme('forums', array('forums' => $forums, 'topics' => $topics, 'parents' => $parents, 'tid' => 17, 'sortby' => 'ASC', 'forums\_per\_page' => 25));  
?>

If the 'template' had been left off of the hook\_theme() definition, the theme hook's default implementation would be assumed to be a function named 'theme\_forums'.

## Default templates

When implemented as a template, the .tpl.php file is required. It should be in the same directory as the .module file (though the 'path' directive can be used to place these templates in another directory or a sub-directory).

Templates should be as much pure HTML as possible, but there are a few functions that are explicitly encouraged in your templates. First is the t() function. Modules should always provide proper translatability, and templates are no exception. Themers need to have the direct text to work with, and translators need all of the text to be passed through t(). Therefore, the use of t() is encouraged in templates.

Another function that is encouraged in templates is format\_date(). Since this function is, really, a presentation function, the presentation layer is the appropriate place for it. However, its use is somewhat arcane and difficult for people who are not familiar with PHP to use. Nonetheless, it should be used in templates.

For other functions, consider whether or not they really are needed at the presentation layer. If they are not, they can be used in the preprocessor layer. All templates may have an optional preprocess function, named template\_preprocess\_HOOK. For example, for our forums theme hook above, its preprocess function will be named template\_preprocess\_forums().

The purpose of the preprocess function is to perform any logic that needs to be performed to make data presentable, and to sanitize any data so that it is safe to output. It is critically important that your output be secure and not contain XSS (Cross Site Scripting) vulnerabilities. And since data that is output often comes from users, this data must be sanitized before it is output. Since we assume that themers are not necessarily developers, we must assume that they are not going to fully understand how to do this; but that's ok, because we can sanitize data in the preprocess function by running it through check\_plain, check\_markup, filter\_xss\_admin or other output sanitizing functions.

Here is a simple example [from Poll module](http://api.drupal.org/api/function/template_preprocess_poll_bar/7):

<?php  
function template\_preprocess\_poll\_bar(&$variables) {  
  if ($variables['block']) {  
    $variables['theme\_hook\_suggestions'][] = 'poll\_bar\_\_block';  
  }  
  $variables['title'] = check\_plain($variables['title']);  
  $variables['percentage'] = round($variables['votes'] \* 100 / max($variables['total\_votes'], 1));  
}  
?>

First, note that the preprocessor function takes a reference to an array of variables. This array will be seeded with the arguments that were sent via the theme() and named by the 'arguments' section of the hook registration. Since this is a reference, simply modifying this array is enough to transport those changes to the template that accompanies it.

This example illustrates three important concepts:

The 'title' field is unsafe, because it comes from user input. It is run through check\_plain so that the template may safely output it.

The theme hook receives the total number of votes and the number of votes for just that item, but the template wants to display a percentage. That kind of work shouldn't be done in a template; instead, the math is performed here. The variables still exist, though; a themer overriding this could easily choose to display something other than a percentage.

The special variable 'theme\_hook\_suggestions' can be used to provide alternative template files to use. This is an array of hook names, and it is last in, first out, which means the last item added to the array will be the first one used. If a template doesn't exist, it will use the next one in the list. The special double underscore in this example is a shorthand way of indicating this– it does not look for a hook name with two underscores or a template with two dashes, rather, it will look for (in this template file case) first poll-bar-block.tpl.php, and failing to find that it will drop the part after the double underscores and it will look for poll-bar.tpl.php.

[Theme Developer module](http://drupal.org/project/devel_themer), which is part of the devel project, includes a template log feature which outputs at bottom of page all the template files which could have been used to theme the current page. This may be handy while building your module, but even more so when themeing a site. Also, don't miss its Themer information popup.

Quick note: Template files should be named with hyphens instead of underscores. If the theme hook is 'forum\_list', the template file should be named 'forum-list.tpl.php'. Also if you are planning to use preprocess functions, do not use hyphens in your theme hook name.

## Theme functions:

Drupal allows you to use functions for your default theme implementations. This is somewhat faster performance than loading template files. New in Drupal 7, theme functions can have preprocess functions just like templates. Copying and modifying template files is still considered more friendly for themers than overriding theme functions in template.php.

Theme functions are named by prepending 'theme\_' to the name of the hook. The arguments given to theme('hook') will be passed straight through, unaltered. The defaults specified in the hook registration will not be provided here; they must be provided as normal PHP argument defaults.

<?php  
/\*\*  
\* Implements hook\_theme().  
\*/  
function dashboard\_theme() {  
  return array(  
    'dashboard' => array(  
      'render element' => 'element',  
    ),  
// ...  
  );  
}  
?>

And the function:

<?php  
/\*\*  
\* Returns HTML for the entire dashboard.  
\*  
\* @param $variables  
\*   An associative array containing:  
\*   - element: A render element containing the properties of the dashboard  
\*     region element, #dashboard\_region and #children.  
\*  
\* @ingroup themeable  
\*/  
function theme\_dashboard($variables) {  
  extract($variables);  
  drupal\_add\_css(drupal\_get\_path('module', 'dashboard') . '/dashboard.css');  
  return '<div id="dashboard">' . $element['#children'] . '</div>';  
}  
?>

Themers can override this function by creating a function named themename\_dashboard().

## Dynamic theming:

In addition to being able to specify alternate templates in a preprocess function, you may also create dynamic theming implementations using wildcards. There are two steps in this process.

First, in hook\_theme, you can specify a pattern. Patterns are simple regular expressions. ^ (beginning of line) is assumed, but $ (end of line) is not. To signify the dynamic portion of the pattern, a double underscore is the general convention; this is not required but it is highly recommended.

Second, when calling the theme() function, instead of a string for the first argument you may pass an array. This array is much like theme\_hook\_suggestions above, but this one is *first in, first out* so the first one seen will be used.

For a practical example, the module Views likes to let each view be themed by name. Upon registration, the hook 'views\_view' would register with the pattern 'views\_view\_\_'. When theming that view, Views would call:

$output = theme(array("views\_view\_\_$view->name", 'views\_view'), $view);

Views will implement a default view for views\_view; if a theme registers 'views\_view\_\_foo' and Views themes a view named 'foo', the specific override will activate and be used instead. Unlike the 'theme\_hook\_suggestions' variable in the preprocessor function, this works for both theme functions as well as templates.

More on [preprocess functions](http://drupal.org/node/223430).

## theme('table') and theme('item\_list'):

Drupal provides a few helpers to build complex HTML constructs easily. These are very useful features, and by using them it is easy to create a consistent look on tables and lists. The downside is that they are not readily accessible to a themer. Instead, they place code that should be at the presentation layer into the logic layer, and only advanced themers are able to do anything with it.

These functions are more acceptable for administration pages.

When creating output that is likely to be changed, it is best to avoid the use of these constructs and create the tables and lists with real HTML code. The forum themes are perfect examples of how to accomplish this and still create HTML code that is consistent

An even bigger problem with using common provided theme functions such as theme('item\_list') is that a theme probably does not want to override every theme\_item\_list() for every list output by Drupal anywhere, but only the one in our module. Therefore, instead of simply theme('item\_list'), we can use theme('item\_list\_\_mymodule\_\_main', $items) which would make it possible for a theme to implement themename\_item\_list\_\_mymodule\_\_main() or themename\_item\_list\_\_mymodule() to override item\_list() only for that specific instance.

An alternative to the double underscore convention is to explicitly name each hook that could be used, in the order of our preference, in an array:

<?php  
  theme(array('somemodule\_itemlist\_alternative', 'item\_list'), $items);  
?>

## Having changes to code take effect:

When new theming functions are added, we must cleare the [theme registry](http://drupal.org/node/173880#theme-registry) to see them.

# Structure (big picture)

## Content:

### Content type

Structure->Content Type

#### Edit

##### There might be custom settings

##### Submission form settings

##### Publish options

##### Display settings

##### Comments Settings

##### Menu Settings

The menus available to place links in for this content type.

When content is created, menu tab will ask for title of the menu and where to put this new content on the menu hierachy, this is controled by the menu settings of content type

##### Main menu

Is the system one

### Manage Fields

There are three compulsary fields that come from node base type

Title, label of this can be changed when the type is created

Url path settings,

Body,

Plus custom fields

### Manage Display

If a field type has multiple formatters registered against this field, then it can be changed here.

### Field API

This is used by content type api (node type api)

This creates a field type(label in field\_info as name , field type, field length,field null in field\_schema, ) that can be used in defining a real instantiated field in a particular content type, once is created, a new table will be created in the back end, as field\_data\_fieldname, the last field of this table is decided by this new field type.

Providing a field requires:

\* Defining a field:

o hook\_field\_info()

o hook\_field\_schema()

o hook\_field\_validate()

o hook\_field\_is\_empty()

\* Defining a formatter for the field (the portion that outputs the field for display):

o hook\_field\_formatter\_info()

o hook\_field\_formatter\_view()

\* Defining a widget for the edit form:

o hook\_field\_widget\_info()

o hook\_field\_widget\_form()

Our module defines the field in field\_example\_field\_info(), field\_example\_field\_validate() and field\_example\_field\_is\_empty(). field\_example\_field\_schema() is implemented in field\_example.install.

Our module sets up a formatter in field\_example\_field\_formatter\_info() and field\_example\_field\_formatter\_view(). These are the API hooks that present formatted and themed output to the user.

And finally, our module defines the widet in field\_example\_field\_widget\_info() and field\_example\_field\_widget\_form(). The widget is the form element used to receive input from the user when the field is being populated.

#### field\_example\_3text\_validate($element, &$form\_state)

Validate the individual fields and then convert them into a single HTML RGB value as text.

#### hook\_field\_formatter\_info()

We need to tell Drupal that we have two different types of formatters for this field. The field formatter can be chosen on the content type editing page-manage display.

#### hook\_field\_formatter\_view()

Implementation of formatter or formatters

#### hook\_field\_info()

An array whose keys are field type names and whose values are arrays describing the field type, with the following key/value pairs

Provides the description of the field.

Label is the name in the drop down of field types.

#### hook\_field\_is\_empty()

hook\_field\_is\_emtpy() is where Drupal asks us if this field is empty. Return TRUE if it does not contain data, FALSE if it does. This lets the form API flag an error when required fields are empty.

#### hook\_field\_schema()

Defines the database schema of the field, using the format used by the Schema API.

The data we will store here is just one 7-character element, even though the widget presents the three portions separately.

All implementations of hook\_field\_schema() must be in the module's .install file

#### hook\_field\_validate ()

This hook gives us a chance to validate content that's in our field. We're really only interested in the $items parameter, since it holds arrays representing content in the field we've defined. If it doesn't validate, we add our own error notification to the $errors parameter

Validate this module's field data.

If there are validation problems, add to the $errors array (passed by reference). There is no return value.

#### hook\_field\_widget\_error()

hook\_field\_widget\_error() lets us figure out what to do with errors we might have generated in hook\_field\_validate(). Generally, we'll just call form\_error()

#### hook\_field\_widget\_form ()

hook\_widget\_form() is where Drupal tells us to create form elements for our field's widget.

You can specify the special validation functions instead hook\_field\_validate() like

'#element\_validate' => array('field\_example\_3text\_validate'),

We provide one of three different forms, depending on the widget type of the Form API item provided.

The 'field\_example\_colorpicker' and 'field\_example\_text' are essentially the same, but field\_example\_colorpicker adds a javascript colorpicker helper.

field\_example\_3text displays three text fields, one each for red, green, and blue. However, the field type defines a single text column, rgb, which needs an HTML color spec. Define an element validate handler that converts our r, g, and b fields into a simulated single 'rgb' form element

#### hook\_field\_widget\_info()

in content type field definition page, the array of this hook will appear as dropdown of WIdGET column.

These widget types will eventually show up in hook\_field\_widget\_form, where we will have to flesh them out.

### Custom content type API

Structure->Content Type

Example defining a node type in code.

This is an example outlining how a module can be used to define a new node type. Our example node type will allow users to specify multiple "colors", a "quantity" and an "image" for their nodes; some kind of rudimentary inventory-tracking system, perhaps?

The basic pattern for defining a node type is to tell Drupal about the node's field types, and view modes. Drupal will then take over and manage the storage for this node type. This differs from Drupal 6, where we would have to handle all the database storage ourselves in the module.

Remember that most node types do not require any custom code, as one simply creates them using the Drupal user interface. Creating a node like this in code is a special case.

Since we only have to define our node type once, most of the code required to do this is moved to the node type's .install file. Drupal 7 has us defining most of our node structure in arrays, and passing those to node\_type\_save(). We use hook\_install() as a convenient place to define these types, and hook\_uninstall() as a convenient place to not only uninstall the data contained in these nodes, but also remove the node types from Drupal's knowledge.

In previous versions of Drupal, "teaser" and "page" were node view modes. In Drupal 7 we can define custom view modes to let the node know how it should return it's data. This module declares a custom view mode called "example\_node\_list".

#### hook\_entity\_info\_alter()

We need to modify the default node entity info by adding a new view mode to be used in functions like node\_view() or node\_build\_content().

#### hook\_field\_formatter\_info()

#### hook\_field\_formatter\_view()

@todo: We need to provide a formatter for the colors that a user is allowed to enter during node creation

#### hook\_help()

#### hook\_install()(.install)

This hook is called when the user enables the module for the first time (or on subsequent enables after the module has been uninstalled). So it's a good place to define our new node type.

We will:

\* Add the body field.

\* Configure the body field.

\* Create color, quantity, and image fields.

\* Create color, quantity, and image instances.

See also

node\_type\_set\_defaults()

field\_info\_instance()

field\_update\_instance()

field\_create\_field()

field\_create\_instance()

#### hook\_menu()

We are providing a default page to illustrate the use of our custom node view mode that will live at <http://example.com/?q=examples/node_example>

node\_example\_page()

Callback that builds our content and returns it to the browser.

This callback comes from hook\_menu().

Return value

a build array a renderable array showing a list of our nodes.

See also

node\_load()

node\_view()

node\_example\_field\_formatter\_view()

#### hook\_theme()

This lets us tell Drupal about our theme functions and their arguments.

A custom theme function.

theme\_example\_node\_color($variables)

By using this function to format our node-specific information, themes can override this presentation if they wish. This is a simplifed theme function purely for illustrative purposes

#### hook\_uninstall(.install)

This hook is called when the user not only has disabled the module, but also uninstalls it from the 'uninstall' tab in the module page.

So it's a perfect time to remove our fields and instances and new node type from the database.

\_node\_example\_installed\_fields()

Returns a structured array defining the fields created by this content type.

This is factored into this function so it can be used in both node\_example\_install() and node\_example\_uninstall().

Return value

An associative array specifying the fields we wish to add to our new node type.

Related topics

\_node\_example\_installed\_instances()

Returns a structured array defining the instances for this content type.

The instance lets Drupal know which widget to use to allow the user to enter data and how to react in different view modes. We are going to display a page that uses a custom "node\_example\_list" view mode. We will set a cardinality of three allowing our content type to give the user three color fields.

This is factored into this function so it can be used in both node\_example\_install() and node\_example\_uninstall().

Return value

An associative array specifying the instances we wish to add to our new node type.

### Custom field type API(Node API)

What node api does is create independent fields that can be enabled to attach to an existing content type

Hook\_form\_alter will attach the new fields to existing editor form and configuration in node type edit form to allow administrator to config.

It is different from a new node type which has its own defined fields

#### Hook\_form\_alter

By implementing this hook, we're able to modify any form. We'll only make changes to two types: a node's content type configuration and edit forms.

We need to have a way for administrators to indicate which content types should have our rating field added. This is done by inserting radios in the node's content type configuration page.

Changes made by this hook will be shown when editing the settings of any content type.

Optionally, hook\_form\_FORM\_ID\_alter() could be used with the function name nodeapi\_example\_form\_node\_type\_form\_alter

#### Hook\_node\_delete

When a node is deleted, we need to remove all related records from our table, including all revisions. For the delete operations we use node->nid

#### Hook\_node\_insert

As a new node is being inserted into the database, we need to do our own database inserts

#### Hook\_node\_insert

Loads the rating information if available for any of the nodes in the argument list.

#### Hook\_node\_update

As an existing node is being updated in the database, we need to do our own database updates.

This hook is called when an existing node has been changed. We can't simply update, since the node may not have a rating saved, thus no database field. So we first check the database for a rating. If there is one, we update it. Otherwise, we call nodeapi\_example\_node\_insert() to create one

#### Hook\_node\_validate

Check that the rating attribute is set in the form submission, since the field is required. If not, send error message

#### Hook\_node\_view

This is a typical implementation that simply runs the node text through the output filters.

Finally, we need to take care of displaying our rating when the node is viewed. This operation is called after the node has already been prepared into HTML and filtered as necessary, so we know we are dealing with an HTML teaser and body. We will inject our additional information at the front of the node copy.

Using node API 'hook\_node\_view' is more appropriate than using a filter here, because filters transform user-supplied content, whereas we are extending it with additional information

#### Hook\_schema(.install)

#### Hook\_uninstall(.install)

We need to clean up our variables data when uninstalling our module.

Our implementation of nodeapi\_example\_form\_alter() automatically creates a nodeapi\_example\_node\_type\_<contentType> variable for each node type the user wants to rate.

To delete our variables we call variable\_del for our variables' namespace, 'nodeapi\_example\_node\_type\_'. Note that an average module would have known variables that it had created, and it could just delete those explicitly. For example, see render\_example\_uninstall(). It's important not to delete variables that might be owned by other modules, so normally we would just explicitly delete a set of known variables.

hook\_uninstall() will only be called when uninstalling a module, not when disabling a module. This allows our data to stay in the database if the user only disables our module without uninstalling it.

#### Hook\_theme

#### Custom theme function

theme\_nodeapi\_example\_rating($variables)

A custom theme function.

By using this function to format our rating, themes can override this presentation if they wish; for example, they could provide a star graphic for the rating. We also wrap the default presentation in a CSS class that is prefixed by the module name. This way, style sheets can modify the output without requiring theme code.

## Menus

### Main menu

This is the system one, some time due to theme, it might not be expandable for children nodes.

But then you view children nodes, the navigation crumbs will always be right.

In some themes the main menu can be hard coded, that is without enabling the main menu on any block, you can still can see main menu, like the default theme.

### Navigation menu

This is the system one too

Expandable children can be displayed, menus defined in hook\_menu will appear in Navigation block automatically

### Management menu

This is the system one

### User menu

This is the system one

### Custom menu

Admin-structure-menu-new menu

Edit menu or Add linkm while hook\_menu provides the urls that can be added to the menu, the same is for system menus.

### API

#### Hook\_menu

Menu items are defined by placing them in an $items array. The array key(in this case 'menu\_example') is the path that defines the menu router entry, so the page will be accessible from the URL example.com/menu\_example.

##### Menu hierarchy

The key of $items that is url as well, decides it is a root level or n level of menu item

The menu will form the links into a tree structure according its route

A

a/b

a/c

##### Type

Registered as : $items['abc/def']

\* MENU\_NORMAL\_ITEM: Normal menu items show up in the menu tree and can be moved/hidden by the administrator.

\* MENU\_CALLBACK: Callbacks simply register a path so that the correct information is generated when the path is accessed, but not in the visible menu items.

\* MENU\_SUGGESTED\_ITEM: Modules may "suggest" menu items that the administrator may enable.

\* MENU\_LOCAL\_ACTION: Local actions are menu items that describe actions on the parent item such as adding a new user or block, and are rendered in the action-links list in your theme.

\* MENU\_LOCAL\_TASK: Local tasks are menu items that describe different displays of data, and are generally rendered as tabs.

\* MENU\_DEFAULT\_LOCAL\_TASK: Every set of local tasks should provide one "default" task, which should display the same page as the parent item.

If the "type" element is omitted, MENU\_NORMAL\_ITEM is assumed.

##### Optional path arguments

Registered as : $items['abc/def']

You can also pass optional path arguments as parameters to the page or theme call back functions, like when 'abc/def/123/foo' is requested, and only ‘'abc/def’ is registered, $ghi will be '123' and $jkl will be 'foo'.

##### Page arguments

array(1, 'foo'), the second of path sec will be used as argument in the argument list of call back, if there is any other optional path arguments, they will be attached to the argument list too.

when path 'abc/def/bar/baz' is requested, page call back will be called with 'def', 'foo', 'bar' and 'baz' as arguments, in that order

##### Access callback

If the page is meant to be accessible to all users, you can set 'access callback' to TRUE. This bypasses all access checks

For a permissioned menu entry, we provide an access callback which determines whether the current user should have access. The default is user\_access(), which we'll use in this case. Since it's the default, we don't even have to enter it.

'access callback' => 'user\_access'

##### Access arguments

The 'access arguments' are passed to the 'access callback' to help it, do its job. In the case of user\_access(), we need to pass a permission as the first argument

'access arguments' => array('access protected menu example')

##### Page callback

page callback should be a function, or a general one like : drupal\_get\_form

##### Menu name

We can choose which menu gets the link. The default is 'navigation', if it is set main mneu, the link will appear on main menu

##### Menu tabs

A menu entry with tabs.

For tabs we need at least 3 things:

1. A parent MENU\_NORMAL\_ITEM menu item (menu\_example/tabs in this example.)

2. A primary tab (the one that is active when we land on the base menu).This tab is of type MENU\_DEFAULT\_LOCAL\_TASK.

3. Some other menu entries for the other tabs, of type MENU\_LOCAL\_TASK

##### Title callback

The menu title can be dynamically created by using the 'title callback' which by default is t(). So, normally do not need t(‘title’), as default call back already calling t()

##### Place holder or wildcards

array(numeric\_position\_value) as the value for 'page arguments'. The numeric\_position\_value is the zero-based index of the portion of the URL which should be passed to the 'page callback'

Drupal provides magic placeholder processing as well, so if the placeholder is '%menu\_example\_arg\_optional', the function menu\_example\_arg\_optional\_load($arg) will be called to translate the path argument to a more substantial object. $arg will be the value of the placeholder. then the return value of menu\_example\_id\_load($arg) will be passed to the 'page callback'.

In addition, if (in this case) menu\_example\_arg\_optional\_to\_arg() exists ,when no arg is provided then a menu link with default arg can be created using the results of that function as a default for %menu\_example\_arg\_optional.

##### Hook\_menu\_alter

array(numeric\_position\_value) as the value for 'page arguments'. The numeric\_position\_value is the zero-based index of the portion of the URL which should be passed to the 'page callback'

#### Navigation Block

Navigation is a block which can be placed in the structure-blocks

Menus defined in hook\_menu will be added to Navigation automatically

In the menus, new menu blocks like Navigation can be added, within this menu block you can add other links to this menu block, the links must be provided, it is not selected

## Block

Blocks are placed in regions which come from theme

A block can be installed as module with block type

A menu is a block, so it can be placed in regions

### Pages:

The block can be placed on only certain pages

### Content types

Only show on certain content types

### Roles

### Users

### API

#### Path: sites/all/modules/modulename

#### Info file : sites/all/modules/modulename/modulename.info

#### Module file : sites/all/modules/modulename/modulename.module

#### Modulename\_help

At admin module list, the help link is provided

#### hook\_block\_info()

Provided the human name for this block, the name will appear in the admin modules list

#### hook\_menu()

##### $items['admin/config/content/current\_posts']

adds a config entry for this block on Admin/Configuration page

###### Title :

The title of configuration page, the title of link on the admin configs page

###### Description:

The description of the link on the admin configs page

###### page callback

The function call if the link is clicked, here is drupal\_get\_form

###### page arguments

The arguments of page call back

Array first item is the form implementation function name

###### Form implementation of above setting

$form['current\_posts\_max'] = array(

'#type' => 'textfield',

'#title' => t('Maximum number of posts'),

'#default\_value' => variable\_get('current\_posts\_max', 3),

'#size' => 2,

'#maxlength' => 2,

'#description' => t('The maximum number of links to display in the block.'),

'#required' => TRUE,

);

###### Form validation for form implementation

$form\_state['values']['current\_posts\_max']; the way to access the variable on form

###### access arguments

Which access control to use hook\_permission()

?????

###### type

MENU\_NORMAL\_ITEM,

Menu type -- A "normal" menu item that's shown in menu and breadcrumbs.

##### $items[modulename]

###### Type

MENU\_CALLBACK

Menu type -- A hidden, internal callback, typically used for API calls

###### Page call back page implementation

Set the title of page

drupal\_set\_title('Johnson current posts11111111');

Get the contents, it is an object or array

Set the theme of items

$page\_array['current\_posts\_arguments'] = array(

'#title' => t('All posts from the last week'),

'#items' => $items,

//Theme hook with suggestion.

'#theme' => 'item\_list\_\_current\_posts',

#### hook\_block\_view()

This is the implementation of view of block normally with a more link to the page

##### set the title of block

$block['subject'] = t('Johnson current posts22222222');

##### get the contents

Set the theme of items

//Pass data through theme function.

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

'#theme' => 'item\_list\_\_current\_posts\_\_block',

'#items' => $items,

);

//Add a link to the page for more entries.

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

'#theme' => 'more\_link\_\_current\_posts',

'#url' => 'current\_posts',

'#title' => t('See the full list of current posts.'),

);

## Taxonomy

# People

## Role

Roles created to group users and to link with permissions.

### Create role

Admin – people-permissions- roles tab

## User

### Admin-configuration-account settings

To change basic settings of user management, like the how many fields are used by user profile.

## Permission

Permission are assigned to roles, and permission is checked when it is required to access or operate in drupal.

For any content type, there are 5 system permissions that can be used by drupal, either through ui or through permission api

#### Admin-people-permission

## Permission API

### hook\_permission

return array(

'create cc real estate listing' =>

array('title' => t('Create access to real estate listing contents')),

'edit any cc real estate listing' =>

array('title' => t('Edit access to any real estate listings')),

'edit own cc real estate listings'=>

array('title' => t('Edit access to their own real estate listing contents')),

'delete any cc real estate listing' =>

array('title' => t('Delete access to any real estate listings')),

'delete own cc real estate listings'=>

array('title' => t('Delete access to their own real estate listing contents')),

);

This will actually create new permissions for this module

### hook\_access

This applies or checks the custom defined permission in hook\_permission with actions

[php]

if ($op == 'create') {

// Only users with permission to do so may create this node type.

return user\_access('create cc real estate listing', $account);

}

[/php]

# Modules: Program Interface, APIs

## Path: sites/all/modules

## Hooks

A hook is a PHP function that is named foo\_bar()

"foo" is the name of the module (whose filename is thus foo.module)

"bar" is the name of the hook

### hook\_help($path, $arg)

path could either be a registered path in hook\_menu

using ‘%’ for wildcard matching

arg can provide further params

To provide a help page for a whole module with a listing on admin/help, your hook implementation should match a path with a special descriptor after a "#" sign: 'admin/help#modulename' The main module help text, displayed on the admin/help/modulename page and linked to from the admin/help page

### hook\_install

This hook is called when the user enables the module for the first time

(or on subsequent enables after the module has been uninstalled).

So it's a good place to define our new node type

### hook\_uninstall

This hook is called when the user not only has disabled the module but also uninstalls it from the 'uninstall' tab in the module page

Only modules that implemented uninstall appear on uninstall tab

### hook\_field\_schema

An associative array with the following keys:

\* columns: An array of Schema API column specifications, keyed by column name. This specifies what comprises a value for a given field. For example, a value for a number field is simply 'value', while a value for a formatted text field is the combination of 'value' and 'format'. It is recommended to avoid having the column definitions depend on field settings when possible. No assumptions should be made on how storage engines internally use the original column name to structure their storage.

\* indexes: (optional) An array of Schema API indexes definitions. Only columns that appear in the 'columns' array are allowed. Those indexes will be used as default indexes. Callers of field\_create\_field() can specify additional indexes, or, at their own risk, modify the default indexes specified by the field-type module. Some storage engines might not support indexes.

\* foreign keys: (optional) An array of Schema API foreign keys definitions.

## Info file

### name

The name appears in modules on admin

### package

The group table name for modules, all modules that have same package name will be grouped together. If it does not exist, then the module will sit under ‘other’

### version

The version appears in modules on admin

### description

The description appears in modules on admin

## Form API

### hook\_element\_info

Allows modules to declare their own Forms API element types and specify their default values.

This hook allows modules to declare their own form element types and to specify their default values. The values returned by this hook will be merged with the elements returned by hook\_form() implementations and so can return defaults for any Form APIs keys in addition to those explicitly mentioned below.

Each of the form element types defined by this hook is assumed to have a matching theme function, e.g. theme\_elementtype(), which should be registered with hook\_theme() as normal.

For more information about custom element types see the explanation at http://drupal.org/node/169815.

Return value

An associative array describing the element types being defined. The array contains a sub-array for each element type, with the machine-readable type name as the key. Each sub-array has a number of possible attributes:

\* "#input": boolean indicating whether or not this element carries a value (even if it's hidden).

\* "#process": array of callback functions taking $element, $form\_state, and $complete\_form.

\* "#after\_build": array of callback functions taking $element and $form\_state.

\* "#validate": array of callback functions taking $form and $form\_state.

\* "#element\_validate": array of callback functions taking $element and $form\_state.

\* "#pre\_render": array of callback functions taking $element and $form\_state.

\* "#post\_render": array of callback functions taking $element and $form\_state.

\* "#submit": array of callback functions taking $form and $form\_state.

\* "#title\_display": optional string indicating if and how #title should be displayed, see theme\_form\_element() and theme\_form\_element\_label().

# Eclipse configuration for drupal model development

## Install hook template to enable code completion

## Download the templates from the Eclipse Hook Templates project page

## In Eclipse, go to Preferences > PHP > Editor > Templates and 'Import' the downloaded XML file. Apply your changes

### Use the templates by typing the name of the hook (eg. 'hook\_form') and pressing ctrl-space to pop up the auto-complete window.

### http://drupal.org/project/eclipse

## File extension support

Under Window -> Preferences (Eclipse

-> Preferences on Mac) menu:

\* Expand the left-hand menu to General -> Content Types. Under Content types on the right, click Text -> PHP Content Type. Add the \*.engine, \*.theme, \*.install, \*.inc, \*.module \*.profile and \*.test file types so that these files will be recognized as PHP. The free plug-in to Eclipse for Drupal developers does this for you and is easy to install.

Under Window -> Preferences (Eclipse

-> Preferences on Mac) menu:

\* In the left-hand menu click on General -> Workspace: Check Text file encoding and select

Other :

UTF-8

Check New text file line delimiter and select

Other:

Unix

Under Window -> Preferences (Eclipse

-> Preferences on Mac) menu:

\* In the left-hand menu click on General -> Editors -> Text

Editors: Check Insert spaces for tabs.

In the left-hand menu click on PHP -> Code Style ->

Formatter: Set Tab policy: Spaces. Set Indentation size to 2.

# dddd