Drupal notes:

Contents

[1. Database : drupal712 9](#_Toc326922993)

[1.1. Database user and password drupaluser/1234 9](#_Toc326922994)

[1.2. url:http://localhost:81/drupal7 9](#_Toc326922995)

[1.3. Site admin : admin/680628 9](#_Toc326922996)

[1.4. Useful urls: 9](#_Toc326922997)

[1.4.1. Useful 9](#_Toc326922998)

[1.4.2. Modules download 9](#_Toc326922999)

[1.4.3. Examples download 9](#_Toc326923000)

[2. Create a git repo: 9](#_Toc326923001)

[2.1. Right click the folder, create repository 9](#_Toc326923002)

[2.2. Add files to repo 9](#_Toc326923003)

[2.3. Go on to github ,create a repo 9](#_Toc326923004)

[2.4. On desktop, create a remote with the url like 9](#_Toc326923005)

[2.5. https://cleancodenz@github.com/cleancodenz/DrupalDev.git 9](#_Toc326923006)

[2.6. Push to remote 9](#_Toc326923007)

[2.7. How to get rid off of message box 9](#_Toc326923008)

[2.7.1. Go to repository, then .git directory, some time it is hidden 9](#_Toc326923009)

[2.7.2. In the core group 9](#_Toc326923010)

[2.7.3. Add 9](#_Toc326923011)

[3. Beautiful url 10](#_Toc326923012)

[3.1. Enable clean url first 10](#_Toc326923013)

[3.2. Install pathAuto module 10](#_Toc326923014)

[3.3. Install token module 10](#_Toc326923015)

[3.4. Administration:config:search and meta data:url alias:pattern tab 10](#_Toc326923016)

[3.5. For all previous contents, enable the automatic url 10](#_Toc326923017)

[4. Theme designing 10](#_Toc326923018)

[4.1. Folder is :sites/all/themes/mytheme 10](#_Toc326923019)

[4.2. .info file: sites/all/themes/mytheme/mytheme.info 10](#_Toc326923020)

[4.3. Defaults 10](#_Toc326923021)

[4.4. Regions in content 10](#_Toc326923022)

[4.4.1. Must have regions[content] = Content 10](#_Toc326923023)

[4.4.2. Output of region 10](#_Toc326923024)

[4.4.3. There are template (.tpl.php) files available for rendering individual blocks 11](#_Toc326923025)

[4.4.4. Order of regions definition 11](#_Toc326923026)

[4.4.5. Manually set content of a region using drupal\_set\_content('header', 'Welcome!') 11](#_Toc326923027)

[4.5. Color scheme, color module integration 11](#_Toc326923028)

[4.6. Clearing the cache: 11](#_Toc326923029)

[4.7. Template.php 11](#_Toc326923030)

[4.8. Add javascripts through behaviours 11](#_Toc326923031)

[5. Blue print 11](#_Toc326923032)

[5.1. Source file structure 11](#_Toc326923033)

[5.1.1. /blueprint/ css folder 11](#_Toc326923034)

[5.1.2. Main css files /blueprint/ 11](#_Toc326923035)

[6. Useful links 11](#_Toc326923036)

[6.1. Drupal 7 examples : 11](#_Toc326923037)

[6.2. Drupal 7 api documentation 12](#_Toc326923038)

[7. Key functions: 12](#_Toc326923039)

[8. Entiy Bundle field 18](#_Toc326923040)

[9. Form API 19](#_Toc326923041)

[9.1. Get Form 21](#_Toc326923042)

[9.2. Prepare Form 21](#_Toc326923043)

[9.3. Process Form 21](#_Toc326923044)

[9.4. #States 21](#_Toc326923045)

[9.5. Form Element Types 22](#_Toc326923046)

[9.5.1. System\_elements\_info()- system 22](#_Toc326923047)

[9.5.1.1. Form 22](#_Toc326923048)

[9.5.1.2. Page 22](#_Toc326923049)

[9.5.1.3. Ajax 22](#_Toc326923050)

[9.5.1.4. Html\_tag 23](#_Toc326923051)

[9.5.1.5. Styles 23](#_Toc326923052)

[9.5.1.6. Submit 23](#_Toc326923053)

[9.5.1.7. Button 23](#_Toc326923054)

[9.5.1.8. Image\_button 23](#_Toc326923055)

[9.5.1.9. Textfield 23](#_Toc326923056)

[9.5.1.10. Machine\_name 23](#_Toc326923057)

[9.5.1.11. Password 23](#_Toc326923058)

[9.5.1.12. Password\_confirm 23](#_Toc326923059)

[9.5.1.13. Textarea 23](#_Toc326923060)

[9.5.1.14. Radios 23](#_Toc326923061)

[9.5.1.15. Radio 23](#_Toc326923062)

[9.5.1.16. Checkboxes 23](#_Toc326923063)

[9.5.1.17. Checkbox 23](#_Toc326923064)

[9.5.1.18. Select 23](#_Toc326923065)

[9.5.1.19. Weight 23](#_Toc326923066)

[9.5.1.20. Date 23](#_Toc326923067)

[9.5.1.21. File 23](#_Toc326923068)

[9.5.1.22. Tableselect 23](#_Toc326923069)

[9.5.1.23. Item 23](#_Toc326923070)

[9.5.1.24. Hidden 24](#_Toc326923071)

[9.5.1.25. Value 24](#_Toc326923072)

[9.5.1.26. Markup 24](#_Toc326923073)

[9.5.1.27. Link 24](#_Toc326923074)

[9.5.1.28. Fieldset 24](#_Toc326923075)

[9.5.1.29. Vertical\_tabs 24](#_Toc326923076)

[9.5.1.30. Container 24](#_Toc326923077)

[9.5.1.31. Actions 24](#_Toc326923078)

[9.5.1.32. Token 24](#_Toc326923079)

[9.5.2. Hook\_elements\_info()-custom 24](#_Toc326923080)

[9.5.3. Element theming 25](#_Toc326923081)

[9.6. Form element values 26](#_Toc326923082)

[9.6.1. Value callback or form\_type\_hook\_value() 26](#_Toc326923083)

[9.6.2. Value callback or form\_type\_hook\_value() 27](#_Toc326923084)

[9.7. Form Theming 27](#_Toc326923085)

[10. Rendering 32](#_Toc326923086)

[10.1. Property:’#Property’ 33](#_Toc326923087)

[10.1.1. #type 33](#_Toc326923088)

[10.1.2. #markup 33](#_Toc326923089)

[10.1.3. #prefix/#suffix 33](#_Toc326923090)

[10.1.4. #pre\_render 33](#_Toc326923091)

[10.1.5. #post\_render 33](#_Toc326923092)

[10.1.6. #theme 33](#_Toc326923093)

[10.1.7. #theme\_wrappers 34](#_Toc326923094)

[10.1.8. #cache 34](#_Toc326923095)

[10.2. Children: 35](#_Toc326923096)

[10.3. HTML generation: 35](#_Toc326923097)

[10.3.1. #theme 35](#_Toc326923098)

[10.3.2. #theme\_wrappers 35](#_Toc326923099)

[10.3.3. #cache 36](#_Toc326923100)

[10.3.4. #attached 36](#_Toc326923101)

[10.3.4.1. Library 37](#_Toc326923102)

[10.3.4.2. Js 37](#_Toc326923103)

[10.3.4.3. Css 39](#_Toc326923104)

[10.3.4.4. callback 40](#_Toc326923105)

[10.3.4.5. hook\_library() 41](#_Toc326923106)

[11. Theming 41](#_Toc326923107)

[11.1. Basics: theme() 41](#_Toc326923108)

[11.1.1. $hook 41](#_Toc326923109)

[11.1.2. $variables 42](#_Toc326923110)

[11.1.2.1. #variables as a render array 42](#_Toc326923111)

[11.1.2.2. #variables passed through 44](#_Toc326923112)

[11.2. Registration Hook\_theme: 44](#_Toc326923113)

[11.3. Default templates 45](#_Toc326923114)

[11.4. Theme functions: 47](#_Toc326923115)

[11.4.1. $attributes 48](#_Toc326923116)

[11.5. Dynamic theming: 48](#_Toc326923117)

[11.5.1. Pattern for suggestions 49](#_Toc326923118)

[11.5.2. Implemented suggestions 49](#_Toc326923119)

[11.6. theme('table') and theme('item\_list'): 50](#_Toc326923120)

[11.7. Having changes to code take effect: 50](#_Toc326923121)

[11.8. Page rendering example : 50](#_Toc326923122)

[12. Structure (big picture) 61](#_Toc326923123)

[12.1. Content: 61](#_Toc326923124)

[12.1.1. Content type 61](#_Toc326923125)

[12.1.1.1. Edit 61](#_Toc326923126)

[12.1.1.1.1. There might be custom settings 61](#_Toc326923127)

[12.1.1.1.2. Submission form settings 61](#_Toc326923128)

[12.1.1.1.3. Publish options 61](#_Toc326923129)

[12.1.1.1.4. Display settings 61](#_Toc326923130)

[12.1.1.1.5. Comments Settings 61](#_Toc326923131)

[12.1.1.1.6. Menu Settings 61](#_Toc326923132)

[12.1.1.1.7. Main menu 61](#_Toc326923133)

[12.1.2. Manage Fields 61](#_Toc326923134)

[12.1.3. Manage Display 61](#_Toc326923135)

[12.1.4. Field API 61](#_Toc326923136)

[12.1.4.1. field\_example\_3text\_validate($element, &$form\_state) 63](#_Toc326923137)

[12.1.4.2. hook\_field\_formatter\_info() 63](#_Toc326923138)

[12.1.4.3. hook\_field\_formatter\_view() 63](#_Toc326923139)

[12.1.4.4. hook\_field\_info() 63](#_Toc326923140)

[12.1.4.5. hook\_field\_is\_empty() 63](#_Toc326923141)

[12.1.4.6. hook\_field\_schema() 63](#_Toc326923142)

[12.1.4.7. hook\_field\_validate () 63](#_Toc326923143)

[12.1.4.8. hook\_field\_widget\_error() 63](#_Toc326923144)

[12.1.4.9. hook\_field\_widget\_form () 64](#_Toc326923145)

[12.1.4.10. hook\_field\_widget\_info() 64](#_Toc326923146)

[12.1.5. Custom content type API 64](#_Toc326923147)

[12.1.5.1. hook\_entity\_info\_alter() 65](#_Toc326923148)

[12.1.5.2. hook\_field\_formatter\_info() 65](#_Toc326923149)

[12.1.5.3. hook\_field\_formatter\_view() 65](#_Toc326923150)

[12.1.5.4. hook\_help() 65](#_Toc326923151)

[12.1.5.5. hook\_install()(.install) 65](#_Toc326923152)

[12.1.5.6. hook\_menu() 66](#_Toc326923153)

[12.1.5.7. hook\_theme() 67](#_Toc326923154)

[12.1.5.8. hook\_uninstall(.install) 67](#_Toc326923155)

[12.1.6. Custom field type API(Node API) 68](#_Toc326923156)

[12.1.6.1. Hook\_form\_alter 68](#_Toc326923157)

[12.1.6.2. Hook\_node\_delete 68](#_Toc326923158)

[12.1.6.3. Hook\_node\_insert 68](#_Toc326923159)

[12.1.6.4. Hook\_node\_insert 68](#_Toc326923160)

[12.1.6.5. Hook\_node\_update 69](#_Toc326923161)

[12.1.6.6. Hook\_node\_validate 69](#_Toc326923162)

[12.1.6.7. Hook\_node\_view 69](#_Toc326923163)

[12.1.6.8. Hook\_schema(.install) 69](#_Toc326923164)

[12.1.6.9. Hook\_uninstall(.install) 69](#_Toc326923165)

[12.1.6.10. Hook\_theme 70](#_Toc326923166)

[12.1.6.11. Custom theme function 70](#_Toc326923167)

[12.2. Menus 70](#_Toc326923168)

[12.2.1. Main menu 70](#_Toc326923169)

[12.2.2. Navigation menu 70](#_Toc326923170)

[12.2.3. Management menu 70](#_Toc326923171)

[12.2.4. User menu 70](#_Toc326923172)

[12.2.5. Custom menu 70](#_Toc326923173)

[12.2.6. API 71](#_Toc326923174)

[12.2.6.1. Hook\_menu 71](#_Toc326923175)

[12.2.6.1.1. Menu hierarchy 71](#_Toc326923176)

[12.2.6.1.2. Type 71](#_Toc326923177)

[12.2.6.1.3. Optional path arguments 71](#_Toc326923178)

[12.2.6.1.4. Page arguments 72](#_Toc326923179)

[12.2.6.1.5. Access callback 72](#_Toc326923180)

[12.2.6.1.6. Access arguments 72](#_Toc326923181)

[12.2.6.1.7. Page callback 72](#_Toc326923182)

[12.2.6.1.8. Menu name 72](#_Toc326923183)

[12.2.6.1.9. Menu tabs 72](#_Toc326923184)

[12.2.6.1.10. Title callback 73](#_Toc326923185)

[12.2.6.1.11. Place holder or wildcards 73](#_Toc326923186)

[12.2.6.1.12. Hook\_menu\_alter 73](#_Toc326923187)

[12.2.6.2. Navigation Block 73](#_Toc326923188)

[12.3. Block 73](#_Toc326923189)

[12.3.1. Pages: 74](#_Toc326923190)

[12.3.2. Content types 74](#_Toc326923191)

[12.3.3. Roles 74](#_Toc326923192)

[12.3.4. Users 74](#_Toc326923193)

[12.3.5. API 74](#_Toc326923194)

[12.3.5.1. Path: sites/all/modules/modulename 74](#_Toc326923195)

[12.3.5.2. Info file : sites/all/modules/modulename/modulename.info 74](#_Toc326923196)

[12.3.5.3. Module file : sites/all/modules/modulename/modulename.module 74](#_Toc326923197)

[12.3.5.4. Modulename\_help 74](#_Toc326923198)

[12.3.5.5. hook\_block\_info() 74](#_Toc326923199)

[12.3.5.6. hook\_menu() 74](#_Toc326923200)

[12.3.5.6.1. $items['admin/config/content/current\_posts'] 74](#_Toc326923201)

[12.3.5.6.1.1. Title : 74](#_Toc326923202)

[12.3.5.6.1.2. Description: 74](#_Toc326923203)

[12.3.5.6.1.3. page callback 74](#_Toc326923204)

[12.3.5.6.1.4. page arguments 74](#_Toc326923205)

[12.3.5.6.1.5. Form implementation of above setting 75](#_Toc326923206)

[12.3.5.6.1.6. Form validation for form implementation 75](#_Toc326923207)

[12.3.5.6.1.7. access arguments 75](#_Toc326923208)

[12.3.5.6.1.8. type 75](#_Toc326923209)

[12.3.5.6.2. $items[modulename] 75](#_Toc326923210)

[12.3.5.6.2.1. Type 75](#_Toc326923211)

[12.3.5.6.2.2. Page call back page implementation 75](#_Toc326923212)

[12.3.5.7. hook\_block\_view() 76](#_Toc326923213)

[12.3.5.7.1. set the title of block 76](#_Toc326923214)

[12.3.5.7.2. get the contents 76](#_Toc326923215)

[12.4. Taxonomy 76](#_Toc326923216)

[13. People 76](#_Toc326923217)

[13.1. Role 76](#_Toc326923218)

[13.1.1. Create role 76](#_Toc326923219)

[13.2. User 76](#_Toc326923220)

[13.2.1. Admin-configuration-account settings 76](#_Toc326923221)

[13.3. Permission 76](#_Toc326923222)

[13.3.1. Admin-people-permission 77](#_Toc326923223)

[13.4. Permission API 77](#_Toc326923224)

[13.4.1. hook\_permission 77](#_Toc326923225)

[13.4.2. hook\_access 77](#_Toc326923226)

[14. Modules: Program Interface, APIs 78](#_Toc326923227)

[14.1. Path: sites/all/modules 78](#_Toc326923228)

[14.2. Hooks 78](#_Toc326923229)

[14.2.1. hook\_help($path, $arg) 78](#_Toc326923230)

[14.2.2. hook\_install 78](#_Toc326923231)

[14.2.3. hook\_uninstall 78](#_Toc326923232)

[14.2.4. hook\_field\_schema 78](#_Toc326923233)

[14.3. Info file 79](#_Toc326923234)

[14.3.1. name 79](#_Toc326923235)

[14.3.2. package 79](#_Toc326923236)

[14.3.3. version 79](#_Toc326923237)

[14.3.4. description 79](#_Toc326923238)

[14.4. Form API 79](#_Toc326923239)

[14.4.1. hook\_element\_info 79](#_Toc326923240)

[15. Eclipse configuration for drupal model development 80](#_Toc326923241)

[15.1. Install hook template to enable code completion 80](#_Toc326923242)

[15.2. Download the templates from the Eclipse Hook Templates project page 80](#_Toc326923243)

[15.3. In Eclipse, go to Preferences > PHP > Editor > Templates and 'Import' the downloaded XML file. Apply your changes 80](#_Toc326923244)

[15.3.1. Use the templates by typing the name of the hook (eg. 'hook\_form') and pressing ctrl-space to pop up the auto-complete window. 80](#_Toc326923245)

[15.3.2. http://drupal.org/project/eclipse 80](#_Toc326923246)

[15.4. File extension support 80](#_Toc326923247)

[16. 81](#_Toc326923248)

[17. dddd 81](#_Toc326923249)

# 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’

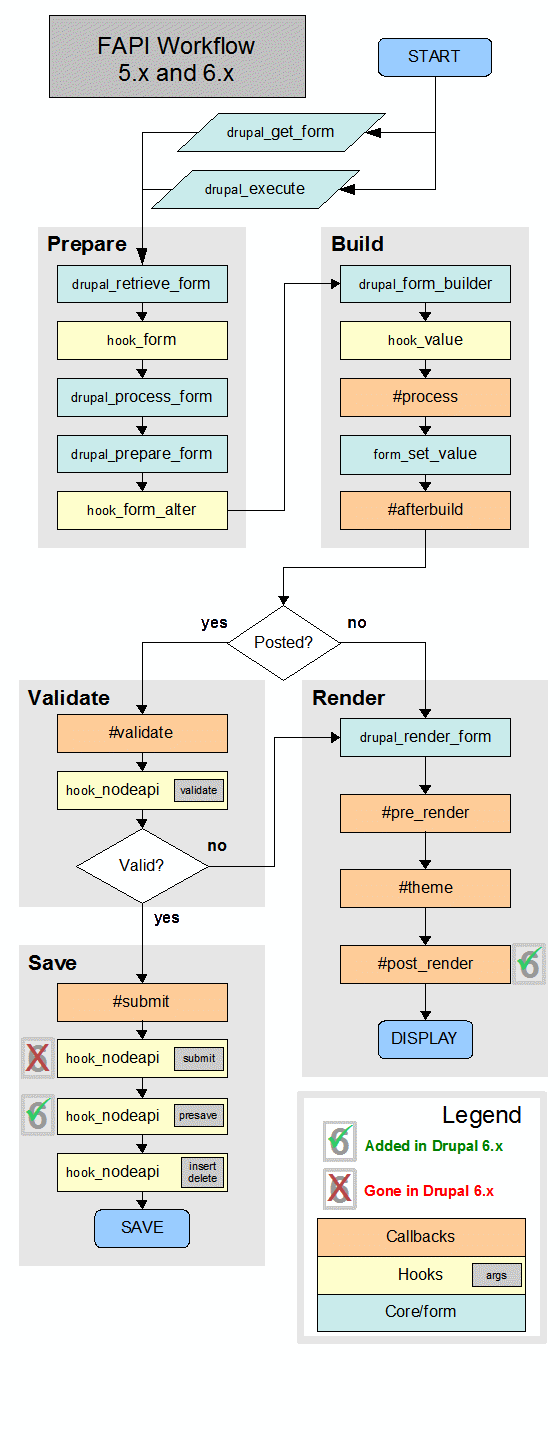
Entity-> node

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

Fields -> fields attached in the bundle

# Form API

Start from drupal\_get\_form with form\_id as implemented function name form\_id($form,$form\_state) then durpal\_build\_form runs through following steps



## Get Form

drupal\_get\_form()

The primary function used with forms is drupal\_get\_form(), which is used for forms presented interactively to a user. Forms can also be built and submitted programmatically without any user input using the drupal\_form\_submit() function.

drupal\_get\_form() handles retrieving, processing, and displaying a rendered HTML form for modules automatically. Here is an example of how to use drupal\_get\_form() and a form builder

$form = drupal\_get\_form('my\_module\_example\_form');

function my\_module\_example\_form($form, &$form\_state) {

$form['submit'] = array(

'#type' => 'submit',

'#value' => t('Submit'),

);

return $form;

}

function my\_module\_example\_form\_validate($form, &$form\_state) {

// Validation logic.

}

function my\_module\_example\_form\_submit($form, &$form\_state) {

// Submission logic.

}

Or with any number of additional arguments:

$extra = "extra";

$form = drupal\_get\_form('my\_module\_example\_form', $extra);

function my\_module\_example\_form($form, &$form\_state, $extra) {

$form['submit'] = array(

'#type' => 'submit',

'#value' => $extra,

);

return $form;

}

The $form argument to form-related functions is a structured array containing the elements and properties of the form. For information on the array components and format, and more detailed explanations of the Form API workflow, see the http://api.drupal.org/api/drupal/developer--topics--forms\_api\_reference.html Form API reference and the http://drupal.org/node/37775 Form API documentation section

In the form builder, validation, submission, and other form functions, $form\_state is the primary influence on the processing of the form and is passed by reference to most functions, so they use it to communicate with the form system and each other. See drupal\_build\_form() for documentation of $form\_state keys.

@param $form\_id

The unique string identifying the desired form. If a function with that name exists, it is called to build the form array. Modules that need to generate the same form (or very similar forms) using different $form\_ids can implement hook\_forms(), which maps different $form\_id values to the proper form constructor function. Examples may be found in node\_forms(), search\_forms(), and user\_forms().

@param ...

Any additional arguments are passed on to the functions called by drupal\_get\_form(), including the unique form constructor function. For example, the node\_edit form requires that a node object is passed in here when it is called. These are available to implementations of hook\_form\_alter() and hook\_form\_FORM\_ID\_alter() as the array $form\_state['build\_info']['args'].

@return

The form array.

drupal\_retrieve\_form() from form\_id() form array. And hook\_form\_alter to add new items to a form

Anything set in your implementation(instance) of the element will override the basic settings in hook\_elements(). For example, if you set #example\_attribute to 'Nothing much' in hook\_form\_alter(), that would override the value set in hook\_elements() of 'Something extra'. You can use this feature to set sane default values for your element and to be sure those values are not empty, while still being able to override them later.

## Build form

drupal\_build\_form()

Build and process a form based on a form id. The form may also be retrieved from the cache if the form was built in a previous page-load. The form is then passed on for processing, validation and submission if there is proper input.

@param $form\_id

The unique string identifying the desired form. If a function with that name exists, it is called to build the form array.

@param $form\_state

An array which stores information about the form. This is passed as a reference so that the caller can use it to examine what in the form changed when the form submission process is complete. Furthermore, it may be used to store information related to the processed data in the form, which will persist across page requests when the 'cache' or 'rebuild' flag is set. The following parameters may be set in $form\_state to affect how the form is rendered:

- build\_info: Internal. An associative array of information stored by Form API that is necessary to build and rebuild the form from cache when the original context may no longer be available:

- args: A list of arguments to pass to the form constructor.

- files: An optional array defining include files that need to be loaded for building the form. Each array entry may be the path to a file or another array containing values for the parameters 'type', 'module' and 'name' as needed by module\_load\_include(). The files listed here are automatically loaded by form\_get\_cache(). By default the current menu router item's 'file' definition is added, if any. Use form\_load\_include() to add include files from a form constructor.

- base\_form\_id: Identification for a base form, as declared in a hook\_forms() implementation.

- rebuild\_info: Internal. Similar to 'build\_info', but pertaining to drupal\_rebuild\_form().

- rebuild: Normally, after the entire form processing is completed and submit handlers have run, a form is considered to be done and drupal\_redirect\_form() will redirect the user to a new page using a GET request (so a browser refresh does not re-submit the form). However, if 'rebuild' has been set to TRUE, then a new copy of the form is immediately built and sent to the browser, instead of a redirect. This is used for multi-step forms, such as wizards and confirmation forms. Normally, $form\_state['rebuild'] is set by a submit handler, since it is usually logic within a submit handler that determines whether a form is done or requires another step. However, a validation handler may already set $form\_state['rebuild'] to cause the form processing to bypass submit handlers and rebuild the form instead, even if there are no validation errors.

- redirect: Used to redirect the form on submission. It may either be a string containing the destination URL, or an array of arguments compatible with drupal\_goto(). See drupal\_redirect\_form() for complete information.

- no\_redirect: If set to TRUE the form will NOT perform a drupal\_goto(), even if 'redirect' is set.

- method: The HTTP form method to use for finding the input for this form. May be 'post' or 'get'. Defaults to 'post'. Note that 'get' method forms do not use form ids so are always considered to be submitted, which can have unexpected effects. The 'get' method should only be used on forms that do not change data, as that is exclusively the domain of 'post.'

- cache: If set to TRUE the original, unprocessed form structure will be cached, which allows the entire form to be rebuilt from cache. A typical form workflow involves two page requests; first, a form is built and rendered for the user to fill in. Then, the user fills the form in and submits it, triggering a second page request in which the form must be built and processed. By default, $form and $form\_state are built from scratch during each of these page requests. Often, it is necessary or desired to persist the $form and $form\_state variables from the initial page request to the one that processes the submission. 'cache' can be set to TRUE to do this. A prominent example is an Ajax-enabled form, in which ajax\_process\_form() enables form caching for all forms that include an element with the #ajax property. (The Ajax handler has no way to build the form itself, so must rely on the cached version.) Note that the persistence of $form and $form\_state happens automatically for (multi-step) forms having the 'rebuild' flag set, regardless of the value for 'cache'.

- no\_cache: If set to TRUE the form will NOT be cached, even if 'cache' is set.

- values: An associative array of values submitted to the form. The validation functions and submit functions use this array for nearly all their decision making. (Note that http://api.drupal.org/api/drupal/developer--topics--forms\_api\_reference.html/7 determines whether the values are a flat array or an array whose structure parallels the $form array.)

- input: The array of values as they were submitted by the user. These are raw and unvalidated, so should not be used without a thorough understanding of security implications. In almost all cases, code should use the data in the 'values' array exclusively. The most common use of this key is for multi-step forms that need to clear some of the user input when setting 'rebuild'. The values correspond to $\_POST or $\_GET, depending on the 'method' chosen.

- always\_process: If TRUE and the method is GET, a form\_id is not necessary. This should only be used on RESTful GET forms that do NOT write data, as this could lead to security issues. It is useful so that searches do not need to have a form\_id in their query arguments to trigger the search.

- must\_validate: Ordinarily, a form is only validated once, but there are times when a form is resubmitted internally and should be validated again. Setting this to TRUE will force that to happen. This is most likely to occur during Ajax operations.

- programmed: If TRUE, the form was submitted programmatically, usually invoked via drupal\_form\_submit(). Defaults to FALSE.

- process\_input: Boolean flag. TRUE signifies correct form submission. This is always TRUE for programmed forms coming from drupal\_form\_submit() (see 'programmed' key), or if the form\_id coming from the $\_POST data is set and matches the current form\_id.

- submitted: If TRUE, the form has been submitted. Defaults to FALSE.

- executed: If TRUE, the form was submitted and has been processed and executed. Defaults to FALSE.

- triggering\_element: (read-only) The form element that triggered submission. This is the same as the deprecated $form\_state['clicked\_button']. It is the element that caused submission, which may or may not be a button (in the case of Ajax forms). This key is often used to distinguish between various buttons in a submit handler, and is also used in Ajax handlers.

- clicked\_button: Deprecated. Use triggering\_element instead.

- has\_file\_element: Internal. If TRUE, there is a file element and Form API will set the appropriate 'enctype' HTML attribute on the form.

- groups: Internal. An array containing references to fieldsets to render them within vertical tabs.

- storage: $form\_state['storage'] is not a special key, and no specific support is provided for it in the Form API. By tradition it was the location where application-specific data was stored for communication between the submit, validation, and form builder functions, especially in a multi-step-style form. Form implementations may use any key(s) within $form\_state (other than the keys listed here and other reserved ones used by Form API internals) for this kind of storage. The recommended way to ensure that the chosen key doesn't conflict with ones used by the Form API or other modules is to use the module name as the key name or a prefix for the key name. For example, the Node module uses $form\_state['node'] in node editing forms to store information about the node being edited, and this information stays available across successive clicks of the "Preview" button as well as when the "Save" button is finally clicked.

- buttons: A list containing copies of all submit and button elements in the form.

- complete form: A reference to the $form variable containing the complete form structure. #process, #after\_build, #element\_validate, and other handlers being invoked on a form element may use this reference to access other information in the form the element is contained in.

- temporary: An array holding temporary data accessible during the current page request only. All $form\_state properties that are not reserved keys (see form\_state\_keys\_no\_cache()) persist throughout a multistep form sequence. Form API provides this key for modules to communicate information across form-related functions during a single page request.

It may be used to temporarily save data that does not need to or should not be cached during the whole form workflow; e.g., data that needs to be accessed during the current form build process only. There is no use-case for this functionality in Drupal core.

- wrapper\_callback: Modules that wish to pre-populate certain forms with common elements, such as back/next/save buttons in multi-step form wizards, may define a form builder function name that returns a form structure, which is passed on to the actual form builder function. Such implementations may either define the 'wrapper\_callback' via hook\_forms() or have to invoke drupal\_build\_form() (instead of drupal\_get\_form()) on their own in a custom menu callback to prepare $form\_state accordingly.

Information on how certain $form\_state properties control redirection behavior after form submission may be found in drupal\_redirect\_form().

@return

The rendered form. This function may also perform a redirect and hence may not return at all, depending upon the $form\_state flags that were set.

## Prepare Form

drupal\_prepare\_form()

Prepares a structured form array by adding required elements, executing any hook\_form\_alter functions, and optionally inserting a validation token to prevent tampering.

Merge the form array with form alter hooks like hook\_form\_alter(), hook\_form\_BASE\_FORM\_ID\_alter(), and hook\_form\_FORM\_ID\_alter() implementations

## Process Form

Drupal\_process\_form()

This function is the heart of form API. The form gets built, validated and in appropriate cases, submitted and rebuilt

### Form builder

form\_builder() finishes building the form by calling element #process functions and mapping user input, if any, to #value properties, and also storing the values in $form\_state['values']. We need to retain the unprocessed $form in case it needs to be cached.

This is one of the three primary functions that recursively iterates a form array. This one does it for completing the form building process. The other two are \_form\_validate() (invoked via drupal\_validate\_form() and used to invoke validation logic for each element) and drupal\_render() (for rendering each element). Each of these three pipelines provides ample opportunity for modules to customize what happens. For example, during this function's life cycle, the following functions get called for each element:

- $element['#value\_callback']: A function that implements how user input is mapped to an element's #value property. This defaults to a function named 'form\_type\_TYPE\_value' where TYPE is $element['#type'].

- $element['#process']: An array of functions called after user input has been mapped to the element's #value property. These functions can be used to dynamically add child elements: for example, for the 'date' element type, one of the functions in this array is form\_process\_date(), which adds the individual 'year', 'month', 'day', etc. child elements. These functions can also be used to set additional properties or implement special logic other than adding child elements: for example, for the 'fieldset' element type, one of the functions in this array is form\_process\_fieldset(), which adds the attributes and JavaScript needed to make the fieldset collapsible if the #collapsible property is set. The #process functions are called in preorder traversal, meaning they are called for the parent element first, then for the child elements.

- $element['#after\_build']: An array of functions called after form\_builder() is done with its processing of the element. These are called in postorder traversal, meaning they are called for the child elements first, then for the parent element. There are similar properties containing callback functions invoked by \_form\_validate() and drupal\_render(), appropriate for those operations. Developers are strongly encouraged to integrate the functionality needed by their form or module within one of these three pipelines, using the appropriate callback property, rather than implementing their own recursive traversal of a form array. This facilitates proper integration between multiple modules. For example, module developers are familiar with the relative order in which hook\_form\_alter() implementations and #process functions run. A custom traversal function that affects the building of a form is likely to not integrate with hook\_form\_alter() and #process in the expected way. Also, deep recursion within PHP is both slow and memory intensive, so it is best to minimize how often it's done. As stated above, each element's #process functions are executed after its #value has been set. This enables those functions to execute conditional logic based on the current value. However, all of form\_builder() runs before drupal\_validate\_form() is called, so during #process function execution, the element's #value has not yet been validated, so any code that requires validated values must reside within a submit handler.

As a security measure, user input is used for an element's #value only if the element exists within $form, is not disabled (as per the #disabled property), and can be accessed (as per the #access property, except that forms submitted using drupal\_form\_submit() bypass #access restrictions). When user input is ignored due to #disabled and #access restrictions, the element's default value is used. Because of the preorder traversal, where #process functions of an element run before user input for its child elements is processed, and because of the Form API security of user input processing with respect to #access and #disabled described above, this generally means that #process functions should not use an element's (unvalidated) #value to affect the #disabled or #access of child elements. Use-cases where a developer may be tempted to implement such conditional logic usually fall into one of two categories:

- Where user input from the current submission must affect the structure of a form, including properties like #access and #disabled that affect how the next submission needs to be processed, a multi-step workflow is needed. This is most commonly implemented with a submit handler setting persistent data within $form\_state based on \*validated\* values in $form\_state['values'] and setting $form\_state['rebuild']. The form building functions must then be implemented to use the $form\_state data to rebuild the form with the structure appropriate for the new state.

- Where user input must affect the rendering of the form without affecting its structure, the necessary conditional rendering logic should reside within functions that run during the rendering phase (#pre\_render, #theme, #theme\_wrappers, and #post\_render).

@param $form\_id

A unique string identifying the form for validation, submission, theming, and hook\_form\_alter functions.

@param $element

An associative array containing the structure of the current element.

@param $form\_state

A keyed array containing the current state of the form. In this context, it is used to accumulate information about which button was clicked when the form was submitted, as well as the sanitized $\_POST data.

## Rebuild Form

drupal\_rebuild\_form()

Constructs a new $form from the information in $form\_state. This is the key function for making multi-step forms advance from step to step. It is called by drupal\_process\_form() when all user input processing, including calling validation and submission handlers, for the request is finished. If a validate or submit handler set $form\_state['rebuild'] to TRUE, and if other conditions don't preempt a rebuild from happening, then this function is called to generate a new $form, the next step in the form workflow, to be returned for rendering.

Ajax form submissions are almost always multi-step workflows, so that is one common use-case during which form rebuilding occurs. See ajax\_form\_callback() for more information about creating Ajax-enabled forms.

@param $form\_id

The unique string identifying the desired form. If a function with that name exists, it is called to build the form array.

@param $form\_state

A keyed array containing the current state of the form.

@param $old\_form

(optional) A previously built $form. Used to retain the #build\_id and #action properties in Ajax callbacks and similar partial form rebuilds. The only properties copied from $old\_form are the ones which both exist in $old\_form and for which $form\_state['rebuild\_info']['copy'][PROPERTY] is TRUE. If $old\_form is not passed, the entire $form is rebuilt freshly. 'rebuild\_info' needs to be a separate top-level property next to 'build\_info', since the contained data must not be cached.

@return

The newly built form.

## Validate Form

drupal\_validate\_form()

Validates user-submitted form data from the $form\_state using the validate functions defined in a structured form array.

@param $form\_id

@param $form

An associative array containing the structure of the form, which is passed by reference. Form validation handlers are able to alter the form structure (like #process and #after\_build callbacks during form building) in case of a validation error. If a validation handler alters the form structure, it is responsible for validating the values of changed form elements in $form\_state['values'] to prevent form submit handlers from receiving unvalidated values.

@param $form\_state

A keyed array containing the current state of the form. The current user-submitted data is stored in $form\_state['values'], though form validation functions are passed an explicit copy of the values for the sake of simplicity. Validation handlers can also $form\_state to pass information on to submit handlers. For example:

$form\_state['data\_for\_submission'] = $data;

This technique is useful when validation requires file parsing, web service requests, or other expensive requests that should not be repeated in the submission step.

### Set Error

form\_set\_error()

Files an error against a form element.

When a validation error is detected, the validator calls form\_set\_error() to indicate which element needs to be changed and provide an error message. This causes the Form API to not execute the form submit handlers, and instead to re-display the form to the user with the corresponding elements rendered with an 'error' CSS class (shown as red by default).

The standard form\_set\_error() behavior can be changed if a button provides the #limit\_validation\_errors property. Multistep forms not wanting to validate the whole form can set #limit\_validation\_errors on buttons to limit validation errors to only certain elements. For example, pressing the "Previous" button in a multistep form should not fire validation errors just because the current step has invalid values. If #limit\_validation\_errors is set on a clicked button, the button must also define a #submit property (may be set to an empty array). Any #submit handlers will be executed even if there is invalid input, so extreme care should be taken with respect to any actions taken by them. This is typically not a problem with buttons like "Previous" or "Add more" that do not invoke persistent storage of the submitted form values. Do not use the #limit\_validation\_errors property on buttons that trigger saving of form values to the database.

The #limit\_validation\_errors property is a list of "sections" within $form\_state['values'] that must contain valid values. Each "section" is an array with the ordered set of keys needed to reach that part of $form\_state['values'] (i.e., the #parents property of the element).

Example 1: Allow the "Previous" button to function, regardless of whether any user input is valid.

$form['actions']['previous'] = array(

'#type' => 'submit',

'#value' => t('Previous'),

'#limit\_validation\_errors' => array(), // No validation.

'#submit' => array('some\_submit\_function'), // #submit required.

);

Example 2: Require some, but not all, user input to be valid to process the submission of a "Previous" button.

$form['actions']['previous'] = array(

'#type' => 'submit',

'#value' => t('Previous'),

'#limit\_validation\_errors' => array(

array('step1'), // Validate $form\_state['values']['step1'].

array('foo', 'bar'), // Validate $form\_state['values']['foo']['bar'].

),

'#submit' => array('some\_submit\_function'), // #submit required.

);

This will require $form\_state['values']['step1'] and everything within it (for example, $form\_state['values']['step1']['choice']) to be valid, so calls to form\_set\_error('step1', $message) or

form\_set\_error('step1][choice', $message) will prevent the submit handlers from running, and result in the error message being displayed to the user. However, calls to form\_set\_error('step2', $message) and form\_set\_error('step2][groupX][choiceY', $message) will be suppressed, resulting in the message not being displayed to the user, and the submit handlers will run despite $form\_state['values']['step2'] and $form\_state['values']['step2']['groupX']['choiceY'] containing invalid values. Errors for an invalid $form\_state['values']['foo'] will be suppressed, but errors flagging invalid values for $form\_state['values']['foo']['bar'] and everything within it will be flagged and submission prevented.

Partial form validation is implemented by suppressing errors rather than by skipping the input processing and validation steps entirely, because some forms have button-level submit handlers that call Drupal API functions that assume that certain data exists within $form\_state['values'], and while not doing anything with that data that requires it to be valid, PHP errors would be triggered if the input processing and validation steps were fully skipped.

http://drupal.org/node/370537

http://drupal.org/node/763376

@param $name

The name of the form element. If the #parents property of your form element is array('foo', 'bar', 'baz') then you may set an error on 'foo' or 'foo][bar][baz'. Setting an error on 'foo' sets an error for every element where the #parents array starts with 'foo'.

@param $message

The error message to present to the user.

@param $limit\_validation\_errors

Internal use only. The #limit\_validation\_errors property of the clicked button, if it exists.

@return

Return value is for internal use only. To get a list of errors, use form\_get\_errors() or form\_get\_error().

## Handling Form

drupal\_form\_submit()

Retrieves, populates, and processes a form. This function allows you to supply values for form elements and submit a form for processing. Compare to drupal\_get\_form(), which also builds and processes a form, but does not allow you to supply values.

There is no return value, but you can check to see if there are errors by calling form\_get\_errors().

@param $form\_id

The unique string identifying the desired form.

@param $form\_state

A keyed array containing the current state of the form. Most important is the $form\_state['values'] collection, a tree of data used to simulate the incoming $\_POST information from a user's form submission. If a key is not filled in $form\_state['values'], then the default value of the respective element is used. To submit an unchecked checkbox or other control that browsers submit by not having a $\_POST entry, include the key, but set the value to NULL.

@param ...

Any additional arguments are passed on to the functions called by drupal\_form\_submit(), including the unique form constructor function. For example, the node\_edit form requires that a node object be passed in here when it is called. Arguments that need to be passed by reference should not be included here, but rather placed directly in the $form\_state build info array so that the reference can be preserved. For example, a form builder function with the following signature:

function mymodule\_form($form, &$form\_state, &$object) {

}

would be called via drupal\_form\_submit() as follows:

$form\_state['values'] = $my\_form\_values;

$form\_state['build\_info']['args'] = array(&$object);

drupal\_form\_submit('mymodule\_form', $form\_state);

For example:

// register a new user

$form\_state = array();

$form\_state['values']['name'] = 'robo-user';

$form\_state['values']['mail'] = 'robouser@example.com';

$form\_state['values']['pass']['pass1'] = 'password';

$form\_state['values']['pass']['pass2'] = 'password';

$form\_state['values']['op'] = t('Create new account');

drupal\_form\_submit('user\_register\_form', $form\_state);

## Redirect Form

drupal\_redirect\_form()

Redirects the user to a URL after a form has been processed.

After a form was executed, the data in $form\_state controls whether the form is redirected. By default, we redirect to a new destination page. The path of the destination page can be set in $form\_state['redirect'], as either a string containing the destination or an array of arguments compatible with drupal\_goto(). If that is not set, the user is redirected to the current page to display a fresh, unpopulated copy of the form.

For example, to redirect to 'node':

$form\_state['redirect'] = 'node';

Or to redirect to 'node/123?foo=bar#baz':

$form\_state['redirect'] = array(

'node/123',

array(

'query' => array(

'foo' => 'bar',

),

'fragment' => 'baz',

),

);

There are several triggers that may prevent a redirection though:

- If $form\_state['redirect'] is FALSE, a form builder function or form validation/submit handler does not want a user to be redirected, which means that drupal\_goto() is not invoked. For most forms, the redirection logic will be the same regardless of whether $form\_state['redirect'] is undefined or FALSE. However, in case it was not defined and the current request contains a 'destination' query string, drupal\_goto() will redirect to that given destination instead. Only setting $form\_state['redirect'] to FALSE will prevent any redirection.

- If $form\_state['no\_redirect'] is TRUE, then the callback that originally built the form explicitly disallows any redirection, regardless of the redirection value in $form\_state['redirect']. For example, ajax\_get\_form() defines $form\_state['no\_redirect'] when building a form in an Ajax callback to prevent any redirection. $form\_state['no\_redirect'] should NOT be altered by form builder functions or form validation/submit handlers.

- If $form\_state['programmed'] is TRUE, the form submission was usually invoked via drupal\_form\_submit(), so any redirection would break the script that invoked drupal\_form\_submit().

- If $form\_state['rebuild'] is TRUE, the form needs to be rebuilt without redirection.

## #States

Drupal\_process\_states

$form['toggle\_me'] = array(

'#type' => 'checkbox',

'#title' => t('Tick this box to type'),

);

$form['settings'] = array(

'#type' => 'textfield',

'#states' => array(

// Only show this field when the 'toggle\_me' checkbox is enabled.

'visible' => array(

':input[name="toggle\_me"]' => array('checked' => TRUE),

),

),

);

array(

'visible' => array(

JQUERY\_SELECTOR => REMOTE\_CONDITIONS,

JQUERY\_SELECTOR => REMOTE\_CONDITIONS,

...

),

)

## Form Element Types

They are render arrays, but have form relevant settings in there as well.

Form elements are already familiar to anyone who uses Form API. They share history with render elements.Form elements array are processed by drupal\_get\_form. Examples of core form elements are 'textfield', 'checkbox' and 'fieldset'. Drupal utilizes hook\_elements\_info() to define these FAPI types, and this occurs in the core function system\_elements\_info().

Each form element has a #type value that determines how it is treated by the Form API and how it is ultimately rendered into HTML. hook\_element\_info() allows modules to define new element types, and tells the Form API what default values they should automatically be populated with. By implementing hook\_element\_info() in your own module, you can create custom form (or render) elements with their own properties, validation and theming.

### System\_elements\_info()- system

#### 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

### Hook\_elements\_info()-custom

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.

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().

### Element theming

if you use hook\_elements() to define an element type, there is an implicit assumption that there will be a theme function with the same name as the element type used to render HTML of its instances. In this example, the elements will be passed to a theme function called theme\_example\_field(), if it exists. You don't need to add #theme to hook\_elements() (note: in Drupal 7, you do need to add #theme to hook\_element\_info()), it is assumed. If a theme function with the expected name does not exist and there is no other #theme set in the element, nothing will appear in the form.

<?php  
function theme\_example\_field($element) {  
  // Put your own rendering logic.  
  $output .= '<div class="title">'. t('My field') .'</div>';  
  $output .= '<div class="value">'. $element['#some\_value'] .'</div>';  
  
  // Or even render child elements standard way  
  // WARNING: Make sure that sub\_element is not and doesn't contain the element of your type, as you'll get a recursion.  
  $output .= '<div class="special-value">'. drupal\_render($element['sub\_element']) .'</div>';  
  
  return $output;  
}  
?>

With a hook\_theme

<?php  
function example\_theme() {  
  return array(  
    'example\_field' => array(  
       'arguments' => array('element' => NULL)  
    ),  
  );  
}  
?>

## Form element values

Normally FAPI assigns #value either based on what posted when the form is executed using drupal\_get\_form() or drupal\_execute(), or if there is nothing posted, then it copies #default\_value.

### Value callback or form\_type\_hook\_value()

The first opportunity after hook\_form\_alter() to change values is in form\_type\_hook\_value(). This is another implied callback for any element defined in hook\_elements(). No attribute is specified for this callback in hook\_elements(); it is implemented simply by creating a function with a name like form\_type\_example\_field\_value, ('form\_type\_'. $element\_name .'\_value') as follows:

<?php  
function form\_type\_example\_field\_value($element, $edit = FALSE) {  
  if (func\_num\_args() == 1) {  
     return $element['#default\_value'];  
  }  
  return $edit;  
}  
?>

The function is called in two different situations with different arguments. When there is no post, you only get one argument, $element, which is an array representing your element. When there is a post, $edit is whatever has been posted as a value for that element. Given that it's not possible to post FALSE from a form, it seems that using FALSE as a default value and branching on it is pretty safe. However, you can assign FALSE if you drupal\_execute your form so it's even better to use func\_num\_args here, as in the example code above.

When the form is unposted the above example will set the value to the default\_value of the field (the usual behavior). If $edit has a value, you could make changes to it before it is set. Anything you return here will be used as the value for the element.

If you just want #value set to #default\_value, you don't need this hook at all, since that will happen anyway. This function is mainly useful if you need to do something non-standard with the value, like transpose an array into an option list.

Some of the core FAPI elements that use hook\_value are select elements, which run drupal\_map\_assoc() over the posted $edit value, and password\_confirm, which expands the default value into two fields.

There is another way to set the value for the field: by adding '#value\_callback' to hook\_element() or to your form element, and providing a custom function that sets the value. That callback will get the same arguments as above, but provides a more flexible way to set a value. You can use that to set a value on a nested sub-element like those CCK uses, to use the same value callback for several different elements, or to add a custom value callback to a standard FAPI element type.

### Process Callback

The next opportunity in the form processing workflow to alter form values is in whatever callbacks are declared in the #process array of the element. These callbacks could be set by hook\_element, or added to the element when the form is created, or both. The callbacks are in an array, and are processed in whatever order they appear in the element, so you can pass an element through several processes, and control the order that this will happen by appending a new callback to the beginning or end of the array to have it process the element first or last.

The #process callback gets four arguments, $element, $edit, $form\_state and $complete\_form, and expects to have $element returned. This is the last place you can override the default value. By this point in the processing, #default\_value has been moved to #value in the element, so to change the value in here you need to alter $element['#value'], not $element['#default\_value']

<?php  
$form['my\_field'] = array(  
  '#type' => 'example\_field',  
  '#title' => t('My example Field'),  
  '#default\_value' => t('Good News!'),  
  '#process' => array('example\_process'),  
  '#element\_validate' => array('example\_validate'),  
  '#example\_attribute' => 'Something extra',  
);  
?>

<?php  
function form\_type\_example\_field\_value($element, $edit = FALSE) {  
  if (func\_num\_args() == 1) {  
     return $element['#default\_value'];  
  }  
  return $edit;  
}

The function is called in two different situations with different arguments. When there is no post, you only get one argument, $element, which is an array representing your element. When there is a post, $edit is whatever has been posted as a value for that element. Given that it's not possible to post FALSE from a form, it seems that using FALSE as a default value and branching on it is pretty safe. However, you can assign FALSE if you drupal\_execute your form so it's even better to use func\_num\_args here, as in the example code above.

When the form is unposted the above example will set the value to the default\_value of the field (the usual behavior). If $edit has a value, you could make changes to it before it is set. Anything you return here will be used as the value for the element.

If you just want #value set to #default\_value, you don't need this hook at all, since that will happen anyway. This function is mainly useful if you need to do something non-standard with the value, like transpose an array into an option list.

Some of the core FAPI elements that use hook\_value are select elements, which run drupal\_map\_assoc() over the posted $edit value, and password\_confirm, which expands the default value into two fields.

There is another way to set the value for the field: by adding '#value\_callback' to hook\_element() or to your form element, and providing a custom function that sets the value. That callback will get the same arguments as above, but provides a more flexible way to set a value. You can use that to set a value on a nested sub-element like those CCK uses, to use the same value callback for several different elements, or to add a custom value callback to a standard FAPI element type.

### Afterbuild Callback

After the element passes through #process and its value is set, it will be passed to any #after\_build callbacks. Just as with #process, there is an array of #after\_build callbacks, so several can be set, and each will get $element and $form\_state. At this point in the processing, you can expect #value to be properly and finally set, so this is an opportunity to make changes to the element that can only be made once #value will not change but before the form is submitted or rendered.

### Validation Callbacks

There are two kinds of validation callbacks, #validate and #element\_validate. The first does validation on the entire form, the second on only a single element.

Form-level validation happens before element validation, and receives as arguments $form and $form\_state. Element validation receives only the element's portion of the form, along with the element's portion of $form\_state.

In the above example, the element validation callback might look like:

<?php  
function example\_field\_validate($element, &$form\_state) {  
  if ($element['#value'] <= 10) {  
    form\_error($element, t('The number must be more than 10.'));  
    form\_set\_value($element, 0, $form\_state);  
  }  
  return $element;  
}  
?>

There are several core FAPI elements that use #element\_validate. They include password\_confirm, which checks that its two nested password fields match and unsets the extra fields once validation is done, and date, which checks that the date values create a valid date.

### Rendering Callbacks

There are two callbacks that come into play after the form has been created and is ready to be presented to the user, or when validation fails and the form is re-drawn. They are #pre\_render and #post\_render. These callbacks allow you to make changes to the form presentation. The core FAPI elements don't use these callbacks, but they are available for use by custom or contributed modules. #pre\_render will get $element as argument. #post\_render gets $content and $element as arguments, where $content is the rendered element.

### Submission Callbacks

The final callback is #submit. Like #process, it is an array which can contain several callbacks which will be called in the order they appear in the array. The submit callbacks get two arguments, $form, and $form\_state. If #submit is set on an element of the form, instead of at the top level of the form, $form will contain only the element's part of the form.

As used by core, #submit seems to make the most sense and work the best if set at the form level and used to do things like update databases or take action based on the entire form results. None of the core FAPI elements use #submit at the element level.

## Form Element Validation

\_form\_validate()

Performs validation on form elements. First ensures required fields are completed, #maxlength is not exceeded, and selected options were in the list of options given to the user. Then calls user-defined validators.

@param $elements

An associative array containing the structure of the form.

@param $form\_state

A keyed array containing the current state of the form. The current user-submitted data is stored in $form\_state['values'], though form validation functions are passed an explicit copy of the values for the sake of simplicity. Validation handlers can also $form\_state to pass information on to submit handlers. For example:

$form\_state['data\_for\_submission'] = $data;

This technique is useful when validation requires file parsing, web service requests, or other expensive requests that should not be repeated in the submission step.

@param $form\_id

A unique string identifying the form for validation, submission, theming, and hook\_form\_alter functions.

## Form Theming

Theme\_form()

drupal\_prepare\_form()

in drupal\_prepare\_form, if there is no #theme defined for this form, the default theme will be $form\_id, while system theme\_form() used as #theme\_wrappers as ‘form’ is added to #theme\_wrappers by default

this means that there will be looking for a default form theme as theme\_$form\_id() as the theme function for form element

and hook\_theme will be defined like

'form\_id' => array(

'render element' => 'form',

),

**array**

'form' =>

**array**

'text' =>

**array**

'#type' => string 'textfield' *(length=9)*

'#title' => string 'Please input something!' *(length=23)*

'#required' => boolean true

'#input' => boolean true

'#size' => int 60

'#maxlength' => int 128

'#autocomplete\_path' => boolean false

'#process' =>

**array**

...

'#theme' => string 'textfield' *(length=9)*

'#theme\_wrappers' =>

**array**

...

'#defaults\_loaded' => boolean true

'#tree' => boolean false

'#parents' =>

**array**

...

'#array\_parents' =>

**array**

...

'#weight' => int 0

'#processed' => boolean true

'#attributes' =>

**array**

...

'#title\_display' => string 'before' *(length=6)*

'#id' => string 'edit-text' *(length=9)*

'#name' => string 'text' *(length=4)*

'#value' => string '' *(length=0)*

'#ajax\_processed' => boolean false

'#sorted' => boolean true

'submit' =>

**array**

'#type' => string 'submit' *(length=6)*

'#value' => string 'Go' *(length=2)*

'#input' => boolean true

'#name' => string 'op' *(length=2)*

'#button\_type' => string 'submit' *(length=6)*

'#executes\_submit\_callback' => boolean true

'#limit\_validation\_errors' => boolean false

'#process' =>

**array**

...

'#theme\_wrappers' =>

**array**

...

'#defaults\_loaded' => boolean true

'#tree' => boolean false

'#parents' =>

**array**

...

'#array\_parents' =>

**array**

...

'#weight' => float 0.001

'#processed' => boolean true

'#required' => boolean false

'#attributes' =>

**array**

...

'#title\_display' => string 'before' *(length=6)*

'#id' => string 'edit-submit' *(length=11)*

'#ajax\_processed' => boolean false

'#sorted' => boolean true

'#form\_id' => string 'theming\_example\_text\_form' *(length=25)*

'#type' => string 'form' *(length=4)*

'#build\_id' => string 'form-gTaCl6eeOrmh5HepXxKHB9Ojquv9QlUEscpOpKWjdxI' *(length=48)*

'form\_build\_id' =>

**array**

'#type' => string 'hidden' *(length=6)*

'#value' => string 'form-gTaCl6eeOrmh5HepXxKHB9Ojquv9QlUEscpOpKWjdxI' *(length=48)*

'#id' => string 'form-gTaCl6eeOrmh5HepXxKHB9Ojquv9QlUEscpOpKWjdxI' *(length=48)*

'#name' => string 'form\_build\_id' *(length=13)*

'#input' => boolean true

'#process' =>

**array**

...

'#theme' => string 'hidden' *(length=6)*

'#defaults\_loaded' => boolean true

'#tree' => boolean false

'#parents' =>

**array**

...

'#array\_parents' =>

**array**

...

'#weight' => float 0.002

'#processed' => boolean true

'#required' => boolean false

'#attributes' =>

**array**

...

'#title\_display' => string 'before' *(length=6)*

'#ajax\_processed' => boolean false

'#sorted' => boolean true

'form\_id' =>

**array**

'#type' => string 'hidden' *(length=6)*

'#value' => string 'theming\_example\_text\_form' *(length=25)*

'#id' => string 'edit-theming-example-text-form' *(length=30)*

'#input' => boolean true

'#process' =>

**array**

...

'#theme' => string 'hidden' *(length=6)*

'#defaults\_loaded' => boolean true

'#tree' => boolean false

'#parents' =>

**array**

...

'#array\_parents' =>

**array**

...

'#weight' => float 0.003

'#processed' => boolean true

'#required' => boolean false

'#attributes' =>

**array**

...

'#title\_display' => string 'before' *(length=6)*

'#name' => string 'form\_id' *(length=7)*

'#ajax\_processed' => boolean false

'#sorted' => boolean true

'#id' => string 'theming-example-text-form' *(length=25)*

'#method' => string 'post' *(length=4)*

'#action' => string '/drupal7/examples/theming\_example/theming\_example\_text\_form' *(length=59)*

'#theme\_wrappers' =>

**array**

0 => string 'form' *(length=4)*

1 => string 'block' *(length=5)*

'#tree' => boolean false

'#parents' =>

**array**

*empty*

'#validate' =>

**array**

*empty*

'#submit' =>

**array**

0 => string 'theming\_example\_text\_form\_submit' *(length=32)*

'#theme' =>

**array**

0 => string 'theming\_example\_text\_form' *(length=25)*

'#processed' => boolean false

'#defaults\_loaded' => boolean true

'#required' => boolean false

'#attributes' =>

**array**

*empty*

'#title\_display' => string 'before' *(length=6)*

'#array\_parents' =>

**array**

*empty*

'#sorted' => boolean true

'#block' =>

**object**(*stdClass*)[*60*]

*public* 'bid' => string '71' *(length=2)*

*public* 'module' => string 'system' *(length=6)*

*public* 'delta' => string 'main' *(length=4)*

*public* 'theme' => string 'bartikcopy' *(length=10)*

*public* 'status' => string '1' *(length=1)*

*public* 'weight' => string '-8' *(length=2)*

*public* 'region' => string 'content' *(length=7)*

*public* 'custom' => string '0' *(length=1)*

*public* 'visibility' => string '0' *(length=1)*

*public* 'pages' => string '' *(length=0)*

*public* 'title' => string '' *(length=0)*

*public* 'cache' => string '-1' *(length=2)*

*public* 'subject' => string '' *(length=0)*

'#weight' => int 2

'#children' => string '' *(length=0)*

'theme\_hook\_suggestions' =>

**array**

*empty*

'zebra' => string 'odd' *(length=3)*

'id' => int 1

'directory' => string 'sites/all/modules/theming\_example' *(length=33)*

'classes\_array' =>

**array**

0 => string 'theming-example-text-form' *(length=25)*

'attributes\_array' =>

**array**

*empty*

'title\_attributes\_array' =>

**array**

*empty*

'content\_attributes\_array' =>

**array**

*empty*

'title\_prefix' =>

**array**

*empty*

'title\_suffix' =>

**array**

*empty*

'user' =>

**object**(*stdClass*)[*8*]

*public* 'uid' => int 0

*public* 'hostname' => string '127.0.0.1' *(length=9)*

*public* 'roles' =>

**array**

1 => string 'anonymous user' *(length=14)*

*public* 'cache' => int 0

'db\_is\_active' => boolean true

'is\_admin' => boolean false

'logged\_in' => boolean false

'is\_front' => boolean false

# Rendering

Drupal\_render($elements)

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()(this is part of form api)

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.

Different #types have themes defined against them, this will drive drupal\_render to right theme().

### #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.

Theme() is called and result is assigned to elements[‘children’], for the page theme in the page.tpl.php might have render in there too, it will become recursive just like no theme function in drupal\_render

The #theme value will be hook name, the whole render array will be passed in as $variables

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>

### #theme\_wrappers

This is applied after the initial theme or child rendering and the implementation should expect $element['#children'] to decorate

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.

The higher place in theme\_wrappers array, will be in outer wrappers.

The #theme\_wrapper each item value will be hook name, the whole render array will be passed in as $variables.

### #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.

t('cache demonstration') => array(

// If your expensive function were to be executed here it would happen

// on every page load regardless of the cache. The actual markup is

// added via the #pre\_render function, so that drupal\_render() will only

// execute the expensive function if this array has not been cached.

'#markup' => '',

'#pre\_render' => array('render\_example\_cache\_pre\_render'),

'#cache' => array(

'keys' => array('render\_example', 'cache', 'demonstration'),

'bin' => 'cache',

'expire' => time() + $interval,

'granularity' => DRUPAL\_CACHE\_PER\_PAGE | DRUPAL\_CACHE\_PER\_ROLE,

),

The above example shows that put the expensive markup building in pre\_render as in drupal\_render, it checks cache before pre\_render, if found the cache, it will just return straight away.

## 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

If no theme defined in the element, all the children of element will be run through drupal\_render respectively and attached to elements[‘children’]

## 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

### #attached

drupal\_process\_attached($elements, $group = JS\_DEFAULT, $dependency\_check = FALSE, $every\_page = NULL)

Libraries, JavaScript, CSS and other types of custom structures are attached to elements using the #attached property. The #attached property is an associative array, where the keys are the attachment types and the values are the attached data. For example:

$build['#attached'] = array(

'js' => array(drupal\_get\_path('module', 'taxonomy') . '/taxonomy.js'),

'css' => array(drupal\_get\_path('module', 'taxonomy') . '/taxonomy.css'),

);

'js', 'css', and 'library' are types that get special handling. For any other kind of attached data, the array key must be the full name of the callback function and each value an array of arguments. For example:

$build['#attached']['drupal\_add\_http\_header'] = array(

array('Content-Type', 'application/rss+xml; charset=utf-8'),

);

External 'js' and 'css' files can also be loaded. For example:

$build['#attached']['js'] = array(

'http://code.jquery.com/jquery-1.4.2.min.js' => array(

'type' => 'external',

),

);

An example of data and type

function drupal\_process\_states(&$elements) {

$elements['#attached']['library'][] = array('system', 'drupal.states');

$elements['#attached']['js'][] = array(

'type' => 'setting',

'data' => array('states' => array('#' . $elements['#id'] => $elements['#states'])),

);

}

From the code of drupal\_process\_attached, we can see that $data can either be the key or $build['#attached']['js'] array, or ‘data’ item of an array, in which case others will be passed in as $options

#### Library

drupal\_add\_library($module, $name, $every\_page = NULL)

In above case $module = ‘ststem’, $name = ‘drupal.states’

#### Js

drupal\_add\_js($data = NULL, $options = NULL)

$options: (optional) A string defining the type of JavaScript that is being added in the $data parameter ('file'/'setting'/'inline'/'external'), or an associative array. JavaScript settings should always pass the string 'setting' only. Other types can have the following elements in the array:

type: The type of JavaScript that is to be added to the page. Allowed values are 'file', 'inline', 'external' or 'setting'. Defaults to 'file'.

* **'file'**: Path to the file relative to [base\_path](http://api.drupal.org/api/drupal/includes%21common.inc/function/base_path/7)().
* **'inline'**: The JavaScript code that should be placed in the given scope.
* **'external'**: The absolute path to an external JavaScript file that is not hosted on the local server. These files will not be aggregated if JavaScript aggregation is enabled.
* **'setting'**: An associative array with configuration options. The array is merged directly into Drupal.settings. All modules should wrap their actual configuration settings in another variable to prevent conflicts in the Drupal.settings namespace. Items added with a string key will replace existing settings with that key; items with numeric array keys will be added to the existing settings array.

scope: The location in which you want to place the script. Possible values are 'header' or 'footer'. If your theme implements different regions, you can also use these. Defaults to 'header'.

group: A number identifying the group in which to add the JavaScript. Available constants are:

JS\_LIBRARY: Any libraries, settings, or jQuery plugins.

JS\_DEFAULT: Any module-layer JavaScript.

JS\_THEME: Any theme-layer JavaScript.

The group number serves as a weight: JavaScript within a lower weight group is presented on the page before JavaScript within a higher weight group.

every\_page: For optimal front-end performance when aggregation is enabled, this should be set to TRUE if the JavaScript is present on every page of the website for users for whom it is present at all. This defaults to FALSE. It is set to TRUE for JavaScript files that are added via module and theme .info files. Modules that add JavaScript within hook\_init() implementations, or from other code that ensures that the JavaScript is added to all website pages, should also set this flag to TRUE. All JavaScript files within the same group and that have the 'every\_page' flag set to TRUE and do not have 'preprocess' set to FALSE are aggregated together into a single aggregate file, and that aggregate file can be reused across a user's entire site visit, leading to faster navigation between pages. However, JavaScript that is only needed on pages less frequently visited, can be added by code that only runs for those particular pages, and that code should not set the 'every\_page' flag. This minimizes the size of the aggregate file that the user needs to download when first visiting the website. JavaScript without the 'every\_page' flag is aggregated into a separate aggregate file. This other aggregate file is likely to change from page to page, and each new aggregate file needs to be downloaded when first encountered, so it should be kept relatively small by ensuring that most commonly needed JavaScript is added to every page.

weight: A number defining the order in which the JavaScript is added to the page relative to other JavaScript with the same 'scope', 'group', and 'every\_page' value. In some cases, the order in which the JavaScript is presented on the page is very important. jQuery, for example, must be added to the page before any jQuery code is run, so jquery.js uses the JS\_LIBRARY group and a weight of -20, jquery.once.js (a library drupal.js depends on) uses the JS\_LIBRARY group and a weight of -19, drupal.js uses the JS\_LIBRARY group and a weight of -1, other libraries use the JS\_LIBRARY group and a weight of 0 or higher, and all other scripts use one of the other group constants. The exact ordering of JavaScript is as follows:

First by scope, with 'header' first, 'footer' last, and any other scopes provided by a custom theme coming in between, as determined by the theme.

Then by group.

Then by the 'every\_page' flag, with TRUE coming before FALSE.

Then by weight.

Then by the order in which the JavaScript was added. For example, all else being the same, JavaScript added by a call to drupal\_add\_js() that happened later in the page request gets added to the page after one for which drupal\_add\_js() happened earlier in the page request.

defer: If set to TRUE, the defer attribute is set on the &lt;script&gt; tag. Defaults to FALSE.

cache: If set to FALSE, the JavaScript file is loaded anew on every page call; in other words, it is not cached. Used only when 'type' references a JavaScript file. Defaults to TRUE.

preprocess: If TRUE and JavaScript aggregation is enabled, the script file will be aggregated. Defaults to TRUE.

#### Css

drupal\_add\_css($data = NULL, $options = NULL)

$options: (optional) A string defining the 'type' of CSS that is being added in the $data parameter ('file', 'inline', or 'external'), or an array which can have any or all of the following keys:

'type': The type of stylesheet being added. Available options are 'file', 'inline' or 'external'. Defaults to 'file'.

* **'file'**: The path to the CSS file relative to the [base\_path](http://api.drupal.org/api/drupal/includes%21common.inc/function/base_path/7)(), or a stream wrapper URI. For example: "modules/devel/devel.css" or "public://generated\_css/stylesheet\_1.css". Note that Modules should always prefix the names of their CSS files with the module name; for example, system-menus.css rather than simply menus.css. Themes can override module-supplied CSS files based on their filenames, and this prefixing helps prevent confusing name collisions for theme developers. See [drupal\_get\_css](http://api.drupal.org/api/drupal/includes%21common.inc/function/drupal_get_css/7)() where the overrides are performed. Also, if the direction of the current language is right-to-left (Hebrew, Arabic, etc.), the function will also look for an RTL CSS file and append it to the list. The name of this file should have an '-rtl.css' suffix. For example, a CSS file called 'mymodule-name.css' will have a 'mymodule-name-rtl.css' file added to the list, if exists in the same directory. This CSS file should contain overrides for properties which should be reversed or otherwise different in a right-to-left display.
* **'inline'**: A string of CSS that should be placed in the given scope. Note that it is better practice to use 'file' stylesheets, rather than 'inline', as the CSS would then be aggregated and cached.
* **'external'**: The absolute path to an external CSS file that is not hosted on the local server. These files will not be aggregated if CSS aggregation is enabled.

'basename': Force a basename for the file being added. Modules are expected to use stylesheets with unique filenames, but integration of external libraries may make this impossible. The basename of 'modules/node/node.css' is 'node.css'. If the external library "node.js" ships with a 'node.css', then a different, unique basename would be 'node.js.css'.

'group': A number identifying the group in which to add the stylesheet. Available constants are:

CSS\_SYSTEM: Any system-layer CSS.

CSS\_DEFAULT: (default) Any module-layer CSS.

CSS\_THEME: Any theme-layer CSS.

The group number serves as a weight: the markup for loading a stylesheet within a lower weight group is output to the page before the markup for loading a stylesheet within a higher weight group, so CSS within higher weight groups take precendence over CSS within lower weight groups.

'every\_page': For optimal front-end performance when aggregation is enabled, this should be set to TRUE if the stylesheet is present on every page of the website for users for whom it is present at all. This defaults to FALSE. It is set to TRUE for stylesheets added via module and theme .info files. Modules that add stylesheets within hook\_init() implementations, or from other code that ensures that the stylesheet is added to all website pages, should also set this flag to TRUE. All stylesheets within the same group that have the 'every\_page' flag set to TRUE and do not have 'preprocess' set to FALSE are aggregated together into a single aggregate file, and that aggregate file can be reused across a user's entire site visit, leading to faster navigation between pages. However, stylesheets that are only needed on pages less frequently visited, can be added by code that only runs for those particular pages, and that code should not set the 'every\_page' flag. This minimizes the size of the aggregate file that the user needs to download when first visiting the website. Stylesheets without the 'every\_page' flag are aggregated into a separate aggregate file. This other aggregate file is likely to change from page to page, and each new aggregate file needs to be downloaded when first encountered, so it should be kept relatively small by ensuring that most commonly needed stylesheets are added to every page.

'weight': The weight of the stylesheet specifies the order in which the CSS will appear relative to other stylesheets with the same group and 'every\_page' flag. The exact ordering of stylesheets is as follows:

First by group.

Then by the 'every\_page' flag, with TRUE coming before FALSE.

Then by weight.

Then by the order in which the CSS was added. For example, all else being the same, a stylesheet added by a call to drupal\_add\_css() that happened later in the page request gets added to the page after one for which drupal\_add\_css() happened earlier in the page request.

'media': The media type for the stylesheet, e.g., all, print, screen. Defaults to 'all'.

'preprocess': If TRUE and CSS aggregation/compression is enabled, the styles will be aggregated and compressed. Defaults to TRUE.

'browsers': An array containing information specifying which browsers should load the CSS item. See drupal\_pre\_render\_conditional\_comments() for details.

#### callback

#### hook\_library()

To define dependencies in library type

drupal\_add\_library($module, $name, $every\_page = NULL)

Adds multiple JavaScript or CSS files at the same time.

A library defines a set of JavaScript and/or CSS files, optionally using settings, and optionally requiring another library. For example, a library can be a jQuery plugin, a JavaScript framework, or a CSS framework. This function allows modules to load a library defined/shipped by itself or a depending module, without having to add all files of the library separately. Each library is only loaded once.

# 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

As an example we are using one of common theme function item\_list, we have such in registry (includes/common.inc/ drupal\_common\_theme())

'item\_list' => array(

'variables' => array('items' => array(), 'title' => NULL, 'type' => 'ul', 'attributes' => array()),

),

Render array will be like

'just\_a\_list\_of\_links' => array(

'#theme' => 'item\_list',

'#items' => array('Item 1', 'Item 2', 'Item 3'),

'#title' => t('My item list'),

'#type' => 'ol',

),

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

And the function interface is like

function theme\_item\_list($variables) {

$items = $variables['items'];

$title = $variables['title'];

$type = $variables['type'];

$attributes = $variables['attributes'];

}

If not found, looks for 'render element' in theme registry,

**function** myrender\_example\_theme()

{

**return array**(

'example\_link\_list' => **array**(

'render element' => 'links',

),

),

);

}

And render array should be like

'just\_a\_list\_of\_links' => **array**(

'#theme' => 'example\_link\_list',

'#children' => **array**(

0 => **array**(

'label' => t('Step 1'),

'url' => 'http://www.example.com/1',

),

1 => **array**(

'label' => t('Step 2'),

'url' => 'http://www.example.com/2',

),

),

),

if found, pass render array element into $variables[the value of 'render element’ in theme registry] , basically it just inserts a new level of final array structure $variables .

So it will be like

$variables[‘links’] =

**array**(

'#theme' => 'example\_link\_list',

'#children' => **array**(

0 => **array**(

'label' => t('Step 1'),

'url' => 'http://www.example.com/1',

),

1 => **array**(

'label' => t('Step 2'),

'url' => 'http://www.example.com/2',

),

),

)

Then in theme implementation

**function** theme\_example\_link\_list($variables = **array**()) {

$links = **array**();

**foreach** ($variables['links']['#children'] **as** $link) {

$links[] = l($link['label'], $link['url']);

}

**return** theme('item\_list', **array**('items' => $links));

}

#### #variables passed through

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

<?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),  
    ),  
//...  
  );  
}  
?>

As a result, the array will passed straight through.

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

## Registration Hook\_theme:

This is actually the interface definition that must be respected by render array and theme function implementation.

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.

If the implementation is a template file, several functions are called before the template file is invoked, to modify the $variables array. These fall into the "preprocessing" phase and the "processing" phase, and are executed (if they exist), in the following order (note that in the following list, HOOK indicates the theme hook name, MODULE indicates a module name, THEME indicates a theme name, and ENGINE indicates a theme engine name):

\* template\_preprocess(&$variables, $hook): Creates a default set of variables for all theme hooks.

\* template\_preprocess\_HOOK(&$variables): Should be implemented by the module that registers the theme hook, to set up default variables.

\* MODULE\_preprocess(&$variables, $hook): hook\_preprocess() is invoked on all implementing modules.

\* MODULE\_preprocess\_HOOK(&$variables): hook\_preprocess\_HOOK() is invoked on all implementing modules, so that modules that didn't define the theme hook can alter the variables.

\* ENGINE\_engine\_preprocess(&$variables, $hook): Allows the theme engine to set necessary variables for all theme hooks.

\* ENGINE\_engine\_preprocess\_HOOK(&$variables): Allows the theme engine to set necessary variables for the particular theme hook.

\* THEME\_preprocess(&$variables, $hook): Allows the theme to set necessary variables for all theme hooks.

\* THEME\_preprocess\_HOOK(&$variables): Allows the theme to set necessary variables specific to the particular theme hook.

## 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().

### $attributes

Some theme function can use $attributes and drupal\_attributes($attributes) to export attributes in html element

With $attributes[key]= array(value1,value2…), this will export key=”value1,value2…” in an html element, when a css class is applied you can

$attributes['class'][] = 'first';

Then $output .= '<li' . drupal\_attributes($attributes) . '>' . $data . "</li>\n";

will get string '<li class=”first”>item1</li>'

## Dynamic theming:

This has two side, one is how to construct a pattern that is going to be used to search registry, the other side of it is how to provide implementations that might match with those patterns

### Pattern for suggestions

This will generate a pattern according to its args and base, base is normally the hook name, the args are url sections.

function theme\_get\_suggestions($args, $base, $delimiter = '\_\_')

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.

Build a list of suggested theme hooks or body classes in order of specificity. One suggestion is made for every element of the current path, though numeric elements are not carried to subsequent suggestions. For example, for $base='page', http://www.example.com/node/1/edit would result in the following suggestions and body classes:

page\_\_node page-node

page\_\_node\_\_% page-node-%

page\_\_node\_\_1 page-node-1

page\_\_node\_\_edit page-node-edit.

Suggestions result is double \_\_ delimited string like page\_\_node

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

### Implemented suggestions

These will be functions that are defined by hook\_theme, or functions defined in template.php under theme

Like bartikcopy\_field\_\_taxonomy\_term\_reference in bartikcopy theme will register in $hooks which is ThemeRegistry an entry like :

field\_\_taxonomy\_term\_reference Array [5]

function bartikcopy\_field\_\_taxonomy\_term\_reference

render element element

base hook field

type theme\_engine

theme path sites/all/themes/bartikcopy

Then a suggestion field\_\_taxonomy\_term\_reference will be matched up and the renderarry will be taken up and run through.

This explained how dynamic theming works from template implementation

## 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 clear the [theme registry](http://drupal.org/node/173880#theme-registry) to see them.

## Page rendering example :

This is based on theme()

Page Hook

$info Array [7]

template page

path sites/all/themes/bartikcopy/templates

type theme\_engine

theme path sites/all/themes/bartikcopy

render element page

preprocess functions Array [5]

0 template\_preprocess // theme.inc

1 template\_preprocess\_page

2 contextual\_preprocess

3 overlay\_preprocess\_page

4 shortcut\_preprocess\_page

process functions Array [4]

0 template\_process

1 template\_process\_page

2 rdf\_process

3 bartikcopy\_process\_page

Page Variables:

$variables Array [10]

#show\_messages true

#theme page

#theme\_wrappers Array [1]

#type page

content Array [5]

sidebar\_first Array [6]

footer\_firstcolumn Array [4]

footer Array [4]

page\_top Array [3]

#children

template\_preprocess:

Page variables is

$variables Array [16]

page Array [10] --- original page variables

theme\_hook\_suggestions Array [0]

zebra odd

id 1

directory sites/all/themes/bartikcopy

classes\_array Array [1]

attributes\_array Array [0]

title\_attributes\_array Array [0]

content\_attributes\_array Array [0]

title\_prefix Array [0]

title\_suffix Array [0]

user stdClass

db\_is\_active true

is\_admin false

logged\_in false

is\_front false

template\_preprocess\_page:

$variables Array [29]

page Array [25]

#show\_messages true

#theme page

#theme\_wrappers Array [1]

#type page

content Array [5]

sidebar\_first Array [6]

footer\_firstcolumn Array [4]

footer Array [4]

page\_top Array [3]

#children

header Array [0]

help Array [0]

page\_bottom Array [0]

highlighted Array [0]

featured Array [0]

sidebar\_second Array [0]

triptych\_first Array [0]

triptych\_middle Array [0]

triptych\_last Array [0]

footer\_secondcolumn Array [0]

footer\_thirdcolumn Array [0]

footer\_fourthcolumn Array [0]

dashboard\_main Array [0]

dashboard\_sidebar Array [0]

dashboard\_inactive Array [0]

theme\_hook\_suggestions Array [2]

zebra odd

id 1

directory sites/all/themes/bartikcopy

classes\_array Array [1]

attributes\_array Array [0]

title\_attributes\_array Array [0]

content\_attributes\_array Array [0]

title\_prefix Array [0]

title\_suffix Array [0]

user stdClass

db\_is\_active true

is\_admin false

logged\_in false

is\_front false

show\_messages true

layout first

base\_path /drupal7/

front\_page /drupal7/

feed\_icons

language stdClass

logo http://localhost:81/drupal7/sites/all/themes/bartikcopy/logo.png

main\_menu Array [2]

secondary\_menu Array [0]

action\_links Array [0]

site\_name drupal on localhost

site\_slogan

tabs Array [3]

contextual\_preprocess:

overlay\_preprocess\_page:

shortcut\_preprocess\_page://modules/shortcut/shortcut.module

template\_process:

$variables Array [33]

[0...30]

page Array [25]

#show\_messages true

#theme page

#theme\_wrappers Array [1]

#type page

content Array [5]

sidebar\_first Array [6]

footer\_firstcolumn Array [4]

footer Array [4]

page\_top Array [3]

#children

header Array [0]

help Array [0]

page\_bottom Array [0]

highlighted Array [0]

featured Array [0]

sidebar\_second Array [0]

triptych\_first Array [0]

triptych\_middle Array [0]

triptych\_last Array [0]

footer\_secondcolumn Array [0]

footer\_thirdcolumn Array [0]

footer\_fourthcolumn Array [0]

dashboard\_main Array [0]

dashboard\_sidebar Array [0]

dashboard\_inactive Array [0]

theme\_hook\_suggestions Array [2]

0 page\_\_examples

1 page\_\_examples\_\_myrender\_example

zebra odd

id 1

directory sites/all/themes/bartikcopy

classes\_array Array [1]

attributes\_array Array [0]

title\_attributes\_array Array [0]

content\_attributes\_array Array [0]

title\_prefix Array [0]

title\_suffix Array [0]

user stdClass

db\_is\_active true

is\_admin false

logged\_in false

is\_front false

show\_messages true

layout first

base\_path /drupal7/

front\_page /drupal7/

feed\_icons

language stdClass

logo http://localhost:81/drupal7/sites/all/themes/bartikcopy/logo.png

main\_menu Array [2]

menu-308 Array [3]

menu-218 Array [2]

secondary\_menu Array [0]

action\_links Array [0]

site\_name drupal on localhost

site\_slogan

tabs Array [3]

#theme menu\_local\_tasks

#primary

#secondary

classes page

attributes

[31...32]

title\_attributes

content\_attributes

template\_process\_page:

bread crumbs,,title etc

bartikcopy\_process\_page:// sites/all/themes/bartikcopy/template.php

$template\_file = sites/all/themes/bartikcopy/templates/page.tpl.php

call

theme\_render\_template

with above template file and $variabbles

$variables Array [38]

[0...30]

page Array [25]

#show\_messages true

#theme page

#theme\_wrappers Array [1]

#type page

content Array [5]

sidebar\_first Array [6]

footer\_firstcolumn Array [4]

footer Array [4]

page\_top Array [3]

#children

header Array [0]

help Array [0]

page\_bottom Array [0]

highlighted Array [0]

featured Array [0]

sidebar\_second Array [0]

triptych\_first Array [0]

triptych\_middle Array [0]

triptych\_last Array [0]

footer\_secondcolumn Array [0]

footer\_thirdcolumn Array [0]

footer\_fourthcolumn Array [0]

dashboard\_main Array [0]

dashboard\_sidebar Array [0]

dashboard\_inactive Array [0]

theme\_hook\_suggestions Array [2]

zebra odd

id 1

directory sites/all/themes/bartikcopy

classes\_array Array [1]

attributes\_array Array [0]

title\_attributes\_array Array [0]

content\_attributes\_array Array [0]

title\_prefix Array [0]

title\_suffix Array [0]

user stdClass

db\_is\_active true

is\_admin false

logged\_in false

is\_front false

show\_messages true

layout first

base\_path /drupal7/

front\_page /drupal7/

feed\_icons

language stdClass

logo http://localhost:81/drupal7/sites/all/themes/bartikcopy/logo.png

main\_menu Array [2]

secondary\_menu Array [0]

action\_links Array [0]

site\_name drupal on localhost

site\_slogan

tabs Array [3]

classes page

attributes

[31...37]

title\_attributes

content\_attributes

breadcrumb <h2 class="element-invisible">You are here</h2><div class="breadcrumb"><a href="/drupal7/">Home</a></div>

title My Render Example

messages

hide\_site\_name false

hide\_site\_slogan false

Child renderarrays will be executed by the above function

rdf\_process://modules/rdf/rdf.module

# 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