Table of Contents

Developing for Drupal ................................................. 1
Contributing to Drupal .................................................. 2
Types of Contributions .................................................. 2
Task list ................................................................. 3
Bug reports ............................................................... 3
How to report bugs effectively .......................................... 3
Feature suggestions .................................................... 10
Patches ................................................................. 10
Diff and patch ........................................................... 10
Diff on Windows ......................................................... 12
Patching with Cygwin ................................................... 13
Gnuwin32 diff and patch ................................................. 14
Patch on Windows ....................................................... 14
Using Cygwin to patch in Windows .................................... 15
HOWTO: Set up a test environment to help review patches ........... 16
Creating and submitting patches ....................................... 18
Tips for reviewing patches ............................................. 19
General guidelines ...................................................... 19
Reviewing process ...................................................... 20
Authoring a patch review ............................................... 20
Severity levels of bugs .................................................. 20
The revision process ..................................................... 21
Criteria for evaluating proposed changes .............................. 22
Maintaining a project on drupal.org .................................... 22
 Downloads and packaging .............................................. 23
Managing releases ....................................................... 23
Orphaned projects ...................................................... 23
Tips for contributing to the core ...................................... 24
List of maintainers ...................................................... 24
Mailing lists ............................................................. 26
Support ................................................................. 26
Development ............................................................ 26
Themes ................................................................. 26
Documentation ........................................................... 26
Translations .............................................................. 26
Consulting ............................................................... 26
CVS commits ............................................................. 27
Infrastructure ........................................................... 27
DrupalCON ............................................................... 27
Subscribe ............................................................... 27
Mailing of project issues ................................................ 27
Coding standards ....................................................... 28
Drupal Coding Standards .............................................. 28
Indenting ............................................................... 28
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control Structures</td>
<td>28</td>
</tr>
<tr>
<td>Function Calls</td>
<td>29</td>
</tr>
<tr>
<td>Function Declarations</td>
<td>29</td>
</tr>
<tr>
<td>Arrays</td>
<td>29</td>
</tr>
<tr>
<td>Comments</td>
<td>30</td>
</tr>
<tr>
<td>Including Code</td>
<td>30</td>
</tr>
<tr>
<td>PHP Code Tags</td>
<td>30</td>
</tr>
<tr>
<td>Header Comment Blocks</td>
<td>31</td>
</tr>
<tr>
<td>Using CVS</td>
<td>31</td>
</tr>
<tr>
<td>Example URLs</td>
<td>31</td>
</tr>
<tr>
<td>Naming Conventions</td>
<td>31</td>
</tr>
<tr>
<td>Functions and Methods</td>
<td>31</td>
</tr>
<tr>
<td>Constants</td>
<td>32</td>
</tr>
<tr>
<td>Global Variables</td>
<td>32</td>
</tr>
<tr>
<td>Filenames</td>
<td>32</td>
</tr>
<tr>
<td>Doxygen formatting conventions</td>
<td>32</td>
</tr>
<tr>
<td>Configuring vim</td>
<td>34</td>
</tr>
<tr>
<td>Indentation</td>
<td>34</td>
</tr>
<tr>
<td>Syntax highlighting</td>
<td>34</td>
</tr>
<tr>
<td>Using these settings only with drupal</td>
<td>35</td>
</tr>
<tr>
<td>Indenting</td>
<td>35</td>
</tr>
<tr>
<td>PHP Code tags</td>
<td>35</td>
</tr>
<tr>
<td>SQL naming conventions</td>
<td>36</td>
</tr>
<tr>
<td>String concatenations</td>
<td>37</td>
</tr>
<tr>
<td>Use of icons</td>
<td>37</td>
</tr>
<tr>
<td>Write E_ALL compliant code</td>
<td>37</td>
</tr>
<tr>
<td>Introductionary notice</td>
<td>37</td>
</tr>
<tr>
<td>E_ALL: a better practice</td>
<td>37</td>
</tr>
<tr>
<td>Common coding mistakes and new coding practice</td>
<td>38</td>
</tr>
<tr>
<td>1- Use of if (isset($var)) or if (!empty($var))</td>
<td>38</td>
</tr>
<tr>
<td>Testing for error notices</td>
<td>39</td>
</tr>
<tr>
<td>Functions</td>
<td>39</td>
</tr>
<tr>
<td>Constants</td>
<td>40</td>
</tr>
<tr>
<td>Control structures</td>
<td>40</td>
</tr>
<tr>
<td>Header comment blocks</td>
<td>40</td>
</tr>
<tr>
<td>CVS</td>
<td>42</td>
</tr>
<tr>
<td>CVS concepts</td>
<td>42</td>
</tr>
<tr>
<td>Using CVS with branches and tags</td>
<td>43</td>
</tr>
<tr>
<td>Windows</td>
<td>44</td>
</tr>
<tr>
<td>Available Branches</td>
<td>45</td>
</tr>
<tr>
<td>Apply for contributions CVS access</td>
<td>45</td>
</tr>
<tr>
<td>CVS GUIs and clients</td>
<td>45</td>
</tr>
<tr>
<td>Cross-platform CVS clients</td>
<td>45</td>
</tr>
<tr>
<td>Eclipse CVS plug-in</td>
<td>45</td>
</tr>
<tr>
<td>Checkout Drupal CVS into the Eclipse Workspace</td>
<td>45</td>
</tr>
<tr>
<td>Installing Eclipse with PHP Support</td>
<td>46</td>
</tr>
<tr>
<td>Checkout Drupal in your Eclipse Workspace</td>
<td>46</td>
</tr>
</tbody>
</table>
CVS front ends for Windows ................. 47
   SmartCVS .................................. 47
   TortoiseCVS ................................ 47
   WinCVS .................................... 48
      Adding a new project ..................... 48
      Checking out a project using WinCVS .... 49
CVS on Mac OS X ............................... 49
   CVL: point and click CVS ................. 49
      Setting up/step by step CVS ............ 50
      Basic CVS with CVL ..................... 51
      Preparing a project .................... 52
      Committing a project ................... 52
Drupal CVS repositories ................. 53
   Main repository ........................... 53
   Contributions repository .............. 54
   Promoting a project to be an official release 55
   Adding/modifying a file to the CVS repository 56
      Adding .................................. 56
      Modifying ............................... 57
   Demoting an official release ............ 57
   Tracking Drupal source with CVS ........ 57
      Example ................................ 57
      Managing contributions ................. 58
      Updating the vendor branch .......... 61
      Summary ................................ 62
   Additional resources ..................... 62
Sandbox maintenance rules .............. 62
Commit messages—providing history and credit 63
   Give details ................................ 63
   Provide a link to the drupal.org issue, if available 63
   Give credit ................................ 63
   Example ................................ 63
Additional references ..................... 64
Drupal’s APIs ................................. 65
   Forms API ................................ 65
      Forms API Quickstart Guide .......... 67
      Forms API reference .................. 67
   Multipage forms with the Forms API ... 67
      Why can’t I make multipage forms with what I already know? 67
      A Fully Working Example .......... 68
Tips and Tricks ............................... 69
   #tree and #parents ......................... 69
   Adding a custom element type & expanding elements 71
   Adding and theming additional fields to a node form 73
   Creating an array of form elements ...... 74
   Creating fieldsets outside forms with minimal code 75
   Creating multi-part forms ............... 75

iii
Creating submit buttons with images ........................................... 78
Easier debugging of forms code .................................................. 79
Getting a form element without a form ....................................... 79
Modifying checkboxes to display in multiple columns .................. 80
Module dependency checker ...................................................... 82
The #ref property .................................................................... 84
Writing forms in tables ............................................................. 87
Upgrading to forms API ............................................................. 88
Example module conversion: Project Module ................................. 89
  Overview ............................................................................... 89
  Conversion Tips ...................................................................... 90
  Form Updater ........................................................................ 90
Common legacy form errors ....................................................... 91
project_settings: Converting a hook_settings implementation (Easy) 92
  Overview ............................................................................... 92
  Original function .................................................................... 93
  Form conversion ...................................................................... 94
  hook_settings changes in 4.7 .................................................. 95
  Default values ....................................................................... 96
  Converted function .................................................................. 96
Core module before and after examples ....................................... 98
  Standard example: Path Form .................................................. 98
    Before ................................................................................. 98
    After .................................................................................. 99
  Fieldsets and advanced fields: system_view_general ................. 100
    Before ................................................................................. 100
    After .................................................................................. 103
  Validation and execution functions: contact_mail_page ............. 113
    Before ................................................................................. 113
    After .................................................................................. 115
  Theming forms: system_themes .............................................. 119
    Before ................................................................................. 119
    After .................................................................................. 120
  Advanced theming: system_user ............................................. 125
    Before ................................................................................. 125
    After .................................................................................. 126
Drupal 4.6 vs. Drupal 4.7 Forms API Flowcharts ............................... 128
  Drupal Pre-4.7 Forms API ..................................................... 128
  Drupal 4.7 Forms API .......................................................... 130
Forms API FAQ ........................................................................ 132
  Validation .............................................................................. 132
  How do you use the #validation arguments? ......................... 132
  How do you validate a URL, e-mail address or integer value? .... 132
  is the nodeapi ‘validate’ op to be used anymore given the validation features of the new api? if so, when? . 133
  Form logic ............................................................................. 133
  How do you handle multiple submit buttons in a form? Where do you put the
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>dispatch logic?</td>
<td>133</td>
</tr>
<tr>
<td>What does #post_process do?</td>
<td>133</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>133</td>
</tr>
<tr>
<td>How do I add a taxonomy selection to my form?</td>
<td>133</td>
</tr>
<tr>
<td>Why was the #attribute_name convention chosen?</td>
<td>133</td>
</tr>
<tr>
<td>Your own drupaldocs or api.drupal.org site</td>
<td>134</td>
</tr>
<tr>
<td>Module developer's guide</td>
<td>136</td>
</tr>
<tr>
<td>Introduction to Drupal modules</td>
<td>136</td>
</tr>
<tr>
<td>Creating modules - a tutorial</td>
<td>136</td>
</tr>
<tr>
<td>01. Getting started</td>
<td>137</td>
</tr>
<tr>
<td>02. Telling Drupal about your module</td>
<td>137</td>
</tr>
<tr>
<td>03. Telling Drupal who can use your module</td>
<td>138</td>
</tr>
<tr>
<td>04. Declare we have block content</td>
<td>140</td>
</tr>
<tr>
<td>05. Generate the block content</td>
<td>141</td>
</tr>
<tr>
<td>06. Installing, enabling and testing the module</td>
<td>144</td>
</tr>
<tr>
<td>07. Create a module configuration (settings) page</td>
<td>145</td>
</tr>
<tr>
<td>08. Generate a page content</td>
<td>148</td>
</tr>
<tr>
<td>09. Letting Drupal know about the new function</td>
<td>149</td>
</tr>
<tr>
<td>10. Adding a ‘more’ link and showing all entries</td>
<td>150</td>
</tr>
<tr>
<td>Conclusion</td>
<td>151</td>
</tr>
<tr>
<td>Documenting your code</td>
<td>151</td>
</tr>
<tr>
<td>How to create your own simple node type (from story node)</td>
<td>151</td>
</tr>
<tr>
<td>How to create your own simple node type (from story node) (Drupal 4.7)</td>
<td>159</td>
</tr>
<tr>
<td>Third party applications integration guide</td>
<td>169</td>
</tr>
<tr>
<td>Session handler issues</td>
<td>169</td>
</tr>
<tr>
<td>Sharing a user base</td>
<td>170</td>
</tr>
<tr>
<td>Theme engine integration</td>
<td>172</td>
</tr>
<tr>
<td>Using Javascript and AJAX</td>
<td>173</td>
</tr>
<tr>
<td>Drupal's Javascript tools</td>
<td>173</td>
</tr>
<tr>
<td>drupal.js functions</td>
<td>173</td>
</tr>
<tr>
<td>Testing for appropriate Javascript support</td>
<td>174</td>
</tr>
<tr>
<td>AJAX data exchanges</td>
<td>174</td>
</tr>
<tr>
<td>Working with CSS class names</td>
<td>174</td>
</tr>
<tr>
<td>Element position</td>
<td>175</td>
</tr>
<tr>
<td>Adding events</td>
<td>175</td>
</tr>
<tr>
<td>Tutorial 1: Creating new Javascript widgets</td>
<td>175</td>
</tr>
<tr>
<td>Tutorial 2: Using existing Javascript widgets: autocomplete</td>
<td>178</td>
</tr>
<tr>
<td>Prebuilt autocomplete functions</td>
<td>179</td>
</tr>
<tr>
<td>Building a custom autocomplete function</td>
<td>179</td>
</tr>
<tr>
<td>Tutorial 3: Creating new widgets with AJAX</td>
<td>180</td>
</tr>
<tr>
<td>AJAX solution components</td>
<td>181</td>
</tr>
<tr>
<td>Example: click_info with AJAX</td>
<td>181</td>
</tr>
<tr>
<td>1. Marking up content</td>
<td>181</td>
</tr>
<tr>
<td>2. The Javascript</td>
<td>182</td>
</tr>
<tr>
<td>Objects and methods</td>
<td>182</td>
</tr>
<tr>
<td>3. The Handler</td>
<td>184</td>
</tr>
<tr>
<td>More examples</td>
<td>186</td>
</tr>
</tbody>
</table>
**Additional tools and approaches**

- Using the APIs available through contributed modules
  - Views
  - Actions and Workflows
  - E-Commerce
  - Location
- Writing .install files
- Install instructions
- Update instructions
- Drupal’s menu building mechanism
- Drupal’s node building mechanism
- Drupal’s page serving mechanism

**Updating your modules**

- Converting 3.0 modules to 4.0
- Converting 4.0 modules to 4.1
  - Required changes
  - Optional changes
- Converting 4.1 modules to 4.2
- Converting 4.2 modules to 4.3
- Creating modules for version 4.3.1
  - Getting Started
    - Telling Drupal about your module
    - Telling Drupal who can use your module
    - Announce we have block content
    - Generate content for a block
    - Installing, enabling and testing the module
    - Create a module configuration (settings) page
    - Adding menu links and creating page content
    - Letting Drupal know about the new function
    - Adding a more link and showing all entries
  - Conclusion
  - How to build up a _help hook
  - How to convert a _system hook
  - How to convert an _auth_help hook
- Converting 4.3 modules to 4.4
  - Menu system
  - Theme system
  - Node system
  - Filter system
  - Hook changes
  - Emitting links
  - Status and error messages
- Converting 4.4 modules to 4.5
  - Menu system
  - Path changes
  - Node changes
  - Filtering changes
Check_output() changes ......................................................... 231
Filter hook ........................................................................... 231
Filter tips .............................................................................. 232
Other changes ....................................................................... 233
Converting 4.5 modules to 4.6 .................................................. 233
  Block system ....................................................................... 233
  Search system ...................................................................... 234
  Module paths ....................................................................... 234
  Database backend .................................................................. 234
  Theme system ...................................................................... 234
  Watchdog messages ............................................................ 235
  Node markers ..................................................................... 235
  Control over destination page after form processing ............ 235
  Confirmation messages ......................................................... 235
  Inter module calls .............................................................. 235
  Node queries ...................................................................... 236
  Text output ......................................................................... 236
Converting XML-RPC using modules ....................................... 237
Converting 4.6 modules to 4.7 .................................................. 239
  Overview of Drupal API changes in 4.7 ................................. 239
  New handling of return values from callbacks ....................... 240
    node definition changes .................................................... 240
    node_load() changes ......................................................... 242
    node_save() changes ......................................................... 242
    node_list() changes ........................................................ 242
  Node titles now handled by node modules ............................. 243
    module_get_node_name deprecated ..................................... 243
    format_name() renamed .................................................... 244
    theme_table() change ......................................................... 244
    check_output() change .................................................... 245
    XML-RPC changes ............................................................ 245
    Taxonomy API change ....................................................... 245
    message_access() removed ................................................. 246
    Unicode string API .......................................................... 246
    conf_url_rewrite() became custom_url_rewrite() ............... 247
    node_delete(): moderately used ....................................... 247
    New order of node hooks .................................................. 248
      hook_nodeapi('settings', ...) replaced by form api ......... 248
      hook_nodeapi('form x', ...) replaced by form api .......... 249
    file_directory variables replaced by functions .................... 249
      array2object replaced by native PHP type conversion .......... 249
      user_load returns FALSE if a user cannot be loaded ......... 250
    MySQL tables are now always UTF-8 encoded ...................... 250
    We no longer use the <base> element ................................. 250
    hook_onload replaced by addLoadEvent() ......................... 251
    hook_search_item replaced by hook_search_page .................. 251
    Extreme long comment on http://drupal.org/node/42388 ....... 251
Developing for Drupal

The Drupal engine is open source. It is possible for each and every user to become a contributor. The fact remains that most Drupal users, even those skilled in programming arts, have never contributed to Drupal even though most of us had days where we thought to ourselves: "I wish Drupal could do this or that ...". Through this section, we hope to make Drupal more accessible to them.

The guide pages found here are collaborative, but not linked to particular Drupal versions. Because of this, documentation can become out of date. To combat this, we are moving most developer documentation into the [Doxygen documentation](#) that is versioned by CVS and generated from the source code. Look there for up-to-date and version-specific information.

- [CVS log messages](#)
- [Browse CVS repository](#)
Contributing to Drupal

Drupal is a collaborative, community-driven project. This means that the software and its supporting features (documentation, the drupal.org website) are collaboratively produced by users and developers all over the world.

There are several ways to contribute to Drupal:

- Improve or enhance the software
- Provide support and documentation for other users (e.g., by posting additions or updates to the Drupal Handbook or answering requests on user forums or issues).
- Provide financial support to Drupal development.

This section focuses on the first of these three.

Types of Contributions

There are two basic types of contributions you can make to Drupal’s code base: (a) "contributed" modules or themes and (b) contributions to the drupal "core".

- "Contributions" are the community-produced modules and themes available on the Drupal site. To make a contribution, you need to apply for contributor privileges, produce your contribution, and then notify the contributions manager to request a review of your work before posting. As long as contributions meet some minimal criteria - they do what they claim to and have some demonstrable benefit without unduly replicating already-available functionality - they are approved.

If you have major enhancements you wish to contribute, doing so via a contributed module is in many ways the easiest way to begin. Contributed code has a relatively low set of requirements to meet.

- In contrast, changes to the Drupal core are made through a thorough consultative process to ensure the overall integrity of the software.

Changes to the Drupal core are generally of three types:

- Bug fixes. These changes respond to identified problems in the existing code.
- New features. These changes are enhancements on what is already available.
- Code maintenance. These changes are to improve the quality of the code or bring it up to date with changes elsewhere in Drupal. This can include bringing code in line with coding standards, improving efficiency (e.g., eliminating unneeded database queries), introducing or improving in-line comments, and doing upgrades for compliance with a new release version.
While you can create your own issues, you can also begin by simply taking on existing tasks on the [task list](#) See the page [Tips for contributing to the core](#) for advice on how to get started as a core contributor.

**Task list**

The [Drupal bug database](#) contains many issues classified as "bite-sized" tasks -- tasks that are well-defined and self-contained, and thus suitable for a volunteer looking to get involved with the project. You don’t need broad or detailed knowledge of Drupal’s design to take on one of these, just a pretty good idea of how things generally work, and familiarity with the coding guidelines. Each task is something a volunteer could pick off in a spare evening or two.

If you start one of these, please notify the other developers by mailing drupal-devel@drupal.org (of course, you should be subscribed to that list). If you have questions as you go, ask the dev list or update the task (updates are sent to the list automatically). Send the patch to the list when ready.

**Bug reports**

If you found a bug, send us the bug report and we will fix it provided you include enough diagnostic information for us to go on. Your bug reports play an essential role in making Drupal reliable.

Bug reports can be posted in connection with any project hosted on drupal.org. You can submit a new bug via the submit issue form. Provide a sensible title for the bug, and choose the project you think you have found the bug in. After previewing the submission, you will need to choose a related component and you will be able to provide more details about the bug, including the description of the problem itself. Please include any error messages you received and a detailed description of what you were doing at the time.

Note that you don’t have to be logged in nor a member of drupal.org to submit bugs.

The first thing we will do when you report a bug is tell you to upgrade to the newest version of Drupal, and then see if the problem reproduces. So you'll probably save us both time if you upgrade and test with the latest version before sending in a bug report.

Always include PHP, database and webserver version information.

**How to report bugs effectively**

**Summary**

- The first aim of a bug report is to let the programmer see the failure with their own eyes. If you can’t be with them to make it fail in front of them, give them detailed instructions so that they can make it fail for themselves.
- In case the first aim doesn’t succeed, and the programmer can’t see it failing themselves, the
second aim of a bug report is to describe what went wrong. Describe everything in detail. State what you saw, and also state what you expected to see. Write down the error messages, especially if they have numbers in.

- When your computer does something unexpected, freeze. Do nothing until you’re calm, and don’t do anything that you think might be dangerous.
- By all means try to diagnose the fault yourself if you think you can, but if you do, you should still report the symptoms as well.
- Be ready to provide extra information if the programmer needs it. If they didn’t need it, they wouldn’t be asking for it. They aren’t being deliberately awkward. Have version numbers at your fingertips, because they will probably be needed.
- Write clearly. Say what you mean, and make sure it can’t be misinterpreted.
- Above all, be precise. Programmers like precision.

Introduction

Anybody who has written software for public use will probably have received at least one bad bug report. Reports that say nothing ("It doesn’t work!"); reports that make no sense; reports that don’t give enough information; reports that give wrong information. Reports of problems that turn out to be user error; reports of problems that turn out to be the fault of somebody else’s program; reports of problems that turn out to be network failures.

There’s a reason why technical support is seen as a horrible job to be in, and that reason is bad bug reports. However, not all bug reports are unpleasant: I maintain free software, when I’m not earning my living, and sometimes I receive wonderfully clear, helpful, informative bug reports.

In this essay I’ll try to state clearly what makes a good bug report. Ideally I would like everybody in the world to read this essay before reporting any bugs to anybody. Certainly I would like everybody who reports bugs to me to have read it.

In a nutshell, the aim of a bug report is to enable the programmer to see the program failing in front of them. You can either show them in person, or give them careful and detailed instructions on how to make it fail. If they can make it fail, they will try to gather extra information until they know the cause. If they can’t make it fail, they will have to ask you to gather that information for them.

In bug reports, try to make very clear what are actual facts ("I was at the computer and this happened") and what are speculations ("I think the problem might be this"). Leave out speculations if you want to, but don’t leave out facts.

When you report a bug, you are doing so because you want the bug fixed. There is no point in swearing at the programmer or being deliberately unhelpful: it may be their fault and your problem, and you might be right to be angry with them, but the bug will get fixed faster if you help them by supplying all the information they need. Remember also that if the program is free, then the author is providing it out of kindness, so if too many people are rude to them then they may stop feeling kind.
"It doesn’t work."
Give the programmer some credit for basic intelligence: if the program really didn’t work at all, they would probably have noticed. Since they haven’t noticed, it must be working for them. Therefore, either you are doing something differently from them, or your environment is different from theirs. They need information; providing this information is the purpose of a bug report. More information is almost always better than less.

Many programs, particularly free ones, publish their list of known bugs. If you can find a list of known bugs, it’s worth reading it to see if the bug you’ve just found is already known or not. If it’s already known, it probably isn’t worth reporting again, but if you think you have more information than the report in the bug list, you might want to contact the programmer anyway. They might be able to fix the bug more easily if you can give them information they didn’t already have.

This essay is full of guidelines. None of them is an absolute rule. Particular programmers have particular ways they like bugs to be reported. If the program comes with its own set of bug-reporting guidelines, read them. If the guidelines that come with the program contradict the guidelines in this essay, follow the ones that come with the program!

If you are not reporting a bug but just asking for help using the program, you should state where you have already looked for the answer to your question. ("I looked in chapter 4 and section 5.2 but couldn’t find anything that told me if this is possible.") This will let the programmer know where people will expect to find the answer, so they can make the documentation easier to use.

"Show me"
One of the very best ways you can report a bug is by showing it to the programmer. Stand them in front of your computer, fire up their software, and demonstrate the thing that goes wrong. Let them watch you start the machine, watch you run the software, watch how you interact with the software, and watch what the software does in response to your inputs.

They know that software like the back of their hand. They know which parts they trust, and they know which parts are likely to have faults. They know intuitively what to watch for. By the time the software does something obviously wrong, they may well have already noticed something subtly wrong earlier which might give them a clue. They can observe everything the computer does during the test run, and they can pick out the important bits for themselves.

This may not be enough. They may decide they need more information, and ask you to show them the same thing again. They may ask you to talk them through the procedure, so that they can reproduce the bug for themselves as many times as they want. They might try varying the procedure a few times, to see whether the problem occurs in only one case or in a family of related cases. If you’re unlucky, they may need to sit down for a couple of hours with a set of development tools and really start investigating. But the most important thing is to have the programmer looking at the computer when it goes wrong. Once they can see the problem happening, they can usually take it from there and start trying to fix it.
"Show me how to show myself"
This is the era of the Internet. This is the era of worldwide communication. This is the era in which I can send my software to somebody in Russia at the touch of a button, and he can send me comments about it just as easily. But if he has a problem with my program, he can’t have me standing in front of it while it fails. "Show me" is good when you can, but often you can’t.

If you have to report a bug to a programmer who can’t be present in person, the aim of the exercise is to enable them to reproduce the problem. You want the programmer to run their own copy of the program, do the same things to it, and make it fail in the same way. When they can see the problem happening in front of their eyes, then they can deal with it.

So tell them exactly what you did. If it’s a graphical program, tell them which buttons you pressed and what order you pressed them in. If it’s a program you run by typing a command, show them precisely what command you typed. Wherever possible, you should provide a verbatim transcript of the session, showing what commands you typed and what the computer output in response.

Give the programmer all the input you can think of. If the program reads from a file, you will probably need to send a copy of the file. If the program talks to another computer over a network, you probably can’t send a copy of that computer, but you can at least say what kind of computer it is, and (if you can) what software is running on it.

"Works for me, so what goes wrong?"
If you give the programmer a long list of inputs and actions, and they fire up their own copy of the program and nothing goes wrong, then you haven’t given them enough information. Possibly the fault doesn’t show up on every computer; your system and theirs may differ in some way. Possibly you have misunderstood what the program is supposed to do, and you are both looking at exactly the same display but you think it’s wrong and they know it’s right.

So also describe what happened. Tell them exactly what you saw. Tell them why you think what you saw is wrong; better still, tell them exactly what you expected to see. If you say "and then it went wrong", you have left out some very important information.

If you saw error messages then tell the programmer, carefully and precisely, what they were. They are important! At this stage, the programmer is not trying to fix the problem: they’re just trying to find it. They need to know what has gone wrong, and those error messages are the computer’s best effort to tell you that. Write the errors down if you have no other easy way to remember them, but it’s not worth reporting that the program generated an error unless you can also report what the error message was.

In particular, if the error message has numbers in it, do let the programmer have those numbers. Just because you can’t see any meaning in them doesn’t mean there isn’t any. Numbers contain all kinds of information that can be read by programmers, and they are likely to contain vital clues. Numbers in error messages are there because the computer is too confused to report the error in words, but is doing the best it can to get the important information to you somehow.
At this stage, the programmer is effectively doing detective work. They don’t know what’s happened, and they can’t get close enough to watch it happening for themselves, so they are searching for clues that might give it away. Error messages, incomprehensible strings of numbers, and even unexplained delays are all just as important as fingerprints at the scene of a crime. Keep them!

If you are using Unix, the program may have produced a core dump. Core dumps are a particularly good source of clues, so don’t throw them away. On the other hand, most programmers don’t like to receive huge core files by e-mail without warning, so ask before mailing one to anybody. Also, be aware that the core file contains a record of the complete state of the program: any “secrets” involved (maybe the program was handling a personal message, or dealing with confidential data) may be contained in the core file.

"So then, I tried..."
There are a lot of things you might do when an error or bug comes up. Many of them make the problem worse. A friend of mine at school deleted all her Word documents by mistake, and before calling in any expert help, she tried reinstalling Word, and then she tried running Defrag. Neither of these helped recover her files, and between them they scrambled her disk to the extent that no Undelete program in the world would have been able to recover anything. If she’d only left it alone, she might have had a chance.

Users like this are like a mongoose backed into a corner: with its back to the wall and seeing certain death staring it in the face, it attacks frantically, because doing something has to be better than doing nothing. This is not well adapted to the type of problems computers produce.

Instead of being a mongoose, be an antelope. When an antelope is confronted with something unexpected or frightening, it freezes. It stays absolutely still and tries not to attract any attention, while it stops and thinks and works out the best thing to do. (If antelopes had a technical support line, it would be telephoning it at this point.) Then, once it has decided what the safest thing to do is, it does it.

When something goes wrong, immediately stop doing anything. Don’t touch any buttons at all. Look at the screen and notice everything out of the ordinary, and remember it or write it down. Then perhaps start cautiously pressing "OK" or "Cancel", whichever seems safest. Try to develop a reflex reaction - if a computer does anything unexpected, freeze.

If you manage to get out of the problem, whether by closing down the affected program or by rebooting the computer, a good thing to do is to try to make it happen again. Programmers like problems that they can reproduce more than once. Happy programmers fix bugs faster and more efficiently.

"I think the tachyon modulation must be wrongly polarised."
It isn’t only non-programmers who produce bad bug reports. Some of the worst bug reports I’ve ever seen come from programmers, and even from good programmers.

I worked with another programmer once, who kept finding bugs in his own code and trying to fix them. Every so often he’d hit a bug he couldn’t solve, and he’d call me over to help. "What’s gone wrong?" I’d ask. He would reply by telling me his current opinion of what needed to be
This worked fine when his current opinion was right. It meant he’d already done half the work and we were able to finish the job together. It was efficient and useful.

But quite often he was wrong. We would work for some time trying to figure out why some particular part of the program was producing incorrect data, and eventually we would discover that it wasn’t, that we’d been investigating a perfectly good piece of code for half an hour, and that the actual problem was somewhere else.

I’m sure he wouldn’t do that to a doctor. "Doctor, I need a prescription for Hydroyyodyne." People know not to say that to a doctor: you describe the symptoms, the actual discomforts and aches and pains and rashes and fevers, and you let the doctor do the diagnosis of what the problem is and what to do about it. Otherwise the doctor dismisses you as a hypochondriac or crackpot, and quite rightly so.

It’s the same with programmers. Providing your own diagnosis might be helpful sometimes, but always state the symptoms. The diagnosis is an optional extra, and not an alternative to giving the symptoms. Equally, sending a modification to the code to fix the problem is a useful addition to a bug report but not an adequate substitute for one.

If a programmer asks you for extra information, don’t make it up! Somebody reported a bug to me once, and I asked him to try a command that I knew wouldn’t work. The reason I asked him to try it was that I wanted to know which of two different error messages it would give. Knowing which error message came back would give a vital clue. But he didn’t actually try it - he just mailed me back and said "No, that won’t work". It took me some time to persuade him to try it for real.

Using your intelligence to help the programmer is fine. Even if your deductions are wrong, the programmer should be grateful that you at least tried to make their life easier. But report the symptoms as well, or you may well make their life much more difficult instead.

"That’s funny, it did it a moment ago."
Say "intermittent fault" to any programmer and watch their face fall. The easy problems are the ones where performing a simple sequence of actions will cause the failure to occur. The programmer can then repeat those actions under closely observed test conditions and watch what happens in great detail. Too many problems simply don’t work that way: there will be programs that fail once a week, or fail once in a blue moon, or never fail when you try them in front of the programmer but always fail when you have a deadline coming up.

Most intermittent faults are not truly intermittent. Most of them have some logic somewhere. Some might occur when the machine is running out of memory, some might occur when another program tries to modify a critical file at the wrong moment, and some might occur only in the first half of every hour! (I’ve actually seen one of these.)

Also, if you can reproduce the bug but the programmer can’t, it could very well be that their computer and your computer are different in some way and this difference is causing the problem. I had a program once whose window curled up into a little ball in the top left corner of
the screen, and sat there and sulked. But it only did it on 800x600 screens; it was fine on my 1024x768 monitor.

The programmer will want to know anything you can find out about the problem. Try it on another machine, perhaps. Try it twice or three times and see how often it fails. If it goes wrong when you’re doing serious work but not when you’re trying to demonstrate it, it might be long running times or large files that make it fall over. Try to remember as much detail as you can about what you were doing to it when it did fall over, and if you see any patterns, mention them. Anything you can provide has to be some help. Even if it’s only probabilistic (such as “it tends to crash more often when Emacs is running”), it might not provide direct clues to the cause of the problem, but it might help the programmer reproduce it.

Most importantly, the programmer will want to be sure of whether they’re dealing with a true intermittent fault or a machine-specific fault. They will want to know lots of details about your computer, so they can work out how it differs from theirs. A lot of these details will depend on the particular program, but one thing you should definitely be ready to provide is version numbers. The version number of the program itself, and the version number of the operating system, and probably the version numbers of any other programs that are involved in the problem.

"So I loaded the disk on to my Windows . . ."

Writing clearly is essential in a bug report. If the programmer can’t tell what you meant, you might as well not have said anything.

I get bug reports from all around the world. Many of them are from non-native English speakers, and a lot of those apologise for their poor English. In general, the bug reports with apologies for their poor English are actually very clear and useful. All the most unclear reports come from native English speakers who assume that I will understand them even if they don’t make any effort to be clear or precise.

- Be specific. If you can do the same thing two different ways, state which one you used. "I selected Load" might mean "I clicked on Load" or "I pressed Alt-L". Say which you did. Sometimes it matters.
- Be verbose. Give more information rather than less. If you say too much, the programmer can ignore some of it. If you say too little, they have to come back and ask more questions. One bug report I received was a single sentence; every time I asked for more information, the reporter would reply with another single sentence. It took me several weeks to get a useful amount of information, because it turned up one short sentence at a time.
- Be careful of pronouns. Don’t use words like "it", or references like "the window", when it’s unclear what they mean. Consider this: "I started FooApp. It put up a warning window. I tried to close it and it crashed." It isn’t clear what the user tried to close. Did they try to close the warning window, or the whole of FooApp? It makes a difference. Instead, you could say "I started FooApp, which put up a warning window. I tried to close the warning window, and FooApp crashed." This is longer and more repetitive, but also clearer and less easy to misunderstand.
- Read what you wrote. Read the report back to yourself, and see if you think it’s clear. If you have listed a sequence of actions which should produce the failure, try following them
yourself, to see if you missed a step.

**Feature suggestions**

How many times you have dreamed "Gee...I wish Drupal could do that" or "I like the xxx feature, but it should work better". If you want to improve Drupal, send us you wishes as a feature suggestions. Your suggestions play an essential role in making Drupal more usable and feature-rich.

The core features provided by Drupal are listed on the [features page](#). You can submit a feature request by creating a new issue connected to the component the feature is related to. Please note that there is a Drupal contributed module named 'Features' which is used on the feature page mentioned above. Every module has a feature request subcategory, and thus the 'Feature' module is not the appropriate place to submit feature requests. To properly file a feature request, first choose the project it is related to and then after hitting preview set the other related options. You will be able to categorize the issue as a feature request with the Issue Information / Category dropdown.

Note that you don’t have to be logged in nor to be a member of drupal.org to suggest features.

**Patches**

Patches are a way to distribute relatively small changes to code. They are the preferred way to contribute bug fixes and other proposed changes to Drupal’s codebase.

A [CVS account](#) is not required to supply patches -- anyone with a Drupal account can go to the [issues queue](#) to upload a patch. The [tips for contributing](#) apply to core as well as contributed modules.

See the sections below for further details on how to actually create and submit patches.

**NOTE:** Patching is something that should never be done on your production site unless you are able to understand the implications of doing so, have sufficient backup and testing performed. **Patching your system without understanding what you have done can lead to loss of data and/or site instabilities**

**Diff and patch**

Diff and patch are two complementary tools for recording and applying changes between two sets of files.

We use them for content control even though we distribute our code via CVS. Why? Because diff and patch provide an immense amount of control. Patches can be submitted via e-mail and in plain text; maintainers can read and judge the patch before it ever gets near a tree. It allows maintainers to look at changes easily without blindly integrating them.
Diff is the first command in the set. It has the simple purpose to create a file called a *patch* or a *diff* which contains the differences between two text files or two groups of text files. Diff can write into different formats, although the unified difference format is preferred. The patches this command generates are much easier to distribute and allow maintainers to see quickly and easily what changed and to make a judgement.

Patch is diff’s complement and takes a patch file generated by diff and applies it against a file or a group of files.

The actual usage of diff and patch is not complicated.

At its simplest, a diff command for comparing two files would be:

```bash
diff old.txt new.txt > oldnew.patch
```

For drupal, we prefer patches in unified format, so we add `-u` to the command line:

```bash
diff -u old.txt new.txt > oldnew.patch
```

It is helpful to keep a reference in the patch file to which function was patched, so the following form of the command is often used. For example, if you have made a change in foo.module, to create a patch against the CVS tree:

```bash
cvs diff -u -F ^function foo.module > foo.patch
```

Or if you had downloaded Drupal instead of checking it out from CVS and were creating a patch against a local copy of foo.module:

```bash
diff -u -F ^function foo.module newfoo.module > foo.patch
```

Note that diff `-u -F^f` is a valid shortcut for the parameters above.

Generally, however, a comparison of two source trees is often desired. A possible command to do so is:

```bash
diff -ruN old new > tree.diff
```

Once a patch is generated, the process of patching the file is even simpler. Based on our examples above, we could do:

```bash
patch < oldnew.patch
```

Or if you want to patch an entire directory, you should use:

```bash
patch -p0 -u < tree.diff
```

To unapply the patch, use:

```bash
patch -p0 -R < tree.diff
```
Diff on Windows

(against a cvs source with the cvs.exe built-in diff. do diff local files, you need a windows diff program, command line or [visual]

Generic:

- find the cvs.exe of your cvs package [WinCVS TortoiseCVS cygwin ...] and make sure it is in your PATH
- cd to your drupal root dir
- cvs diff -u [[-r rev1|-D date1] [-r rev2|-D date2]] [file_to_diff] [&gt; file_to_diff.patch]
  - -u: unified format
  - -r: revision(s) to diff
    - no -r: compare the working file with the revision it was based on
    - one -r: compare that revision with your current working file
    - two -r: compare those two revisions
  - -D: use a [date_spec] to specify revisions. examples: "1972-09-24 20:05", "24 Sep 1972 20:05".
  - file_to_diff: path to the file or directory you want to diff. if you specify a directory, the output will include the diff of all differing files in this directory and all subdirectories.
  - &gt; file_to_diff.patch: creates a patch - saves the diff in file_to_diff.patch instead of outputting it on stdout. if you send a patch, make sure it has the proper line endings
  - see the [CVS manual] for a complete list of and additional options

via WinCVS GUI

Just select the file you edited and right-mouse-click &gt; "diff selection" (or press the "diff selection"-icon on the toolbar, or do Menubar &gt; "Query" &gt; "diff selection"). This brings up a "Diff settings" dialog box that offers some limited options as "revisions to diff" and "ignore whitespace/case" [update 2003-Feb-07: starting with [WinCvs 1.3b1] "Full diff options are available from the diff dialog")). The resulting diff is output to the WinCVS-Console and can be copied and pasted.

via WinCVS/TortoiseCVS external diff

- WinCVS: Menubar &gt; "Admin" &gt; "Preferences" &gt; "WinCVS" &gt; "External diff program ". This program will be invoked by the "Diff selection" when "Use the external diff" is checked.
- TortoiseCVS: CVS &gt; "Preferences" &gt; "External diff application". This program will be invoked by "CVS Diff ...

Some external visual diff programs for Windows:

- [Araxis Merge](commercial)
- ExamDiff
- CSDiff
- for those who can live w/ java: [Guiffy](commercial)
- [WinMerge]
- you may find more [here](#)
Notes:

- While these programs do a nice job in showing file differences visually, side by side, *non of them* (as i can tell) allows to actually save the difference in unified format (most allow to save a *standard* diff, though) - **update**: TortoiseCVS lets you save patches. It does unified format by default. See its Make Patch option. Note that this 'Make Patch' option can make recursive patches when applied to directories.
- You *cannot* specify the "-u" in the External diff preferences (eg "diff -u") as this will result in 'Unable to open 'diff -u' (The system cannot find the file specified.)". A workaround for this is to, in the preferences, specify a batch-file that calls the external diff with the -u option. Another workaround is meta-diff which allows for launching of special diff programs for certain file types.)

**line endings**: an issue with using diff on windows is that generated patches have windows line endings, which makes them impossible to apply on unix boxes. unfortunately, there seems to be no way to convince "cvs diff" to output unix line endings so the only way for making a proper patch on windows that i see is to convert / filter the output from "cvs diff" to unix line endings:

* filter: pipe "cvs diff"s output through some dos2unix tool (like the one from Robert B. Clark or like cygwins’s dos2unix / d2u):

```
cvs diff [options] file_to_diff  | unix2dos -u > file_to_diff.patch
```

* convert: save "cvs diff"s output to a file:

```
cvs diff [options] file_to_diff > file_to_diff.patch
```

and manually convert file_to_diff.patch to unix line endings. every developers editor should be capable of this; besides, there are many dos2unix versions that operate on files.

### Patching with Cygwin

1. Get the latest version of Cygwin at [Cygwin.com](http://www.cygwin.com) What you download is actually a small setup file that will reconnect to the web for more data when executed.
2. Run setup.exe. Chose "install from the internet", an appropriate directory, connection type etc. matching your system. In the next window, chose a server near to you (btw: no problem to cancel & restart with a different server, if the first choice is slow).
3. In the next window you get a list of available packages - now it’s important to add "patch" to the list, since it is not included by default. You have to find it by pressing the "view"-Button once to see all of the packages (view cycles between "category", "all" and "updated"). Scroll down until you see "patch" in the column "Packages". Klick the small Icon with the text "skip" way to the left in the line - “skip” will change to a version number. This means the package will be installed. (at this point you could also deactivate a whole bunch of the other packages, if you don’t need them. If you don’t, the setup-process takes a while.)
4. Run cygwin. In the explorer, you will see that a unix-like directory-structure has been created inside the installation directory. when you run the program for the first time, a user-directory is created below the "home"-directory. This will be your default directory.
when working with cygwin - the easiest way is to put the original file & the patch there (so you don’t have to enter a lengthy path to your commands).

5. Patch! :) In the open shell (command window) just enter the command:

   patch -p0 original.filename nameofyour.patch

6. If you get errors while patching, try to use the original file fresh from the drupal server. I used a module from my working server, and the patch command threw all kinds of errors. Seems as if the line-numbers have been messed up somehow by copying & viewing. Although I do not understand how this could happen.

7. Before you copy the patched file to your server, rename the old one... you never know if it really works in your environment.

Gnuwin32 diff and patch

An alternative for Windows only users is GnuWin32.

Got to the download link and get the patch and diffutil packages. They work well with standard *nix system examples found here.

Note: If you add the bin directory c:\Program Files\GnuWin32\bin to your Windows system path (go look up how to do this on a Windows how to site) then you will be able to run them at a CMD prompt anywhere on your system.

Patch on Windows

I haven’t found any Windows-GUI for patch, so the only choice is a Windows port of the Unix command-line tool. If you know of a Windows GUI for patch, please let me know

- Cygwin - a UNIX environment for Windows, including many standard UNIX-tools (including diff and patch)
- pre-compiled binaries, available from various places. Some I’ve found:
  - [http://www.squirrel.nl/people/jvromans/tpj0403-0016b.html](http://www.squirrel.nl/people/jvromans/tpj0403-0016b.html)

Note:

I found many of the precompiled binaries have problems with pathnames etc. and do not work properly. So I would recommend installing cygwin - it takes a while, but after that, you have a nice Unix environment that works.

* and i tried a lot: checking out all files with unix line endings, various -kb options, external diffs, patched cvs versions ... nothing. for a discussion of this, check [CVS and binary files](http://www.squirrel.nl/people/jvromans/tpj0403-0016b.html)

You can try also integrated diff/patch packages like [GNU diffutils for Windows](http://www.gnu.org/software/emacs/windows/faq11.html#patch) or get the [GNU utilities for Win 32](http://www.gnu.org/software/emacs/windows/faq11.html#patch)
Using Cygwin to patch in Windows

To use Cygwin to patch in Windows:

1. In your c: drive, create two folders; one named C:\cygwinstuff and another named C:\patchfiles

2. Go to [www.cygwin.com](http://www.cygwin.com) and download the setup file.

2. Click on the file downloaded in step 2 to start the install

The next few steps will walk you through the install process for Cygwin.

4. Choose **Install from Internet**

5. For Root Install Directory: Choose C:\cygwin

6. For Local package directory, choose c:\cygwinstuff -- the folder you created in step 1.

7. Choose the appropriate internet connection -- for most of us, that’s the **Direct Connection**

8. Choose a download site -- I generally pick one at random, but for those of you who are more precise, you can use the domain names to try and choose a site on the same continent.

9. Select packages: This is where things can get tricky. Just make sure to select the **patchutils** under the **Devel** section.

10. Begin the install.

11. Once Cygwin has installed, move the file(s) you want to patch, and the patch file, into the C:\patchfiles folder you created in step one.

12. Open Cygwin. Cygwin gives you what you have been looking for: the command line. You will need the following commands:

   - `ls -al` -- lists the contents of your current directory
   - `cd ..` -- moves you up to the parent directory (note the space between cd and .. if you come from a DOS background!)
   - `cd foldername` -- moves you to a specific folder visible from your current directory

13. When you open Cygwin and type `ls -al`, you will see the contents of your current directory. Different versions of Cygwin start you in different directories. Regardless of where you start, you can use `cd c:` to navigate to your C: drive (under Cygwin, this will be called /cygwin/c).

14. When you are at C:, you will be able to see the patchfiles folder you created in step one. Use `cd patchfiles` to navigate into this folder.
15. Type `ls -al`. You should see the files you moved into this folder in step 11.

16. Type `patch filetobepatched < patchfile` or `patch -p0 < patchfile` if there are many files to be patched.

These instructions ought to get you started.

For a more comprehensive overview of the options available while patching, see `patch` manual (man) page.

**HOWTO: Set up a test environment to help review patches**

This guide is aimed at a developer or end user who would like to participate in the Drupal core patch review process, but is unsure of where to start. A similar process will work for contributed modules, but this page’s focus is on core. There are five main sections to this guide:

1. Retrieving a copy of Drupal HEAD from CVS
2. Configuring Drupal HEAD
3. Finding and applying patches
4. Testing patches for versions other than Drupal HEAD
5. Creating a test environment from an existing installation of Drupal

This document assumes the reader has shell access to some sort of Linux/Unix/BSD/Cygwin platform.

**Retrieving a copy of Drupal HEAD from CVS**

Drupal HEAD where work on the next version of Drupal always occurs, so it is against this version of Drupal that any new features will be developed. Testing these features requires a functional copy of the development version of Drupal, which is available from Drupal’s CVS repository. For more information on CVS, please refer to the CVS section of the handbook.

1. Begin by creating a directory to contain the new version of Drupal. For example,
   ```
   cd /path/to/web/root/
   ```

2. The next step is to retrieve a copy of Drupal HEAD from the server. Execute the command:
   ```
   cvs -z9 -d:pserver:anonymous:anonymous@cvs.drupal.org:/cvs/drupal checkout -d drupal-cvs drupal
   ```

   This will download the Drupal HEAD files to a folder called `drupal-cvs`.

**Configuring Drupal HEAD**

Drupal HEAD’s installation and configuration is generally done exactly the same as a standard version of Drupal (see the Installing Drupal section of the handbook for more information). However, there are a few additional changes which are recommended:
1. Create a folder in the `sites/` directory with the same name as the domain name of the site, and copy the `settings.php` file there. This prevents customized settings from being overwritten when Drupal HEAD is updated. For example,

```bash
mkdir sites/example.com
cp sites/default/settings.php sites/example.com/settings.php
```

2. After completing the steps in `INSTALL.txt`, test the installation to ensure it is working properly.

**Finding and applying patches**

Patches can be found in the [Drupal Patch Queue](http://drupal.org/). Choose from patches which match the Drupal installation (in this case, `cvs`; see below for instructions on testing patches on other versions of Drupal). An excellent guide to reviewing patches can be found at [Tips for reviewing patches](http://drupal.org/.contributors-guide/patches) in the Contributor’s Handbook.

Once an interesting patch has been found, the process to apply the patch in order to test it is as follows:

1. Ensure Drupal HEAD is updated to the most recent version. This can be done at any time by executing the following command inside the Drupal HEAD root directory:

   ```bash
cvs update -dP
   ```

2. Download a copy of the most recent version of the patch (often patches are revised further down the issue page) by issuing the command:

   ```bash
   wget http://drupal.org/files/issues/patch-name.patch
   ```

3. Apply the patch by issuing the following:

   ```bash
   patch -p0 -u < patch-name.patch
   ```

Finally, test the patch out rigorously and submit feedback to the issue tracker, in order to help identify problems and improve the functionality of Drupal.

**Testing patches for versions other than Drupal HEAD**

Not all patches in the patch queue are for Drupal HEAD; bug fixes and security updates to release versions of Drupal will also appear here. To setup a test environment for patches other than those intended for Drupal HEAD (for example, Drupal 4.6.1), generally the same steps as above are followed with the following exception:

When retrieving a copy of Drupal from CVS, the same checkout command is used, however a branch must be specified in order to retrieve a version of Drupal other than HEAD. A list of available branches is available from the [Using CVS with branches and tags](http://drupal.org/using-cvs) page of the Contributor’s Handbook.

For example, to checkout a test version of Drupal 4.6.1, use the command:

```
cvs -z9 -d:pserver:anonymous:anonymous@cvs.drupal.org:/cvs/drupal
checkout -r DRUPAL-4-6-1 drupal
```
Remember that the version assigned to a patch and the version of Drupal to which it is applied must match.

**Creating a test environment from an existing installation of Drupal**

The best way to see how a patch will affect an already-live installation of Drupal is to apply it directly. However, since patches can sometimes yield unexpected results, the best course of action is always to apply them to a copy of the live installation rather than the installation itself.

Please see [Copying Your Live Site to a Test Site (Command-Line version)] from the CivicSpace upgrade information for more information on how to do this.

Some additional tips

- test with a meaningful database. One that contains different roles, blocks, nodes and comments. Or one that has no comments, etc. testing with a clean installation makes very little sense for a lot of tests. The `devel module` contains scripts to generate content, taxonom and users
- a good way to maintain patches and to test only certain maintained patches is to do so on a dedicate installation. That way you can be sure that patches do not conflict, or that a previous test breaks the current one.

Here is what I do:

1. `mkdir 28245` (the number is the issue/bug/node id)
2. `mysqlhotcopy` the "clean" database in there with dbname 28245
3. edit settings.php to use `myhost/28245` as url and 28245 as database.

Might seem a lot of work, but once I got used to it it works very fast.

**Creating and submitting patches**

The process of submitting a patch can seem daunting at first. This text is a collection of suggestions which can greatly increase the chances of your change being accepted.

The easiest way to get set up for making and sending patches is to get CVS working. Then you can just type: `cvs diff -u -F^f [file to patch]` to generate a patch. To output it to a file, go: `cvs diff -u -F^f [file to patch] > [outfile]`

**Coding style:**

- If your code deviates too much from the [Code Conventions](#) it is more likely to be rejected without further review and without comment.

  `diff -u`:

  Use `diff -u` or `diff -urN` to create patches: when creating your patch, make sure to create it in "unified diff" format, as supplied by the `-u` argument to `diff`. Patches should be based in the root source directory, not in any lower subdirectory. Make sure to create patches against a "vanilla", or unmodified source tree.
diff -F^f:

Use the additional \(-F^f\) argument to \texttt{diff} to create patches that are easier to read. \(-F^f\) tells \texttt{diff} to include the last matching line in the header of the created patch. This will be the last function definition if the files adhere to the Drupal Code Conventions.

Describe your changes:

Describe the technical detail of the change(s) your patch includes and try to be as specific as possible. Note that we prefer technical reasoning above marketing: give us clear reasons why "this way" is good. Justify your changes and try to carry enough weight. It is important to note the version to which this patch applies.

Separate your changes:

Separate each logical change into its own patch. For example, if your changes include both bug fixes and performance enhancements, separate those changes into two or more patches. If your changes include an API update, and a new module which uses that new API, separate those into two patches.

Verifying your patch

The CVS review team is overloaded reviewing patch submissions. Please make their lives easier by assuring the following:

- Test your code!
- Make sure your code is clean and secure. If your patch is just a quick hack, then don’t set your issue to \texttt{Patch} status.
- Patch against HEAD. If you only have a patch against a prior revision, then don’t assign your issue to \texttt{Patch} status

Submitting your patch:

Patches should be submitted \([\text{via the issue tracker}]\). Create a bug report or feature request, attach your patch using the file upload form and set the issue’s status to \texttt{patch}. Setting the status to patch is important as it adds the patch to the \texttt{patch queue}.

Tips for reviewing patches

General guidelines

- Write down your thoughts as you are reviewing, not afterwards.
  Keep a text editor open for typing in and write down your thoughts immediately. When you are done, clean it up and structure it.
- Take a look at both the big picture and the details.
  Simply saying "I [don’t] like this feature" or "-1" is of no use and is strongly discouraged. Similarly, diving into a patch and saying nothing but "there is a typo in function so and so" could be a waste of time as you could be continuing a patch that has no hope of being committed.
- No patch can save the world, not even a Drupal core patch. If it works and does something useful on its own, then it is good to go. One of the worst things you can do is elaborate on other, equivalent approaches and suggest complex extensions to the patch. Additional features can be added later. A scalpel is often better than the Swiss Knife.
- Don’t stop reviewing at the first sign of trouble.
  If there are bugs, imagine how things should’ve worked. If there are usability problems, try to think of a better interface.
Pay attention to what the submitter has said about the patch.
If some things are not clear, write down your questions. You should have a good idea of what the patch does and who it is for before you start.

Apply a patch to a CVS tree if you can.
This allows you to immediately create an updated patch. Often it takes more time to describe small changes (like typos and code style) than it is to make those changes yourself.

Reviewing process

- Challenge the proposed changes.
  What are the benefits? Are there disadvantages? Does it fit in the general ideas of Drupal? Does it make good use of our systems or does it re-invent the wheel? Does it belong in core? Can you see the changes being used elsewhere too?

- Test the functionality.
  Does it do what it should? Does it work under different settings (e.g. clean URLs) and with different modules?

- Examine the usability.
  If new pages are added, are they are in a logical place? If controls are added, are they intuitive and logical to use? Does it come with appropriate contextual help? Is the interface consistent?

- Review the code.
  Is the code understandable and documented? Does it make appropriate use of APIs or does it re-invent the wheel? Are there possible security issues? Does it conform to our coding conventions? Is the functionality separated into logical pieces, or just mashed together? Are appropriate sections made themable?

Authoring a patch review

When posting your issue follow-up, it is important to think critically and speak positively

- Your goal is to propose an even better solution than the patch currently envisions. Do not post negative feedback without a corresponding solution. A good review helps a patch proceed, instead of obstructing it. It is a rare case that a submitted patch has absolutely no merit and warrants a purely negative review.

- Post your findings in a clear and structured manner to the original issue. If you can, include an improved patch.

Severity levels of bugs

These are bugs that hold back a release

Extreme critical
When a bug breaks all core. Possible examples: sessions, bootstrap one of the form elements or something like that. These are to be fixed immediately because Drupal HEAD is not usable at all.
critical bugs
  render a whole (or most of a) module unusable. Example: you can’t post a forum topic.
major bugs
  a smaller part of a module. Example: You can’t add a menu with menu module.

These are bugs that do not hold a release:

Normal bugs
  just bugs, we could even release with these. Just one functionality. Example: the category filter not working on node admin screen.
minor bugs
  cosmetical stuff, notices etc. Often also referred to as ’tasks’ or features.

The revision process

Changes to the Drupal core are usually made after consideration, planning, and consultation. They are also made on a priority basis—fixes come before additions, and changes for which there is a high demand come before proposals that have gone relatively unnoticed. Any potential change has to be considered not only on its own merits but in relation to the aims and principles of the project as a whole.

The particular stages that a new feature goes through vary, but a typical cycle for a significant change might include:

- General discussion of the idea, for example through a posting in a drupal.org forum. This can be a chance to gauge support and interest, scope the issue, and get some direction and suggestions on approaches to take. If you’re considering substantive changes, starting out at the discussion level - rather than jumping straight into code changes - can save you a lot of time.
- Posting an issue through the drupal.org project system.
- Discussion raising issues on the proposed direction or solution, which may include a real-time meeting through IRC.
  Individual Drupal community members may vote for (+1) or against (-1) the change. While informal, this voting system can help quantify support.
- Producing a patch with specific proposed code changes.
- Review of the changes and further discussion.
- Revisions to address issues.
- Possible application of the patch.

The process of discussion and revision might be repeated several times to encompass diverse input. At any point in the process, the proposal might be:

- Shelved as impractical or inappropriate.
- Put off until other logically prior decisions are made.
- Rolled into another related initiative.
Superceded by another change.

If you submit suggestions that don’t end up being adopted, please don’t be discouraged! It doesn’t mean that your ideas weren’t good—just that they didn’t end up finding a place. The discussion itself may have beneficial outcomes. It’s all part of collaboratively building a quality open source project.

Criteria for evaluating proposed changes

The following criteria are used by core developers in reviewing and approving proposed changes:

- **The changes support and enhance Drupal project aims**
- **The proposed changes are current.** Especially for new features, priority is usually given to development for the "HEAD" (the most recent development version of the code, also referred to as the CVS version) as opposed to released versions. There may have been significant changes since the last release, so developing for the CVS version means that
- **The proposed change doesn’t raise any significant issues or risks.** Specifically, issues that have been raised in the review process have been satisfactorily addressed.
- **The changes are well coded.** At a minimum, this means coding in accordance with the Drupal coding standards. But it also means that the coding is intelligent and compact. Elegant solutions will have greater support than cumbersome ones that accomplish the same result.
- **There is demonstrated demand and support for the change.** Demand is indicated by, e.g., comments on the drupal.org issues system or comments in forums or the drupal-dev email list.
- **The change will be used by a significant portion of the installed Drupal base** as opposed being relevant only to a small subset of Drupal users.
- **The benefits of the change justifies additional code and resource demands.** Every addition to the code base increases the quantity of code that must be actively maintained (e.g., updated to reflect new design changes or documentation approaches). Also, added code increases the overall Drupal footprint through, e.g., added procedure calls or database queries. Benefits of a change must outweigh these costs.

Maintaining a project on drupal.org

Each drupal.org project (a contributed theme, module or translation) needs to be maintained in the contributions repository. Before creating a project page on drupal.org, apply for a CVS account and commit your project to the repository. If you are not using the drupal.org infrastructure, you can’t setup a project page on drupal.org nor can you offer your module for
download at drupal.org.

To get your project listed on drupal.org after it has been committed to CVS, fill in the form at [http://drupal.org/node/add/project_project/](http://drupal.org/node/add/project_project/). Make sure that the ‘Short project name’ matches the directory name in the CVS repository. For example, the `contributions/modules/my_module` module has the short name `my_module`.

Note that the newly created project will not be instantly available as it will need to be approved by one of the administrators. After that, it will appear soon after you committed some code/updates to the contributions repository. Once the project page became available, people will be able to file bugs against your project, add tasks or request new features. Your project will also become available for download.

**Downloads and packaging**

As soon your project page has been activated and assuming it is properly configured, drupal.org will automatically package your project and make it available for download. Projects are packaged once or twice a day so your project will not be available instantly.

**Managing releases**

Releases are handled using CVS branches. By default, only the CVS HEAD version (development version) of your project is packaged and offered for download.

However, if you branch your project using the `DRUPAL-4-5` branch name, drupal.org will package the Drupal 4.5 compatible release of your project. For this to work, you must use the correct branch names. A list of valid branch names can be found in the contributions repository’s FAQ.txt.

As projects are only packaged once or twice a day, it might take up to 24 hours for new releases to become available on the website or for updates to propagate to the downloads.

If you found a bug that needs to be fixed in several releases of your project, make sure to commit the fix to the different branches unless you are no longer maintaining certain releases of your project.

Branching and releases are restricted to the `modules`, `themes`, `theme-engines` and `translations` directories in the contributions repository. Personal sandboxes in the sandbox directory can’t be branched, won’t be packaged and can’t get a project page on drupal.org.

**Orphaned projects**

If you are no longer capable of maintaining your project, please add a note to your project page and ask in the forums whether someone is willing to take over maintenance. Proper communication is key so make sure to mark your project as orphaned. If you found a new maintainer or if you are willing to maintain an orphaned project, get in touch with a site maintainer so we can transfer maintainership.
Tips for contributing to the core

The following tips might improve the chances of your contributions being accepted. Supplying patches is how to make changes to core or contributed code.

- Start small. Review other patches and offer constructive suggestions and improvements. Tackle a few small bugs from the issue queue. Prove yourself. Before you wade in with a substantial change, develop a profile as a dependable, collaborative contributor.
- Take a step back and objectively evaluate whether the changes are appropriate for the Drupal core. Ask yourself:
  - Is the feature already implemented? Search the forums and issue tracker.
  - Could the feature be implemented as a contributed module rather than a patch to the core?
  - Will the change benefit a substantial portion of the Drupal install base?
  - Is the change sufficiently general for others to build upon cleanly?
- Be explanatory, provide descriptions and illustrations, make a good case. Don’t count on others downloading, installing, and testing your changes. Rather, show them in a nutshell what your changes would mean. Anticipate and address questions or concerns. If appropriate, provide screenshots.
- Be collaborative. Be friendly and respectful. Acknowledge the effort others put in.
- Be open to suggestions and to other ways of accomplishing what you’re aiming for.
- Be persistent. If you don’t get any response right away, don’t necessarily give up. If you’re still convinced your idea has merit, find another way to present it. Request another developer - ideally, one familiar with your work - to take the time to review your issue.
- Respond, in a timely way, to suggestions, requests, or issues raised. Revise your work accordingly.
- If some time has gone by, update your changes to work with the current CVS version.

List of maintainers

This comes from MAINTAINERS.txt and may not be completely up to date. It roughly lists what users in the community maintain core Drupal modules. See Drupal core for more information on what this means.

BOOTSTRAP
Moshe Weitzman

BLOG API
James Walker

DISTRIBUTED AUTHENTICATION MODULES
Moshe Weitzman
DOCUMENTATION COORDINATOR
Charlie Lowe

FILTER SYSTEM
Steven Wittens

FORM SYSTEM
Károly Négyesi

LOCALE MODULE
Gábor Hojtsy

MENU SYSTEM
Richard Archer

TABLESORT API
Moshe Weitzman

PATH MODULE
Matt Westgate

POSTGRESQL PORT
Piotr Krukowiecki

SECURITY COORDINATOR
Károly Négyesi

STATISTICS MODULE
Jeremy Andrews

SEARCH
Steven Wittens

XML-RPC SERVER/CLIENT
Károly Négyesi

DEBIAN PACKAGE
Hilko Bengen

TRANSLATIONS COORDINATOR
Gerhard Killesreiter

THE REST:
Dries Buytaert
Mailing lists

Support
A list for support questions.

Development
A list for Drupal developers.

Themes
A list for Drupal theme developers/designers.

Documentation
A list for documentation contributors.

Translations
A list for Drupal UI translators.

Consulting
A mailing list for Drupal consultants and Drupal service/hosting providers.
CVS commits

A list with all CVS commit messages.

Infrastructure

A list for Drupal infrastructure maintainers (eg. drupal.org, CVS, mailing lists).

DrupalCON

A list for the organization of Drupal conferences and events.

Subscribe

E-mail address:

Mailing lists:

- support
- development
- themes
- documentation
- translations
- consulting
- drupal-cvs
- infrastructure
- drupal-con

Mailing of project issues

Every project issue for Drupal with patch status is emailed to the drupal-devel mailing list when updated. These are mailed to promote peer review of code potentially going into Drupal. Other issues are not emailed because it would make the mailing list less useful as email volume increases. You can subscribe to project issue updates for any contributed module, theme, or translation.
Coding standards

Drupal Coding Standards

*Note: The Drupal Coding Standards applies to code that is to become a part of Drupal. This document is based on the [PEAR Coding standards](#).*

Indenting

Use an indent of 2 spaces, with no tabs.

Control Structures

These include if, for, while, switch, etc. Here is an example if statement, since it is the most complicated of them:

```php
if (condition1 || condition2) {
    action1;
}
elseif (condition3 && condition4) {
    action2;
}
else {
    defaultaction;
}
```

Control statements should have one space between the control keyword and opening parenthesis, to distinguish them from function calls.

You are strongly encouraged to always use curly braces even in situations where they are technically optional. Having them increases readability and decreases the likelihood of logic errors being introduced when new lines are added.

For switch statements:

```php
switch (condition) {
    case 1:
        action1;
        break;
    case 2:
        action2;
        break;
    default:
        defaultaction;
        break;
}
```
Function Calls

Functions should be called with no spaces between the function name, the opening parenthesis, and the first parameter; spaces between commas and each parameter, and no space between the last parameter, the closing parenthesis, and the semicolon. Here’s an example:

```php
$var = foo($bar, $baz, $quux);
```

As displayed above, there should be one space on either side of an equals sign used to assign the return value of a function to a variable. In the case of a block of related assignments, more space may be inserted to promote readability:

```php
$short = foo($bar);
$long_variable = foo($baz);
```

Function Declarations

```php
function funstuff_system($field) {
  $system['description'] = t("This module inserts funny text into posts randomly.");
  return $system[$field];
}
```

Arguments with default values go at the end of the argument list. Always attempt to return a meaningful value from a function if one is appropriate.

Arrays

Arrays should be formatted with a space separating each element and assignment operator, if applicable:

```php
$some_array = array('hello', 'world', 'foo' => 'bar');
```

Note that if the line spans longer than 80 characters (often the case with form and menu declarations), each element should be broken into its own line, and indented one level:

```php
$form['title'] = array(
  '#type' => 'textfield',
  '#title' => t('Title'),
  '#size' => 60,
  '#maxlength' => 128,
  '#description' => t('The title of your node.'),
);
```

Note the comma at the end of the last array element--this is not a typo! It helps prevent parsing errors if another element is placed at the end of the list later.
Comments

Inline documentation for classes should follow the Doxygen convention. More information about Doxygen can be found here:

- Document block syntax
- Comment commands

Note that Drupal uses the following docblock syntax:

```
/**
 * Comments.
 */
```

And all Doxygen commands should be prefixed with a @ instead of a /.

Non-documentation comments are strongly encouraged. A general rule of thumb is that if you look at a section of code and think "Wow, I don’t want to try and describe that", you need to comment it before you forget how it works.

C style comments (/* */) and standard C++ comments (//) are both fine. Use of Perl/shell style comments (#) is discouraged.

Including Code

Anywhere you are unconditionally including a class file, use require_once(). Anywhere you are conditionally including a class file (for example, factory methods), use include_once(). Either of these will ensure that class files are included only once. They share the same file list, so you don’t need to worry about mixing them - a file included with require_once() will not be included again by include_once().

**Note**: include_once() and require_once() are statements, not functions. You don’t need parentheses around the filename to be included.

PHP Code Tags

Always use <?php ?> to delimit PHP code, not the <? ?> shorthand. This is required for Drupal compliance and is also the most portable way to include PHP code on differing operating systems and setups.

Note that the final ?> should be omitted from all code files--modules, includes, etc. The closing delimiter is optional, and removing it helps prevent unwanted white space at the end of files which can cause problems elsewhere in the system. More information is available from the [PHP Code tags](phphandbook) portion of the handbook.
Header Comment Blocks

All source code files in the core Drupal distribution should contain the following comment block as the header:

```php
<?php
// $Id$
```

This tag will be expanded by the CVS to contain useful information

```php
<?php
// $Id: CODING_STANDARDS.html,v 1.7 2005/11/06 02:03:52 webchick Exp $```

Using CVS

Include the Id CVS keyword in each file. As each file is edited, add this tag if it’s not yet present (or replace existing forms such as "Last Modified:", etc.).

The rest of this section assumes that you have basic knowledge about CVS tags and branches.

CVS tags are used to label which revisions of the files in your package belong to a given release. Below is a list of the required CVS tags:

- **DRUPAL-X-Y**
  
  (required) Used for tagging a release. If you don’t use it, there’s no way to go back and retrieve your package from the CVS server in the state it was in at the time of the release.

Example URLs

Use "example.com" for all example URLs, per [RFC 2606](http://www.example.com).

Naming Conventions

Functions and Methods

Functions and methods should be named using lower caps and words should be separated with an underscore. Functions should in addition have the grouping/module name as a prefix, to avoid name collisions between modules.

Private class members (meaning class members that are intended to be used only from within the same class in which they are declared; PHP 4 does not support truly-enforceable private namespaces) are preceded by a single underscore. For example:

```php
_node_get()
```

```php
$this->_status
```
**Constants**

Constants should always be all-uppercase, with underscores to separate words. Prefix constant names with the uppercased name of the module they are a part of.

**Global Variables**

If you need to define global variables, their name should start with a single underscore followed by the module/theme name and another underscore.

**Filenames**

All documentation files should have the filename extension ".txt" to make viewing them on Windows systems easier. Also, the filenames for such files should be all-caps (e.g. README.txt instead of readme.txt) while the extension itself is all-lowercase (i.e. txt instead of TXT).

Examples: README.txt, INSTALL.txt, TODO.txt, CHANGELOG.txt etc.

**Doxygen formatting conventions**

Doxygen is a documentation generation system. The documentation is extracted directly from the sources, which makes it much easier to keep the documentation consistent with the source code.

There is an excellent [manual] at the Doxygen site. The following notes pertain to the Drupal implementation of Doxygen.

**General documentation syntax**

To document a block of code, the syntax we use is:

```
/**
 * Documentation here
 */
```

Doxygen will parse any comments located in such a block. Our style is to use as few Doxygen-specific commands as possible, so as to keep the source legible. Any mentions of functions or file names within the documentation will automatically link to the referenced code, so typically no markup need be introduced to produce links.

**Documenting files**

It is good practice to provide a comment describing what a file does at the start of it. For example:
The line immediately following the @file directive is a short description that will be shown in the list of all files in the generated documentation. Further description may follow after a blank line.

**Documenting functions**

All functions that may be called by other files should be documented; private functions optionally may be documented as well. A function documentation block should immediately precede the declaration of the function itself, like so:

```php
/**
 * Verify the syntax of the given e-mail address.
 * Empty e-mail addresses are allowed. See RFC 2822 for details.
 * @param $mail
 *   A string containing an email address.
 * @return
 *   TRUE if the address is in a valid format.
 */
function valid_email_address($mail) {
```

The first line of the block should contain a brief description of what the function does. A longer description with usage notes may follow after a blank line. Each parameter should be listed with a @param directive, with a description indented on the following line. After all the parameters, a @return directive should be used to document the return value if there is one. Functions that are easily described in one line may omit these directives, as follows:

```php
/**
 * Convert an associative array to an anonymous object.
 */
function array2object($array) {
```

The parameters and return value must be described within this one-line description in this case.

**Documenting hook implementations**

Many modules consist largely of hook implementations. If the implementation is rather standard and does not require more explanation than the hook reference provides, a shorthand documentation form may be used:
/**
 * Implementation of hook_help().
 */
function blog_help($section) {

This generates a link to the hook reference, reminds the developer that this is a hook implementation, and avoids having to document parameters and return values that are the same for every implementation of the hook.

Documenting themeable functions

In order to provide a quick reference for theme developers, we tag all themeable functions so that Doxygen can group them on one page. To do this, add a grouping instruction to the documentation of all such functions:

/**
 * Format a query pager.
 * ...
 * @ingroup themeable
 */
function theme_pager($tags = array(), $limit = 10, $element = 0, $attributes = array()) {
    ...
}

The same pattern can be used for other functions scattered across multiple files that need to be grouped on a single page.

Configuring vim

Indentation

The following commands will indent your code the right amount, using spaces rather than tabs and automatically indent after you start

set expandtab
set tabstop=2
set shiftwidth=2
set autoindent
set smartindent

Syntax highlighting

If you enjoy syntax highlighting, it is may be worth remembering that many of drupal’s php files are *.module or *.inc, among others.
Using these settings only with drupal

- Copy your .vimrc to .vimrc-drupal
- Append these settings to the end
- Run `vim -u ~/.vimrc-drupal`

To make this easier (using bash on linux), you could create an alias by typing:

```bash
alias vid="vim -u ~/.vimrc-drupal"
```

This allows you to just use the vid command instead of vi when you want to edit a drupal file.

Vim seems to syntax highlight *.inc files properly by default but doesn't know that *.module is php content. For *.modules use this snippet in .vimrc:

```vim
if has("autocmd")
  " Drupal *.module files.
  augroup module
  autocmd BufRead *.module set filetype=php
  augroup END
endif
```

**Indenting**

Use an indent of 2 spaces, with no tabs. No trailing spaces.

**PHP Code tags**

Always use `<?php ?>` to delimit PHP code, not the `<? ?>` shorthand. This is required for Drupal compliance and is also the most portable way to include PHP code on differing operating systems and setups.

Note that as of Drupal 4.7, the `?>` at the end of code files (modules, includes, etc.) is purposely omitted. The full discussion that led to this decision is available from the [no ?> needed at the end of modules](https://www.drupal.org/node/118975) discussion on the drupal-devel mailing list, but can be summarized as:

- Removing it eliminates the possibility for unwanted whitespace at the end of files which can cause "header already sent" errors, XHTML/XML validation issues, and other problems.
- PHP.net itself removes the closing delimiter from the end of its files (example: [prepend.inc]), so this can be seen as a "best practice."
SQL naming conventions

- Don’t use (ANSI) SQL / MySQL / PostgreSQL / MS SQL Server / ... Reserved Words for column and/or table names. Even if this may work with your (MySQL) installation, it may not with others or with other databases. Some references:
  - (ANSI) SQL Reserved Words
  - MySQL Reserved Words: 4.x, 3.23.x, 3.21.x
  - PostgreSQL Reserved Words
  - MS SQL Server Reserved Words

Some commonly misused keywords: TIMESTAMP, TYPE, TYPES, MODULE, DATA, DATE, TIME, ...
See also [bug] SQL Reserved Words

- Capitalization, Indentation
  - UPPERCASE reserved words
  - lowercase (or Capitalize) table names
  - lowercase column names

Example:

```sql
SELECT r.rid, p.perm
FROM {role} r
LEFT JOIN {permission} p ON r.rid = p.rid -- may be on one line with prev.
ORDER BY name
```

- Naming
  - Use plural or collective nouns for table names since they are sets and not scalar values.
  - Name every constraint (primary, foreign, unique keys) yourself. Otherwise you’ll see funny-looking system-generated names in error messages. This happened with the `moderation_roles` table which initially defined a key without explicite name as KEY (mid). This got mysqlump’ed as KEY `mid` (mid) which resulted in a syntax error as mid() is a mysql function (see [bug] mysql --ansi cannot import install database).
  - Index names should begin with the name of the table they depend on, eg. INDEX `users_sid_idx`.

References:

- Joe Celko - Ten Things I Hate About You
- Joe Celko - SQL for Smarties: Advanced SQL Programming
- RDBMS Naming conventions
- SQL Naming Conventions
String concatenations

Always use a space between the dot an the concatenate part, unless it is a quote. So there is no space between the dot and the quote.

Examples:

```php
<?php
$string = 'Foo'. $bar;
$string = $bar .'foo';
$string = bar() .'foo';
?>
```

When you concatenate simple variables, you can use double quotes and add the string inside, otherwise use single quotes.

Example:

```php
<?php
$string = "Foo $bar"
?>
```

Use of icons

When you include icons, take a few guidelines into consideration:

- For contributions: Save the icons in your module’s directory
- For core: Save the icons in the /misc directory
- Use the Tango icons whenever possible
- If no icons exist, refer to the icon style guidelines of Tango

Write E_ALL compliant code

Introductionary notice

This document is a very recent addition to the coding guidelines, and is still subject to discussion among developers. Do NOT post comments on this page yet. One issue will be created for each point introduced here. Those issues will be used for discussion and patch submission. I’ll keep updating this page as the discussion progresses. When a consensus has been reached on each point, this introduction will be deleted. Eventually, when everything has been patched up, I’ll delete the mention about Drupal not being E_STRICT compliant.

E_ALL: a better practice

Currently, the Drupal code is not E_STRICT compliant. When running a Drupal site with E_ALL, each page view creates scores of error notices messages. Many developers agree that it would be good if the source of Drupal could be brought up to par with commonly accepted good practice.
The purpose of this document is twofold:

1. to show common coding mistakes that prevent Drupal from being E_STRICT compliant.
2. to set better coding guidelines to use in new code and patches.

Once those guidelines are accepted, it is only a matter of time and some developers’ efforts before all the previous coding mistakes are patched up. Then, we will be able to run Drupal with the E_ALL directive.

Common coding mistakes and new coding practice

1- Use of if (isset($var)) or if (!empty($var))

(Note: this first issue is still under review by the developers. See discussion and patches here: http://drupal.org/node/34434. This section will be updated according to the discussion there, until a broad consensus is reached and relevant patches committed).

If you want to test if an array has been set to any value, don’t use:

```php
<?php
if ($foo) {} // (the string 'foo' in 'isset($foo)')
?>
```

but:

```php
<?php
// either
if (isset($foo)) {} // $foo=0 (zero) and $foo= '' return TRUE
// or
if (!empty($foo)) {} // use this when 0 or '' are not expected
// and are not valid values for $foo.
?>
```

The difference between isset() and !empty() is that unlike !empty(), isset() will return TRUE even if the variable is set to an empty string or to the integer 0. In order to decide which one to use, consider whether 0 or '' are valid and expected values for your variable.

The following code is wrong:

```php
<?php
function _form_builder($form, $parents = array(), $multiple = FALSE) {
    // (...
    if ($form["#input"]) {
        // some code (...
    }
}
?>
```
Here, the variable \$form is passed on to the function. If \$form['#input'] has been set to any value, some code is executed. The problem is that testing this way outputs the following error message:

notice: Undefined index: #input in includes/form.inc on line 194.

Even though the array \$form is already declared and passed to the function, each the array’s index must be explicitly declared. The previous code should read:

```php
<?php
function _form_builder($form, $parents = array(), $multiple = FALSE) {
    // (...)
    if (!empty($form['#input'])) {
        // some code (...)
    }
}?

Beware!

The function isset() returns TRUE when the variable is set to the integer 0, but FALSE if the variable is set to NULL. In some cases, is_null() is a better choice, especially when testing the value of a variable returned by an SQL query.

Testing for error notices

If you wish to help to clean up Drupal’s code so that it complies with E_STRICT, you can set up a test site and in includes/common.inc change:

```php
<?php
if ($errno & (E_ALL ^ E_NOTICE)) {
?
``` to:

```php
<?php
if ($errno & (E_ALL )) {
?
```

Functions

Functions should be named using lower caps and words should be separated with an underscore. Functions should have the grouping/module name as a prefix, to avoid name collisions between modules.
Constants

Constants should always be all-uppercase, with underscores to separate words. Prefix constant names with the uppercased name of the module they are a part of.

Control structures

These include if, for, while, switch, etc. Here is an example if statement, since it is the most complicated of them:

```php
if ((condition1) || (condition2)) {
    action1;
}
elseif ((condition3) && (condition4)) {
    action2;
}
else {
    defaultaction;
}
```

Control statements should have one space between the control keyword and opening parenthesis, to distinguish them from function calls.

You are strongly encouraged to always use curly braces even in situations where they are technically optional. Having them increases readability and decreases the likelihood of logic errors being introduced when new lines are added.

For switch statements:

```php
switch (condition) {
    case 1:
        action1;
        break;
    case 2:
        action2;
        break;
    default:
        defaultaction;
        break;
}
```

Header comment blocks

All Drupal source code files should start with a header containing the `$Id$ keyword:

```php
<?php
// $Id: CODING_STANDARDS,v 1.1 2001/11/05 07:32:17 natrak Exp $
```
Note that everything after the starting $Id$ and before the closing $ is automatically generated by CVS - you shouldn’t edit this manually. If you add a new file to CVS, just write // $Id$. 
CVS

CVS (the **concurrent version system**) is a tool to manage software revisions and release control in a multi-developer, multi-directory, multi-group environment. It comes in very handy to maintain local modifications.

Thus, CVS helps you if you are part of a group of people working on the same project. In large software development projects, it’s usually necessary for more than one software developer to be modifying modules of the code at the same time. Without CVS, it is all too easy to overwrite each others’ changes unless you are extremely careful.

In addition, CVS helps to keep track of all changes. Therefore, the CVS server has been setup to mail all CVS commits to all maintainers. Thus, it does not require any effort to inform the other people about the work you have done, and by reading the mails everyone is kept up to date.

**CVS concepts**

Drupal uses Concurrent Versions System (CVS) to co-ordinate development, which can be thought of as a system for controlling the contents of a library. Describing CVS in terms of a library, books, and editions is only a metaphor - unlike a real, physical library, no matter how many people check out a book, it will always be available to the next person.

As with any library though, there are rules of behaviour towards other users and how you treat the library materials, no need to worry about overdue library tickets or leaving coffee stains on pages, but please be sure to act responsibly when contributing content to the library.

**Repository**

A repository can be thought of as a book, Drupal has two repositories which can be checked out (downloaded to your local computer):

- **Drupal**
  the core Drupal code, i.e. what is downloaded as ‘Drupal’.

- **Contributions**
  modules, themes, translations, etc. supplied by contributors, i.e. all Drupal material that is not in the core.

**Branch**

CVS tracks different versions of content. Imagine different editions of a text book, each edition including amendments and additions - each edition is known as a ‘branch’, and has a branch tag to identify it, e.g. Drupal 4.5. Every branch corresponds to a particular version of Drupal that gets released.

**Head**
This is a special edition of the book (repository) which has not been published yet, or in
CVS speak has not been "branched" yet. Think of "head" as a manuscript of the next edition of
the book. Amendments and additions are added to the manuscript, it’s proof-read and tested,
and once it’s ready it gets published (branched) as a new edition.

**Working copy/Work area**

Users can check out a copy of a book to work on locally, the original remains in the library
and can be checked out by other users. When checking out a book remember that it is an
edition (branch or head) of the book (repository). The copy of the book which is on the
user’s local computer is known as the "Working copy", "Work area" or "Work directory".

Changes the user makes to the local copy can be sent back to the library for inclusion in the
repository.

**Project**

Modules, themes or translations can be added (committed) by users to the Contributions
book (repository) at any time. It’s possible to add to the manuscript (head) of Contributions,
or any of the previous editions (branches). The website drupal.org tracks user additions to
the Contributions book (repository) as "Projects". Each project has a description page from
which it can be downloaded, and where issues and feature requests can be added by users.

**Patch**

Only a few Drupal developers are able to make changes directly to the Drupal book
(repository). All other users who want to include changes they’ve made while working on
their local copy of the Drupal book (repository), must create a file showing the differences
between the local version and the version at drupal.org. This differences file is known as a
"patch", and is sent to the rest of the development community by creating an "issue" at
drupal.org and attaching the file.

**Using CVS with branches and tags**

To manage the different Drupal versions, we use tags and branches. A branch specifies a major
Drupal version. For example, all 4.4.x versions belong in the DRUPAL-4-4 branch. Whenever we
release a specific version, we create a tag. A tag is a marker which defines a snapshot of all the
files in the CVS at a certain moment. For example, the tag DRUPAL-4-4-0 specifies all files at the
time of the 4.4.0 release. The HEAD branch is special and is used to refer to the latest
development version.

For an up-to-date complete list of branches and tags, see "Show files using tag:" at [ViewCVS](at the bottom).

Here’s a quick guide on using tags and branches. This assumes you have successfully checked
out the main and contributions repositories. In my case, I usually make a folder in my home
directory for each cvs server. In Drupal’s case, cvs.drupal.org. For instance, you’ve made the
CVS folder and here you check out your copy of the CVS version of Drupal.
~cvs.drupal.org$ cvs -d :pserver:anonymous@cvs.drupal.org:/cvs/drupal login
~cvs.drupal.org$ cvs co drupal

This should leave you with a folder `cvs.drupal.org/drupal` which contains the current CVS code. You can keep this up-to-date by going into the `cvs.drupal.org/drupal` directory and using the command

~/cvs.drupal.org/drupal$ cvs update -dP

which will give you the latest copy, create any new directories that exist in the repository (-d), and trim unused directories (-P). Note that you don’t need to specify the server at this point since the `drupal` directory contains a CVS folder that contains the repository and root information.

You’ve also done this for the contributions,

~cvs.drupal.org$ cvs -d :pserver:anonymous@cvs.drupal.org:/cvs/drupal-contrib login
~cvs.drupal.org$ cvs co contributions

Which leaves you with the latest contributions in the `cvs.drupal.org/contributions` directory.

But now you want to have a nice copy of the 4.3.0 version, and you don’t want to have to download the tgz file all the time. The CVS maintainer has branched the drupal repository and tagged it to keep track of this release. If you download it directly with the release tag it’s going to overwrite your drupal folder. You probably want to keep it simple and use this command:

~cvs.drupal.org$ cvs -d :pserver:anonymous@cvs.drupal.org:/cvs/drupal -q checkout -d drupal-4.3 -r DRUPAL-4-3 drupal

That’s going to create a new directory `cvs.drupal.org/drupal-4.3.0` that contains the 4.3.0 version. Once it’s been checked out, you don’t need to worry about specifying it again. The CVS directory in the `drupal-4.3.0` directory has the tag information along with repository and root information like we saw before. Just go into the `drupal-4.3` folder and execute

~/cvs.drupal.org/drupal-4.3$ cvs update -dP

to keep your copy of the 4.3.0 branch up to date.

**Windows**

You may have noticed this was geared towards the linux user. Sorry, I haven’t used the windows clients for CVS. I’m sure they would work, I just haven’t tried them (for very long). You could probably do this same thing on your windows box by installing one of the windows GUIs or using [Cygwin](http://www.cygwin.com) a GNU/UNIX client for windows. Probably the easiest way it to use [TortoiseCVS](http://www.tortoisecvs.org)
Available Branches

The available branches currently are:

- HEAD
- DRUPAL-4-7
- DRUPAL-4-6
- DRUPAL-4-5
- DRUPAL-4-4
- DRUPAL-4-3
- DRUPAL-4-2
- DRUPAL-4-1
- DRUPAL-4-0
- DRUPAL-3-0

The DRUPAL-4-3-0 tag we used above is a marker in the DRUPAL-4-3 branch. Other tags for DRUPAL-4-3 are DRUPAL-4-3-1 and DRUPAL-4-3-2.

Apply for contributions CVS access

CVS GUIs and clients

There are a number of CVS ‘front-ends’ or GUIs which aim to improve on the command-line tools of CVS. These tools are grouped here by operating system/platform.

Cross-platform CVS clients

There are a number of GUI clients which run on multiple operating systems and platforms

Eclipse CVS plug-in

Perhaps the best CVS front-end I’ve used is the cross-platform tool Eclipse (www.eclipse.org). It has the functionality to do almost anything you can do with the command line, but you can use the console if you need to. As well, you can use it to edit the PHP code with another plugin.

Checkout Drupal CVS into the Eclipse Workspace

Eclipse is a good way for Developers to Contribute to Drupal, it has integrated CVS Support and a good PHP Plugin for full PHP4/PHP5 Support including Debugger, Syntax Highlighting, code completion and more.
Installing Eclipse with PHP Support

First you need a copy of eclipse. The Eclipse Homepage is http://www.eclipse.org. The Installation is Easy. You need the Java SDK and the right Eclipse SDK package for your OS. Extract the zip file and start Eclipse.

For PHP Support you need [PHP Eclipse]. Installation is the same as for eclipse, download the latest stable for your Eclipse Version and unzip it in your Eclipse folder.

**Attention:** Please install PHP support before you start with your Projects, it makes anything easier for you.

Checkout Drupal in your Eclipse Workspace

The easiest way to get Drupal CVS into your Eclipse Workbench is the "Import from CVS" feature.

1. Start Eclipse and open the *File/Import...* Dialog.
2. Choose "Checkout Projects from CVS", click "Next" and "Create a new repository Location", click "Next".
3. For Anonymous Access to the Drupal CVS insert the following information:
   - **Host:** cvs.drupal.org
   - **Repository Path:** /cvs/drupal
   - **User:** anonymous
   - **Password:** anonymous
   - **Connection type:** pserver
   Also make sure "Save Password" is checked.

As a developer for contrib you must type
   - **Repository Path:** /cvs/drupal-contrib
   - **User:** your username as filled out in the CVS access form
   - **Password:** your password

4. In the "checkout Module" Dialog select "Use an existing module" and choose the module you want, "drupal" for the Drupal Core, or "contribution" for the 3rd Party modules and themes. Click "Next"
5. 
6. Check out the Module with the "New Project Wizard" and select the Branch you want, this is usually "HEAD", now press "Finish".
7. In the "New Project Wizard" Select PHP/PHP Project as Project Type. This Project type only appears if you have PHP Eclipse installed, if you don't have PHP Eclipse installed use "Simple/Project" instead.
8. Give the module an useful Name like "drupal-contrib" and press finish.

9. The CVS Checkout starts, the contrib module take a long time (around 15 Minutes), so don’t be impatient.

**CVS front ends for Windows**

There are many CVS GUI front ends for Windows. Please add a new section if you know of additional ones

**SmartCVS**

SmartCVS is another multi-platform CVS client based on java.

It has the typical features, like built-in File Compare/Merge, Show Transactions or List Repository Files.

You can download the latest version from [http://www.smartcvs.com/](http://www.smartcvs.com/). The checkout / update process is similar to all of the CVS gui programs.

**TortoiseCVS**

[TortoiseCVS](http://www.tortoisecvs.org/download.shtml) lets you work with files under CVS version control directly from Windows Explorer. It’s freely available under the GPL. The following tutorial teaches how to use TortoiseCVS with Drupal.

- Download TortoiseCVS from [http://www.tortoisecvs.org/download.shtml](http://www.tortoisecvs.org/download.shtml) and install it.
- In Windows Explorer, select the folder under which you want the Drupal source directory to live. Right-click on it. There are two new sections in the context menu - **CVS Checkout** and **CVS >**. Select **CVS Checkout**.
- Fill in the following fields:
  - **Protocol**: Password server (:pserver:)
  - **Server**: [cvs.]drupal.org
  - **Repository folder**: /cvs/drupal (main distro) or /cvs/drupal-contrib (contributions)
  - **User name**: anonymous
  - **Module**: drupal (main distro) or contributions (contributions)

and press "OK".

- You will be asked for password. Enter anonymous and press "OK".
- A log window which monitors the checkout process will appear. Checking out the whole CVS repository will take a while.
- If everything works, you will see the message "Success, CVS operation completed" at the
end of the log. A new directory (named like the module selected before) with the sources will be created.

- To bring your Drupal source tree up-to-date, select it’s root folder ("drupal" / "contributions"), right-click it and do a "CVS Update".

The process above retrieves the freshest files from the repository (the so-called HEAD branch). These are sometimes unstable. To get Drupal modules and themes that are stable and ready for production (which you can also download from the [Drupal downloads page](http://drupal.org/project/downloads)), follow the process described above, but before hitting "OK" you need to:

- Click on the "Revision" tab on the CVS checkout dialog.
- Enable "Get tag/branch".
- Enter `DRUPAL-4-1-0` or `DRUPAL-4-00` depending on the version you are using in the tag/branch field.
- Hit OK.

You can also generate patch files with TortoiseCVS. Just select the files which you have patched in Windows Explorer. Then right click into the `CVS => Make Patch` menu item. Then you may wish to read [Creating and sending your patches](http://drupal.org/project/downloads)

**WinCVS**

WinCVS is another graphical CVS client available for MS Windows and for Macs. You can download the latest version from [http://www.wincvs.org/](http://www.wincvs.org/) The checkout / update process is similar to the one described above.

**Adding a new project**

These instructions are written for WinCVS, but the concepts probably apply to Tortoise as well.

Assumptions: a working installation of WinCVS. A CVS account at drupal.org. A checked-out contributions/modules (or themes or whatever) directory with CVS directories intact giving the correct login (not anonymous) information.

**Do not use Remote -> Import module.** If you are used to working on Sourceforge this may take some getting used to - remember that Drupal contributions use common branches/tags so you don’t want to be creating a whole new CVS module, just adding a subdirectory to the existing tree.

In fact it is easy to add your project, provided you have the contributions directory created when checking out with its CVS subdirectory intact. If you don’t, just check out any project using the [previous instructions](http://drupal.org/project/downloads) to get it back. From then on the process is really simple.

Copy your project’s directory into the contributions/modules directory (or contributions/themes or whatever). It should show up in the right hand pane of WinCVS without a tick in the folder icon. Select the folder in the right hand pane and select Modify -> Add. This will create the directory on the CVS server: you can then add and commit the files within your project.
Checking out a project using WinCVS

These instructions are written for WinCVS, but the concepts probably apply to Tortoise as well. They are intended for developers with a CVS account.

Assumptions: a working installation of WinCVS. A CVS account at drupal.org.

You may have checked out modules anonymously before. You will need to start again as the CVS Root definition (which is stored in each local copy of a checked out directory) needs to contain your login information. It is a good idea to start with an empty folder to keep things clean - I have a folder DrupalSandbox on my Desktop.

1. Use View -> Browse location -> Change to set the local path to DrupalSandbox or whatever your folder is.
2. Select Remote -> Checkout module
3. On the Checkout settings tab, tick CVSROOT and click the ‘...’ button to the right of the box.
4. In Protocol select pserver, and in Repository path enter /cvs/drupal-contrib. Then under Keyword select username and then the Edit button to enter your username; do the same for password (this is of course the username and password you stated when requesting a CVS account). Select Hostname and enter cvs.drupal.org. Click OK.
5. Now still in Checkout settings, under Module name and path on the server enter contributions/modules/image (or whatever). Other than CVSROOT the check boxes on this tab should all be unchecked.
6. Click OK, and WinCVS should connect to the server and check out the code. It will put it in a subdirectory contributions/modules/image (or contributions/theme-engines/phptemplate or whatever). You can move the image directory to your DrupalSandbox folder if you want, but keep contributions/modules etc. for later.

You can now work with WinCVS to manage your local copy of the files, updating and committing as necessary.

CVS on Mac OS X

There are a number of good CVS clients for Mac OS X.

CVL: point and click CVS

For Mac users who are unused to the command line, CVS can at first look a bit daunting. Fortunately there is an application called CVL that provides control of CVS through a point and click interface. Most of the instructions for using CVL will also apply when using other applications to control CVS.
Setting up/step by step CVS

Here is a step-by-step guide to installing and setting up CVL.

1. **Install CVS.** The first step of using any application is of course to install it - CVS is installed by default with the Apple Developer tools, so if you haven’t installed these yet, download the latest version and install them. They also include a lot of other useful stuff like Project Builder and File Merge (Apple Developer Membership is required, but free).

2. **Install CVL**

   The CVL package and complete installation instructions are available at [http://www.sente.ch/software/cvl/](http://www.sente.ch/software/cvl/)

3. **Setup CVL to work with CVS.**

   Set CVS to ignore the ‘hidden’ .DS_Store files which OS X creates in each folder. To do this you need to open the Terminal (Application->Utilities->Terminal), and type the following:

   ```
   cd  
   pico  
   .DS_Store 
   ```

   takes you to root of your account

   opens the Pico text application

   specifies which file types you want CVS to ignore

   Now press the keys: **Control** and **x** at the same time

   This closes the document you’ve just written, it will ask you if you want to save it - press **y** for yes, then type in the name of the file **.cvsignore** (note the ‘.’) and press return. You’ve finished with the Terminal, so you can quit it.

4. **Create a folder to put the CVS files into.** The best place to do this is in the ‘Sites’ folder, to make it easy to use them through the Apache server built into your system. You can name the folder anything you want, my one is called ‘drupal_cvs’.

5. **Step five** open the CVL application, you now need to Checkout (download) the latest version of the Drupal CVS like this:

   Tools->Repositories->Show Repositories

   Click Add
Choose a repository dialog box will appear.

In Repository type choose pserver.

CVS User: your Username (that you applied for CVS with)

Host: cvs.drupal.org

Path: /cvs/drupal (main distro) or /cvs/drupal-contrib (contributions)

Password: your password (that you applied for CVS with)

Click Add.

Next go to Tools->Repositories->Show Repositories

The Drupal repository is now listed in the Repositories window. Select it and press Checkout...

Checkout Module dialog box appears.

Choose Module: drupal (main distro) or contributions (contributions)

New work area location: Choose... select the folder you created in step four.

Press Checkout.

Wait patiently, this may take some time, as the whole of the Repository needs to be downloaded - you can see this happening if you open the console window (Tools->Console->Show Console), don’t worry if you don’t see anything at first, CVL usually thinks about what it’s doing for a minute or two before taking action.

When this is finished you will have a copy of the Drupal repository files in the folder you created on your hard drive, this is your Work Area, where you work on projects before uploading them to the repository for others to use.

**Basic CVS with CVL**

You now have a Work Area on your hard drive which is a mirror of the Repository on the Drupal server. You can see this by using the CVL menu Work Area->Open Recent and selecting the repository you just downloaded (drupal or contributions).

You can use this work area in the same way you would any other folder on your hard drive - create new files with BBEdit (or whatever you use), drag files to the trash, add new folders, delete folders - it’s just a regular folder.

Once you’ve done some work you want to upload back to the Drupal server here’s what you do:
Update the CVS by selecting the folder the new work is in, then Control+Click on the folder and choose Update from the contextual menu that pops up (or through the menu File->Update). CVS now shows any new files or folders that you have added (with a blue * in front).

Next you need to tell CVS to mark the files and folders for upload next time you send your changes to the Drupal repository. To do this select the files and folders and Control+Click, choose Add To Work Area (or through the menu File->Add To Work Area).

To upload your work to the Drupal repository, select your files and folders and Control+Click, choose Commit... (or through the menu File->Commit...). CVS will now add your work to the Drupal repository.

**Preparing a project**

New module, theme or translation projects should be started in the Drupal CVS contributions repository.

In the Finder, go to the folder where you saved the CVS contributions working copy, and create a new folder in the appropriate subfolder. For example, if you are working on a new module, create new folder in the module folder. Name the new folder to whatever you want to call the project. Try to make the name short and descriptive. Avoid spaces, use "_" to separate words, but read the Developer guidelines to understand how underscores in module names may interact with code behaviour.

In CVL, open the CVS contributions work area, navigate to the folder containing the new folder you just created, control-click on it and select "Refresh" from the menu that pops up.

The new folder, and any files you put in it, should now show up in CVL with a blue * next to it.

The blue * signifies that the files have not been added to the work area yet.

In CVL select the new folder, control-click on it and select "Mark File(s) for Addition" from the menu that pops up.

The blue * will now change to a green + next to each of the new files and folders, signifying that the files are part of the working copy and can be added to the repository at drupal.org once you want to commit them.

**Committing a project**

Once your new module, theme or translation is complete you may want to add it to Drupal’s contributed repository, and create a project for it at www.drupal.org.

1. **Add your project to CVS**

   Your new project should first be added to the trunk of the contrib repository, which is know as the ’cvs’ or ’HEAD’ branch. (see Setting up)
Add the files to the ‘cvs’ version of the contrib repository (see [Preparing a project]). Once added to CVS, the project folder and each of it’s files will have a green ‘+’ next to it, this means they are ready to be committed.

Select the project’s folder, for example:

- **banana.module**

  contrib/modules/banana/banana.module

  select folder ‘banana’

With the project folder selected, control-click, select ‘Commit...’

A dialog box will appear into which you can type a log message. The log message should briefly explain what new features have been added to this version of the files or what bugs have been fixed.

2. **Add your project to the Drupal Project tracker**

Your files are now in the contrib repository, now you need to make drupal.org aware of your new project.

By creating a ‘project’ at www.drupal.org the files in CVS become available for download on the ‘Downloads’ page, it also allows users to submit feature requests and bug reports for the project.

Log in to www.drupal.org, in the side account block click on "create content", then click "project".

Fill in the project form page to create the new project. The project will appear on drupal.org in a day or two.

**Drupal CVS repositories**

**Main repository**

There are two ways to access the latest Drupal sources in the main CVS repository. If you just want to have a quick look at some files, use the [ViewCVS web interface](http://example.com). If you need the complete source tree to study and work with the code, follow these steps:

- If you don’t have it yet, install a recent copy of [CVS](http://example.com). If you are on Windows, you may check [CVS front ends for Windows](http://example.com). Mac OS X users may find the tutorial on the CVL front end for CVS in section [CVL: point and click CVS](http://example.com) helpful. Note that CVS uses port 2401 so you may need to open this on your firewall in order to perform these commands.
- To check out the latest drupal sources, run the command:
$ cvs -z6 -d:pserver:anonymous:anonymous@cvs.drupal.org:/cvs/drupal checkout drupal

This will create a directory called drupal containing the latest drupal source tree (-z6: use gzip compression so the transfer takes less time. -d: specify the ‘directory’ where the files are located and how to access them).

- To move the files in drupal to a different directory (for example, drupal_head), run the command:

  $ mv drupal/* drupal/.htaccess drupal_head/
  $ rmdir drupal

- Once you have a copy of the Drupal source tree, use

  $ cvs update -dP

  in the source root dir to update all files to their latest versions (-d: Create any (new) directories that exist in the repository if they’re missing from the working directory. -P: Prune empty directories - directories that got removed in the repository will be removed in your working copy, too).

- You can also use the command

  $ cvs -q update -dP

  to suppress informational messages related to the update (-q: quiet).

If you can’t or don’t want to use CVS, you can download nightly CVS snapshots from http://drupal.org/files/projects/drupal-cvs.tar.gz.

Other useful methods of performing the checkout are:

Checkout a specific Drupal version

$ cvs -z6 -d:pserver:anonymous:anonymous@cvs.drupal.org:/cvs/drupal co -r DRUPAL-4-6 drupal

Where DRUPAL-4-6 is one of the branches identified at the bottom of Using CVS with branches and tags

Checkout Drupal from a specific date

$ cvs -z6 -d:pserver:anonymous:anonymous@cvs.drupal.org:/cvs/drupal co -D "06 Oct 2005" drupal

Contributions repository

The Contributions repository is a separate CVS repository where people can submit their modules, themes, translations, etc. See the contributions FAQ.txt and README.txt for more information.
As the **Main repository** you can browse it via the **web interface**. For anonymous (read-only) access, do the following:

- Login by running the command

  ```
  $ cvs -z6 -d:pserver:anonymous@cvs.drupal.org:/cvs/drupal-contrib login
  ```

  The required password is ‘anonymous’ (without the quotes).

- To check out the latest drupal contributions, run the command:

  ```
  $ cvs -z6 -d:pserver:anonymous@cvs.drupal.org:/cvs/drupal-contrib checkout contributions
  ```

- To check out only specific contributions, do

  ```
  cvs -z6 -d:pserver:anonymous@cvs.Drupal.org:/cvs/drupal-contrib checkout contributions/<directory/of/contrib>
  ```

  where `<directory/of/contrib>` is a direct directory listing of the contrib. For example modules/views

- To check out contributions for a certain Drupal version, do

  ```
  $ cvs -z6 -d:pserver:anonymous@cvs.drupal.org:/cvs/drupal-contrib checkout -r <version tag> contributions
  ```

  where `<version tag>` is one of the tags listed under "Q: How do I control the releases of my module/theme?" [here](#).

- $ cvs -z6 update -dP

  in the source root dir.

If you want to add your own modules, themes, translations, etc., you need CVS write access:

**Promoting a project to be an official release**

To promote a project from the HEAD of the CVS tree to an official release state, the author needs to branch to the current stable branch, say DRUPAL-4-6.

To properly branch your code perform the following in your local workspace.

**Modules:**

```
cvs tag -b DRUPAL-4-7 modules/mymodule
```

**Themes:**

```
cvs tag -b DRUPAL-4-7 themes/mytheme
```

Note that branching is restricted to the ‘modules’, ‘themes’, ‘theme-engines’ and ‘translations’ directories. You can’t branch your sandbox.
For more information on CVS branches see:
http://www.cvshome.org/docs/manual/cvs-1.11.6/cvs_5.html#SEC54
http://cvsbook.red-bean.com/cvsbook.html#Branches

All later updates on that released version need to be done from the selected branch and committed to that branch.

**Adding/modifying a file to the CVS repository**

**Adding**

If you would like to add a file or a directory, first you need to download the parent directory. Once again: the *parent* directory, not the directory you want to add something to, but the parent of it. I can not find any logic in this, but this is so.

First, issue the following command:

```bash
export CVSROOT=:pserver:icvslogin:cvspasswd@cvs.drupal.org:/cvs/drupal-contrib
```

To add a new file to a module:

```bash
cvs co contributions/modules/modulename
cd contributions/modules
cp sourcefile modulename
cvs add modules/sourcefilename
cvs commit
```

Say you want to create a sandbox named *mysandbox*. Then do the following:

```bash
cvs co contributions/sandbox/weblinks [1]
cd contributions
mkdir sandbox/mysandbox
cvs add sandbox/mysandbox
cvs commit
```

[1] *does not really matter which directory, just sg. from sandbox. I like this one, because it is small.*

Of course, you may do several things in one commit, adding files, removing files, updating files. Neither remove (cvs remove) nor update (cvs update) is such a tedious process. Warning: you need to check out with your CVS account, because CVS ignores CVSROOT for existing checkouts.
Modifying

in the sandbox (modules etc.) directory issue:

cvs up mysandbox
cvs commit

you can update only one file:

cd mysandbox
cvs up myfile
cvs commit

Demoting an official release

To demote a project from an official release to the HEAD of the CVS tree, the author needs to branch to remove the current stable branch, say DRUPAL-4-7.

Modules:
cvs rtag -d -a DRUPAL-4-7 contributions/modules/mymodule

Themes:
cvs rtag -d -a DRUPAL-4-7 contributions/themes/mytheme

Tracking Drupal source with CVS

Note: The following assumes you have both basic knowledge of CVS and your own local repository set up and working.

If you’ve been modifying the Drupal source code for your own purposes (or developing a module or theme) and manually applying your changes to the Drupal source every time it updates, you may be glad to learn that CVS can help make this easier.

This is usually referred to as ‘tracking third-party sources’ and requires knowledge of the CVS concepts branching, release tags, and the vendor tag. We’ll work through an example here and explain these concepts as we go.

Example

Lets assume we’d like to track current Drupal CVS HEAD, and start by downloading the source. In this case we’ll export using anonymous CVS (we could also just download a tarball).

Begin by logging in to the anonymous CVS server, the required password is ‘anonymous’:

cvs -d:pserver:anonymous@cvs.drupal.org:/cvs/drupal login
Then export the newest development version of drupal using the HEAD release tag:

cvs -d:pserver:anonymous@cvs.drupal.org:/cvs/drupal export -r HEAD drupal

Now that we have a local copy of the drupal source we can import it into our own CVS repository. In this example we import with a log message including the date ’-m "message text"’, a module location/name of ’sites/drupal’ (customize that to suit your own CVS repository), a vendor tag of ’drupal’ and a release tag of ’HEAD20040110’. We also use the -ko option to prevent keyword expansion (this preserves the CVS $Id$ tags used on drupal.org):

cd drupalcvs import -ko -m "Import CVS HEAD on Jan 10th 2004" sites/drupal drupal HEAD20040110

Before we can customize we need to checkout into a working directory. Then we can modify a file or files and commit:

cvs checkout drupalcd drupal...modify a file or files...cvs commit

We now have a drupal module with a special ‘vendor branch’ (identified by the vendor tag), which contains the drupal source files we imported, and a main trunk with our modified files. Any files modified at this point are now HEAD on the main trunk of the module, whilst the unmodified files remain HEAD on the vendor branch (HEAD being what is produced by cvs update). For an individual file (fileone.php) the version history now looks like something like this:

```
HEAD

[Main trunk] fileone.php *-------------+ 1.2 +
\ +-----+

[Vendor Branch]
+ 1.1.1.1 +
+--------+
{tag:HEAD20040110}
```

## Managing contributions

Now I’ve been thinking about the best way to extend this method to incorporate contributions (eg, modules from the separate drupal.org contributions CVS repository or any other non-drupal-core code). I would welcome comments/suggestions on a good model.

The model I’m currently using:

1. Maintain updated working copies of drupal HEAD and contributions HEAD from drupal.org locally, something like:

   [HOST:~/drupal]$ ls -1
   contributions
   drupal
   imports

2. **Import drupal into my local CVS repository with vendor/release tags as per method outlined above**

3. Maintain customized versions of drupal using CVS branches which can be diff’d/updated against the latest drupal I’ve imported into my local CVS repository as suggested at the end
of the method above.

4. Make a temporary copy of the complete, updated directory of whatever contribution I want to add to Drupal’s core in a different and otherwise empty local directory so I can import it to my local CVS repository alone. For example, say I want to include the event module:

```
[MP:~/drupal/contributions/modules]$ cp -r event ../../imports
[MP:~/drupal/contributions/modules]$ cd ../../imports/
[MP:~/drupal/imports]$ ls -1a
.
.. event
```

5. Import the contribution with its own vendor/release tags into the proper place in the HEAD branch of Drupal I already have in my local CVS repository. In this example, I import the entire event module from the temporary directory that contains the event module directory and only that directory. Note how the event module is imported into the proper directory (i.e., "drupal/modules") of the Drupal project in my local CVS repository:

```
[MP:~/drupal/imports]$ cvs import -ko -m "Import module event HEAD on 04 OCT 2004" drupal/modules module_event MODULE_EVENT_20041004
```

```
   cvs import: Importing /home/mediaped/repos/drupal/modules/event
   I drupal/modules/event/CVS
   N drupal/modules/event/CHANGELOG
   N drupal/modules/event/CREDITS
   N drupal/modules/event/INSTALL
   N drupal/modules/event/LICENSE
   N drupal/modules/event/README
   N drupal/modules/event/TODD
   N drupal/modules/event/event.css
   N drupal/modules/event/event.module
   N drupal/modules/event/event.mysql
   N drupal/modules/event/event.ppgsql
   N drupal/modules/event/fields.inc
   cvs import: Importing /home/mediaped/repos/drupal/modules/event/po
   I drupal/modules/event/po/CVS
   N drupal/modules/event/po/event.pot
   N drupal/modules/event/po/event.de.po
   N drupal/modules/event/po/event.es.po
   N drupal/modules/event/po/event/he.po
   N drupal/modules/event/po/event/hu.po
   No conflicts created by this import
```

6. Now the contribution is included in the HEAD of the Drupal project in my local CVS repository, so if I update a working copy of that local Drupal project (using the update -d option to ensure I get any new contribution directories) or checkout a new HEAD working copy, the newly added contribution will be in that working copy.

7. And, if I would like to add the contribution to one of the branches I have to maintain customized, separate versions of Drupal, I just need to update a working copy of that branch with the release tag I established when importing the contribution into my local CVS repository. For example, to get the event module in a working copy of a branch tagged "drupalcustomA":

```
```
Note: if you try an update using your new tag and cvs complains that the tag doesn’t exist, apparently you have to keep trying different cvs commands using that tag until cvs updates CVSROOT/val-tags. Once you get the tag to work once, it will work from then on. See the section here on error "cvs [checkout aborted]: no such tag".

8. This adds the event module files to that working copy, but I still have to commit them to that branch in my local CVS repository:

   [MP:~/projects/drupalcustomA]$ cvs -q ci -m "adding event module"
   Checking in modules/event/... 
   ... 
   done 

9. Then I can see that the event module is now a part of that branch by checking the status -v of one of the event module files:

   [MP:~/projects/drupalcustomA]$ cvs st -v modules/event/INSTALL

   File: INSTALL Status: Up-to-date
      Working revision: 1.1.2.1 Tue Oct 5 06:46:30 2004
      Repository revision: 1.1.2.1
   /repos/drupal/modules/event/INSTALL,v
      Sticky Tag: drupalcustomA (branch: 1.1.2)
      Sticky Date: (none)
      Sticky Options: -ko
   Existing Tags:
      drupalcustomA (branch: 1.1.2)
      MODULE_EVENT_20041004 (revision: 1.1.1.1)
      module_event (branch: 1.1.1)

10. Later, if there are significant changes to the event module, I’ll update it using the same vendor tag, but a different release tag, just as it is demonstrated in the method above for drupal itself. My local drupal head will get the latest event module and I can get it in my customized
branches by updating them with the new release tag I make when importing the new event module.

11. I will import every contribution I wish to include in my local CVS repository this way, giving unique vendor/release tags to each. That way I can pick and choose which contributions are included in any of my customized branches of drupal by updating each branch with the release tags of only the contributions I want in that branch.

**Updating the vendor branch**

At some later point the drupal source code will have been updated and we’ll want to add the updated version to our repository. We do this by repeating the process described above, we get a fresh copy of the source from drupal.org, and import using the same vendor tag but change the release tag from 'HEAD20040110' to reflect the newer version:

```cvs
  cvs import -ko -m "Import CVS HEAD on Jan 11th 2004" sites/drupal drupal HEAD20040111
```

This updates the vendor branch, a single files revision history can now appear four different ways, depending on whether it has been modified by us, by the vendor (drupal.org), by both, or not at all.

If the file was modified only by us, our modified version remains the head revision:

```
[Main trunk] fileone.php *------------------ 1.2 +
  \                    +-----+
  \                                  +-----+
[Vendor Branch]                   +---------+
            +-----+ 1.1.1.1 +
                +---------+ 1.1.1.1.1 +
                      (tag:HEAD20040110)
```

If the file was modified only by the vendor, the new version becomes the HEAD revision:

```
[Main trunk] filetwo.php *
  \                    HEAD
  \                                  +-----+
[Vendor Branch]                   +---------+
                +-----+ 1.1.1.1 +
                      +---------+
                        (tag:HEAD20040110) (tag:HEAD20040111)
```

And if the file was modified by both us and the vendor:

```
[Main trunk] filethree.php *------------------ 1.2 +
  \                    +-----+
[Vendor Branch]                   +---------+
            +-----+ 1.1.1.1 +
                +---------+
                      (tag:HEAD20040110) (tag:HEAD20040111)
```
Our version of filethree.php remains the HEAD revision, but this is clearly not desirable since it doesn’t carry the latest changes. In fact, during our import of the latest source CVS would have warned us of conflicts between the two versions of filethree.php, we need to merge the changes to remove this conflict:

cvs checkout -jHEAD20040110 -jHEAD20040111 drupal

Examine the merged file to ensure the changes CVS made were sane and then ‘cvs commit’ the changes back to the main trunk. Leaving us with a new revision which becomes HEAD:

```
[Main trunk] filethree.php *----------+ 1.2 +-------+ 1.3 +
  \ +------+
[Vendor Branch] + 1.1.1.1 +-------+ 1.1.1.2 +
  \ +------+
(tag:HEAD20040110) (tag:HEAD20040111)
```

**Summary**

It should now be clear that using the CVS vendor tag to create a vendor branch in your own drupal module you can track changes to the drupal source code whilst also maintaining and developing your own customizations and new features for drupal. This example has been kept very simple for the purposes of explanation, but the basic process can be used to achieve many different things, some examples:

- Track a specific release of Drupal (e.g. 4.3, or 4.2), instead of the development (CVS HEAD) version.
- Maintain your customized sites with modules, themes, static pages, images etc all added to your CVS repository, whilst still tracking and importing updates to the drupal core.
- Branch your module to maintain several customized web sites off a single tracked branch of the drupal core.

Reading the following additional resources is highly recommended.

**Additional resources**

- Article by Nick Patavalis: The mechanics and a methodology for tracking 3rd party sources with CVS
- The section “Tracking third-party sources” in the CVS manual
- The section “Tracking third-party sources” in a book on CVS

**Sandbox maintenance rules**

1. Always document your changes.
2. Split different set of patches into different directories. It takes longer to find the set of files relating to one change if it is mixed in with 2 other patches.
3. Keep the documentation current. Try to keep some track of your reasoning too. If I read in a README that change X wasn’t a good idea after all it makes the reviewer wonder why.
4. Document the status of your patch. It is important to know if this is an early test, or considered stable and workable but the author of the patch.
5. All patches should be against the latest CVS version of Drupal, and include in the README when it was last synced.
6. Don’t use a sandbox for developing modules. There is a different directory structure for that.
7. If your patch is 4 lines long don’t bother to put it in a sandbox. Just mail it to the devel list and find out quicker if people like it or not. Small patches are quick to check and find out if work. Sandboxes should be for more extensive changes.
8. Try to maintain patches in the sandbox. They are so much easier to check than compete files. If you are using CVS then you can use diff (cvs -H diff)
9. Please make sure your script passes the code-style.pl script. It isn’t perfect, and sometimes a bit too strict, but it will ensure some level of compliance with the coding standards.

Commit messages--providing history and credit

As a module or theme maintainer or contributor, you may be making commits to the Drupal contributions CVS repository on a regular basis, both for your own code and changes contributed by others. Appropriate comments along with those commit messages are key to providing context that others need to understand the development of your code.

Give details

Provide at least a quick summary of what is contained in the commit.

Provide a link to the drupal.org issue, if available

If the change you’re committing relates to an issue on drupal.org, give the issue number, preceded by a pound sign (#). This convention is interpreted by the CVS module on drupal.org to provide a link back to the issue.

Give credit

If others have contributed to the change you’re committing, take the time to give them credit.

Example

A sample commit message:

Patch #46746 by Matt: fixed inconsistent encoding of path aliases.
Fixes broken URLs on profile pages.

For other examples, have a look at commit messages as available on [http://www.drupal.org/cvs](http://www.drupal.org/cvs)
Additional references

- CVS book
- CVS docs
- CVS FAQ
- CVS guide from TLDP
Drupal’s APIs

If you are interested in developing Drupal modules or hacking away at the Drupal core then this is the place to find details about all the functions and classes defined in Drupal.

We now use Doxygen to automatically generate documentation from the latest drupal sources. This allows us to ensure that documentation is up-to-date, and to simultaneously track multiple versions of the documentation.

API Documentation is available from api.drupal.org/ for:

- Drupal CVS HEAD
- Drupal 4.6.x
- Drupal 4.5.x

Please also read the Drupal Coding Standards page, which contains some guidelines for writing Doxygen comments.

Note: Drupaldocs.org is being phased out

Forms API

This outline is meant as a starting point for those who wish to further documentation of Drupal’s form API. Many of the basics have already been addressed in the Quickstart Guide which we may wish to expand into full forms doc.

I. Overview
II. Form creation
   A. Overview
   B. Building individual fields
      1. Example
      2. Declaring field names
         a) Nesting
         b) Tree attribute
      3. Declaring properties/attributes
         a) Link to list of all properties/attributes available
   C. Building form groups
      1. Overview
      2. Example
      3. Fieldset attribute
      4. Collapsible form groups
   D. Declaring markup
      1. Prefix/Suffix
         a) Example
      2. Markup attribute
a) Example
E. Weighting elements/groups
   1. Overview
   2. Example
F. Returning forms
   1. drupal_get_form()
      a) When to use drupal_get_form
      b) Example
   2. Returning form elements
      a) When to return form elements
      b) Example

III. Theming Forms
A. Overview
   1. When to use form markup
   2. When to use separate theming function
B. Example
C. form_render()
   1. Rendering individual elements
   2. Final render
D. Looping through the $form array
   1. Using element_children
E. Theming individual form elements
F. Theming form elements in a table
   1. Building the rows

IV. Validating forms
A. Overview
B. Example

V. Executing forms
A. Overview
B. Example
C. Converting from the POST/switch approach
D. confirm_form()
   1. Overview
   2. Example

VI. Overriding Forms â this section needs work. No code has been written for these features as yet AFAIK
A. Altering form elements
   1. Overview
   2. Example
B. Custom theming
   1. Overview
   2. Example
C. Custom validation
Forms API Quickstart Guide

For an introduction to how the forms API works, check out the
[Forms API Quickstart Guide](#)

Forms API reference

For a comprehensive listing of form elements and their associated attributes, please see the
[Forms API Reference](#)

Multipage forms with the Forms API

In most cases, web-based forms are one-page affairs: fill in all these fields, click "Submit", and get your result. Naturally, Drupal’s Forms API excels at making these types of forms. Occasionally, however, there is a need to make a multipage form: a form which stretches across multiple pages and isn’t truly "submitted" until the final part is completed. This document addresses an approach to these types of forms. We will assume the following requirements, which are probably more than you actually need, but contain useful tips for style and code structure.

- Your multipage form has four pages.
- Your multipage form isn’t finished until the "Submit" on page 4.
- Your multipage form is a node type, and exists in hook_form.
- Your multipage form hides certain elements on different parts.
- Your multipage form requires certain elements to be filled in.

Why can’t I make multipage forms with what I already know?

Before we begin, the most worrisome question is: why does this document need to exist? To answer that, you’ll have to understand the regular workflow of a form in the Drupal Forms API. At its simplest:

- Build
- Validate
- Submit
- Display
The "build" phase is what happens when you code your form - Drupal will take all your form elements and build them internally. Since this built form is often pre-filled with data (in the case of editing, or based on values from \$_POST), the next step is "validation", which determines if the values of the form are legitimate for the elements in question (ie., #required fields have values, select #type types have values which match an entry in its $options, etc.). Finally, there’s the optional "submit" which only happens if the form has been submitted, and the "display" which kicks in when the form is actually being visually rendered in the browser.

This is just dandy for your regular one-page forms.

But, when you consider multipage forms, there comes a time when you need two "build" phases: one for the previous part (that the user has just "submitted") so that it may be properly validated, and another build phase for the current part (that the user is about to view):

- Build the previous part for validation purposes.
- Validate the data received from the previous part.
- Submit (as before and above, again optional).
- Build the next part if there were no validation errors.
- Display the next part.

That’s where the Forms API #pre_render comes in. The #pre_render is an array of function names that you set during your normal "build" phase, and which will be called the split-second before the form is actually displayed to the user. This creates the following workflow, which gives us exactly what we need: the chance to modify our form for the current part after the validation phase for the previous part has finished:

- Build
- Validate
- Submit
- #pre_render
- Display

A Fully Working Example

Let’s show off some code. As per our feature requirements at the beginning of this document, we’re doing a custom multipage node type, so take a look at \_multipage_form_example_form in the fully functional multipage_form_example.module:

- Always build your form as if all your form elements were on a single page - this keeps your declarations all in one place, which is stylistically neater. The hiding of the elements on their specific parts will be handled in the #pre_render function.
- Don’t use #required in these declarations either.
- You can use fieldsets to group your form elements together, but you won’t be able to set the #type until the #pre_render function. If you set the fieldset in the initial declarations, you’ll have to choose between having the fieldset appear on every page of your multipage (which may be fine for some forms), or not passing the fieldset’s child values from part to part (which would generally be a bad idea, unless the fieldset happened to occur on the
Besides #pre_render, which we’ll get to shortly, the other important part of our multipage is the powerful hook_form_alter of Drupal’s Forms API. Look at multipage_form_example_form_alter: you’ll notice that we initialize or, in the case of an existing value, set the current part number that we’re on, and modify it as needed depending on if the “Back” or “Preview” has been clicked. Notice that we also modify the #validate and #submit hooks. Instead of just adding our custom functions, we merge with any existing values, which is especially important for custom node types (as we’ll need the nodeapi to kick in as normal once our multipage is finally submitted).

The last crucial bit needed for multipages is that #pre_render function we’ve been talking about - astute readers may have guessed that it is run twice: once in multipage_form_example_form_alter to set the validation requirements for our current/previous page, and then again by Drupal’s Forms API, which is when we know that we should increment our counter and show the next form part (as validation has been passed successfully for our previous part).

multipage_form_example_pre_render is the workhorse behind our demo module, and is responsible for setting visibility of elements, whether they are #required, what buttons are available for clicking, and any other visual indication that the user has progressed from page to page. Our logic here is based around using in_array to determine if our current part number is in an array of valid part numbers - which makes it quite simple to allow an element to appear on multiple pages without increasing the size of the code with equality checks.

More information about multipage forms is sprinkled throughout this demo code. Realize, however, that there are still hiccups along the way - ‘checkboxes’ and multiselect arrays (or any hidden form value that requires array values) are still difficult to pull off. These issues will probably be addressed in a future version of Drupal.

**Tips and Tricks**

Post your tips and tricks and sample module conversions, as well as any forms API-related questions you’ve had answered here as comments. They’ll be incorporated into the text.

**#tree and #parents**

All forms are a tree, for example:

```
    (root)
      |   /
      Foo / \   
      / \    
     Bar / \  
     /   \   
    Baz /    
```
In code, this is written:

$form['foo']['bar']['baz']$

As long as the #tree attribute is true at any point in the tree, the form element is aware that it is
in a tree, and traverses the tree towards the root (from baz, to bar, to foo). Along the way, the
names of the modules passed are stored in #parents. #parents is used to create the name/ID of
the form element itself.

So, if:

$form['foo']['#tree'] == TRUE
And
$form['foo']['bar']['#tree'] == TRUE
And
$form['foo']['bar']['baz']['#tree'] == TRUE

Then #parents for baz will be array('foo', 'bar', 'baz') and the name of the element in the HTML
will be $edit['foo']['bar']['baz']

If, on the other hand,

$form['foo']['bar']['baz']['#tree'] == FALSE

Then #parents will only be array('baz') and the name of the element in the HTML $edit['baz']

There are shortcuts to traversing the full tree each time. If you set #tree = TRUE at a closer point
to the root of the tree, as in:

$form['foo']['#tree'] = TRUE

and you have not specifically set #tree anywhere else, then it will cascade and make all of the
sub-elements’ #tree = TRUE. This is very useful because otherwise you would need to write
#tree = TRUE for each element in the tree.

A common use of #tree is fieldsets. Another example is the checkboxes element type where #tree
is set to TRUE internally before expanding to multiple checkbox elements.

The following is the section of code from which deals with #tree and #parents, taken from
_form_builder():
foreach (element_children($form) as $key) {
    // don't squash an existing tree value
    if (!isset($form[$key]['#tree'])) {
        $form[$key]['#tree'] = $form['#tree'];
    }

    // don't squash existing parents value
    if (!isset($form[$key]['#parents'])) {
        // Check to see if a tree of child elements is present. If so, continue down the tree if required.
        $form[$key]['#parents'] = $form[$key]['#tree'] && $form['#tree'] ? array_merge($form['#parents'], array($key)) : array($key);
    }
}

Note that the code is not the same to the explanation above for performance reasons: instead each element walking towards to the root as long as #tree is TRUE, we pass #parents down as long as #tree is TRUE.

You can set #parents manually, but the need for this is rare. More common is to read #parents to determine where in the form tree the current element is.

Adding a custom element type & expanding elements

the start and end date selection boxes proved to be a particular challenge in my upgrade of eventrepeat module. i handled it by creating my own form element type.

first, use hook_elements to declare the new type--‘eventrepeat_date’ is the name of the new type. also, if you want any default values, be sure to declare them either here or in the process function:

function eventrepeat_elements() {
    $type['eventrepeat_date'] = array('#input' => TRUE,);
    return $type;
}

here’s how the element type is invoked when in a $form array. notice in particular how the #process attribute is declared here as an array--the key is the name of the callback used to expand the type into multiple elements (you may not need a function such as this if your new type is simple), and the value is an array of arguments to be passed to the callback function. normally, #process would be declared as a default value in the element type declaration, but since i needed to pass $node and $edit as function arguments in this case, i declare it for each instance of the element type:

$form['end_controls']['end_date'] = array('#type' =>
    'eventrepeat_date', '#title' => t('Repeat end date'), '#process' =>
    array('_eventrepeat_form_date' => array($edit, $node,
    'eventrepeat_end')));

next, write the expand function. the first argument is always the element that is being passed for processing, and the other arguments are the optional arguments i added in my #process attribute. notice that i’m adding to $element and returning it:

function _eventrepeat_form_date($element, $edit = NULL, $node = NULL, $prefix = NULL) {
    //get current year, and drop next 10 years into an array

71
$date = getdate(time());
$curyear = $date['year'];
$years = array(0 => '--'.t('Select').'--');
while ($i < 10) {
    $years[$curyear + $i] = $curyear + $i;
    $i++;
}
//months array
$months = array('--'.t('Select').'--',
    t('January'),
    t('February'),
    t('March'),
    t('April'),
    t('May'),
    t('June'),
    t('July'),
    t('August'),
    t('September'),
    t('October'),
    t('November'),
    t('December'));
//days array
$days = array(0 => '--'.t('Select').'--');
for ($i = 1; $i <= 31; $i++) {
    $days[$i] = $i;
}
//compose the select boxes, and add the exception editor button if necessary
$element[$prefix.'month'] = array('#type' => 'select',
    '#default_value' => $edit ? $edit[$prefix.'month'] : ($node->{$prefix.'month'} ? $node->{$prefix.'month'} : 0), '#options' => $months);
$element[$prefix.'day'] = array('#type' => 'select', '#default_value' => $edit ? $edit[$prefix.'day'] : ($node->{$prefix.'day'} ? $node->{$prefix.'day'} : 0), '#options' => $days);
$element['comma'] = array('#type' => 'markup', '#value' => ', ');
$element[$prefix.'year'] = array('#type' => 'select', '#default_value' => $edit ? $edit[$prefix.'year'] : ($node->{$prefix.'year'} ? $node->{$prefix.'year'} : 0), '#options' => $years);
if ($prefix == 'eventrepeat_EXDATE_edit') {
    $element['exception_button'] = array('#type' => 'submit', '#value' => t('Add/Delete Exception'));
}
return $element;
then, a theming function to pretty it up. This is basically a theming function for a form item, but I put an inline container and $element['#children'] (which is the constructed select boxes, etc. from my expand function) into it. Naming convention for the theme function is theme_typename:

```php
function theme_eventrepeat_date($element) {
  return theme('form_element', $element['#title'], '<div class="container-inline">'. $element['#children']. '</div>', $element['#description'], $element['#id'], $element['#required'], $element['#error']);
}
```

### Adding and theming additional fields to a node form

Adding a form element to an existing node form is a matter of using hook_form_alter and checking for the appropriate form id.

Observe the following example from taxonomy.module:

```php
function taxonomy_form_alter($form_id, &$form) {
  if (isset($form['type']) && $form['type']['#value'] . '_node_form' == $form_id) {
    ...
    // Add your form array here, for example:
    $form['taxonomy']['tags'][$vocabulary->vid] = array('#type' => 'textfield', '#default_value' => $typed_string, '#maxlength' => 100, '#autocomplete_path' => 'taxonomy/autocomplete/'. $vocabulary->vid, '#required' => $vocabulary->required, '#title' => $vocabulary->name, '#description' => t('A comma-separated list of terms describing this content (Example: funny, bungie jumping, "Company, Inc.").'));
  }
}
```

The only thing in the form_alter hook should be things related to the form declaration itself. If you’d like to theme this in a particular way (for example, place the results in a table), set the form’s #theme attribute:

```php
$form['custom_form']['#theme'] = 'custom_form';
```

Then later, declare a function to handle the theming of this form. Here’s an example from system.module:

```php
function theme_system_user($form) {
  foreach (element_children($form) as $key) {
    $row = array();
    if (is_array($form[$key]['description'])) {
```
Creating an array of form elements

As someone who’s done a lot of work with the "old" form methods, the new Forms API is a lot of new information to chew on. Almost everything is different. In the spirit of practicality, here’s how I solved the riddle of "how to I make an array of form elements?" The answer is #tree.

In this case I want to create an array of select fields to assign one of a pre-defined set of values to a given piece of data. It’s meant to help assign CVS field headers (and the content in their columns) to node data rows. Another example might be a mass-categorization page for lots of posts.

Here’s the code before:

```php
<?php
$fields = array('pre', 'defined', 'values', 'to', 'assign');
foreach ($array_of_stuff as $i => $value) {
    $output .= form_select('', 'assign'[$i], $edit['assign'][$i], $fields));
}
?>
```

Here’s the code after:

```php
<?php
$form['assign'] = array('#tree' => 1);
$fields = array('pre', 'defined', 'values', 'to', 'assign');
foreach ($array_of_stuff as $i => $value) {
    $form['assign'][$i] = array(
        '#type' => 'select',
        '#title' => '',
        '#default_value' => $edit['assign'][$i],
        '#options' => $fields
    );
}
?>
```
I’m sure all this information is contained in the big docs about Forms API, but it can be a bit dense. I hope this helps people in getting to know the new API, which offers a lot more power once you learn it’s ins and outs.

Creating fieldsets outside forms with minimal code

Every now and again there comes a time when you want to use form groups (HTML fieldset elements) without having a real form, simply because they make good containers for content.

With the old form code, we could happily write something like the following:

```
<?php
$output .= form_group(t('My Fieldset'), $group);
?>
```

I dreaded the thought of creating a new $form, and all kinds of form API code just to create a fieldset, but luckily we can accomplish the same effect with only a minimal amount of fuss:

```
<?php
$output .= theme('fieldset', array('#title' => t('My Fieldset'), '#children' => $group));
?>
```

Creating multi-part forms

There’s no question that the new forms API will make life easier for Drupal developers. However, not everything about them is intuitive. Creating multi-part forms is one of them. I spent many, many hours trying to figure out the best way to do this. Granted, I’m not a hard-core coder like some others Drupalers out there, but I think even advanced coders will also have their share of frustration. And so to help you avoid the pain I went through, I offer this tip. Note that what follows assumes you are familiar with forms API.

It wasn’t until I studied the code in node.module that I was actually able to figure this out. Specifically, the three functions in that module that I learned from were node_admin_nodes(), node_multiple_delete_confirm(), and node_multiple_delete_confirm_submit(). Together, these functions allow users to select which nodes to delete (part 1 of the multi-part form) and then confirm the deletion (part 2 of the the form).

But rather than use that module as a basis for this discussion, I created a simple module that simulates those functions more compactly to avoid losing the forest through the tangles of code in node.module.

Anyway, here’s the boiled-down, multi-part form simulation module:

```
<?php
function formtest_menu() {

    // If we aren't caching, generate the initial menu
    if (!$may_cache) {
        $items[] = array('path' => 'formtest', 'title' => t('initial
```
# form()

  'callback' => 'formtest_page',
  'access' => TRUE,
  ];
}

return $items;
}

// Main function
function formtest_page() {
  // If user has already performed an operation, handle it.
  if ($_POST['op'] == 'Submit' || $_POST['edit']['confirm']) {
    return formtest_confirm_form();
  }

  // Main form
  // This is a parent element so #tree is set to TRUE
  $form['checkboxes'] = array(
    #tree' => TRUE,
  );
  // Checkbox #1 child element
  $form['checkboxes'][1]['checked'] = array(
    #type' => 'checkbox',
    '#title' => 'Checkbox #1',
    '#default_value' => 0,
  );
  $form['checkboxes'][1]['title'] = array(
    #type' => 'hidden',
    '#value' => 'Checkbox #1',
  );
  // Checkbox #2 child element
  $form['checkboxes'][2]['checked'] = array(
    #type' => 'checkbox',
    '#title' => 'Checkbox #2',
    '#default_value' => 0,
  );
  $form['checkboxes'][2]['title'] = array(
    #type' => 'hidden',
    '#value' => 'Checkbox #2',
  );
  $form['submit'] = array(
    #type' => 'submit',
    '#value' => 'Submit',
  );
  return drupal_get_form('simple_form', $form);
}

function formtest_confirm_form() {
  $edit = $_POST['edit'];
  // Here, we create a form element that will hold the $_POST data
// which contains an array representing the checkboxes the user checked.
// Notice we give the first dimension of this element the name "checkboxes".
// This is the same name as the parent form element in the formtest_page.
// Giving it the same name allows us to add children to the parent (with $_POST
// acting as our surrogate parent). We must do this so we can pass our data on
// to the formtest_confirm_submit().
$form['checkboxes'] = array('tree' => TRUE);
// Now we populate the element with our data.
foreach ($edit['checkboxes'] as $checkbox_id => $data) {
    if ($data['checked']) {
        $form['checkboxes'][$checkbox_id] = array(
            '#type' => 'hidden',
            '#value' => $checkbox_id,
        );
        $checked[] = check_plain($data['title']);
    }
}
$form['data'] = array(
    '#type' => 'markup',
    '#value' => theme('item_list', $checked),
);
$form['operation'] = array(
    '#type' => 'hidden',
    '#value' => 'delete',
);
$output = confirm_form('formtest_confirm', $form,
    t('delete these boxes?'),
    'formtest', t('This action cannot be undone.'),
    t('Delete'), t('Cancel'))
);
return $output;
}
function formtest_confirm_submit($form_id, $edit) {
    //Simulate deletion of our data
    _formtest_simulate_deletions($edit);
    //Return user back to main page.
    drupal_goto('formtest');
}
//This code is for simulation purposes only.
function _formtest_simulate_deletions($edit) {
    drupal_set_message('If this code was real, the checkbox(es) you deleted would not appear.');
Drupal Handbook

}
?>

25 Aug 2006

// Code to delete data here

You can actually install this chunk of code in your modules directory and see it work. At the
time of this writing, you’ll need cvs but by the time you read this, it will likely work with
version 4.7+ of Drupal.
The code basically speaks for itself. There are a few tricky catches in there. Lengthy comments
have been supplied where this is the case. The three functions to focus on are formtest_page(),
formtest_confirm_form(), formtest_confirm_submit(). The other functions aren’t important to
creating multi-part forms.
This code is extremely simple. A more realistic example would be more complex and require to
be very careful to run security checks on the $_POST data which ultimately ends up in our form.
Under normal circumstances, this is a no-no.
If anyone can offer suggestions on how to improve this code or simplify it, please share!
This tip courtesy of: Steve Dondley, Dondley Communications with special thanks to Earl Miles
(a.k.a. merlinofchaos) for guidance offered.

Creating submit buttons with images
Instead of the default and boring grey submit buttons, it’s possible to use clickable images to
submit forms. This has been possible with HTML for a long time.
Here’s how to do it with the new forms api:
1) Place the following code in your module:
/**
* Custom form element to do our nice images.
*/
function hook_elements() { // Change this line
$type[’imagebutton’] = array(’#input’ => TRUE, ’#button_type’ =>
’submit’, ’#form_submitted’ => TRUE, ’#name’ => ’op’);
return $type;
}
function theme_imagebutton($element) {
return ’<input type="image" class="form-’. $element[’#button_type’]
.’" name="’. $element[’#name’] .’" value="’.
check_plain($element[’#default_value’]) .’" ’.
drupal_attributes($element[’#attributes’]) . ’ src="’ .
$element[’#image’] . ’" alt="’ . $element[’#title’] . ’" title="’ .
$element[’#title’] . "\" />\n";
}

78


Be sure to "hook" in hook_elements() above with the name of your module.

2. Now use the following piece of code in your forms to create the submit button:

```php
$form['submit'] = array(
    '#type' => 'imagebutton',
    '#image' => '/submit.jpg', // provide the path to your image here
    '#title' => 'submit button',
    '#default_value' => 'submit',
);
```

Special thanks to Earl Miles, (aka merlinofchaos) for this tip.

**Easier debugging of forms code**

If you want to make your life easier, while debugging you change this section in form.inc:

```php
// Recurse through all child elements.
$count = 0;
foreach (element_children($form) as $key) {
    // Recurse through all child elements.
    $count = 0;
    foreach (element_children($form) as $key) {
        if (!is_array($form[$key])) {
            echo "the form has a wrong key $key";
            continue;
        }
    }
    if (!is_array($form[$key])) {
        echo "the form has a wrong key $key";
        continue;
    }
    if (!is_array($form[$key])) {
        echo "the form has a wrong key $key";
        continue;
    }
    if (!is_array($form[$key])) {
        echo "the form has a wrong key $key";
        continue;
    }
    if (!is_array($form[$key])) {
        echo "the form has a wrong key $key";
        continue;
    }
        }
```

this way, if you miss a # from a property name, you will get some immediate feedback.

**Getting a form element without a form**

I’m using AJAX to replace form elements within a form and I need to get a form element, such as a select box, without the surrounding form. Here’s how!

Build your $form like normal, then call this:

```php
<?php
print form_render(_form_builder('my_select', $form));
?>
```
Modifying checkboxes to display in multiple columns

This tip will show you how to modify the checkboxes element type so that checkboxes display in multiple columns instead of one single column. This is particularly useful for modules that would have users select from a very long list of choices. This tip will help you present the checkboxes in a more compact form, making your site more user friendly. **Disclaimer:** This document shares how I solved this particular problem and doesn’t claim to be anything more. There may be other ways of accomplishing the same thing. If you are a forms API guru and know an easier way, please share!

The first job is to override the expand_element_checkboxes() function in the form.inc. Here’s how it looks:

```php
<?php
function expand_checkboxes($element) {
    $value = is_array($element['#value']) ? $element['#value'] : array();
    $element['#type'] = 'checkbox';
    $element['#tree'] = TRUE;
    if (count($element['#options']) > 0) {
        if (!isset($element['#default_value']) || $element['#default_value'] == 0) {
            $element['#default_value'] = array();
        }
        foreach ($element['#options'] as $key => $choice) {
            if (!isset($element[$key])) {
                $element[$key] = array('#type' => 'checkbox', '#processed' => TRUE, '#title' => $choice, '#default_value' => in_array($key, $value), '#attributes' => $element['#attributes']);
            }
        }
    }
    return $element;
}
?>
```

To override it, the first step is to actually put the new function in our module. Here it is:

```php
<?php
function expand_checkbox_columns($element) {
    $value = is_array($element['#value']) ? $element['#value'] : array();
    $element['#type'] = 'checkboxes';
    $element['#tree'] = TRUE;
    if (count($element['#options']) > 0) {
        if (!isset($element['#default_value']) || $element['#default_value'] == 0) {
            $element['#default_value'] = array();
        }
        foreach ($element['#options'] as $key => $choice) {
```
$$\text{class} = ($\text{column} \% \text{element}[\#columns']) \&\& \text{column} ?
'checkbox-columns' : 'checkbox-columns-clear';$

if (!isset($element[$key])) {
    $element[$key] = array(\#type => 'checkbox', \#processed => TRUE, \#title => $choice, \#default_value => in_array($key, $value), \#attributes => $element[\#attributes'], \#prefix => '<div class="'. $class .'">', \#suffix => '</div>');
}
$column++;
}

return $element;
?>

The above function contains the logic that will create our multi-column list of checkboxes. Notice it is the same as the expand_checkboxes_function except it has a different name, "expand_checkbox_columns", and it has some code that puts some div tags in front of the checkboxes. The div tags are how we ultimately control which column the checkboxes appear. Also notice this line:

```

<?php $element[\#type'] = 'checkboxes';
?>
```

That line is in there to suppress a bug in the current version of forms api. In the interests of brevity, I’ll skip over exactly why that’s there. But basically, all it does is fool forms.inc that all it’s dealing with is your standard checkboxes element type.

The next step is to let Drupal know about the new "expand_checkbox_columns" function you just created. You do that by putting a hook_elements function in your module like so:

```

<?php
function pol_tracker_elements() {
    $type['checkbox_columns'] = array(\#input => TRUE, \#process => array('expand_checkbox_columns' => array()), \#tree => TRUE);
    return $type;
}
?>
```

Simple enough. But see the other tip about creating your own elements for more details on what exactly this does.

Next, of course, we need some code that actually puts our new element function to use. Here’s an example:
Couple of things to point out here. First the '#type' is set to 'checkbox_columns'. This is how we ensure our new expand_checkbox_columns function gets called. Second, notice the '#columns' property. This is how you can change the number of columns displayed. If you look at the new expand_checkbox_columns() function we created above, you'll see that it makes use of the '#columns' value.

Our last step is to add some CSS to our style sheet so this will work. Here's how:

```
.checkbox-columns .form-item {
    width: 15em;
    margin-right: 10px;
    float: left;
    display: inline;
}
.checkbox-columns-clear .form-item {
    width: 15em;
    margin-right: 10px;
    clear: left;
    float: left;
    display: inline;
}
```

And that should do it. Notice that you can control the width of the columns by changing the width property (set to 15em). You will want to change this according to the length of your longest checkbox selection to give your list a nice neat appearance.

**Module dependency checker**

Do you have a module that depends on other modules? Have you ever wished there was a way to ensure they were installed before your module is enabled?? Well, I've figured out a fairly elegant way for contrib modules to do simple dependency checking, via hook_form_alter. Here's how it's done:
1. Add this section of code to your module’s hook_form_alter:

```php
<?php
/* this is a check for module dependencies. the only way we
   can ensure this check happening when the module is initially
   enabled is to insert the check for when the form is initially
   built, which will also be caught when the admin/module page is
   reloadd upon submission. this means we never want to call this
   function when the form has been submitted, so make sure there’s
   no $_POST. */
if ($form_id == 'system_modules' && !$_POST) {
    modulename_system_module_validate($form);
}
?>
```

2. Add the dependency check function to the module:

```php
<?php
/**
 * Validates module dependencies for the module.
 *
 * @param $form The form array passed from hook_form_alter.
 *
 * Set the $module variable to a string which is the name of the
 * module, minus
 * the .module extension. Set $dependencies to an array of module
 * names which
 * the module is dependent on--each element is a string which is the
 * module name
 * minus the .module extension. Note that this will not check for
 * any dependencies
 * for the modules this module depends on--only those that are
 * explicitly listed in
 * the $dependencies array.
 */
function modulename_system_module_validate(&$form) {
    $module = 'modulename';
    $dependencies = array('dependentmodule1', 'dependentmodule2', 'etc');
    foreach ($dependencies as $dependency) {
        if (!in_array($dependency, $form['status']['#default_value'])) {
            $missing_dependency = TRUE;
            $missing_dependency_list[] = $dependency;
        }
    }
    if (in_array($module, $form['status']['#default_value']) && isset($missing_dependency)) {
        db_query("UPDATE {system} SET status = 0 WHERE type = 'module'
```
AND name = '%s', $module);
$key = array_search($module, $form['status'][#default_value]);
unset($form['status'][#default_value][$key]);
drupal_set_message(t('The module %module was deactivated--it requires the following disabled/non-existant modules to function properly: %dependencies', array('%module' => $module, '%dependencies' => implode(', ', $missing_dependency_list))), 'error');
}
}?></div>

I think the code comments explain the feature pretty well. The main trick is that you can’t check the modules dependencies upon it’s initial form submission in admin/modules because, well, it’s not enabled yet... :): fortunately, there’s a drupal_goto which sends you right back to the admin/modules page–so we can catch it there and manually disable the module in the database.

This works both when the module is initially enabled, and if somebody happens to try and disable any dependent modules later. To keep it simple, I just warn the user and disable the module. This solution is not foolproof (with layered dependencies it’s possible that a module might pass when it shouldn’t–but those are almost non-existent in Drupal ATM AFAIK), but it will work for probably 98% of the use cases--and providing this protection is as hard as dropping in the code above and setting your dependencies.

**The #ref property**

#ref is a form property that creates a reference that can then be accessed at any point during the form processing cycle.

This is a very useful property if for example you wish to make changes to a property on _validate, and see those changes on _submit.

Here is an example from [http://drupal.org/node/51063](http://drupal.org/node/51063)

**Background info**

On OG’s invite form is a textarea called ‘mails’ where people may enter a list of e-mail addresses or usernames, either comma- or newline-separated:

```php
<?php
function og_invite_form() {
    $form['mails'] = array('#type' => 'textarea', '#title' => t('Email addresses or usernames'), '#description' => t('Enter up to %max email addresses or usernames. Separate multiple addresses by commas or new lines. Each will receive an invitation message from you.', array('%max' => $max)));
}?>```
During the _validate process, the textarea input is parsed, a series of checks take place to ensure valid e-mail addresses exist:

```php
<?php
function og_invite_validate($form_id, $form_values, $form) {
    $mails = $form_values['mails'];
    $mails = str_replace("\n", ",", $mails);
    $emails = explode(’,’, $mails);
    if (count($emails) > $max) {
        form_set_error('mails', t("You may not specify more than %max email addresses or usernames."), array('max' => $max));
    } else {
        $valid_emails = array();
        $bad = array();
        foreach ($emails as $email) {
            $email = trim($email);
            if (empty($email)) {
                continue;
            }
            if (valid_email_address($email)) {
                $valid_emails[] = $email;
            } else {
                $account = user_load(array('name' => check_plain($email)));
                if ($account->mail) {
                    $valid_emails[] = $account->mail;
                } else {
                    $bad[] = $email;
                }
            }
        }
        if (count($bad)) {
            form_set_error('mails', t("invalid email address or username: "). implode(' ', $bad));
        }
    }
}
?>
```

At the end of this process, we have a nice array called $valid_emails, which contains all of our valid e-mail addresses. However, we normally can’t change form values during the _validate stage.
The problem

This means that when we get to our _submit function, we have to do the vast majority of this processing again:

1. Replacing the newlines with commas
2. Exploding the string into an array
3. Looping through each value
4. Trimming it and seeing if it’s empty
5. Checking whether it’s an e-mail or a username

What a waste, when $valid_emails is right there with all the data we need in it!

The solution

In steps the ‘#ref’ property. With this, we can store a reference to the $valid_emails array, and then look at it when we’re in _submit!

So first, add a property to the form itself to hold the valid e-mails:

```php
<?php
og_invite_form:
   $form['valid_emails'] = array('type' => 'value', 'value' => array());
?>
```

Next, in the validate function, after all of the processing is complete, store the value of the $valid_emails array in the ‘valid_emails’ element’s #ref property:

```php
<?php
og_form_validate:
   // ... all of that code from above ...
   if (count($bad)) {
      form_set_error('mails', t('invalid email address or username: ').
         implode(' ', $bad));
   } else {
      // Store valid e-mails so we don’t have to go through that
      // looping again on submit
      $form['valid_emails']['#ref'] = $valid_emails;
   }
?>
```

Finally, in the submit function, retrieve the value:

```php
<?php
function og_invite_submit() {
```
$emails = $form_values['valid_emails'];
?>

And now we can just loop through them knowing that they’ve already been verified by the validation function.

**Writing forms in tables**

When you want your form elements to be presented in tables (as, more generally, whenever you want to customize the presentation of your form), you follow a two-step process. First, you generate the appropriate form elements. Then, when it comes time to present the form, you pass it through a theme.

Good examples of how to do this are in the issue.inc file in the Project module.

The simplest is used to present the issue filtering interface, seen at the top of the page [http://drupal.org/project/issues](http://drupal.org/project/issues). Note that the form elements here are enclosed in a table, so that they appear beside each other in a row.

To accomplish this, the elements are first generated in the function `project_issue_query_result()`:

```php
<?php
// Make quick search form:
$form['projects'] = array(
    '#type' => 'select',
    '#title' => t('Project'),
    '#default_value' => $query->projects,
    '#options' => $projects,
);
// etc.
?>
```

A `drupal_get_form()` call generates the form output:

```php
<?php
$group .= drupal_get_form('project_issue_query_result_quick_search', $form);
?>
```

Normally this would return a standard-formatted form, but in this case the module includes a theme function to customize the output of this form: `theme_project_issue_query_result_quick_search()`.

The theme function simply takes the form elements and generates an array of cells, then outputs the result as a table:
Look at other forms in issue.inc for more complex examples, e.g.,
project_issue_admin_states_page(), which uses nested arrays to generate multi-line
tables with one row per record.

### Upgrading to forms API

**NOTE:** Further details about how the form API works can also be found in the API reference on
the following pages:

- [Flowcharts](#)
- [Forms API Quickstart Guide](#) (links to drupaldocs.org)
- [Forms API Reference](#) (links to drupaldocs.org)

Also see the [Form Updater module](#) which can help you get a head start in converting your
modules.

Before embarking on the journey to update a module to the new forms API, it is helpful to
understand more about the conversion process in general, and the benefits it provides. First, it
should be noted that for many module maintainers, converting their module to use the new
forms API will be the most difficult and time-consuming conversion yet. "Why is this so much
harder to implement than the old approach?" This thought will probably come up more than
once if the module being converted has any degree of complexity regarding forms. And the
answer to the question is twofold:

1. The new API is more complex.
2. The approach is fundamentally different.

It’s much easier to get a handle on reason #1 if reason #2 is understood and accepted. :)
Taking the time to learn and implement the API has the following major advantages:

1. Security
   - It filters the variables passed through _POST, so that only variables that were on the form will be accepted as input
   - It validates select, checkboxes and radios form elements so that only listed options can be checked. Example: the filter_form now validates itself (huge security benefit)
   - It allows you to add additional validation through the form_validate hook, which means that it’s a lot simpler to write validation code, and to reuse said validation code. Example: you only need to write the valid_integer function once, and then you can specify whichever form elements need to conform to that.

2. Extensibility
   - Forms can be reordered
   - Additional form fields can be added
   - The default way a form is validated/executed can be overridden
   - Additional form element types can be added
   - Behavior of default element types can be overridden

3. Themability
   - Themers can override the default layout of any form or a part of it in their theme
   - Themers can individually theme a single element, a group of elements, or all elements of the same type. Example: one can write a module which would theme multiple selects with checkboxes.

These advantages are available in any form, including those generated by Drupal core!

In addition, this functionality is a major and necessary step towards CCK.

**Example module conversion: Project Module**

**Overview**

This document provides step-by-step instructions on how various parts of the Project Module were translated to the Drupal 4.7 Forms API. It covers a wide range of situations module developers are likely to encounter, including:

- Converting a hook_settings implementation
- Converting various hook_form implementations
- Breaking form markup out into theme functions
- Working with fieldsets and form inputs
- Implementing a 2-page "wizard" style form

The functions are organized by approximate order of difficulty, with the easiest at the beginning and the most difficult towards the end. While in most cases a reader may skip directly to sections which interest him/her, it is highly recommended that the first function translation be read in its entirety, in order to provide a detailed look at how the process of form conversion works.
If you wish to follow along, you can obtain a copy of the project module from November 5, 2005 (just before the new form changes went through) by performing the command:

```
$ cvs -d:pserver:anonymous:anonymous@cvs.drupal.org:/cvs/drupal-contrib
   checkout -D "05 Nov 2005" -d "project" -P
   "contributions/modules/project"
```

**Conversion Tips**

**Form Updater**

Download the Form Updater module and install it in the usual fashion (download it, extract it, copy the folder to your modules directory, and enable it through administer >> modules). Then click on the new form updater link in the menu to bring up the form updater interface.

**Form Updater module**

Then simply copy and paste the contents of .module and .inc files, and it will automatically attempt to find legacy form function calls and recommend Drupal 4.7 equivalents:
Form Updater recommendations

While this module is not a complete solution to forms API conversion (it does not catch everything, and human intervention is still required in order to place the code in the correct place), it is nevertheless an essential tool.

Common legacy form errors

If you try and use a contributed module in Drupal HEAD (Drupal 4.7) you may be confronted with errors about any of the following functions:

- _form_get_error
- form
- form_checkbox
- form_checkboxes
- form_group
- form_radio
- form_radios
- form_textarea
- form_textfield
This is an indication that this module has yet to be converted to the Drupal 4.7 Forms API. Lines which contain calls to any of these functions inside a module will need to be translated to the new API.

When performing the conversion, once you have identified a hook or function where an error is occurring, it is often helpful to comment out the entire inside of the function, and then uncomment lines as you get them converted, reloading the offending page to view your changes as you go forward.

**project_settings: Converting a hook_settings implementation (Easy)**

**Overview**

The first form to investigate is the project module’s settings form. This is a very straight-forward form conversion which only requires taking existing form_* function calls and converting them into arrays.

**Project settings form**
Original function

The following is the original project_settings function which generates this form:

```php
function project_settings() {
  $project_directory = file_create_path(variable_get('project_directory_issues', 'issues'));
  if (!file_check_directory($project_directory)) {
    $error['project_directory_issues'] = theme('error', t('Directory does not exist, or is not writable.'));
  }
  $versions = array(-1 => t('all')) + project_releases_list();
  $output = form_textfield(t('Release directory'), 'project_release_directory', variable_get('project_release_directory'),
```
'Leave this blank if project maintainers are to create their own release packages. This is useful if releases are generated by an external tool.'));
** $output .= form_radios(t('Unmoderate projects with releases'),
  'project_release_unmoderate',
  variable_get('project_release_unmoderate', 0), array('Disabled',
  'Enabled'));
** $output .= form_checkbox(t('Browse projects by releases'),
  'project_browse_releases', 1, variable_get('project_browse_releases',
  1), t('Checking this box will cause the project browsing page to have version subtabs.'));
** $output .= form_radios(t('Default release overview'),
  'project_release_overview', variable_get('project_release_overview',
  -1), $versions, t('Default release version to list on the overview page'));
** $output .= form_textfield(t('Issue directory'),
  'project_directory_issues', variable_get('project_directory_issues',
  'issues'), 30, 255, t("Subdirectory in the directory '%dir' where attachment to issues will be stored.", array('%dir' =>
  variable_get('file_directory_path', 'files') .'/')));
  if (module_exist('mailhandler')) {
    // TODO: move this stuff to mailhandler.module?
    $items = array(t(' <none>'));
    $result = db_query('SELECT mail FROM {mailhandler} ORDER BY mail');
    while ($mail = db_result($result, $i++)) {
      $items[$mail] = $mail;
    }
  }
** $output .= form_select(t('Reply-to address on e-mail notifications'),
  'project_reply_to', variable_get('project_reply_to',
  ''), $items);
  }
** return $output;
}

Note that lines prefixed with ** indicate a line that will cause problems with Drupal 4.7.

Form conversion

Begin by navigating to administer >> settings >> project. You will receive an error:

Fatal error: Call to undefined function: form_textfield() in \modules\project\project.module on line 119

The error tells the name of the file from which the error is originating: project.module. Copy and paste the contents of this file into the Form Updater module, and it will identify six calls to the old forms API (out of seven lines identified as problematic). Here is the first call as an
example:

form_textfield(t('Release directory'), 'project_release_directory', variable_get('project_release_directory', ''), 50, 255, t('Leave this blank if project maintainers are to create their own release packages. This is useful if releases are generated by an external tool.'));

This creates a textarea with:

- a title of Release Directory
- an internal name of project_release_directory
- a default value containing either the contents of the project_release_directory variable in the Drupal variable table, or an empty string if that does not exist
- a size (or length) of 50 characters
- a maximum length of 255 characters
- a description of "Leave this blank if project maintainers are to create their own release packages. This is useful if releases are generated by an external tool."

In the new API, this translates to:

```php
$form['project_release_directory'] = array(
    '#type' => 'textfield',
    '#title' => t('Release directory'),
    '#default_value' => variable_get('project_release_directory', ''),
    '#size' => 50,
    '#maxlength' => 255,
    '#description' => t('Leave this blank if project maintainers are to create their own release packages. This is useful if releases are generated by an external tool.'),
);
```

Notice that the element name is now moved to the name of the array index in $form (in this case, project_release_directory). Also note that each array index is labeled specifically what it means. #title => t('Release Directory') is the form element’s title (or text displayed above it). #default_value is its default value, and so on. You are no longer required to memorize what order function arguments go in, or to refer to the API reference to determine this. Attributes can be placed in any order (although it is customary to keep them in the same order throughout), or even removed altogether if they are not required.

Replace the form_textfield call with the code generated by Form Updater. Repeat for each element that Form Updater finds.

**hook_settings changes in 4.7**

Save project.module at this point and reload the settings page.
Project settings page after replacing form values

While there is no longer an error displayed, something is clearly amiss. Where did the form go? Check the very last highlighted line for the answer:

```
return $output;
```

In Drupal 4.6, `hook_settings` returned an HTML string containing the form output. In Drupal 4.7, `hook_settings` (as well as `hook_form`) return the actual `$form` array instead. Change this line to:

```
return $form;
```

Upon reloading the page, you should see the full form as expected.

**Default values**

One last minor thing is that there are a series of default values (determined by the `system_elements` function) assigned to form elements. These are redundant if declared again in your form. In this case, the only setting that applies is the `#return_value => 1` in the checkbox field.

**Converted function**

The following is the completed code after translating project_settings:

```php
function project_settings() {
    $project_directory = file_create_path(variable_get('project_directory_issues', 'issues'));
    if (!file_check_directory($project_directory)) {
        $error['project_directory_issues'] = theme('error', t('Directory does not exist, or is not writable.'));
    }
```
$versions = array(-1 => t('all')) + project_releases_list();
$form['project_release_directory'] = array(
    '#type' => 'textfield',
    '#title' => t('Release directory'),
    '#default_value' => variable_get('project_release_directory', ''),
    '#size' => 50,
    '#maxlength' => 255,
    '#description' => t('Leave this blank if project maintainers are to
create their own release packages. This is useful if releases are
generated by an external tool.'),
);
$form['project_release_unmoderate'] = array(
    '#type' => 'radios',
    '#title' => t('Unmoderate projects with releases'),
    '#default_value' => variable_get('project_release_unmoderate', 0),
    '#options' => array('Disabled', 'Enabled'),
);
$form['project_browse_releases'] = array(
    '#type' => 'checkbox',
    '#title' => t('Browse projects by releases'),
    '#default_value' => variable_get('project_browse_releases', 1),
    '#description' => t('Checking this box will cause the project
browsing page to have version subtabs.'),
);
$form['project_release_overview'] = array(
    '#type' => 'radios',
    '#title' => t('Default release overview'),
    '#default_value' => variable_get('project_release_overview', -1),
    '#options' => $versions,
    '#description' => t('Default release version to list on the
overview page'),
);
$form['project_directory_issues'] = array(
    '#type' => 'textfield',
    '#title' => t('Issue directory'),
    '#default_value' => variable_get('project_directory_issues', issues),
    '#size' => 30,
    '#maxlength' => 255,
    '#description' => t("Subdirectory in the directory '%dir' where
attachment to issues will be stored.", array('%dir' =>
variable_get('file_directory_path', 'files') .'/')),
);
if (module_exist('mailhandler')) {
    // TODO: move this stuff to mailhandler.module ?
    $items = array(t('<none>'));
    $result = db_query('SELECT mail FROM {mailhandler} ORDER BY mail');
while ($mail = db_result($result, $i++)) {
    $items[$mail] = $mail;
}
$form['project_reply_to'] = array(
    '#type' => 'select',
    '#title' => t('Reply-to address on e-mail notifications'),
    '#default_value' => variable_get('project_reply_to', ''),
    '#options' => $items,
);
return $form;
}

Core module before and after examples

The following examples are taken directly from Drupal core, and illustrate the difference between the previous forms approach and the new forms API. The ‘after’ examples are notated to further describe some of the salient points of the API. They are arranged in order of difficulty--it’s recommended that you go through them in order.

Please note that editorial comments, which call attention to certain aspects of the examples, begin with ###

Standard example: Path Form

This example illustrates the creation of a simple form with a few fields and a submit button.

Before

```php
<?php
function path_form($edit = '') {
    $form .= form_textfield(t('Existing system path'), 'src', $edit['src'], 50, 64, t('Specify the existing path you wish to alias. For example: node/28, forum/1, taxonomy/term/1+2.'));
    $form .= form_textfield(t('New path alias'), 'dst', $edit['dst'], 50, 64, t('Specify an alternative path by which this data can be accessed. For example, type "about" when writing an about page. Use a relative path and don’t add a trailing slash or the URL alias won’t work.'));
    if ($edit['pid']) {
        $form .= form_hidden('pid', $edit['pid']);
        $form .= form_submit(t('Update alias'));
    } else {
        $form .= form_submit(t('Create new alias'));
    }
    return $form
}
?>
```
After

```php
function path_form($edit = '') {
    ### Notice that name of the form field is declared as $form['src'], for example.
    ### The type of element is declared by using '#type'. If a form element needs a
    ### pre-filled value, use '#default_value'
    $form['src'] = array(
        '#type' => 'textfield',
        '#title' => t('Existing system path'),
        '#default_value' => $edit['src'],
        '#size' => 60,
        '#maxlength' => 64,
        '#description' => t('Specify the existing path you wish to alias. For example: node/28, forum/1, taxonomy/term/1+2.'),
    );
    $form['dst'] = array(
        '#type' => 'textfield',
        '#default_value' => $edit['dst'],
        '#size' => 60,
        '#maxlength' => 64,
        '#description' => t('Specify an alternative path by which this data can be accessed. For example, type "about" when writing an about page. Use a relative path and don\'t add a trailing slash or the URL alias won\'t work.'),
    );
    if ($edit['pid']) {
        ### Note the declaration of the types. Also, all values that are
        ### not subject to user input
        ### use the '#value' attribute
        $form['pid'] = array('#type' => 'hidden', '#value' => $edit['pid']);
        $form['submit'] = array('#type' => 'submit', '#value' => t('Update alias'));
    } else {
        $form['submit'] = array('#type' => 'submit', '#value' => t('Create new alias'));
    }
    ### This is the master function that coordinates building, validating, executing, and displaying
    ### the constructed form. The first arg is the $form_id of the form, which by convention
    ### is usually the name of the function in which the form is created. Second arg is the
}
```
Fieldsets and advanced fields: system_view_general

This rather lengthy example introduces how fieldsets are handled, and also shows some additional field types.

Before

<?php
function system_view_general() {
    global $conf;
    // General settings:
    $group = form_textfield(t('Name'), 'site_name',
        variable_get('site_name', 'drupal'), 70, 70, t('The name of this website.'));
    $group .= form_textfield(t('E-mail address'), 'site_mail',
        variable_get('site_mail', ini_get('sendmail_from')), 70, 128, t('A valid e-mail address for this website, used by the auto-mailer during registration, new password requests, notifications, etc.'));
    $group .= form_textfield(t('Slogan'), 'site_slogan',
        variable_get('site_slogan', ''), 70, 128, t('The slogan of this website. Some themes display a slogan when available.'));
    $group .= form_textarea(t('Mission'), 'site_mission',
        variable_get('site_mission', ''), 70, 5, t('Your site’s mission statement or focus.'));
    $group .= form_textarea(t('Footer message'), 'site_footer',
        variable_get('site_footer', ''), 70, 5, t('This text will be displayed at the bottom of each page. Useful for adding a copyright notice to your pages.'));
    $group .= form_textfield(t('Anonymous user'), 'anonymous',
        variable_get('anonymous', 'Anonymous'), 70, 70, t('The name used to indicate anonymous users.'));
    $group .= form_textfield(t('Default front page'), 'site_frontpage',
        variable_get('site_frontpage', 'node'), 70, 70, t('The home page displays content from this relative URL. If you are not using clean URLs, specify the part after "?q=". If unsure, specify "node."'));
    // We check for clean URL support using an image on the client side.
    ### NOTE: Description shortened because of issue with formatting
    $group .= form_radios(t('Clean URLs'), 'clean_url',
        variable_get('clean_url', 0), array(t('Disabled'), t('Enabled')), t('This option makes Drupal emit clean URLs ...'));
    variable_set('clean_url_ok', 0);
    global $base_url;
}
// We will use a random URL so there is no way a proxy or a browser could cache the "no such image" answer.

```php
$period = drupal_map_assoc(array(3600, 10800, 21600, 32400, 43200, 86400, 172800, 259200, 604800, 1209600, 2419200), 'format_interval');

$period['1000000000'] = t('Never');
```

```php
$group = form_textfield(t('Default 403 (access denied) page'), 'site_403', variable_get('site_403', ''), 70, 70, t('This page is displayed when the requested document is denied to the current user. If you are not using clean URLs, specify the part after "?q=". If unsure, specify nothing.'));
```

```php
$group = form_textfield(t('Default 404 (not found) page'), 'site_404', variable_get('site_404', ''), 70, 70, t('This page is displayed when no other content matches the requested document. If you are not using clean URLs, specify the part after "?q=". If unsure, specify nothing.'));
```

```php
$group = form_select(t('Error reporting'), 'error_level', variable_get('error_level', 1), array(t('Write errors to the log and to the screen'), t('Where Drupal, PHP and SQL errors are logged. On a production server it is recommended that errors are only written to the error log. On a test server it can be helpful to write logs to the screen.')), t('Write errors to the log'));
```

```php
$group = form_select(t('Discard log entries older than'), 'watchdog_clear', variable_get('watchdog_clear', 604800), $period, t('The time log entries should be kept. Older entries will be automatically discarded. Requires crontab.'));
```

```php
$output = form_group(t('Error handling'), $group);
```

```php
// Caching:
$group = form_radios(t('Cache support'), 'cache', variable_get('cache', 0), array(t('Disabled'), t('Enabled')), t('Enable or disable the caching of rendered pages. When caching is enabled, Drupal will flush the cache when required to make sure updates take effect immediately. Check the <a href="%documentation">cache documentation</a> for information on Drupal\'s cache system.\n', array('%documentation' => url('admin/help/system#cache', NULL, NULL, 'cache'))));
```

```php
$output = form_group(t('Cache settings'), $group);
```

```php
// File system:
$directory_path = variable_get('file_directory_path', 'files');
file_check_directory($directory_path, FILE_CREATE_DIRECTORY, 'file_directory_path');
$directory_temp = variable_get('file_directory_temp', FILE_DIRECTORY_TEMP);
file_check_directory($directory_temp, FILECREATE_DIRECTORY, 'file_directory_temp');
```
$group = form_textfield(t('File system path'), 'file_directory_path', $directory_path, 70, 255, t('A file system path where the files will be stored. This directory has to exist and be writable by Drupal. If the download method is set to public this directory has to be relative to Drupal installation directory, and be accessible over the web. When download method is set to private this directory should not be accessible over the web. Changing this location after the site has been in use will cause problems so only change this setting on an existing site if you know what you are doing.'));
$group = form_textfield(t('Temporary directory'), 'file_directory_temp', $directory_temp, 70, 255, t('Location where uploaded files will be kept during previews. Relative paths will be resolved relative to the file system path.'));
$group = form_radios(t('Download method'), 'file_downloads', variable_get('file_downloads', FILE_DOWNLOADS_PUBLIC), array(FILE_DOWNLOADS_PUBLIC => t('Public - files are available using http directly.'), FILE_DOWNLOADS_PRIVATE => t('Private - files are transferred by Drupal.')), t('If you want any sort of access control on the downloading of files, this needs to be set to <em>private</em>. You can change this at any time, however all download URLs will change and there may be unexpected problems so it is not recommended.'));
$output = form_group(t('File system settings'), $group);
// Image handling:
$group = ''; $toolkits_available = image_get_available_toolkits(); if (count($toolkits_available) > 1) {
    $group = form_radios(t('Select an image processing toolkit'), 'image_toolkit', variable_get('image_toolkit', image_get_toolkit()), $toolkits_available);
} $group = image_toolkit_invoke('settings'); if ($group) {
    $output = form_group(t('Image handling'), $group);
}
// Date settings:
$zones = _system_zonelist();
// Date settings: possible date formats
$dateshort = array('Y-m-d H:i', 'm/d/Y - H:i', 'd/m/Y - H:i', 'Y/m/d - H:i',
    'm/d/Y - g:ia', 'd/m/Y - g:ia', 'Y/m/d - g:ia', 'M j Y - H:i', 'j M Y - H:i', 'Y M j - H:i',
    'M j Y - g:ia', 'j M Y - g:ia', 'Y M j - g:ia');
$datemedium = array('D, Y-m-d H:i', 'D, m/d/Y - H:i', 'D, d/m/Y - H:i', 'D, m/d/Y - g:ia',
    'D, m/d/Y - g:ia', 'D, m/d/Y - g:ia', 'D, Y/m/d - g:ia',
    'Y, M j - H:i', 'Y, M j - g:ia', 'Y, M j - g:ia', 'Y, M j - g:ia');
'F j, Y - g:ia', 'j F, Y - g:ia', 'Y, F j - g:ia');
' l, F j, Y - g:ia', 'l, j F, Y - g:ia', 'l, Y, F j - g:ia');

// Date settings: construct choices for user
foreach ($dateshort as $f) {
    $dateshortchoices[$f] = format_date(time(), 'custom', $f);
}
foreach ($datemedium as $f) {
    $datemediumchoices[$f] = format_date(time(), 'custom', $f);
}
foreach ($datelong as $f) {
    $datelongchoices[$f] = format_date(time(), 'custom', $f);
}
$group = form_select(t('Default time zone'), 'date_default_timezone',
variable_get('date_default_timezone', 0), $zones, t('Select the default site time zone.'));
$group .= form_radios(t('Configurable time zones'), 'configurable_timezones',
variable_get('configurable_timezones', 1), array(t('Disabled'), t('Enabled')), t('Enable or disable user-configurable time zones. When enabled, users can set their own time zone and dates will be updated accordingly.'));
$group .= form_select(t('Short date format'), 'date_format_short',
variable_get('date_format_short', $dateshort[0]), $dateshortchoices,
 t('The short format of date display.'));
$group .= form_select(t('Medium date format'), 'date_format_medium',
variable_get('date_format_medium', $datemedium[0]), $datemediumchoices,
 t('The medium sized date display.'));
$group .= form_select(t('Long date format'), 'date_format_long',
variable_get('date_format_long', $datelong[0]), $datelongchoices,
 t('Longer date format used for detailed display.'));
$group .= form_select(t('First day of week'), 'date_first_day',
variable_get('date_first_day', 0), array(0 => t('Sunday'), 1 =>
t('Monday'), 2 => t('Tuesday'), 3 => t('Wednesday'), 4 =>
t('Thursday'), 5 => t('Friday'), 6 => t('Saturday')), t('The first day of the week for calendar views.'));
$output .= form_group(t('Date settings'), $group);
return $output;
}?

After

<?php
function system_view_general() {
    // General settings:
    ### Notice the type declaration as a fieldset. '#title' in this case
will be the legend
### for the fieldset. Also note the two properties which make it a
collapsed fieldset
### by default the collapsing attributes are FALSE, so they need to
be declared if a
### collapsed fieldset is desired
$form[‘general’] = array(
 ’#type’ => ’fieldset’,
 ’#title’ => t(‘General settings’),
 ’#collapsible’ => TRUE,
 ’#collapsed’ => TRUE,
);
### A form element is placed under a fieldset by adding it under the
### array. Also note that size and maxlength attributes aren’t
declared here. Many attributes
### have system defaults which are used if no value is explicitly
declared
$form[‘general’][‘site_name’] = array(
 ’#type’ => ’textfield’,
 ’#title’ => t(‘Name’),
 ’#default_value’ => variable_get(‘site_name’, ‘drupal’),
 ’#description’ => t(‘The name of this web site.’),
);
$form[‘general’][‘site_mail’] = array(
 ’#type’ => ’textfield’,
 ’#title’ => t(‘E-mail address’),
 ’#default_value’ => variable_get(‘site_mail’,
 ini_get(‘sendmail_from’)),
 ’#maxlength’ => 128,
 ’#description’ => t(‘A valid e-mail address for this website, used
by the auto-mailer during registration, new password requests,
notifications, etc.’),
);
$form[‘general’][‘site_slogan’] = array(
 ’#type’ => ’textfield’,
 ’#title’ => t(‘Slogan’),
 ’#default_value’ => variable_get(‘site_slogan’, ’’),
 ’#maxlength’ => 128,
 ’#description’ => t(‘The slogan of this website. Some themes
display a slogan when available.’),
);
### Declaration of a text area. This field is using the default
’#cols’ width, so it’s not
### declared explicitly, and a custom ’#rows’ attribute
$form[‘general’][‘site_mission’] = array(
 ’#type’ => ’textarea’,

104
'#title' => t('Mission'),
'#default_value' => variable_get('site_mission', ''),
'#rows' => 5, '#description' => t('Your site\'s mission statement or focus.'),
);
$form['general']['site_footer'] = array(
  '#type' => 'textarea',
  '#title' => t('Footer message'),
  '#default_value' => variable_get('site_footer', ''),
  '#rows' => 5,
  '#description' => t('This text will be displayed at the bottom of each page. Useful for adding a copyright notice to your pages.'),
);
$form['general']['anonymous'] = array(
  '#type' => 'textfield',
  '#title' => t('Anonymous user'),
  '#default_value' => variable_get('anonymous', 'Anonymous'),
  '#description' => t('The name used to indicate anonymous users.'),
);
$form['general']['site_frontpage'] = array(
  '#type' => 'textfield',
  '#title' => t('Default front page'),
  '#default_value' => variable_get('site_frontpage', 'node'),
  '#description' => t('The home page displays content from this relative URL. If you are not using clean URLs, specify the part after "?q=". If unsure, specify "node".'),
);
// We check for clean URL support using an image on the client side.
### Radio buttons element. Notice the '#options' attribute, which takes the the same array
### structure for it’s value as pre-4.7 did.
### NOTE: Description shortened because of issue with formatting
$form['general']['clean_url'] = array(
  '#type' => 'radios',
  '#title' => t('Clean URLs'),
  '#default_value' => variable_get('clean_url', 0),
  '#options' => array(t('Disabled'), t('Enabled')),
  '#description' => t('This option makes Drupal emit clean URLs ...'),
);
variable_set('clean_url_ok', 0);
global $base_url;
// We will use a random URL so there is no way a proxy or a browser
could cache the "no such image" answer.
### The markup type. This is the default type for forms elements.
'#/value' is used since
### it can’t be altered by the user, and the value is output directly
as markup.

```php
$form['general']['clean_url_test'] = array(
    '#type' => 'markup',
    '#value' => '<img style="position: relative; left: -1000em;" src="'. $base_url . '/system/test/'. user_password(20) .''.png" alt="'/',
);
// Error handling:
$form['errors'] = array(
    '#type' => 'fieldset',
    '#title' => t('Error handling'),
    '#collapsible' => TRUE,
    '#collapsed' => TRUE
);
$form['errors']['site_403'] = array(
    '#type' => 'textfield',
    '#title' => t('Default 403 (access denied) page'),
    '#default_value' => variable_get('site_403', ''),
    '#description' => t('This page is displayed when the requested document is denied to the current user. If you are not using clean URLs, specify the part after "?q=". If unsure, specify nothing.'),
);
$form['errors']['site_404'] = array(
    '#type' => 'textfield',
    '#title' => t('Default 404 (not found) page'),
    '#default_value' => variable_get('site_404', ''),
    '#description' => t('This page is displayed when no other content matches the requested document. If you are not using clean URLs, specify the part after "?q=". If unsure, specify nothing.'),
);

### A select element. '#default_value' will be the initially selected value when the form is rendered. Again note the '#options' attribute, which takes an array as it’s value.
$form['errors']['error_level'] = array(
    '#type' => 'select',
    '#title' => t('Error reporting'),
    '#default_value' => variable_get('error_level', 1),
    '#options' => array(t('Write errors to the log'), t('Write errors to the log and to the screen')),
    '#description' => t('Where Drupal, PHP and SQL errors are logged. On a production server it is recommended that errors are only written to the error log. On a test server it can be helpful to write logs to the screen.'),
);
$period = drupal_map_assoc(array(3600, 10800, 21600, 32400, 43200, 86400, 172800, 259200, 604800, 1209600, 2419200), 'format_interval');
$period['1000000000'] = t('Never');
$form['errors']['watchdog_clear'] = array(
    '#type' => 'select',
    '#title' => t('Discard log entries older than'),
    '#default_value' => variable_get('watchdog_clear', 604800),
    '#options' => $period,
    '#description' => t('The time log entries should be kept. Older entries will be automatically discarded. Requires crontab.'),
);

// Caching:
$form['cache'] = array(
    '#type' => 'fieldset',
    '#title' => t('Cache settings'),
    '#collapsible' => TRUE,
    '#collapsed' => TRUE,
);
$form['cache']['cache'] = array(
    '#type' => 'radios',
    '#title' => t('Page cache'),
    '#default_value' => variable_get('cache', CACHE_DISABLED),
    '#options' => array(CACHE_DISABLED => t('Disabled'), CACHE_ENABLED => t('Enabled')),
    '#description' => t('Drupal has a caching mechanism which stores dynamically generated web pages in a database. By caching a web page, Drupal does not have to create the page each time someone wants to view it, instead it takes only one SQL query to display it, reducing response time and the server’s load. Only pages requested by "anonymous" users are cached. In order to reduce server load and save bandwidth, Drupal stores and sends compressed cached pages.'),
);

$period = drupal_map_assoc(array(0, 60, 180, 300, 600, 900, 1800, 2700, 3600, 10800, 21600, 32400, 43200, 86400), 'format_interval');
$form[0] = t('none');
$form['cache']['cache_lifetime'] = array(
    '#type' => 'select',
    '#title' => t('Minimum cache lifetime'),
    '#default_value' => variable_get('cache_lifetime', 0),
    '#options' => $period,
    '#description' => t('Enabling the cache will offer a sufficient performance boost for most low-traffic and medium-traffic sites. On high-traffic sites it can become necessary to enforce a minimum cache lifetime. The minimum cache lifetime is the minimum amount of time that will go by before the cache is emptied and recreated. A larger minimum cache lifetime offers better performance, but users will not see new content for a longer period of time.'),
);
// File system:
$form['files'] = array('type' => 'fieldset', 'title' => t('File system settings'), 'collapsible' => TRUE, 'collapsed' => TRUE);
$directory_path = variable_get('file_directory_path', 'files');
file_check_directory($directory_path, FILE_CREATE_DIRECTORY, 'file_directory_path');
$form['files']['file_directory_path'] = array('type' => 'textfield', 'title' => t('File system path'), 'default_value' => $directory_path, 'maxlength' => 255, 'valid' => 'directory', 'description' => t('A file system path where the files will be stored. This directory has to exist and be writable by Drupal. If the download method is set to public this directory has to be relative to Drupal installation directory, and be accessible over the web. When download method is set to private this directory should not be accessible over the web. Changing this location after the site has been in use will cause problems so only change this setting on an existing site if you know what you are doing.'));
$directory_temp = variable_get('file_directory_temp', ini_get('upload_tmp_dir'));
file_check_directory($directory_temp, FILE_CREATE_DIRECTORY, 'file_directory_temp');
### Note the use of the '#valid' attribute--the value is a custom validation function
### for this element. See the full doc for more details
$form['files']['file_directory_temp'] = array('type' => 'textfield', 'title' => t('Temporary directory'), 'default_value' => $directory_temp, 'maxlength' => 255, 'valid' => 'directory', 'description' => t('Location where uploaded files will be kept during previews. Relative paths will be resolved relative to the file system path.'));
$form['files']['file_downloads'] = array('type' => 'radios', 'title' => t('Download method'), 'default_value' => variable_get('file_downloads', FILE_DOWNLOADS_PUBLIC), 'options' => array(FILE_DOWNLOADS_PUBLIC => t('Public - files are available using http directly.'), FILE_DOWNLOADS_PRIVATE => t('Private - files are transferred by Drupal.')), 'description' => t('If you want any sort of access control on the downloading of files, this needs to be set to <em>private</em>. You can
change this at any time, however all download URLs will change and there may be unexpected problems so it is not recommended.

*/

// Image handling:
$group = array();
$toolkits_available = image_get_available_toolkits();
if (count($toolkits_available) > 1) {
  $group['image_toolkit'] = array(
    '#type' => 'radios',
    '#title' => t('Select an image processing toolkit'),
    '#default_value' => variable_get('image_toolkit',
      image_get_toolkit()),
    '#options' => $toolkits_available,
  );
}
$group['toolkit'] = image_toolkit_invoke('settings');
if (is_array($group)) {
  $form['image'] = array(
    '#type' => 'fieldset', '#title' => t('Image handling'),
    '#collapsible' => TRUE, '#collapsed' => true);
  ### Notice here how the above created $group is merged into the
  ### $form array
  $form['image'] = array_merge($form['image'], $group);
}

// Feed settings
$form['feed'] = array(
  '#type' => 'fieldset',
  '#title' => t('RSS feed settings'),
  '#collapsible' => TRUE,
  '#collapsed' => TRUE,
);  
$form['feed']['feed_default_items'] = array(
  '#type' => 'select',
  '#title' => t('Number of items per feed'),
  '#default_value' => variable_get('feed_default_items', 10),
  '#options' => drupal_map_assoc(array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 25, 30)),
  '#description' => t('The default number of items to include in a
  feed.'),
);  
$form['feed']['feed_item_length'] = array(
  '#type' => 'select',
  '#title' => t('Display of XML feed items'),
  '#default_value' => variable_get('feed_item_length', 'teaser'),
'#options' => array(
    'title' => t('Titles only'),
    'teaser' => t('Titles plus teaser'),
    'fulltext' => t('Full text')
),
'#description' => t('Global setting for the length of XML feed items that are output by default.'),
);

// Date settings:
$zones = _system_zonelist();

// Date settings: possible date formats
$dateshort = array('Y-m-d H:i', 'm/d/Y - H:i', 'd/m/Y - H:i', 'Y/m/d - H:i',
    'm/d/Y - g:ia', 'd/m/Y - g:ia', 'Y/m/d - g:ia',
    'M j Y - g:ia', 'j M Y - g:ia', 'Y M j - g:ia');
$datemedium = array('D, Y-m-d H:i', 'D, m/d/Y - H:i', 'D, d/m/Y - H:i',
    'D, m/d/Y - g:ia', 'D, d/m/Y - g:ia', 'D, Y/m/d - g:ia',
    'F j, Y - g:ia', 'j F, Y - g:ia', 'Y, F j - g:ia');
    'l, F j, Y - g:ia', 'l, j F, Y - g:ia', 'l, Y, F j - g:ia');

// Date settings: construct choices for user
foreach ($dateshort as $f) {
    $dateshortchoices[$f] = format_date(time(), 'custom', $f);
}
foreach ($datemedium as $f) {
    $datemediumchoices[$f] = format_date(time(), 'custom', $f);
}
foreach ($datelong as $f) {
    $datelongchoices[$f] = format_date(time(), 'custom', $f);
}

$form['dates'] = array(
    '#type' => 'fieldset',
    '#title' => t('Date settings'),
    '#collapsible' => TRUE,
    '#collapsed' => TRUE,
);
$form['dates']['date_default_timezone'] = array(
    '#type' => 'select',
    '#title' => t('Default time zone'),
    '#default_value' => variable_get('date_default_timezone', 0),
    '#options' => $zones, '#description' => t('Select the default site time zone.'),
);
$form['configurable_timezones'] = array(
    '#type' => 'radios',
    '#title' => t('Configurable time zones'),
    '#default_value' => variable_get('configurable_timezones', 1),
    '#options' => array(t('Disabled'), t('Enabled')),
    '#description' => t('Enable or disable user-configurable time zones. When enabled, users can set their own time zone and dates will be updated accordingly.'),
);

$form['date_format_short'] = array(
    '#type' => 'select',
    '#title' => t('Short date format'),
    '#default_value' => variable_get('date_format_short', $dateshort[0]),
    '#options' => $dateshortchoices,
    '#description' => t('The short format of date display.'),
);

$form['date_format_medium'] = array(
    '#type' => 'select',
    '#title' => t('Medium date format'),
    '#default_value' => variable_get('date_format_medium', $datemedium[0]),
    '#options' => $datemediumchoices,
    '#description' => t('The medium sized date display.'),
);

$form['date_format_long'] = array(
    '#type' => 'select',
    '#title' => t('Long date format'),
    '#default_value' => variable_get('date_format_long', $datelong[0]),
    '#options' => $datelongchoices,
    '#description' => t('Longer date format used for detailed display.'),
);

$form['date_first_day'] = array(
    '#type' => 'select',
    '#title' => t('First day of week'),
    '#default_value' => variable_get('date_first_day', 0),
    '#options' => array(0 => t('Sunday'), 1 => t('Monday'), 2 => t('Tuesday'), 3 => t('Wednesday'), 4 => t('Thursday'), 5 => t('Friday'), 6 => t('Saturday')),
    '#description' => t('The first day of the week for calendar views.'),
);

// Site offline/maintenance settings
$form['site_status'] = array(
    '#type' => 'fieldset',
);
''#title' => t('Site maintenance'),
'collapsible' => TRUE,
'collapsed' => TRUE,
);
$form['site_status']['site_offline'] = array(
    'type' => 'radios',
    'title' => t('Site status'),
    'default_value' => variable_get('site_offline', 0),
    'options' => array(t('Online'), t('Offline')),
    'description' => t('When set to "Online", all visitors will be able to browse your site normally. When set to "Offline", only users with the "administer site configuration" permission will be able to access your site to perform maintenance, all other visitors will see the site offline message configured below.'),
);
$form['site_status']['site_offline_message'] = array(
    'type' => 'textarea',
    'rows' => 5,
    'title' => t('Site offline message'),
    'default_value' => variable_get('site_offline_message', t('%site is currently under maintenance. We should be back shortly. Thank you for your patience.'), array('%site' => variable_get('site_name', t('This drupal site')))),
    'description' => t('Message to show visitors when site is offline.'),
);
// String handling: report status and errors.
$form['strings'] = array(
    'type' => 'fieldset',
    'title' => t('String handling'),
    'collapsible' => TRUE,
    'collapsed' => TRUE,
);
$form['strings'] = array_merge($form['strings'], unicode_settings());
### Note here that since this form is eventually being returned to
### hook_settings, drupal_get_form is not called, and only the $form array is returned. In any case where
### a calling function will process the form array, return only the
### $form array without
### calling drupal_get_form. This will be the case for most core
### hooks.
return $form;
?>
Validation and execution functions: contact_mail_page

This example demonstrates how to make use of the API’s validation and execution functions.

**Before**

```php
<?php
function contact_mail_page() {
    global $user;
    if (!flood_is_allowed('contact', CONTACT_HOURLY_THRESHOLD)) {
        $output = t("You can’t send more than %number messages per hour. Please try again later.", array('%number' => CONTACT_HOURLY_THRESHOLD));
    } else {
        if (isset($_POST['edit'])) {
            $edit = $_POST['edit'];
        }
        if ($edit) {
            // Validate the fields:
            if (!$edit['name']) {
                form_set_error('name', t('You must enter a name.'));
            }
            if (!$edit['mail'] || !valid_email_address($edit['mail'])) {
                form_set_error('mail', t('You must enter a valid e-mail address.'));
            }
            if (!$edit['subject']) {
                form_set_error('subject', t('You must enter a subject.'));
            }
            if (!$edit['message']) {
                form_set_error('message', t('You must enter a message.'));
            }
            if (!$edit['category']) {
                // Look if there is only one category
                $result = db_query('SELECT category FROM {contact}');
                if (db_num_rows($result) == 1) {
                    $category = db_fetch_object($result);
                    $edit['category'] = $category->category;
                } else {
                    form_set_error('category', t('You must select a category.'));
                }
            }
        }
        form_validate($edit, $user->name . $user->mail);
        if (!form_get_errors()) {
```
// Prepare the sender:
$from = $edit['mail'];
// Compose the body:
$message[] = t("%name sent a message using the contact form at %form:", array('%name' => $edit['name'], '%form' => url($_GET['q'], NULL, NULL, TRUE)));
$message[] = $edit['message'];
// Tidy up the body:
foreach ($message as $key => $value) {
    $message[$key] = wordwrap($value);
}
// Format the category:
$subject = '['.$edit['category'] .'] '. $edit['subject'];
// Prepare the body:
$body = implode("\n\n", $message);
// Load the category information:
$contact = db_fetch_object(db_query("SELECT * FROM {contact} WHERE category = '%s'", $edit['category']));
// Send the e-mail to the recipients:
user_mail($contact->recipients, $subject, $body, "From: $from\nReply-to: $from\nX-Mailer: Drupal\nReturn-path: $from\nErrors-to: $from");
if ($edit['copy']) {
    user_mail($from, $subject, $body, "From: $from\nReply-to: $from\nX-Mailer: Drupal\nReturn-path: $from\nErrors-to: $from");
}
// Send an auto-reply if necessary:
if ($contact->reply) {
    user_mail($from, $subject, wordwrap($contact->reply), "From: $from\nReply-to: $contact->reply\nX-Mailer: Drupal\nReturn-path: $contact->reply\nErrors-to: $contact->reply");
}
// Log the operation:
flood_register_event('contact');
watchdog('mail', t('"%name-from sent an e-mail regarding %category." <$from>"), array('%name-from' => theme('placeholder', $edit['name']) ." "<$from>"), '%category' => theme('placeholder', $contact->category)));
// Set a status message:
subject(drupal_set_message(t('Your message has been sent.')));
// Jump to contact page:
user_goto('contact');
}
else if ($user->uid) {
$edit['name'] = $user->name;
$edit['mail'] = $user->mail;
}
$result = db_query('SELECT category FROM {contact} ORDER BY category');
$categories[] = '--';
while ($category = db_fetch_object($result)) {
    $categories[$category->category] = $category->category;
}
if (count($categories) > 1) {
    $output = variable_get('contact_form_information', t('You can leave us a message using the contact form below.'));
    $output .= form_textfield(t('Your name'), 'name', $edit['name'], 60, 255, NULL, NULL, TRUE);
    $output .= form_textfield(t('Your e-mail address'), 'mail', $edit['mail'], 60, 255, NULL, NULL, TRUE);
    $output .= form_textfield(t('Subject'), 'subject', $edit['subject'], 60, 255, NULL, NULL, TRUE);
    if (count($categories) > 2) {
        $output .= form_select(t('Category'), 'category', $edit['category'], $categories, NULL, NULL, NULL, NULL, TRUE);
    }
    $output .= form_textarea(t('Message'), 'message', $edit['message'], 60, 5, NULL, NULL, TRUE);
    $output .= form_checkbox(t('Send me a copy.'), 'copy', $edit['copy']);
    $output .= form_token($user->name . $user->mail);
    $output .= form_submit(t('Send e-mail'));
    $output = form($output);
} else {
    $output = t('The contact form has not been configured.'));
}
return $output;
}?>

After

<?php
function contact_mail_page() {
    global $user;
    if (!flood_is_allowed('contact', CONTACT_HOURLY_THRESHOLD)) {
        $output = t("You can’t send more than %number messages per hour. Please try again later.", array('%number' => CONTACT_HOURLY_THRESHOLD));
    }
} else {
    if ($user->uid) {
        $name = $user->name;
        $mail = $user->mail;
    }

    $result = db_query('SELECT category FROM {contact} ORDER BY category');
    $categories[] = '--';
    while ($category = db_fetch_object($result)) {
        $categories[$category->category] = $category->category;
    }

    if (count($categories) > 1) {
        ### Implementation of the token feature. See the full doc for more information
        $form['#token'] = $user->name . $user->mail;
        $form['contact_information'] = array(
            '#type' => 'markup',
            '#value' => variable_get('contact_form_information', t('You can leave us a message using the contact form below.')),
        );

        $form['name'] = array(
            '#type' => 'textfield',
            '#title' => t('Your name'),
            '#maxlength' => 255,
            '#default_value' => $name,
            '#required' => TRUE,
        );

        $form['mail'] = array(
            '#type' => 'textfield',
            '#title' => t('Your e-mail address'),
            '#maxlength' => 255,
            '#default_value' => $mail,
            '#required' => TRUE,
        );

        $form['subject'] = array(
            '#type' => 'textfield',
            '#title' => t('Subject'),
            '#maxlength' => 255,
            '#required' => TRUE,
        );

        if (count($categories) > 2) {
            $form['category'] = array('#type' => 'select', '#title' => t('Category'), '#options' => $categories, '#required' => TRUE);
        }

        $form['message'] = array(
            '#type' => 'textarea',
        );
To insert a validation function for the specified form, simply create a validate function, appending _validate to the form_id of the form you wish to validate. The function has two args--the form_id of the form being validated, and the already built form array of the form being validated.

```php
<?php
function contact_mail_page_validate($form_id, &$form) {
    ### Here the global variable where form values are stored is brought
    ### for possible editing
    global $form_values;
    ### To check values, simply access them w/ the same name with which
    they were declared
    if (!$form['name']) {
        form_set_error('name', t('You must enter a name.'));
    }
    if (!$form['mail'] || !valid_email_address($form['mail'])) {
        form_set_error('mail', t('You must enter a valid e-mail address.'));
    }
    if (!$form['subject']) {
        form_set_error('subject', t('You must enter a subject.'));
    }
    if (!$form['message']) {
        form_set_error('message', t('You must enter a message.'));
    }
    if (!$form['category']) {
```
// Look if there is only one category
$result = db_query('SELECT category FROM {contact}');
if (db_num_rows($result) == 1) {
    $category = db_fetch_object($result);
    ### Here the global form array is edited before passing it along
    to the execute function
    $form_values['category'] = $category->category;
} else {
    form_set_error('category', t('You must select a valid category.'));
}

To insert an execute function for the specified form, create an execution function, appending _submit to the form_id of the form you wish to execute. The function has two args—the form_id of the form being executed, and the already built form array of the form being executed. Notice how the $_POST['edit']/switch statement approach is eliminated, which has major security benefits

<?php
function contact_mail_page_submit($form_id, $edit) {
    // Prepare the sender:
    $from = $edit['mail'];
    // Compose the body:
    ### Note how the form values are accessed the same way they were
    ### accessed in the validate
    ### function
    $message[] = t("%name sent a message using the contact form at
form:", array('%name' => $edit['name'], '%form' => url($_GET['q'],
NULL, NULL, TRUE)));
    $message[] = $edit['message'];
    // Tidy up the body:
    foreach ($message as $key => $value) {
        $message[$key] = wordwrap($value);
    }
    // Format the category:
    $subject = '[' . $edit['category'] . ']. $edit['subject'];
    // Prepare the body:
    $body = implode("\n\n", $message);
    // Load the category information:
    $contact = db_fetch_object(db_query("SELECT * FROM {contact} WHERE
category = "' . $edit['category'] . "");
    // Send the e-mail to the recipients:
    user_mail($contact->recipients, $subject, $body, "From:"
// If the user requests it, send a copy.
// Here the checkbox value is examined in the form array
if ($edit[‘copy’]) {
    user_mail($from, $subject, $body, "From: $from
Reply-to: $from
X-Mailer: Drupal
Return-path: $from
Errors-to: $from”);
}
// Send an auto-reply if necessary:
if ($contact->reply) {
    user_mail($from, $subject, wordwrap($contact->reply), "From: $contact->recipients
Reply-to: $contact->recipients
X-Mailer: Drupal
Return-path: $contact->recipients
Errors-to: $contact->recipients”);
}
// Log the operation:
flood_register_event(‘contact’);
watchdog(‘mail’, t(‘%name-from sent an e-mail regarding %category.’), array(’%name-from’ => theme(‘placeholder’, $edit[‘name’] . " <$from>"),
’%category’ => theme(‘placeholder’, $contact->category)));
// Set a status message:
subject drupal_set_message(t(‘Your message has been sent.’));
// Jump to contact page:
### Note that the function ends with the same redirect as in the original approach,
### so the page is reloaded after the form is executed
drupal_goto(‘contact’);
?>

**Theming forms: system_themes**

This function demonstrates how to make use of multiple checkboxes, theming of form elements in a table, and a separate theming function which allows for rendering of the table and inline HTML elements.

**Before**

```php
<?php
function system_themes() {
    system_listing_save();
    $form = system_theme_listing();
    $form .= form_submit(t(‘Save configuration’));
    print theme(‘page’, form($form));
}
function system_theme_listing() {
    $themes = system_theme_data();
```
ksort($themes);
foreach ($themes as $info) {
    $info->screenshot = dirname($info->filename) . '/screenshot.png';
    $row = array();
    // Screenshot column.
    $row[] = file_exists($info->screenshot) ? theme('image',
        $info->screenshot, t('Screenshot for %theme theme', array('%theme' =>
            $info->name)), '', 'class="screenshot"', false) : t('no screenshot');
    // Information field.
    $row[] = htmlspecialchars(dirname($info->filename)). '</em>' .
    // enabled, default, and operations columns
    $row[] = array('data' => form_checkbox('', 'status[]'. $info->name,
        1, $info->status), 'align' => 'center');
    $row[] = array('data' => form_radio('', 'theme_default',
        $info->name, (variable_get('theme_default', 'bluemarine') ==
            $info->name) ? 1 : 0), 'align' => 'center');
    if (function_exists($info->prefix . '_settings') ||
        function_exists($info->prefix . '_features')) {
        $row[] = array('data' => l(t('configure'),
            'admin/themes/settings/' . $info->name), 'align' => 'center');
    } else {
        $row[] = '';
    }
    $rows[] = $row;
}
$header = array(t('Screenshot'), t('Name'), t('Enabled'),
    t('Default'), t('Operations'));
$output = form_hidden('type', 'theme');
$output .= theme('table', $header, $rows);
return $output;
}

<?php
function system_themes() {
    $themes = system_theme_data();
    ksort($themes);
    foreach ($themes as $info) {
        $info->screenshot = dirname($info->filename) . '/screenshot.png';
        $screenshot = file_exists($info->screenshot) ? theme('image',
            $info->screenshot, t('Screenshot for %theme theme', array('%theme' =>
                $info->name)), '', array('class' => 'screenshot'), false) : t('no screenshot');
        // enabled, default, and operations columns
        $row[] = array('data' => form_checkbox('', 'status[]'. $info->name,
            1, $info->status), 'align' => 'center');
        $row[] = array('data' => form_radio('', 'theme_default',
            $info->name, (variable_get('theme_default', 'bluemarine') ==
                $info->name) ? 1 : 0), 'align' => 'center');
        if (function_exists($info->prefix . '_settings') ||
            function_exists($info->prefix . '_features')) {
            $row[] = array('data' => l(t('configure'),
                'admin/themes/settings/' . $info->name), 'align' => 'center');
        } else {
            $row[] = '';
        }
        $rows[] = $row;
    }
    $header = array(t('Screenshot'), t('Name'), t('Enabled'),
        t('Default'), t('Operations'));
    $output = form_hidden('type', 'theme');
    $output .= theme('table', $header, $rows);
    return $output;
}

}}
### Note both the use of ’#markup’ to drop the screenshot info into the form, and
### the grouping of the screenshot/description under the theme name in the form array
$form[$info->name][’screenshot’] = array(’#type’ => ’markup’,
 ’#value’ => $screenshot);

### Use of a form item. Notice how ’#value’ is used, because the value is not editable
### by a user
$form[$info->name][’description’] = array(’
   ’#type’ => ’item’,
   ’#title’ => $info->name,
   ’#value’ => dirname($info->filename),
);

### Here the options array for all checkboxes is built. Notice that the theme name is used
### for the key--this will be important later when the form is themed
$options[$info->name] = ’’;

### Here the status array is built conditionally--only checkboxes that are checked are added to this array. Notice in this array the theme name is put in the element’s value,
### unlike the $options array
if ($info->status) {
    $status[] = $info->name;
} else {
    // Dummy element for form_render. Cleaner than adding a check in the theme function.
    $form[$info->name][’operations’] = array();
}

### Note that links can also be included in a markup element.
 Markup can hold any
### kind of markup that needs to get into the form
$form[$info->name][’operations’] = array(’
   ’#type’ => ’markup’,
   ’#value’ => l(t(’configure’),
   ’admin/themes/settings’ . $info->name),
);

else {
    // Dummy element for form_render. Cleaner than adding a check in the theme function.
    $form[$info->name][’operations’] = array();
}

### Now that all checkbox options have been built, and all checked boxes are know, the
### checkboxes element can be declared. Notice that the $status array
is dropped directly
### into '#default_value'
$form[‘status’] = array(
   ’#type’ => ’checkboxes’,
   ’#options’ => $options,
   ’#default_value’ => $status,
);
### Radio button groups are built the same basic way, using the same
$options array in
### this case
$form[‘theme_default’] = array(
   ’#type’ => ’radios’,
   ’#options’ => $options,
   ’#default_value’ => variable_get(’theme_default’, ’bluemarine’)
);
### Notice that the two submit buttons are grouped under a ‘buttons’
group in the form array
### This will be important when we examine the execute function
$form[‘buttons’][’submit’] = array(’#type’ => ’submit’, ’#value’ =>
t(’Save configuration’) );
$form[‘buttons’][’reset’] = array(’#type’ => ’submit’, ’#value’ =>
t(’Reset to defaults’) );
### Drop the form array into the master form function
return drupal_get_form(’system_themes’, $form);
}
?>

In this case, the theming of the form is fairly complex, so a custom theming function is used. Custom theme functions are declared by prepending theme_ to the form_id of the form you wish to theme. The single arg is the constructed form array.

<?php
function theme_system_themes($form) {
   ### The constructed form array has a number of internal
   record-keeping elements, so directly
   ### looping through the array would result in errors. The
   element_children function
   ### extracts only those form elements that have a value.
   foreach (element_children($form) as $key) {
      ### Here the table rows are constructed using the previous
      convention of a rows array.
      ### Notice that there is a check to make sure that the particular
      form array element
      ### has valid theme info in it, otherwise an empty row is added
      $row = array();
      if (is_array($form[$key][’description’])) {
         ### In order to manually render a portion of the form,
form_render is called. It’s
### single argument is the section of the form array that is to
be rendered. Note
### that form_render is recursive--it will render all form array
elements in the portion
### of the array that you declare.
### $row[] = form_render($form[$key][‘screenshot’]);
$row[] = form_render($form[$key][‘description’]);
### $form[‘status’] is the checkboxes element. Notice that by
using $form[‘status’][$key],
### only the checkbox for the current theme $key gets rendered.
If the rendered checkbox
### has a matching value in the above created $status array
(which was passed to the
### checkboxes element), then it will be rendered as a checked
box
$row[] = array(‘data’ => form_render($form[‘status’][$key]),
’align’ => ’center’);
if ($form[‘theme_default’]) {
### The radio buttons are rendered using the same logic as the
checkboxes
$row[] = array(‘data’ =>
form_render($form[‘theme_default’][$key], ‘align’ => ’center’);
$row[] = array(‘data’ =>
form_render($form[$key][‘operations’]), ‘align’ => ’center’);
}
$rows[] = $row;
### Now the table is created using the usual theme_table approach
$header = array(t(‘Screenshot’), t(‘Name’), t(‘Enabled’),
t(‘Default’), t(‘Operations’));
$output = theme(‘table’, $header, $rows);
### The rendering code remembers which form elements have already
been rendered--therefore,
### to render any remaining elements (in this case the submit
buttons), simply call form_render
### using the entire form array, and only unrendered elements will be
rendered. It’s good
### practice to always end with this, in case other modules may have
used form_alter to
### include additional form elements
$output .= form_render($form);
### Finally, the constructed output is returned in the standard
fashion.
return $output;}
A custom execute function also exists for this form. Note that the form is executed before it is themed/displayed.

```php
<?php
function system_themes_execute($form_id, $values) {
    db_query("UPDATE {system} SET status = 0 WHERE type = 'theme'");
    # $POST['op'] can be examined just as before to determine which
    # button was pressed
    if ($POST['op'] == t('Save configuration')) {
        if (is_array($values['status'])) {
            # Only those checkboxes that were checked are returned in the
            processed form array
            foreach ($values['status'] as $key => $choice) {
                // If theme status is being set to 1 from 0, initialize block
                data for this theme if necessary.
                if (db_num_rows(db_query("SELECT status FROM {system} WHERE
                    type = 'theme' AND name = '%s' AND status = 0", $key))) {
                    system_initialize_theme_blocks($key);
                }
                db_query("UPDATE {system} SET status = 1 WHERE type = 'theme'
                    and name = '%s', $key);
            }
        } else {
            variable_del('theme_default');
        }
    }
    drupal_set_message(t('The configuration options have been saved.'));
    # Redirecting back to the page, which will reload the form with the
    updated data
    drupal_goto('admin/themes');
}
?>
```
Advanced theming: system_user

This example demonstrates how to build and implement a custom theming function that themes only a portion of the form array. This can be used when:

- A section of the form needs to be themed identically multiple times, in which case the function can be written once and reused.
- A form array needs to be returned without the use of drupal_get_form (as in the case of a core hook), but a portion of the array needs to be custom themed.

Before

```php
<?php
function system_user($type, $edit, &\$user, $category = NULL) {
    if ($type == 'form' && $category == 'account') {
        $allthemes = list_themes();
        // list only active themes
        foreach ($allthemes as $key => $theme) {
            if ($theme->status) {
                $themes[$key] = $theme;
            }
        }
        if (count($themes) > 1) {
            $rows = array();
            foreach ($themes as $key => $value) {
                $row = array();
                // Screenshot column.
                $screenshot = dirname($value->filename) . '/screenshot.png';
                $row[] = file_exists($screenshot) ? theme('image', $screenshot, t('Screenshot for %theme theme', array('%theme' => $value->name)), '', 'class="screenshot"', false) : t('no screenshot');
                // Information field.
                $field = '<strong>' . $value->name . '</strong> ';
                $row[] = $field;
                // Selected column.
                $row[] = array('data' => form_radio('', 'theme', $key, ($edit['theme'] == $key) ? 1 : 0), 'align' => 'center');
                $rows[] = $row;
            }
        }
    }
}
```
$header = array(t('Screenshot'), t('Name'), t('Selected'));
$data[] = array('title' => t('Theme settings'), 'data' => form_item('', theme('table', $header, $rows), t('Selecting a different theme will change the look and feel of the site.')), 'weight' => 2);

if (variable_get('configurable_timezones', 1)) {
    $zones = _system_zonelist();
    $data[] = array('title' => t('Locale settings'), 'data' => form_select(t('Time zone'), 'timezone', strlen($edit['timezone']) ? $edit['timezone'] : variable_get('date_default_timezone', 0), $zones, t('Select your current local time. Dates and times throughout this site will be displayed using this time zone.')), 'weight' => 2);
}

return $data;

After

<?php

function system_user($type, $edit, &$user, $category = NULL) {
    if ($type == 'form' && $category == 'account') {
        $themes = list_themes();
        ksort($themes);
        // Reset to follow site default theme if user selects the site default
        if ($key == variable_get('theme_default', 'bluemarine')) {
            $key = '';
            if ($edit['theme'] == variable_get('theme_default', 'bluemarine')) {
                $edit['theme'] = '';
            }
        }
        ### Notice the use of the '#weight' attribute--individual elements can now be weighted
    ### to reorder the form. Also note the '#theme' attribute--this is a reference to the
    ### custom theming function for this section of the form. The function is named by
    ### prepending theme_ to the value of the '#theme' attribute.
    $form['themes'] = array(
        '#type' => 'fieldset', '#title' => t('Theme configuration'),
        '#description' => t('Selecting a different theme will change the look and feel of the site.'), '#weight' => 2, '#collapsible' => TRUE,
        '#collapsed' => FALSE, '#theme' => 'system_user');
foreach ($themes as $info) {
    $info->screenshot = dirname($info->filename) . '/screenshot.png';
    $screenshot = file_exists($info->screenshot) ? theme('image',
        $info->screenshot, t('Screenshot for %theme theme', array('%theme' => $info->name)), '', array('class' => 'screenshot'), false) : t('no screenshot');
    $form['themes'][$info->name]['screenshot'] = array('#type' => 'markup', '#value' => $screenshot);
    $form['themes'][$info->name]['description'] = array('#type' => 'item', '#title' => $info->name, '#value' => dirname($info->filename));
    $options[$info->name] = '';}

if (variable_get('configurable_timezones', 1)) {
    $zones = _system_zonelist();
    $form['locale'] = array('#type' => 'item', '#title' => t('Locale settings'), '#weight' => 6);
    $form['locale']['timezone'] = array('#type' => 'select', '#title' => t('Time zone'),
        '#default_value' => strlen($edit['timezone']) ? $edit['timezone'] : variable_get('date_default_timezone', 0),
        '#options' => $zones, '#description' => t('Select your current local time. Dates and times throughout this site will be displayed using this time zone.'));
}

### This is an implementation of a core hook, hook_user--so the constructed form array is returned without calling drupal_get_form. This makes the use of the individual theming function necessary, otherwise no theming information could be passed back to the hook

return $form;}

This is the individual theming function whose callback was specified above. Notice that it follows the same basic approach as the custom theming function in the previous example, including returning $output. The only difference is that this function only themes the part of the form where it’s callback was declared.

<?php
function theme_system_user($form) {
    foreach (element_children($form) as $key) {
$row = array();
if (is_array($form[$key]['description'])) {
    $row[] = form_render($form[$key]['screenshot']);
    $row[] = form_render($form[$key]['description']);
    $row[] = form_render($form['theme'][$key]);
}
$rows[] = $row;
$header = array(t('Screenshot'), t('Name'), t('Selected'));
$output = theme('table', $header, $rows);
return $output;
}

Drupal 4.6 vs. Drupal 4.7 Forms API Flowcharts

The following shows a comparison in the way forms were done pre-4.7 and how they are done in Drupal 4.7. Attached below is a Dia file containing an editable version of the diagrams.

See the print-friendly version if images are being hidden by menus.

Drupal Pre-4.7 Forms API
Example:

```php
<?php
function example_menu($may_cache) {
    $items = array();
    if ($may_cache) {
        $items[] = array('path' => 'example_page', 'callback' => 'example_page', 'title' => 'example page', 'access' => TRUE);
    }
    return $items;
}

function example_page() {
    $op = isset($_POST['op']) ? $_POST['op'] : '';
    $edit = isset($_POST['edit']) ? $_POST['edit'] : '';
    if ($op == t('Submit')) {
        if (!($edit['field_1'] >= 1 && $edit['field_1'] <= 2)) {
            form_set_error('field_1', t('Enter a value between 1 and 2 '));
        }
        if (!form_get_errors()) {
            db_query('INSERT INTO {example} (data) VALUES (%d)', $edit['field_1']);
        }
    }
}
```
Drupal 4.7 Forms API
Example:

function example_menu($may_cache) {
    $items = array();
    if ($may_cache) {
        $items[] = array('path' => 'example_page', 'callback' => 'example_page', 'title' => 'example page', 'access' => TRUE);
    }
    return $items;
}

function example_page() {
    $form['field_1'] = array('#type' => 'textfield', '#default_value' => 1, 'title' => 'first textfield', 'description' => t('Enter a value between 1 and 2 '));
    $form['field_2'] = array('#type' => 'submit', '#value' => t('Submit'));
}


drupal_get_form('example_form_id', $form);
}
function example_form_id_validate($form_id, $form_values) {
  if (!$form_values['field_1'] >= 1 && $form_values['field_1'] <= 2))
  {
    form_set_error('field_1', t('Enter a value between 1 and 2 '));
  }
}
function example_form_id_submit($form_id, $form_values) {
  db_query('INSERT INTO {example} (data) VALUES (%d)', $form_values['field_1']);
  drupal_goto();
}

Forms API FAQ
Validation
How do you use the #validation arguments?

Example:
#valid => 'integer' , #validation arguments => array(1, 13)

This would map to:
valid_integer($form['element'], 1, 13)

where 1,13 could be min/ max allowed values.

If you have multiple validation functions, use:
#valid => array('integer', 'uid'), #validation arguments = array(array(1, 13), array('anonymous'))

which would map to: valid_integer($form['element'], 1, 13) and valid_uid($form['element'],
'anonymous') where the second parameter could be role, for instance, so you could see if it’s a
valid uid in a certain role or '#valid' => 'filename', '#validation_arguments' => 'rwx' where it
becomes valid_filename($form, $permissions)

How do you validate a URL, e-mail address or integer value?

Until they are written for core, you would need to write a valid_url, valid_email and
valid_integer function.
**is the nodeapi 'validate' op to be used anymore given the validation features of the new api? if so, when?**

nodeapi validate is for node objects. It gets called by the form api though (node_validate), but it can also be called from code. (programmatically creating nodes)

**Form logic**

**How do you handle multiple submit buttons in a form? Where do you put the 'dispatch logic'?**

Examine $_POST['op'] to find the pressed button. It’s a good idea to have an execute function for a form, and the dispatch logic can go in there.

**What does #post_process do?**

Post process is used to make alterations to the form, after the form has already been processed.

This particularly comes into play during Node Preview. We can’t display unfiltered data to the user from the $_POST, so we have a post_process on the node form which adds the node_preview form element using the already filtered values. See Image.module for an example.

**Miscellaneous**

**How do I add a taxonomy selection to my form?**

As of Drupal 4.7, you no longer have to do this -- it is done automatically!

So, where you used to have something like:

```php
if (function_exists('taxonomy_node_form')) {
    $output = implode('', taxonomy_node_form($node-&gt;type, $node));
}
```

You now have... nothing!

**Why was the #attribute_name convention chosen?**

Observe the following example:

```php
$form['fieldset'] = array('#type' =&gt; 'fieldset', '#title' =&gt; 'something'); $form['fieldset']['title'] = array('#type' =&gt; 'textfield')
```

If we didn’t use the # in the beginning, the new title field would override the title property of thefieldset. we tried using defines, which looked nice, but were troublesome, and eventually we went with the #, because if you look at your html page DOM with your dom inspector, you see all the text is named as '#TEXT', so it seemed like using that for standardization seemed like a good idea.
Your own drupaldocs or api.drupal.org site

I decided to spend the evening figuring out how to make my own copy of drupaldocs so that if drupaldocs should ever become unavailable, I wouldn’t be stuck. Hopefully this will help other people as well. :)  

Step 1: Gathering the files

You’ll need two things:

1. A copy of the API module; found in the contributions repository, under /modules/api/
2. A copy of the development documentation; found in the contributions repository, under /docs/developer/

Step 2: Installation

1. Copy the /api/ directory into your /modules/ directory. This will include api.module, api.css and parser.inc.
2. Copy the /developer/ directory to your site’s top directory.
3. Next, execute the commands in api.mysql on your database. I just copied and pasted the file contents into PHPMyAdmin, but you could also import it from the commandline, using a command similar to:

4. mysql -u username -p password < api.mysql

5. Finally, enable the API module, by going to administer > modules, checking the Enable box next to the api module, and finally clicking Save Configuration. This will give you two additional menu items: API reference (at the very top) and administer > API reference.

Step 3: Indexing

1. Go to administer > API reference.
2. Under branches to index, enter a Short name, Long name, and Directory for your Drupal installation. For example, my Drupal installation is from HEAD, so I entered:

3. Short name: HEAD
   Long name: Drupal HEAD
   Directory: /absolute/path/to/your/drupal/installation (in my case, this was /home/username/drupal) -- Make sure to leave off the trailing slash!

   Click Save changes when finished.

4. Repeat for any other Drupal branches you might have installed. In my case, I only had one, so I left it as-is.
5. Next, click the Index PHP manual pages button. This will parse PHP-specific function names and so on so they can be referenced in the code samples.
6. NOTE: My host has disabled remote use of file_get_contents() for security reasons, so I had to replace line 13 in parser.inc with cURL function calls:

```php
// $source = file_get_contents($location);
$ch = curl_init();
curl_setopt($ch, CURLOPT_URL, $location);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);
$source = curl_exec($ch);
curl_close($ch);
```

7. Click Reindex to tag all files for reindexing. A message will appear at the top notifying you that this has been done, and informing you that the index will be rebuilt during the next few runs of cron.php. Waiting for cron.php will take way too long, though (it only indexes 10 files at a time, and cron.php runs every 5 minutes). Sooo...

**Step 4: Finishing touches**

1. Instead, open the link to [cron.php](#) and reload the page numerous times, until you quit seeing messages like Parsing "/some/path/somefile.ext".... For me, this was after about 10 or so refreshes, but it will probably be much more if you have a full-featured Drupal installation.

2. NOTE: Right after the message about parsing path.module I received a bunch of errors which read:

   Warning: Unexpected character in input: '\' (ASCII=92) state=1 in /path/to/drupal/modules/api/parser.inc on line 571

   However, this didn’t seem to affect anything in the end (though I admittedly haven’t had the chance to look too closely either).

3. Finally, and I have no idea why this is, but it seems you have to go back to [administer > modules](#) and Save Configuration again. After that, clicking on [API reference](#) up at the top will supply a sub-menu containing whatever long name(s) you entered in the previous steps, which will then take you to the documentation for your installation of Drupal (without this step, API reference is just a blank page).

And voila!

The one thing I couldn’t get working were the example modules (e.g. "How to define blocks"). They’re in the /module/developer/examples/ folder, but for some reason don’t match up with where the API module thinks they should be. Any hints anyone might have would be greatly appreciated. :)

Notes from chx. For me, example modules worked fine. But I needed to raise PHP memory limit significantly.
Module developer’s guide

Developer documentation can be found at [http://drupaldocs.org/](http://drupaldocs.org/) and in the remainder of the Drupal developer’s guide below.

- drupaldocs.org documents the Drupal APIs and presents an overview of Drupal’s building blocks along with handy examples.
- The Drupal developer guide provides guidelines as how to upgrade your modules (API changes) along with development tips/tutorials.

Introduction to Drupal modules

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

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

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

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

See also the overview of module hooks which is generated from the Drupal source code.

Creating modules - a tutorial

This tutorial describes how to create a module for the current Drupal release (formerly 4.5.*). It is an update to the tutorial for Drupal 4.3. Please see comments there, also.

A module is a collection of functions that link into Drupal, providing additional functionality to your Drupal installation. After reading this tutorial, you will be able to create a basic block module and use it as a template for more advanced modules and node modules.
This tutorial will not necessarily prepare you to write modules for release into the wild. It does not cover caching, nor does it elaborate on permissions or security issues. Use this tutorial as a starting point, and review other modules and the Drupal handbook and Coding standards for more information.

This tutorial assumes the following about you:

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

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

01. Getting started

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

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

```php
<?php

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

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

The module is not operational yet: it hasn’t been activated. We’ll activate the module later in the tutorial.

Download source and rename to onthisdate.module before saving in your Drupal installation.

02. Telling Drupal about your module

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

```php
function onthisdate_help($section='') {

137
The $section variable provides context for the help: where in Drupal or the module are we looking for help. The recommended way to process this variable is with a switch statement. You’ll see this code pattern in other modules.

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

The admin/modules#description case is used by the Drupal core as the module description on the modules administration page (/admin/modules or ?q=admin/modules).

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

[More information about the help hook](#)

[Download source](#) and rename to onthisdate.module before saving in your Drupal installation.

03. Telling Drupal who can use your module

The next function to write is the permissions function, using the _perm hook. This is where you will define the names of the permissions of your module. This function function doesn’t grant permission, it just specifies what permissions are available for this module. Access based on these permissions is defined later in the {module}_access function, later in the tutorial.

At this point, we’ll use the most general permission string, and give access to anyone who can access site content or administrate the module by using the "access content" string. This means, if a user can see the site content, s/he can also see the content from this module:
```php
/**
 * Valid permissions for this module
 * @return array An array of valid permissions for the onthisdate module
 */
function onthisdate_perm() {
    return array('access content');
} // function onthisdate_perm()
?>

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

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

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

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

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

The current naming convention is "action_verb modulename". For example:

```php
function newmodule_perm() {
    return array('access newmodule', 'create newmodule', 'administer newmodule');
} // function newmodule_perm()
?>
```

The setup of the module is now done. Next, we’ll start generating content.

[Download source] and rename to onthisdate.module before saving in your Drupal installation.
04. Declare we have block content

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

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

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

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

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

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

Download source and rename to onthisdate.module before saving in your Drupal installation.

05. Generate the block content

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

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

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

```php
<?php
/**
 * Generate HTML for the onthisdate block
 * @param op the operation from the URL
 * @param delta offset
 * @returns block HTML
 */
function onthisdate_block($op='list', $delta=0) {
  // listing of blocks, such as on the admin/block page
  if ($op == "list") {
    $block[0]["info"] = t("On This Date");
    return $block;
  } else if ($op == 'view') {
    // our block content
    // Get today’s date
    $today = getdate();
    // calculate midnight one week ago
    $start_time = mktime(0, 0, 0, $today['mon'], ($today['mday'] - 7), $today['year']);
    // we want items that occur only on the day in question, so
    // calculate 1 day
    $end_time = $start_time + 86400;
    // 60 * 60 * 24 = 86400 seconds in a day
    ...
  }
}
```
The next step is the SQL statement that will retrieve the content we’d like to display from the database. We’re selecting content from the node table, which is the central table for Drupal content. We’ll get all sorts of content type with this query: blog entries, forum posts, etc. For this tutorial, this is okay. For a real module, you would adjust the SQL statement to select specific types of content (by adding the 'type' column and a WHERE clause checking the 'type' column).

**Note:** the table name is enclosed in curly braces: `{node}`. This is necessary so that your module will support database table name prefixes. You can find more information on the Drupal website by reading the [Table Prefix (and sharing tables across instances)](https://www.drupal.org/manual/1) page in the Drupal handbook.

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

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

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

```php
<?php
// get the links
$queryResult = db_query($query);
// content variable that will be returned for display
$block_content = '';
while ($links = db_fetch_object($queryResult)) {
    $block_content .= l($links-&gt;title, 'node/' . $links-&gt;nid) . '<br'/>
}
// check to see if there was any content before setting up
// the block
if ($block_content == '') {
    /* No content from a week ago. If we return nothing, the block
    * doesn’t show, which is what we want. */
    return;
}
// set up the block
$block['subject'] = 'On This Date';
$block['content'] = $block_content;
return $block;
?>
```
Notice the actual URL is enclosed in the l() function. l generates <a href="link"> links, adjusting the URL to the installation’s URL configuration of either clean URLS: http://(sitename)/node/2 or not http://(sitename)/?q=node/2

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

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

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

```php
<?php
function onthisdate_block(&$op='list', $delta=0) {
    // listing of blocks, such as on the admin/block page
    if ($op == "list") {
        $block[0]["info"] = t("On This Date");
        return $block;
    } else if ($op == 'view') {
        // our block content
        $block_content = ";;
        // Get today’s date
        $today = getdate();
        // calculate midnight one week ago
        $start_time = mktime(0, 0, 0, $today[‘mon’],
                           ($today[‘mon’] - 7), $today[‘year’]);
        // we want items that occur only on the day in question, so
        // calculate 1 day
        $end_time = $start_time + 86400;
        // 60 * 60 * 24 = 86400 seconds in a day
        $query = "SELECT nid, title, created FROM ".
                "{node} WHERE created >= " . $start_time .
                " AND created <= " . $end_time . ";;
        // get the links
        $queryResult = db_query($query);
        while ($links = db_fetch_object($queryResult)) {
            $block_content .= l($links->title, ‘node’.$links->nid) . ‘<br ‘/‘;
        }
        // check to see if there was any content before setting up the
        // block
        if ($block_content == '') {
```
Our module is now functional - we can install, enable and test it.

Download source and rename to onthisdate.module before saving in your Drupal installation.

06. Installing, enabling and testing the module

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

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

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

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

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

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

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

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

Now, head to another page, say, select the modules menu. In some themes, the blocks are displayed after the page has rendered the content, and you won’t see the change until you go to new page.
If you have content that was created a week ago, the block will display with links to the content. If you don’t have content, you’ll need to fake some data. You can do this by creating a blog, forum topic or book page, and adjust the “Authored on:” date to be a week ago.

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

07. Create a module configuration (settings) page

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

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

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

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

```php
<?php
    // check the user has content access
    if (!user_access("access content")) {
        return message_access();
    }
    ...
?>
```
To minimize the number of permissions an administrator has to deal with, we’re going to use the global administration permission for administrating our module:

```php
<?php
...
// check the user has administration access
if (!user_access('access administration pages')) {
    return message_access();
}
...
?>
```

This example is actually a bit artificial, as settings hook pages are available only to users with the ‘access administration pages’ permissions - this check isn’t technically needed. As you’ll see later in the tutorial, page access is most easily controlled in the _menu hook. For this tutorial example, however, we’ll leave in the permissions check.

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

For 4.5-4.6 modules, use the form_textfield function:

```php
<?php
function onthisdate_settings() {
    // only administrators can access this function
    if (!user_access('access administration pages')) {
        return message_access();
    }
    $output .= form_textfield(t("Maximum number of links"),
    "onthisdate_maxdisp",
    variable_get("onthisdate_maxdisp", "3"), 2, 2,
    t("The maximum number of links to display in the block.")); return $output;
}
?>
```

For 4.7 modules, use the updated form API:

```php
<?php
function onthisdate_settings() {
    // only administrators can access this module
    if (!user_access('access administration pages')) {
        return message_access();
    }
    $form['onthisdate_maxdisp'] = array(
        'type' => 'textfield',
        'title' => t("Maximum number of links"),
        'default_value' => variable_get("onthisdate_maxdisp", 3),
        'description' => t("The maximum number of links to display in the block."),
        'maxlength' => '2',
        'size' => '2');
```
This function uses several powerful Drupal form handling features. We don’t need to worry about creating an HTML text field or the form, as Drupal will do so for us for this settings page. We use `variable_get` to retrieve the value of the system configuration variable "onthisdate_maxdisp", and define the default value to be 3. We use the `form_textfield` function to create the form and a text box of size 2, accepting a maximum length of 2 characters. We also use the translate function of `t()`. There are other form functions that will automatically create the HTML form elements for use. For now, we’ll just use the `form_textfield()` function for 4.5 - 4.6 nodes, and the forms API for 4.7 and later.

When you save a settings variable for any module, the variable (in our case, 'onthisdate_maxdisp') and the value is stored in the variables table. Programmatically, you can retrieve the values with the `variable_get('variable_name', default_value)` function.

Of course, we’ll need to use the configuration value in our SQL SELECT, so we’ll need to adjust our query statement in the onthisdate_block function. One way to do this is with a LIMIT value in our query:

```php
<?php
$limitnum = variable_get("onthisdate_maxdisp", 3);
$query = "SELECT nid, title, created FROM " .
"{node} WHERE created >= " . $start_time .
" AND created <= " . $end_time . " LIMIT " . $limitnum;
?>
```

However, this method may or may not translate across databases (really). Better to use one of Drupal's select methods. In this case, let’s leave the original query the same, and call `db_query_range`:

```php
<?php
$limitnum = variable_get("onthisdate_maxdisp", 3);
$query = "SELECT nid, title, created FROM " .
"{node} WHERE created >= " . $start_time .
" AND created <= " . $end_time;
$arResult = db_query_range($query, 1, $limitnum);
?>
```

You can test the settings page by editing the number of links displayed and noticing the block content adjusts accordingly. Navigate to the settings page: admin/settings/onthisdate or administer » settings » onthisdate. (If the page doesn’t exist, you may have to disable and enable the module for the system to register the new settings page.) Adjust the number of links and save the configuration. Notice the number of links in the block adjusts accordingly.

**Note:** We don’t have any validation with this input. If you enter "c" in the maximum number of links, you’ll break the block.
08. Generate a page content

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

```php
<?php
function onthisdate_all() {}
?>
```

We’re going to use much of the code from the block function. We’ll write this ExtremeProgramming style, and duplicate the code. If we need to use it in a third place, we’ll refactor it into a separate function. For now, copy the code to the new function onthisdate_all().

Contrary to all our other functions, ‘all’, in this case, is not a Drupal hook.

If you want to call this function from another module, use the standard naming scheme we’ve been using: modulename_action. It can be called using the function `module_invoke` function. If you want the function to remain private (because, say, it’s merely a helper function in your module) and easily accessible by only your module, prefix the function name with an underscore. We want the former.

```php
<?php
function onthisdate_all() {
    // content variable that will be returned for display
    $page_content = '';  // Get today’s date
    $today = getdate();
    // calculate midnight one week ago
    $start_time = mktime(0, 0, 0, $today['mon'], ($today['mday'] - 7), $today['year']);
    // we want items that occur only on the day in question,
    // so calculate 1 day
    $end_time = $start_time + 86400;
    // 60 * 60 * 24 = 86400 seconds in a day
    $query = "SELECT nid, title, created FROM \" . 
        "\{node\} WHERE created >= '" . $start_time . 
        "' AND created <= '" . $end_time . 
        "'";
    // get the links (no range limit here)
    $queryResult = db_query($query);
    while ($links = db_fetch_object($queryResult)) {
        $page_content .= l($links->title, 'node/' . $links->nid).'<br />';  // Generate a page content
```

In this example, we are generating a page that lists all the content created a week ago. We are using the `$query` string to select all nodes with a `created` timestamp between the start time (midnight one week ago) and the end time (1 day after the start time). We then iterate over the returned results, building the content to be displayed on the page.
We have the page content at this point, but we want to do a little more with it than just return it. When creating pages, we need to send the page content to the theme for proper rendering. We use this with the theme() function. Themes control the look of a site. As noted before, we’re including layout in the code. This is bad, and should be avoided. It is, however, the topic of another tutorial, so for now, we’ll include the formatting in our content:

```php
<?php
print theme("page", $page_content);
?>
```

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

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

```php
function onthisdate_all() {
    ...
    // check to see if there was any content before
    // setting up the block
    if ($page_content == '') {
        // no content from a week ago, let the user know
        $page_content = "No events occurred on this site on this date in history.";
        return;
    }
    print theme("page", $page_content);
}
?>
```

Even though we have this function that will output links to the content generated a week ago, we haven’t specified what URL will cause this page to render. We’ll do that next.

[Download source] and rename to onthisdate.module before saving in your Drupal installation.

09. Letting Drupal know about the new function

As mentioned previously, the function we just wrote isn’t a ‘hook’: it’s not a Drupal recognized name. We need to tell Drupal how to access the function when displaying a page. We do this with the menu() hook. The menu() hook defines the association between a URL and the function that creates the content for that url. The hook also does permission checking, if desired.
<?php
function onthisdate_menu() {
    $items = array();
    $items[] = array("path" => 'onthisdate',
                    "title" => t('on this date'),
                    "callback" => 'onthisdate_all',
                    "access" => user_access('access content'),
                    "type" => MENU_CALLBACK);
    return $items;
}
?>

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

More information on the menu system

As mentioned before the menu hook can handle permission checking before rendering the page. The 'access' entry in the menu item array is where this check is done. If you added a value in your permissions array in the perm hook function, you can use that string as a parameter in the user_access function. If the user isn’t in a role that has that permission, the page will not render for the user.

If the module has not be enabled, enable it. If you have already enabled it, in order to reset the menu definitions in the system, you’ll need to disable, then reenable it.

Now, navigate to /onthisdate (or ?q=onthisdate) and see what you get.

Download source and rename to onthisdate.module before saving in your Drupal installation.

10. Adding a 'more' link and showing all entries

We now have a function that creates a page with all the content created a week ago. Let’s link to it from the block with a "more" link.

Add these lines just before that $block['subject'] line. These lines will add the more link to the end of the $block_content variable before saving it to the $block['content'] variable:

<?php
// add a more link to our page that displays all the links
    $block_content .= 
        "<div class="more-link"">".
        l(t("more"), "onthisdate", array("title" => t("More events on")),
This will add the more link to the block. Note the extra parameters used in the \[l()\] function. You can add additional values, such as 'class', in the array to customize the link.

[Download source] and rename to onthisdate.module before saving in your Drupal installation.

**Conclusion**

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

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

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

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

**Documenting your code**

Drupal uses PHPDoc syntax for code documentation with a couple exceptions. See the \[api.module docs\] This api.module creates all the system documentation which you see on [http://drupaldocs.org]. You can use this module to host your own offline copy of drupaldocs.org along with Contrib documentation if you wish.

**How to create your own simple node type (from story node)**

How to create your own simple node type...
A simple node type is the most basic of node types such as 'page' and 'story' node types, with a title, teaser, body, etc.

Why would you want or need this...
The answer stems from a discussion about the [Difference Between Page and Story](#) and that ultimately each node type allows for separate theming, categorization, descriptions, etc...

This example walks you through copying the 'story.module' and making the appropriate changes to the php code for your new simple node type.

There are no database changes or hacking of code, the result will be a new module file that you will need to upload into it's own module directory.

What you need...
- means to download/upload files to your modules directory
- a text editor
- a little php knowledge is helpful, but not required
- a technical name for your node type (16 character limit) - no spaces, numbers or punctuation unless you know what you’re doing (don’t use the name of a module you already have)
- a user-friendly name for your node type (and a plural version of this name) (this is how your node type will appear in most places on your site) - spaces and numbers are ok but again no punctuation unless you know what you’re doing, note: you can use the same name as your technical name above
- a module description - a short description of your node type that will appear on the administer->modules page - don’t use quotes or apostrophes unless you know what you are doing
- a create content description - a short description of your node type that will appear on the create content page - don’t use quotes or apostrophes unless you know what you are doing

In the following **bolded** text in the code boxes is intended to show where the code changes. Quotes and apostrophes in the code are important, please place close attention. It should be noted also that these directions were developed from drupal version 4.6

As an example... say you want a simple press release node type.
- technical name = **release**
- user-friendly name = **press release**
- user-friendly plural = **press releases**
- module description = **Enables the creation of press releases.**
- create content description = **Create a press release.**

- Download the **story.module** file from the modules directory
- Save it as your technical name **.module** in our example the file will be named **release.module**
- Open the new file in your text editor

Now, without getting into too many details on drupal modules and php code.... The first line of code should be
<?php

don’t touch that

The next line of code should be


You should remove this, so that there is no confusion with the story module

The next couple lines of code should be

/**
 * @file
 * Enables users to submit stories, articles or similar content.
 */

change the bolded text above to your user-friendly plural, in our example press releases seen below

/**
 * @file
 * Enables users to submit press releases.
 */

The story.module file allows for story node types. The module file implements the following 6 drupal hooks

- hook_help
- hook_node_name
- hook_perm
- hook_access
- hook_menu
- hook_form

Before each hook is implemented you should you will see a php comment

for hook_help...

/**
 * Implementation of hook_help().
 */

and for hook_node_name...

/**
 * Implementation of hook_node_name().
 */

e tc...
After the comment, the hook function is called, but instead of the word hook the word story is used as this module was originally for story nodes.

for hook_help...

function story_help($section) {

and for hook_node_name...

function story_node_name($node) {

e tc...

In the function calls change story to your technical name, in our example release

for hook_help...

function story_help($section) {

becomes

function release_help($section) {

and for hook_node_name

function story_node_name($node) {

becomes

function release_node_name($node) {

e tc...

We will be changing all 6 in the following steps, plus a few more things...

Here are the changes, moving down the file, starting at the top
Again noting that in the following bolded text in the code boxes is intended to show where the code changes. Quotes and apostrophes in the code are important, please place close attention.

1. hook_help
hook_help shows helpful information in various places...

hook_help orginal code from story.module

function story_help($section) {
    switch ($section) {
        case 'admin/modules#description':
            return t('Allows users to submit stories, articles or similar content.');
        case 'node/add#story':
            return t('Stories are articles in their simplest form: they have a title, a teaser and a body, but can be extended by other modules. The teaser is part of the body...')
    }
too. Stories may be used as a personal blog or for news articles.’);
}
}

here we need our technical name and our two descriptions as indicated

function technical_help($section) {
    switch ($section) {
        case 'admin/modules#description':
            return t('module description');
        case 'node/add#technical':
            return t('create content description');
    }
}

hook_help using our press release example the code becomes

function release_help($section) {
    switch ($section) {
        case 'admin/modules#description':
            return t('Enables the creation of press releases. ’);
        case 'node/add#release':
            return t('Create a press release. ’);
    }
}

2. hook_node_name

hook_node_name registers the node type with drupal

hook_node_name original code from story.module

function story_node_name($node) {
    return t('story');
}

here we our technical name and our user-friendly name as indicated

function technical_node_name($node) {
    return t('user-friendly name');
}

hook_node_name using our press release example the code becomes

function release_node_name($node) {
    return t('press release');
}

3. hook_perm

hook_perm assigns names for permissions that may exist for this node type (these need to agree with hook_access below)

hook_perm original code from story.module
function story_perm() {
    return array('create stories', 'edit own stories');
}

here we need our technical name and our plural as indicated

function technical_perm() {
    return array('create plural', 'edit own plural');
}

hook_perm using our press release example the code becomes

function release_perm() {
    return array('create press releases', 'edit own press releases');
}

4. hook_access
hook_access checks your permissions against a user and what the permission allows (these need to agree with hook_perm above)

hook_access original code from story.module

function story_access($op, $node) {
    global $user;

    if ($op == 'create') {
        return user_access('create stories');
    }

    if ($op == 'update' || $op == 'delete') {
        if (user_access('edit own stories') && ($user->uid == $node->uid)) {
            return TRUE;
        }
    }
}

here we need our technical name and plural as indicated

function technical_access($op, $node) {
    global $user;

    if ($op == 'create') {
        return user_access('create plural');
    }

    if ($op == 'update' || $op == 'delete') {
        if (user_access('edit own plural') && ($user->uid == $node->uid)) {
            return TRUE;
        }
    }
}

hook_access using our press release example the code becomes

function release_access($op, $node) {
    global $user;

    if ($op == 'create') {
        return user_access('create press releases');
    }
if ($op == 'update' || $op == 'delete') {
    if (user_access('edit own press releases') && ($user->uid == $node->uid)) {
        return TRUE;
    }
}

5. hook_menus

hook_menus allow the node type to show in various menus, in this case just the create content menu (note the 'create' permission should agree with those above)

hook_menu original code from story.module

function story_menu($may_cache) {
    $items = array();
    if ($may_cache) {
        $items[] = array('path' => 'node/add/story', 'title' => t('story'),
                         'access' => user_access('create stories'));
    }
    return $items;
}

here we need our technical name, user-friendly name and plural as indicated

function technical_menu($may_cache) {
    $items = array();
    if ($may_cache) {
        $items[] = array('path' => 'node/add/technical', 'title' => t('user-friendly name'),
                         'access' => user_access('create plural'));
    }
    return $items;
}

hook_menu using our press release example the code becomes

function release_menu($may_cache) {
    $items = array();
    if ($may_cache) {
        $items[] = array('path' => 'node/add/release', 'title' => t('press release'),
                         'access' => user_access('create press releases'));
    }
    return $items;
}

6. hook_form

hook_form allows for a form for adding, editing and deleting nodes of the type

hook_form original code from story.module
function story_form(&$node) {
    $output = ''; 
    if (function_exists('taxonomy_node_form')) {
        $output .= implode('', taxonomy_node_form('story', $node));
    }
    $output .= form_textarea(t('Body'), 'body', $node->body, 60, 20, '', NULL, TRUE); 
    $output .= filter_form('format', $node->format);
    return $output; 
}

here we need just our technical name as indicated

function technical_form(&$node) {
    $output = ''; 
    if (function_exists('taxonomy_node_form')) {
        $output .= implode('', taxonomy_node_form('technical', $node));
    }
    $output .= form_textarea(t('Body'), 'body', $node->body, 60, 20, '', NULL, TRUE); 
    $output .= filter_form('format', $node->format);
    return $output; 
}

hook_form using our press release example the code becomes

function release_form(&$node) {
    $output = ''; 
    if (function_exists('taxonomy_node_form')) {
        $output .= implode('', taxonomy_node_form('release', $node));
    }
    $output .= form_textarea(t('Body'), 'body', $node->body, 60, 20, '', NULL, TRUE); 
    $output .= filter_form('format', $node->format);
    return $output; 
}

That's it for the code changes. Hopefully not too difficult.

Next

- Save the file, you're now ready to upload
- Create a directory in the modules directory with your technical name in our example the directory will be named release
- Upload your new module file to your new directory
- Go to your site, administer -> modules, you should see your new module listed, enable it as with any other module.

Note: if you go to your administer -> modules page and get a white screen, it means you have an error in the php code of your new module, delete the file from your new directory and everything should be fine.

You need to fix your new module, but it is probably easier to start over, re-read the
directions and follow them carefully, if it still doesn’t work, then consider these directions a bunch of bunk and try something else.

Your new node type should work the same as with page and story nodes, you will need to enable access to them (administer->access control), configure them (administer->content->configure), allow for categorization (administer->categories), etc.

Sorry for the long directions, I hope they are understandable and usable.

**How to create your own simple node type (from story node) (Drupal 4.7)**

How to create your own simple node type...

- A simple node type is the most basic of node types such as ‘page’ and ‘story’ node types, with a title, teaser, body, etc.
- Why would you want or need this...
  
  The answer stems from a discussion about the [Difference Between Page and Story](#) and that ultimately each node type allows for separate theming, categorization, descriptions, etc...

- This example walks you through copying the ‘story.module’ and making the appropriate changes to the php code for your new simple node type.
- There are no database changes or hacking of code, the result will be a new module file that you will need to upload into its own module directory.
- What you need...
  - means to download/upload files to your modules directory
  - a text editor
  - a little php knowledge is helpful, but not required
  - some information about your node type
    - a technical-name for your node type - limit 32 characters - don’t use the name of an existing module - no spaces, numbers or punctuation unless you know what you’re doing
    - a user-friendly-name for your node type (and a plural version of this name) (this is how your node type will appear in most places on your site) - spaces and numbers are ok but again no punctuation unless you know what you’re doing, note: you can use the same name as your technical-name above
    - a module-description - a short description of your node type that will appear on the administer->modules page - don’t use quotes or apostrophes unless you know what you are doing
    - a create-content description - a short description of your node type that will appear on the create content page - don’t use quotes or apostrophes unless you know what you are doing
    - a help-description - a short description of your node type that will appear on the help page - don’t use quotes or apostrophes unless you know what you are doing
    - drupal-help-text and link - (these are specifically provided for you below)
• an optional **title-caption** for the title field in the node entry/edit form if something other than 'Title' is desired - don’t use quotes or apostrophes unless you know what you are doing

• an optional **body-caption** for the body field in the node entry/edit form if something other than 'Body' is desired - don’t use quotes or apostrophes unless you know what you are doing

• In the following **bolded** text in the code boxes is intended to show where the code changes. Quotes and apostrophes in the code are important, please place close attention.

As an example... say you want a simple press release node type.

• technical-name = release
• user-friendly-name = press release
• user-friendly-plural = press releases
• module-description = Enables the creation of press releases.
• create-content-description = Create a press release.
• help-description = Press Releases are notices sent to various media outlets informing them of some note-worthy information.
• drupal-help-text = This module was created by [your name here], following directions posted at drupal.org <a href="%release">How to create your own simple node type (from story node) (Drupal 4.7)</a>IMPORTANT: Note the use of our technical-name **release** in the drupal help text above (underlined for emphasis only), you should change the underlined text to your technical-name. You should also change [your name here] to your actual name. The rest of the text should be left as is (unless you know what you are doing).
• drupal-help-link = http://www.drupal.org/node/40684
• title-caption (optional) = Press Release Title
• body-caption (optional) = Press Release Body

I specifically use an example that has a different technical-name and user-friendly-name so that it is a little more clear in the changes where the two are used, but many node-types that exist (or that you may want to create) use the same value.

• Download the story.module file from the modules directory
• Save it as your technical-name .module in our example the file will be named release.module
• Open the new file in your text editor

Now, without getting into too many details on drupal modules and php code....
The first line of code should be

```php
<?php
```

don’t touch that

The next line of code will be something like...
You should remove this, so that there is no confusion with the story module.

The next couple lines of code should be

```php
/**
 * @file
 * Enables users to submit stories, articles or similar content.
 */
```

change the bolded text above to your user-friendly-plural, in our example **press releases** seen below

```php
/**
 * @file
 * Enables users to submit press releases.
 */
```

The story.module file allows for story node types. The module file implements the following 7 Drupal hooks:

- hook_help
- hook_node_info
- hook_perm
- hook_access
- hook_menu
- hook_validate
- hook_form

Before each hook is implemented you should see a PHP comment for hook_help...

```php
/**
 * Implementation of hook_help().
 */
```

and for hook_node_info...

```php
/**
 * Implementation of hook_node_info().
 */
```

e tc...

After the comment, the hook function is called, but instead of the word **hook** the word **story** is used as this module was originally for story nodes.

for hook_help...
function story_help($section) {

and for hook_node_info...

function story_node_info() {

etc...

In the function calls change story to your technical-name, in our example release
for hook_help...

function story_help($section) {

becomes

function release_help($section) {

and for hook_node_info

function story_node_info() {

becomes

function release_node_info() {

etc...

We will be changing all 7 in the following steps, plus a few more things...

Here are the changes, moving down the file, starting at the top
Again noting that in the following bolded text in the code boxes is intended to show where the
code changes. Quotes and apostrophes in the code are important, please place close attention.

1. hook_help
hook_help shows helpful information in various places...
NOTE - this function has the most changes, so if you can get thru this one, the rest are a breeze.
for the curious: this 4.7 hook_help implementation was expanded with additional help text from
4.6

hook_help original

function story_help($section) {
switch ($section) {
  case 'admin/help#story':
    $output = '<p>'. t('The story module is used to create a
    content post type called <em>stories</em>.') .'</p>
    $output .= '<p>'. t('Stories are articles in their simplest form: they have a title,
    a teaser and a body. Stories are typically used to post news articles
    or as a group blog.') .'</p>
    $output .= '<p>'. t('The story administration interface allows
    for complex configuration. It provides a submission form, workflow,
    default view permission, default edit permission, permissions for permission,
    and attachments. Trackbacks can also be enabled.') .'</p>
    $output .= t('<p>You can</p>
to change this function, we need most of our new node type information...

function technical-name_help($section) {
  switch ($section) {
    case 'admin/help#technical-name':
      $output = '<p>'. t('The technical-name module is used to create a content post type called <em>user-friendly-plural</em>.') .'</p>';
      $output .= '<p>'. t('The user-friendly-name administration interface allows for complex configuration. It provides a submission form, workflow, default view permission, default edit permission, permissions for permission, and attachments. Trackbacks can also be enabled.') .'</p>';
      return $output;
    case 'admin/modules#description':
      return t('Allows users to submit stories, articles or similar content.');</n

hook_help using our press release example

function release_help($section) {
  switch ($section) {
    case 'admin/help#release':
      $output = '<p>'. t('The release module is used to create a content post type called <em>press releases</em>.') .'</p>';
      return $output;
    case 'admin/modules#description':
      return t('Press Releases are notices sent to various media outlets');
    case 'node/add#technical-name':
      return t('Stories are articles in their simplest form: they have a title, a teaser and a body, but can be extended by other modules. The teaser is part of the body too. Stories may be used as a personal blog or for news articles.');
  }
}
There is a function story_node_info() which returns an array containing the technical name and the user-friendly name of the node type.

```
function story_node_info() {
  return array('story' =>
    array('name' => t('story'),
      'base' => 'story'));
}
```

To change this function, we need our technical-name and our user-friendly-name.

```
function technical-name_node_info() {
  return array('technical-name' =>
    array('name' => t('user-friendly-name'),
      'base' => 'technical-name'));
}
```

There is also a function release_node_info() which returns an array containing the technical name and the user-friendly name of the node type.

```
function release_node_info() {
  return array('release' =>
    array('name' => t('press release'),
      'base' => 'release'));
}
```

```
3. hook_perm
hook_perm assigns names for permissions that may exist for this node type (these need to agree with hook_access below)
for the curious: this 4.7 hook_perm implementation is the same as the 4.6 implementation
```
hook_perm original

```php
function story_perm() {
    return array('create stories',
                  'edit own stories');
}
```

to change this function, we need our technical-name and our user-friendly-plural

```php
function technical-name_perm() {
    return array('create user-friendly-plural',
                  'edit own user-friendly-plural');
}
```

hook_perm using our press release example

```php
function release_perm() {
    return array('create press releases',
                  'edit own press releases');
}
```

4. hook_access

hook_access checks your permissions against a user and what the permission allows (these need to agree with hook_perm above)

for the curious: this 4.7 hook_access implementation is the same as the 4.6 implementation

hook_access original

```php
function story_access($op, $node) {
    global $user;

    if ($op == 'create') {
        return user_access('create stories');
    }

    if ($op == 'update' || $op == 'delete') {
        if (user_access('edit own stories')
            && ($user->uid == $node->uid)) {
            return TRUE;
        }
    }
}
```

to change this function, we need our technical-name and user-friendly-plural

```php
function technical-name_access($op, $node) {
    global $user;

    if ($op == 'create') {
        return user_access('create user-friendly-plural');
    }

    if ($op == 'update' || $op == 'delete') {
        if (user_access('edit own user-friendly-plural')
            && ($user->uid == $node->uid)) {
            return TRUE;
        }
    }
}
hook_access using our press release example

```php
function release_access($op, $node) {
  global $user;

  if ($op == 'create') {
    return user_access('create press releases');
  }

  if ($op == 'update' || $op == 'delete') {
    if (user_access('edit own press releases')
      && ($user->uid == $node->uid)) {
      return TRUE;
    }
  }
}
```

5. **hook_menu**

hook_menu allows the node type to show in various menus, in this case just the create content menu (note the ‘create’ permission should agree with above)

for the curious: this 4.7 hook_menu implementation is the same as the 4.6 implementation

hook_menu original

```php
function story_menu($may_cache) {
  $items = array();

  if ($may_cache) {
    $items[] = array('path' => 'node/add/story',
      'title' => t('story'),
      'access' => user_access('create stories'));
  }

  return $items;
}
```

to change this function, we need our technical-name, user-friendly-name and user-friendly-plural

```php
function technical-name_menu($may_cache) {
  $items = array();

  if ($may_cache) {
    $items[] = array('path' => 'node/add/technical-name',
      'title' => t('user-friendly-name'),
      'access' => user_access('create user-friendly-plural'));
  }

  return $items;
}
```

hook_menu using our press release example
function release_menu($may_cache) {
    $items = array();

    if ($may_cache) {
        $items[] = array('path' => 'node/add/release',
                         'title' => t('press release'),
                         'access' => user_access('create press releases'));
    }

    return $items;
}

6. hook_validate
hook_validate checks posts when submitted to see if information submitted is valid
for the curious: this hook was not used in the 4.6 version

hook_validate original

function story_validate($node) {
    node_validate_title($node);
}

to change this function, we need our technical-name (note only the function name changes with
this hook)

function technical-name_validate($node) {
    node_validate_title($node);
}

hook_validate using our press release example

function release_validate($node) {
    node_validate_title($node);
}

7. hook_form
hook_form allows for a form for adding, editing and deleting nodes of the type
for the curious: this 4.7 implementation of hook_form is substantially different from 4.6

hook_form original

function story_form(&$node) {

    $form['title'] = array('#type' => 'textfield',
                           '#title' => t('Title'),
                           '#required' => TRUE, '#default_value' => $node->title);

    $form['body'] = array('#type' => 'textarea',
                         '#title' => t('Body'),
                         '#default_value' => $node->body, '#rows' => 20, '#required' => TRUE);

    $form['format'] = filter_form($node->format);

    $form['log'] = array('#type' => 'fieldset', '#title' => t('Log message'),
                         '#collapsible' => TRUE, '#collapsed' => TRUE);

    $form['log']['message'] = array('#type' => 'textarea', '#default_value' => $node->log,
                                     '#description' => t('An explanation of the additions or updates being made to
                                     help other authors understand your motivations.'))
to change this function, we need our technical-name and our two optional captions (note the optional captions are underlined for emphasis and do not need to be changed)

function technical-name_form(&$node) {

$form['title'] = array('#type' => 'textfield',
                    '#title' => t('title caption'),
                    '#required' => TRUE, '#default_value' => $node->title);

$form['body'] = array('#type' => 'textarea',
                     '#title' => t('body caption'),
                     '#default_value' => $node->body, '#rows' => 20, '#required' => TRUE);
$form['format'] = filter_form($node->format);

$form['log'] = array('#type' => 'fieldset', '#title' => t('Log message'),
                     '#collapsible' => TRUE, '#collapsed' => TRUE);
$form['log']['message'] = array('#type' => 'textarea', '#default_value' => $node->log,
                             '#description' => t('An explanation of the additions or updates being made to help other authors understand your motivations.'))

return $form;
}

hook_form using our press release example

function release_form(&$node) {

$form['title'] = array('#type' => 'textfield',
                     '#title' => t('Press Release Title'),
                     '#required' => TRUE, '#default_value' => $node->title);

$form['body'] = array('#type' => 'textarea',
                     '#title' => t('Press Release Body'),
                     '#default_value' => $node->body, '#rows' => 20, '#required' => TRUE);
$form['format'] = filter_form($node->format);

$form['log'] = array('#type' => 'fieldset', '#title' => t('Log message'),
                     '#collapsible' => TRUE, '#collapsed' => TRUE);
$form['log']['message'] = array('#type' => 'textarea', '#default_value' => $node->log,
                             '#description' => t('An explanation of the additions or updates being made to help other authors understand your motivations.'))

return $form;
}

That's it for the code changes. Hopefully not too difficult.

Next

- Save the file, you're now ready to upload
- Create a directory in the modules directory with your technical-name; in our example the directory will be named release
- Upload your new module file to your new directory
Go to your site, administer -> modules, you should see your new module listed, enable it as with any other module.

Note: if you go to your administer -> modules page and get a white screen, it means you have an error in the php code of your new module, delete the file from your new directory and everything should be fine.

You need to fix your new module, but it is probably easier to start over, re-read the directions and follow them carefully, if it still doesn’t work, then consider these directions a bunch of bunk and try something else.

Your new node type should work the same as with page and story nodes, you will need to enable access to them (administer->access control), configure them (administer->settings->content types), allow for categorization (administer->categories), etc.

Sorry for the long directions, I hope they are understandable and usable.

Third party applications integration guide

This section addresses developer needs beyond implementing new modules. Existing sites with unique applications and needs may require additional development to integrate, migrate or co-exist with Drupal. This will be a repository of tips and techniques shared by those who managed to develop such integration projects. Some tips may be unique to a third party application, and some may be reusable by several applications. The most frequent asked questions relate to:

- Sharing a user base
- Sharing authentication
- Synchronizing a dual user base
- External authentication
- Session handler issues
- Theme engine integration
- Import scripts

Session handler issues

Managing your session issues will greatly depend on how your application handles sessions, But the Drupal side of things is straight forward. The session.inc file is included in the bootstrap.inc file.

If you decide to exclusively use your application’s session handler, you can simply comment out that line from bootstrap.inc. Or you may decide to allow both to do different parts, where your application session handles authentication, and let Drupal sessions handle the rest of Drupal functions. If you decide to use both, make sure the two session variables are not in conflict. Drupal uses$_SESSION.
If you decide to eliminate Drupal sessions, these are the parts of Drupal core you will need to modify. Drupal core will use sessions for 4 functions:

Authentication: This part can be ignored, as you will be using your application’s session to handle this part, to identify your users and their roles, as explained in the authentication section.

Content filtering: Drupal sessions are used to store the current filters you use to filter your site’s content list in the content administration page. This is only done in the node module. If you decide to store this data into your application session, this is the part where you will need to save these filters into your session:

```php
<?php
// Initialize/reset filters
if (!isset($_SESSION['node_overview_filter']) || !is_array($_SESSION['node_overview_filter']) || $op == t('Reset')) {
    $_SESSION['node_overview_filter'] = array();
}

$session = &$_SESSION['node_overview_filter'];
$filter = $edit['filter'];
if (($op == t('Filter') || $op == t('Refine')) && isset($filter)) {
    if (isset($filters[$filter]['options'][$edit[$filter]])) {
        $session[] = array($filter, $edit[$filter]);
    }
}
if ($op == t('Undo')) {
    array_pop($session);
}
?>
```

Messages: These are the messages Drupal displays to the user following an action, such "The changes have been saved." These messages are set into the session in one central place, functions `drupal_set_message()` and `drupal_get_messages()` in bootstrap.inc. You can use those locations to store the message into your application session.

Comment preferences: This is done in the comment module to store comment preferences for guests. You can modify function `comment_save_settings()` in the comment module to store these into your application session.

Sharing a user base

The two obvious choices are to either have Drupal use a third party user table, or the other way around (Synchronization is covered in another section). This section will address how Drupal can be made to use an alternative user table.

There is a strategic location in includes/database.inc, where Drupal performs table prefix translation. By trapping the `'users'` you have the power to rewrite all Drupal users query, and can perform your own SQL rewrites.
This has the advantage of not modifying other Drupal files, or minimizing any needed edits.

Use something similar (and cleaner) to this code to perform your own user table query manipulation. This is used at the beginning of db_prefix_tables(), before Drupal performs any table translation.

```php
<?php
if (strpos($sql, '{users}')) {
    if (eregi('^update', $sql)) {
        // someone installed a module that updates the user table
        // this is not supported, issue and error, and quit.
        CallCustomException();
    }

    $DrupalUserSQL = array(
        '@{users}@i',
        '@= u.uid@i',
        '@=u.uid@i',
        '@u.uid =@i',
        '@u.uid=@i',
        '@u.uid@i', // general select
        '@u.uid@i', // group by
        '@u.uid@i', // order by
        '@u.name@i', // group by
        '@u.name@i', // order by
        '@u.pass@i',
        '@u.mail@i',
        '@u.language@i',
        '@u.picture@i',
        '@u.picture, @i',
        '@u.picture, @i',
        '@u.data@i',
        '@, u.data@i',
        '@u.data, @i',
        '@u.status@i'});

    $ThirdPartyUserSQL = array(
        TABLE_PREFIX . 'user', // custom db prefix
        '= u.userid',
        '=u.userid',
        'u.userid =',
        'u.userid=',
        'u.userid as uid',
        'u.userid',
        'u.userid',
        'u.username',
        'u.username',
        'u.password',
        'u.email',
        'u.email',
    );
```
The exception/error thrown above is trap any locations during the development where the users table is being updated. It could remain there, to ensure all new modules have been tested or modified to handle user table updates.

Note that you will probably still need to edit or replace the user module, since allowing both Drupal and your third party application to administer users can have negative side effects.

### Theme engine integration

Your site’s theme may have special content and features that do not exist in Drupal themes. This section will explain how you can use your own themes using Drupal’s theme engines. If all you need to achieve a common "look and feel", you do not need to modify a theme engine, you can simply create a new template based on your site’s design elements.

Drupal currently has the following theme engines: Xtemplate, PHPTemplate, Smarty and Plain PHP. You can read an in-depth explanation of these template engines in the [Theme developer’s guide](#).

Of these four theme engines, the simplest for integration purposes is the Plain PHP method.

In most cases, your theme integration will require a header and a footer. Those two regions will require elements from both Drupal and your application, particularly to include Java scripts used by both applications and onload elements.
Having read the Plain PHP guide, the integration with your templates will simply require including your application’s template engine (if not already included elsewhere, like in settings.php), and adding any required globals from within your plain PHP theme.

Example:

"Example to follow after checking why PHP snippets cannot be added here.."

**Using Javascript and AJAX**

As of version 4.7, Drupal includes built-in methods for implementing Javascript. Using these methods when you use Javascript will help to keep your code clean and to ensure compatibility with other modules’ implementations.

A couple of simple principles guide Drupal’s Javascript approach:

- All pages should be perfectly functional without scripts. Javascript provides alternatives or supplements - not replacements - for standard elements.
- No Javascript is hard-coded onto pages. Rather, actions are attached dynamically to page elements--and only if needed Javascript support is present.

**Drupal’s Javascript tools**

Drupal’s javascript toolkit has three basic components:

1. **drupal.js**
   - The javascript file drupal.js contains methods for implementing Javascript solutions, including AJAX (see below).

2. **PHP functions**
   - Specific Drupal functions in PHP help Javascript developers. These include `drupal_add_js()`, used for adding a js file to a page; `drupal_call_js()`, used to generate a javascript call; and `drupal_to_js()`, used to translate data from Drupal PHP to Javascript objects.

3. **Prebuilt tools**
   - As well as the generic tools in drupal.js and the PHP Javascript functions, Drupal core ships with three prebuilt tools: collapse, used to collapse and expand form fieldsets; progress, used to indicate progress of an action; and autocomplete, used to simulate drop-down select options when users enter text into a textbox.

**drupal.js functions**

The file `drupal.js` is a collection of utility functions providing functionality commonly needed in Javascript applications. As `drupal.js` is loaded automatically the first time a module calls `drupal_add_js()`, its methods are always available to other scripts.
Testing for appropriate Javascript support

function `isJsEnabled()` tests for availability of a number of Javascript methods commonly needed and used in Drupal Javascript applications. This function is commonly used in combination with `addLoadEvent()` so that a particular behaviour can be implemented only if appropriate Javascript support is available. This test is referred to as a "killswitch". An example is the first lines of `autocomplete.js`:

```javascript
// Global Killswitch
if (isJsEnabled()) {
   addLoadEvent(autocompleteAutoAttach);
}
```

AJAX data exchanges

Two functions in `drupal.js` provide cross-browser methods for AJAX--dynamic data exchanges between the client and server. These can be used, e.g., to dynamically refresh selected page content based on user actions.

function `HTTPGet()` and its pair function `HTTPPost()` are used to send data to a server and specify a handler for the response.

function `HTTPGet(uri, callbackFunction, callbackParameter)` accepts three arguments: the uri to post to, the function to handle the response, and a callback parameter--usually, a reference to the calling object. The uri will generally be a Drupal path pointing (via a menu item) to the function that parses the client request and sends back a response. Since a Get operation sends information in url-encoded arguments, the data being sent is typically appended to the uri. If a node id is being sent, for example, the uri might look like:

`'modulename/action/' + nid;`

In the PHP handler function, the `nid` could then be accessed as `arg(2)`.

function `HTTPPost(uri, callbackFunction, callbackParameter, object)` is similar to `HTTPGet()` except that it uses a Post rather than Get operation and includes the additional argument, `object`. This is the data to be posted to the server.

For more information on using these functions, see the [tutorial on using AJAX](tutorial).  

Working with CSS class names

CSS class names play an important part in Drupal’s Javascript implementation. Several `drupal.js` methods focus on adding, removing, and manipulating class selectors.

If you’re wanting to add a class name to an element, you might be tempted simply to set its `classname` attribute:
elt.classname = 'name';

But this method would overwrite any existing class name. By using function `addClass(node, className)`, you ensure that a new name is added to any existing ones. `addClass()` takes as arguments the node being acted upon and the specified class name.

Similarly, `removeClass(node, className)` (same two arguments) removes only the specified class name, leaving any others in place.

`hasClass(node, className)` tests if a given element has a given class name, while `toggleClass(node, className)` will add a class name if its not present, or remove it if it is.

**Element position**

`function absolutePosition(el)` will return the absolute position of an element on the screen. The return value is an object with properties of `x` and `y`, representing the absolute x and y coordinates of the element.

**Adding events**

Similar to `addClass()`, two functions provide methods for adding events to elements. `addLoadEvent(func)` adds a function to the window onload event. This is typically used in combination with `isJsEnabled()`, see above.

`function addSubmitEvent(form, func)` adds a function to a given form’s submit event. This is useful for changing the behaviour of a form. See `autocomplete.js` for a sample usage (to prevent a form from submitting if the suggestions popup is open).

**Tutorial 1: Creating new Javascript widgets**

If you want to implement functionality not already available, you’ll create a new Javascript file and then write PHP calls that add to the page what’s needed for the Javascript to work.

So, to come up with a simple if fairly useless example, say we want to write a module that allows users to click on specific words and get a message with further information. Here’s how we might do it.

- **Use CSS class selectors to identify the elements to add the Javascript to.**

  You might be familiar with the approach of adding Javascript to an element through parameters, like this:

  `<span onclick="doSomething()">click me</span>`

  In Drupal, we consider such hard-coding inelegant and also error-prone. Instead, we provide a way to identify elements that Javascript can be attached to selectively, after the page loads. Anything that can identify an element will do (e.g., an id attribute), but we tend to use CSS class selectors. So, instead of the above, we would output something like:
<span class="clickInfo">click me</span>

Often you'll need to pass information to your scripts. In this case, we want to pop up a message—but of what? The simplest way to pass the information is to use another attribute of the element you're giving the class to—in our case, a <span> element. Candidates are id and title attributes. Example:

<span class="clickInfo" title="Hello world">click me</span>

Of course, rather than manually outputting the class and other attributes, you'll want to write a PHP function that adds them. In our case, in a click_info.module file, we might use something like the following:

```php
<?php
function click_info_make($text, $words) {
    foreach ($words as $word => $message) {
        $text = str_replace ($word, '<span class="clickInfo" title="' . $message . '">
            ' . $word . '</span>', $text);
    }
    return $text;
}
?>
```

Now we can add the needed elements to a particular string through a call to that function. So, if we want to make every occurrence of the word "Drupal" clickable, with the message being "rocks!", we could use a _nodeapi hook:

```php
/**
 * Implementation of hook_nodeapi().
 */
function click_info_nodeapi(&$node, $op, $teaser, $page) {
    switch ($op) {
    case 'view':
        $words = array('Drupal' => 'rocks!');
        $node->body = click_info_make($node->body, $words);
    }
}
?>
```

- **Put your functionality in a new Javascript file.**

In most or all cases, you'll want to create a new Javascript file for your new functionality.

In the file, you'll need at the very least an "autoattach" function. The purpose of this is to attached the desired behaviours to page elements. Note that function and variable names in Drupal Javascript are written in "camel case". Call your javascript file, and its "autoattach" function, after the behaviour. So in our case, we would create a clickInfo.js file, and include in it a clickInfoAutoAttach function.
The first thing you need to do in your autoattach function is answer the question "What elements do I attach this behaviour to?" Typically, in your autoattach function you will first select all elements of a given node type, then iterate through them and see if they have right class. In our case, we’re looking for (a) span elements that (b) have class set to clickInfo. When we find one, we want to attach a particular behaviour to it: that, when clicked, it will pop an alert up, with the data we’ve encoded in an attribute called info. Just to be fancy, we’re going to make the onclick action trigger a custom function. So our autoattach function will look like this:

```javascript
function clickInfoAutoAttach() {
    var spans = document.getElementsByTagName('span');
    for (var i = 0; span = spans[i]; i++) {
        if (span && hasClass(span, 'clickInfo')) {
            // Read in the message from the 'title' attribute
            span.message = span.getAttribute('title');
            // Set the title to NULL, so no tooltip will display
            span.removeAttribute('title');
            span.onclick = function() {
                alert(this.message);
            }
        }
    }
}

Now all we need to do is ensure this code is run when it should be. To do this, we add a call at the beginning of our script using two more drupal.js functions:

```javascript
if (isJsEnabled()) {
    addLoadEvent(clickInfoAutoAttach);
}
```

This snippet (a) tests if appropriate Javascript support is present, and, if so (b) registers our autoattach function to be called when the page has loaded (and, therefore, all the page elements are available to have behaviours attached to them).

- **Add style (optional)**
  In many cases, but not all, you’ll want to change the display of the elements you’re working with. This is best done through an included .css file. In our case, we want to let users know that specific words/phrases are clickable. We can do this through a .css file, click_info.css:

  ```html.js
  .clickInfo {
    background: yellow;
    cursor: pointer;
  }
  ```

  Note the html.js selector: it selects the class 'js' on the <html> tag. Drupal adds this class through JavaScript, so it means this style will only be used if JavaScript is enabled.

- **Send the needed files**
We’ve got all the needed elements, so all that remains is to send them to the user. We do this with `drupal_add_js()`. Ideally, we’ll add the Javascript only when we know it’s needed on a page—i.e., when we’ve created target elements. In this case, since we add those elements in a `nodeapi()` hook, we can add calls to that function.

```php
<?php
/**
 * Implementation of hook_nodeapi().
 */
function click_info_nodeapi(&$node, $op, $teaser, $page) {
    switch ($op) {
    case 'view':
        $path = drupal_get_path('module', 'click_info');
        drupal_add_js($path . '/click_info.js');
        theme_add_style($path . '/click_info.css');
        $words = array('Drupal' => 'rocks');
        $node->body = click_info_make($node->body, $words);
        break;
    }
}
?>
```

Note that we don’t need a separate call for adding `drupal.js`; it’s added automatically with the first `drupal_add_js()` call. Similarly, we don’t need to worry about scripts being added twice, as a static variable in `drupal_add_js()` ensures that already-added files are skipped.

See the [completed module](#) To use it:

- Install and enable the module
- Create a new node (e.g., a ‘story’), and include the word Drupal.
- View the page. The word Drupal should be highlighted. Click on it to get the alert ‘rocks’.

**Tutorial 2: Using existing Javascript widgets: autocomplete**

This tutorial will lead you through the steps of implementing an autocomplete textfield in your module.

Autocomplete is implemented in Drupal through AJAX. When users type into a textbox, code on the client page dynamically loads new data from the server (a Drupal website) and uses these data to update the user display (provide a drop-down list of matching options, which the user can select from).

Handily, all the mechanics of exchanging data between client and server and of updating the display are handled by the widget, autocomplete.js. Implementing a particular autocomplete textfield requires two parts: (a) a caller (the textfield, with special additions) and (b) a handler, which is a PHP function that parses the request and returns a response.
Prebuilt autocomplete functions

Two autocomplete functions ship with the Drupal core. Each is referenced by an "autocomplete_path"--the uri to which autocomplete requests are sent.

- **user_autocomplete()**
  Use this function to load matching user names. Autocomplete path: user/autocomplete.

- **taxonomy_autocomplete()**
  Use this function to load matching taxonomy terms from a given vocabulary. Autocomplete path: taxonomy/autocomplete.

If one of these matches your needs, then all you need to do is include the special #autocomplete_path selector in a form field. Here's an example for user autocomplete (from comment.module):

```php
<?php
$form['admin']['author'] = array(  
'#type' => 'textfield',  
'#parents' => array('author'),  
'#title' => t('Authored by'),  
'#size' => 30,  
'#maxlength' => 60,  
'#autocomplete_path' => 'user/autocomplete',  
'#default_value' => $author,  
'#weight' => -1);
?>
```

For taxonomy autocomplete, you include a vocabulary id, as in this example from taxonomy.module:

```php
<?php
$form['taxonomy']['tags'][$vocabulary->vid] = array(  
'#type' => 'textfield',  
'#default_value' => $typed_string,  
'#maxlength' => 100,  
'#autocomplete_path' => 'taxonomy/autocomplete/'. $vocabulary->vid,  
'#required' => $vocabulary->required,  
'#title' => $vocabulary->name,  
'#description' => t('A comma-separated list of terms describing this content (Example: funny, bungie jumping, "Company, Inc.").'));
?>
```

Building a custom autocomplete function

If you want to make your own autocomplete function to answer a need not already met, there are a couple of additional steps.

- Write a handler function. This will receive an autocomplete request and return data to the client in a form ready to be parsed by automplete.js.

  ```php
  user_autocomplete() is a good model:
  ```
  ```php
  /**
   * Retrieve a pipe delimited string of autocomplete suggestions for existing users
   */
  ```
function user_autocomplete($string) {
    $matches = array();
    $result = db_query_range("SELECT name FROM {users} WHERE LOWER(name) LIKE LOWER('%s%%')", $string, 0, 10);
    while ($user = db_fetch_object($result)) {
        $matches[$user->name] = check_plain($user->name);
    }
    print drupal_implode_autocomplete($matches);
    exit();
}

Note that you are (a) finding matching records based on user input (b) constructing an array of matches (c) imploding the array in a special parsable way and (d) outputting the results.

Also note that we need to escape the user name in the $matches array’s values: this is because the values are HTML. Not escaping would open up XSS holes. On the other hand, it also means that you can mark-up the autocomplete suggestions any way you like (for example, by adding a small user picture to each match).

• Create a menu path to your function. Users need to be able to reach your handler. The user_menu() lines are:
<?php
    $items[] = array('path' => 'user/autocomplete', 'title' => t('user autocomplete'), 'callback' => 'user_autocomplete', 'access' => $view_access, 'type' => MENU_CALLBACK);
?>

• Reference your path in form textfields. Having put a handler in place, you can now reference it as an autocomplete path in form fields, as with the existing user/autocomplete and taxonomy/autocomplete handlers.

Security note: make sure your autocomplete handler has appropriate menu permissions set on it, and respects existing access control mechanisms. Otherwise, you might be exposing sensitive information through the autocomplete.

Tutorial 3: Creating new widgets with AJAX

AJAX is a catchy acronym used for Javascript applications that dynamically load data from a server, enabling the updating of content without fully refreshing a page.

In Drupal, AJAX functionality is provided through functions in the Javascript file drupal.js. Different web browsers handle client-server communications differently; these functions provide cross-browser methods.
Some AJAX widgets (autocomplete, progress) ship with Drupal, see [Tutorial 2](#).

How you build your own AJAX widgets will of course depend a lot on what you’re wanting to do. But here are some basic steps to get you started.

### AJAX solution components

At their most basic, AJAX widgets will have three components.

- **Caller**
  Page element(s) that call an AJAX update, generally as a result of a user action (click, mouseover, etc.).

- **PHP handler**
  A PHP function that receives the user input and returns data to the caller.

- **JS handler**
  A Javascript function that receives the PHP response and acts on it (e.g., updates user display).

### Example: click_info with AJAX

In [Tutorial 1](#) we coded a stunningly useless module for popping up messages when users click on select words. Here, we’ll extend that example to load dynamically the message text from the server using - you guessed it - AJAX.

If you haven’t already read it, you might want to glance over [Tutorial 1](#) first.

So here it is, AJAX in three easy steps.

#### 1. Marking up content

As with non-AJAX Javascripting, we want to begin with plain HTML elements, to which behaviours will be attached dynamically after the page loads. The only difference here is that the elements need to have a way of directing the Javascript to the appropriate path on the server. That is, they need to pass to the Javascript the path to post requests to, which will lead to the PHP handler (see below).

Let’s call the path of our PHP handler `click_info_ajax/example_handler`.

In Tutorial 1 we wrote a function that added a CSS class selector and a special `info` attribute to `span` elements.

```php
<?php
function click_info_make($text, $words) {
    foreach ($words as $word => $message) {
        $text = str_replace ($word, '<span class="clickInfo" t="' . $message . '"><span class="clickInfo" t="' . $word . '</span>', $text);
    }
    return $text;
}
```
Here we’ll do something similar, only we’ll attach the PHP handler path instead, with the word appended to the uri (so it can later be fed into the Javascript as an argument). Note that we generate the uri with `url()`, so that it will work whether or not clean urls are enabled. (We’re using `substr_replace()` instead of `str_replace()` so the behaviour will be attached only to the first occurrence of the word.)

```php
<?php
/**
 * Add spans to words.
 */
function click_info_ajax_make($text, $words) {
    foreach ($words as $word) {
        $pos = strpos($text, $word);
        if ($pos === false) {
            continue;
        }
        $text = substr_replace($text, '<span class="clickInfo" title="'.url('click_info_ajax/example_handler/').$word.'">').'</span>', $pos, strlen($word));
    }
    return $text;
}
?>
```

Now the Javascript will have what it needs to post a request to the server.

2. The Javascript

Step 2 is writing Javascript to post information to the server, and to interpret the response.

**Objects and methods**

To attach our new behaviour, we’re going to create a new object type, ”click info data base” (CIDB, for short) and assign an object instance to our page element. The AJAX functionality will be methods of this new object. This approach might sound complicated at first, but, don’t worry, it’s actually straightforward.

We’ll start with the `click_info.js` file we wrote in Tutorial 1.

```javascript
if (isJsEnabled()) {
    addLoadEvent(clickInfoAutoAttach);
}

function clickInfoAutoAttach() {
    var spans = document.getElementsByTagName('span');
    for (var i = 0; span = spans[i]; i++) {
        if (span && hasClass(span, 'clickInfo')) {
```
// Read in the message from the 'title' attribute
span.message = span.getAttribute('title');
// Remove the title, so no tooltip will display
span.removeAttribute('title');
span.onclick = function() {
    alert(this.message);
}
}
}

In that case we were popping up text loaded from the element itself. Now, we’ll get the data from the server instead. So, instead of attaching a behaviour directly to the element (a span), we’ll create a new object type and use that object to do both the behaviour attaching and the AJAX handling.

if (isJsEnabled()) {
    addLoadEvent(clickInfoAutoAttach);
}

function clickInfoAutoAttach() {
    var cidb = [];
    var spans = document.getElementsByTagName('span');
    for (var i = 0; span = spans[i]; i++) {
        if (span && hasClass(span, 'clickInfo')) {
            // Read in the path to the PHP handler
            uri = span.getAttribute('title');
            // Remove the title, so no tooltip will display
            span.removeAttribute('title');
            // Create an object with this uri. Because
            // we feed in the span as an argument, we’ll be able
            // to attach events to this element.
            if (!cidb[uri]) {
                cidb[uri] = new CIDB(span, uri);
            }
        }
    }
}

Now we need to define the CIDB object type and give it AJAX methods--the ability to send requests (using a function defined in drupal.js and to receive and act on responses.

We declare a new object type in Javascript simply by creating a function and setting properties and/or methods. In this case, we’ll add a method that calls the drupal.js function HTTPGet().

Like its pair HTTPPost() (used when you want to use a Post rather than a Get operation), HTTPGet() provides a cross-browser method for posting data to the server. It takes three arguments: the uri being requested, the method that should be called when data is returned, and a reference to the calling object (so that the return data can be linked with the correct object instance).

We can use the prototype method to add a new method to the database object--receive, which will handle the data returned by the server.
function CIDB(elt, uri) {
    var db = this;
    // By making the span element a property of this object,
    // we get the ability to attach behaviours to that element.
    this.elt = elt;
    this.uri = uri;
    this.elt.onclick = function() {
        HTTPGet(db.uri, db.receive, db);
    }
}

function CIDB.prototype.receive = function(string, xmlhttp, cidb) {
    if (xmlhttp.status != 200) {
        return alert('An HTTP error ' + xmlhttp.status + ' occurred.
' + cidb.uri);
    }
    // We have access to the span element, since it's an attribute of the cidb object.
    // Remove the 'clickInfo' class, to show that this is already clicked.
    // We do this with another of the functions in drupal.js
    removeClass(cidb.elt, 'clickInfo');
    alert(string);
}

3. The Handler

To handle the requests sent by AJAX clients, you need at least two pieces of code in your
module: (1) a request handler function, and (2) a menu item allowing access to the handler.

The PHP handler is simply a function that receives input and returns data. In some cases you
might wish to return XML or some other sort of encoded data to be parsed by your Javascript
(see user_autocomplete() for an example), or fully formed HTML elements to be appended
to user display.

In our case, all we need is a phrase that's going to be output.

<?php
function click_info_ajax_example_handler() {
    // We've appended the word in question to the uri as the third
    // argument.
    $string = arg(2);
    switch ($string) {
    case 'Drupal':
        print t('Great software!');
        break;
    default:
        print t('Nothing');
    }
    exit();
}
<?php
header("Cache-Control: no-cache, must-revalidate");
header("Expires: Mon, 26 Jul 1997 05:00:00 GMT");
?>

Now make sure users can reach your handler:

<?php
/**
 * Implementation of hook_menu
 */
function click_info_ajax_menu($may_cache) {
    $items = array();
    if ($may_cache) {
        $items[] = array(
            'path' => 'click_info_ajax/example_handler',
            'title' => t('click info example'),
            'access' => user_access('access content'),
            'type' => MENU_CALLBACK,
            'callback' => 'click_info_ajax_example_handler'
        );
    }
    return $items;
}
?>

Now ensure you send the JS and CSS files as needed (see the section "Send the needed files" in Tutorial 1), and you’re away.

See the completed module To use it:

- Install and enable the module
- Create a new node (e.g., a ‘story’), and include the word Drupal.
- View the page. The word Drupal should be highlighted. Click on it to get the alert ‘Great software!’.
More examples

Of course, this tutorial has only scratched the surface of what can be done with AJAX. To find out more, study the AJAX widgets that ship with Drupal (e.g., autocomplete.js), or contributed modules using AJAX. Examples include chat_window.js and autorefresh.

Happy coding!

Additional tools and approaches

Aside from the Javascript and AJAX approaches used in Drupal core, there are other approaches for introducing "Web 2.0" functionality into Drupal.

See the new S/P Ajax module which introduces effects using the increasingly popular Scriptaculous library, built on Prototype. S/P Ajax includes numerous effects, including draggable page elements, pulsing message, and even expanding menus (through the helper module S/P Magic Menus).

See also How to use ajax for your modules (with Xajax) for tips on using the Xajax toolkit.

Using the APIs available through contributed modules

An application program interface (API) is a set of methods and routines for building software applications. As well as the core APIs (the node, user, and taxonomy systems, etc.), there are several important APIs made available through contributed modules. Using these contributed APIs can both greatly ease your module development and also increase interoperability with other Drupal modules.

Views

The Views module provides a flexible method to control how lists of posts are retrieved and presented. See the Views Module Developer API for detailed instructions on how to expose your module’s tables and fields to views operations.

Actions and Workflows

The actions module allows the configuration of Drupal actions. A Drupal action is a specially written PHP function whose parameters are configured through the web. For example, the Send Email action has parameters Recipient, Subject, and Message.

The workflow module allows the creation and assignment of arbitrary workflows to Drupal node types. Workflows are made up of workflow states. For example, a workflow with the states Draft, Review, and Published could be assigned to the Story node type.
Together, actions and workflows provide powerful and handy module implementation tools.

**E-Commerce**

E-Commerce comes with a developed API to facilitate extension of the suite of modules. The developer documentation provides a quick introduction to the API designed to get you started developing E-Commerce solutions.

**Location**

The Location module provides geographic functionality related to addresses, and includes an API for working with and extending the included functionality. For developer documentation, see the files location_API.txt and extending_support.txt included with the Location distribution.

**Writing .install files**

As of version 4.7, modules authors can use .install files to do module setup work.

A .install file is run the first time a module is enabled, and is used to do setup required by the module. The most common task is creating database tables and fields (which prior to version 4.7 was done manually).

**Install instructions**

Install instructions are enclosed in a _install() function. An typical example, creating a module table:

```
<?php

function nodereference_install() {
    switch ($GLOBALS['db_type']) {
        case 'mysql':
            db_query("CREATE TABLE {node_field_nodereference_data} (
                vid int unsigned NOT NULL default '0',
                field_name varchar(32) NOT NULL default '',
                delta int unsigned NOT NULL default '0',
                field_nid int unsigned NOT NULL default '0',
                PRIMARY KEY (vid,field_name,delta)
            ) /*!40100 DEFAULT CHARACTER SET utf8 */;");
            break;
        case 'pgsql':
            db_query("CREATE TABLE {node_field_nodereference_data} (
                vid integer unsigned NOT NULL default '0',
                field_name varchar(32) NOT NULL default '',
                delta integer unsigned NOT NULL default '0',
                field_nid integer unsigned NOT NULL default '0',
            ");
```
Update instructions

`install` files can also include update instructions, in `_update_x()` functions (where x is an incrementing integer). Example:

```php
function tinymce_update_1() {
    return _system_update_utf8(array('tinymce_settings',
                                      'tinymce_role'));
}
?>
```

This update calls a system function that converts tables to the UTF-8 character set.

**Drupal’s menu building mechanism**

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

menu_execute_active_handler()
menu_get_menu()
menu_build()

'menu' callback fills _menu array

menu table overlays _menu array
database

add custom items
place parent and child items
build visible tree using access permissions
parse path from q variable

_pathindex array
pathindex integer
visible array
menu id array
children array
path
title

_items array
menu id array
'path'
'title'
'type'
access
pid
weight
callback
callback arguments
children array

path in pathindex?

permission to view path?

marshall callback arguments and additional parameters
call_user_func_array()
This continues our examination of how Drupal serves pages. We are looking specifically at how the menu system works and is built, from a technical perspective. See the excellent overview in the menu system documentation.

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

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>path</td>
<td>string</td>
<td>the partial URL to the page for this menu item</td>
</tr>
<tr>
<td>title</td>
<td>string</td>
<td>the title that this menu item will have in the menu</td>
</tr>
<tr>
<td>type</td>
<td>integer</td>
<td>a constant denoting the menu item type (see comments in menu.inc)</td>
</tr>
<tr>
<td>access</td>
<td>boolean</td>
<td></td>
</tr>
<tr>
<td>pid</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>weight</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>callback</td>
<td>string</td>
<td>name of the function to be called if this menu item is selected</td>
</tr>
<tr>
<td>callback arguments</td>
<td>array</td>
<td></td>
</tr>
</tbody>
</table>

An array called $menu_item_list is populated by sending a ‘menu’ callback to all modules with ‘menu’ hooks (that is, they have a function called foo_menu() where foo is the name of the module). So each module has a chance to register its own menu items. It is interesting that when the node module receives the menu callback through node_menu(), and the path is something like ‘node/1’ as it is in our present case, the complete node is actually loaded via the node_load() function so it can be examined for permissions. The $node variable into which it was loaded then goes out of scope, so the node is gone and needs to be rebuilt completely later on. This seems like a golden opportunity for the node module to cache the node.

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

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

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

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

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

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

**Drupal’s node building mechanism**

(This walkthrough done on pre-4.5 CVS code in August 2004.)
The node_page controller checks for a $_POST['op'] entry and, failing that, sets $op to arg(1) which in this case is the ‘1’ in node/1. A numeric $op is set to arg(2) if arg(2) exists, but in this
case it doesn’t (‘1’ is the end of the URL, remember?) so the $op is hardcoded to ‘view’. Thus, we succeed in the ‘view’ case of the switch statement, and are shunted over to node_load(). The function node_load() takes two arguments, $conditions (an array with nid set to desired node id -- other conditions can be defined to further restrict the upcoming database query) for which we use arg(1), and $revision, for which we use _GET['revision']. The ‘revision’ key of the _GET array is unset so we need to make brief stop at error_handler because of an undefined index error. That doesn’t stop us, though, and we continue pell-mell into node_load using the default $revision of -1 (that is, the current revision). The actual query that ends up being run is

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

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

We now have a complete node that looks like the following:
All of the above are strings except the roles array.

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

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

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

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

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

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

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

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

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

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

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

**Drupal’s page serving mechanism**

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

http://127.0.0.1/~vandyk/drupal/?q=node/1
A visual companion to this narration can be found here; you may want to print it out and follow along. Before we start, let’s dissect the URL. I’m running on an OS X machine, so the site I’m serving lives at /Users/vandyk/Sites/. The drupal directory contains a checkout of the latest Drupal CVS tree. It looks like this:

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

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

So, let’s pick up the show with the execution of index.php which looks very simple and is only a few lines long.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The actual check for suspicious input data is carried out by valid_input_data() which lives
in common.inc. It simply goes through an array it’s been handed (in this case the $_REQUEST
array) and checks all keys and values for the following "evil" strings: javascript, expression, alert,
dynsrc, datasrc, data, lowsrc, applet, script, object, style, embed, form, blink, meta, html, frame,
iframe, layer, ilayer, head, frameset, xml. If any of these are matched watchdog records a
warning and Drupal dies (in the PHP sense). I wondered why both the keys and values of the
$_REQUEST array are examined. This seems very time-consuming. Also, would it die if my URL
ended with "/?xml=true" or "/?format=xml"?
The next step in common.inc’s executable code is a call to `locale_init()` to set up locale data. If the user is not an anonymous user and has a language preference set up, the two-character language key is returned; otherwise, the key of the single-entry global array $language is returned. In our case, that’s "en".

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

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

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

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

We’re done.
Updating your modules

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

Converting 3.0 modules to 4.0

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

Drupal 3.0:

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

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

Drupal 4.0:

```php
function form($form, $method = "post", $action = 0, $options = 0)
// Example
$form = form_hidden("nid", $nid);
print form($form);
```

Converting 4.0 modules to 4.1

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

Required changes

Modified block hook:

Drupal 4.0:

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

Modified taxonomy API:

Changes: in function taxonomy_get_tree()

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

Drupal 4.0:

function taxonomy_get_tree($vocabulary_id, &$tree, $parent = 0, $depth = -1, $key = "tid")

Drupal 4.1:

$tree = taxonomy_get_tree($vocabulary_id, $parents = 0, $depth = -1, $key = "tid")

Optional changes

- Take advantage of new taxonomy functions
taxonomy_get_vocabulary_by_name($name) and
taxonomy_get_term_by_name($name)
- Take advantage of pager functions
- Move hardcoded markup from modules to themes, using theme_invoke
Converting 4.1 modules to 4.2

Some points [posted by Axel] on [drupal-devel] on migrating 4.1.0 modules to CVS [updated and added to by ax]:

- the big clean URL patch: Over the weekend, [dries] bit the bullet and converted every single URL in Drupal's code. meaning we'll [can] have clean URLs like http://foo.com/archive/2003/01/06, http://foo.com/user/42, http://foo.com/blog and so on. meaning, for the code:
  
  - drupal_url(array("mod" => "search", "op" => "bla"), "module", $anchor = ")
    became
    url("search/bla")
    with the first url part being the module, the second (typically) being the operation (op); more arguments are handled differently per module convention.
  
  - l("view node", array("op" => "view", "id" => $nid), "node", $anchor = ", $attributes = array())
    became
    l("view node", "node/view/$nid", $attributes = array(), $query = NULL)

- similar,
  
  - lm(), which meant "module link" and used to be
    module.php?mod=bla&op=blub..., is now l("title", "bla/blub/..."); and
  
  - la(), which meant "admin link" and used to be admin.php?mod=bla&op=blub..., is now l("title", "admin/bla/blub/..."

- After fixing those functions, you'll need to edit your _page() function and possibly others so that they get their arguments using the arg() function (see includes/common.inc. These arguments used to be globals called "mod", "op", "id" etc. now these same arguments must be accessed as arg(1), arg(3), for example.

- $theme->function() became theme("function"). see [drupal-devel] renaming 2 functions [drupal-devel] theme("function") vs $theme->function()and [drupal-devel] [CVS] theme()

- &lt;module&gt;_conf_options() became &lt;module&gt;_settings() - see [drupal-devel] renaming 2 functions note that doesn't get an extra menu entry, but is accessed via "site configuration > modules > modules settings"

- the administration pages got changed quite a lot to use a "database driven link system" and become more logical/intuitive - see [drupal-devel] X-mas commit: administration pages this first try resulted in poor performance and a not-so-good api, so it got refactored - see [PATCH] menus this, as of time ax is writing this, isn’t really satisfying, neither (you cannot build arbitrary menu-trees, some forms don’t work (taxonomy > add term), ...), so it probably will change again. and i won’t write more about this here.

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

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

• [from comment_is_new function lost]
  - comment_is_new($comment)
  + node_is_new($comment-&gt;nid, $comment-&gt;timestamp)

please add / update / correct!

Converting 4.2 modules to 4.3

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

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

so that the table prefix can be dynamically prepended to the table name. See the original [feature request] and the corresponding discussion at the mailing list for details.

New help system
From Michael Frankowski message:

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

If the URL of the admin page matches a URL in a _help hook then the text from that _help hook is displayed on the top of the admin page. If there is no match, the block it not displayed. Because Drupal matches URLs in order to stick "other" stuff in the _help hook we have taken to sticking descriptors after a "#" sign. So far, the following descriptors are recognised:
<table>
<thead>
<tr>
<th>Descriptor</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>admin/system/modules#name</td>
<td>The name of a module (unused, but there)</td>
</tr>
<tr>
<td>admin/system/modules#description</td>
<td>The description found on the admin/system/modules page.</td>
</tr>
<tr>
<td>admin/help#modulename</td>
<td>The module’s help text, displayed on the admin/help page and through the module’s individual help link.</td>
</tr>
<tr>
<td>user/help#modulename</td>
<td>The help for a distributed authorization module</td>
</tr>
</tbody>
</table>

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

**Creating modules for version 4.3.1**

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

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

This tutorial assumes the following about you:

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

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

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

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

```php
<?php

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

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

**Telling Drupal about your module**

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

```php
<?php
function onthisdate_help($section) {
}
?>
```

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

```php
/* Commented out until bug fixed */

function onthisdate_help($section) {
    switch($section) {
        case "admin/system/modules#name":
            $output = "onthisdate";
            break;
        case "admin/system/modules#description":
            $output = "Display a list of nodes that were created a week ago.";
```
You will eventually want to add other cases to this switch statement to provide real help messages to the user. In particular, output for "admin/help#onthisdate" will display on the main help page accessed by the admin/help URL for this module (/admin/help or ?q=admin/help).

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

**Telling Drupal who can use your module**

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

```php
<?php
function onthisdate_perm() {
    return array("administer onthisdate");
}
?>
```

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

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

You’ll need to adjust who has permission to view your module on the administer » accounts » permissions page. We’ll use the user_access function to check access permissions later.
Be sure your permission strings must be unique to your module. If they are not, the permissions page will list the same permission multiple times.

**Announce we have block content**

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

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

```php
function onthisdate_block($op='list', $delta=0) {
}
?>
```

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

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

**Generate content for a block**

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

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

First, we need to calculate the time (in seconds since epoch start, see
time format) for midnight a week ago, and 11:59pm a week ago. This part
of the code is Drupal independent, see the PHP website ([http://php.net/](http://php.net/))
for more details.

```php
<?php
function onthisdate_block($op='list', $delta=0) {
    // listing of blocks, such as on the admin/system/block page
    if ($op == "list") {
        $block[0]["info"] = t("On This Date");
        return $block;
    } else {
        // our block content
        // Get today’s date
        $today = getdate();
        // calculate midnight one week ago
        $start_time = mktime(0, 0, 0,
                             $today['mon'], ($today['mday'] - 7),
                             $today['year']);
        // we want items that occur only on the day in question, so
        calculate 1 day
        $end_time = $start_time + 86400; // 60 * 60 * 24 = 86400 seconds
        in a day
        ...
    }
}
?>
```

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

**Note:** the table name is enclosed in curly braces: `{node}`.
This is necessary so that your module will support database table name
prefixes. You can find more information on the Drupal website by
reading the [Table Prefix (and sharing tables across instances)](http://www.php.net/) page in
the Drupal handbook.
<?php
$query = "SELECT nid, title, created FROM ".
 "{node} WHERE created >= ". $start_time . 
 "' AND created <= ". $end_time . "'";

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

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

```php
// get the links
$queryResult = db_query($query);
// content variable that will be returned for display
$block_content = '';
while ($links = db_fetch_object($queryResult)) {
    $block_content .= '<a href="'.url('node/view/').$links->nid .'">'.
    '.$links->title .'</a><br />';
}
// check to see if there was any content before setting up the block
if ($block_content == '') {
    /* No content from a week ago. If we return nothing, the block
     * doesn’t show, which is what we want. */
    return;
}
// set up the block
$block['subject'] = 'On This Date';
$block['content'] = $block_content;
return $block;
?>
```

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

Also, we return an array that has ‘subject’ and ‘content’ elements. This is what Drupal expects from a block function. If you do not include both of these, the block will not render properly.
You may also notice the bad coding practice of combining content with layout. If you are writing a module for others to use, you will want to provide an easy way for others (in particular, non-programmers) to adjust the content’s layout. An easy way to do this is to include a class attribute in your link, and not necessarily include the <br /> at the end of the link. Let’s ignore this for now, but be aware of this issue when writing modules that others will use.

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

```php
<?php
function onthisdate_block($op='list', $delta=0) {
    // listing of blocks, such as on the admin/system/block page
    if ($op == "list") {
        $block[0]["info"] = t("On This Date");
        return $block;
    } else {
        // our block content
        // content variable that will be returned for display
        $block_content = "";
        // Get today’s date
        $today = getdate();
        // calculate midnight one week ago
        $start_time = mktime(0, 0, 0, $today["mon"], ($today["mday"] - 7), $today["year"]);
        // we want items that occur only on the day in question, so calculate 1 day
        $end_time = $start_time + 86400; // 60 * 60 * 24 = 86400 seconds in a day
        $query = "SELECT nid, title, created FROM " . 
            "{node} WHERE created >= " . $start_time . " AND created <= " . $end_time . ";"
        // get the links
        $queryResult = db_query($query);
        while ($links = db_fetch_object($queryResult)) {
            $block_content .= '<a href="' . url('node/view/' . $links->nid).'">' . 
                $links->title . '</a><br />
        }
        // check to see if there was any content before setting up the block
        if ($block_content == "") {
            // no content from a week ago, return nothing.
            return;
        }
        // set up the block
    }
}
```
Installing, enabling and testing the module

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

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

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

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

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

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

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

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

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

Enable the module by selecting the enabled checkbox for the ‘On This Date’ block and save your blocks. Be sure to adjust the location (left/right) if you are using a theme that limits where blocks are displayed.
Now, head to another page, say select the module. In some themes, the blocks are displayed after the page has rendered the content, and you won’t see the change until you go to new page.

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

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

### Create a module configuration (settings) page

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

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

```php
<?php
function onthisdate_settings() {
    // only administrators can access this module
    if (!user_access("admin onthisdate")) {
        return message_access();
    }
}
?>
```

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

```php
<?php
...
// check the user has content access
if (!user_access("access content")) {
    return message_access();
}
...
?>
```
We’d like to configure how many links display in the block, so we’ll create a form for the administrator to set the number of links:

```php
function onthisdate_settings() {
    // only administrators can access this module
    if (!user_access("admin onthisdate")) {
        return message_access();
    }
    $output .= form_textfield(t("Maximum number of links"), "onthisdate_maxdisp",
        variable_get("onthisdate_maxdisp", "3"), 2, 2,
        t("The maximum number of links to display in the block."));
    return $output;
}
?>
```

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

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

```php
<?php
    $limitnum = variable_get("onthisdate_maxdisp", 3);
    $query = "SELECT nid, title, created FROM " .
        "{node} WHERE created >= ‘" . $start_time .
        "’ AND created <= ‘" . $end_time . "’ LIMIT " . $limitnum;
?>
```

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

Navigate to the settings page: admin/system/modules/onthisdate or administer » configuration » modules » onthisdate. Adjust the number of links and save the configuration. Notice the number of links in the block adjusts accordingly.
Note: We don’t have any validation with this input. If you enter "c" in the maximum number of links, you’ll break the block.

Adding menu links and creating page content

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

```php
function onthisdate_all() {
    // content variable that will be returned for display
    $page_content = '';
    // Get today’s date
    $today = gettimeofday();
    // calculate midnight one week ago
    $start_time = mktime(0, 0, 0, $today['mon'], ($today['mday'] - 7), $today['year']);
    $today['year']);
    // we want items that occur only on the day in question, so calculate 1 day
    $end_time = $start_time + 86400; // 60 * 60 * 24 = 86400 seconds in a day
    // NOTE! No LIMIT clause here! We want to show all the code
    $query = "SELECT nid, title, created FROM {node} WHERE created >= '".$start_time.' AND created <= '".$end_time.'";
    // get the links
    $queryResult = db_query($query);
    while ($links = db_fetch_object($queryResult)) {
        $page_content .= '<a href="'.url('node/view/'.$links->nid)."">".$links->title . '</a><br />
    }
...
```
We have the page content at this point, but we want to do a little more with it than just return it. When creating pages, we need to send the page content to the theme for proper rendering. We use this with the theme() function. Themes control the look of a site. As noted above, we’re including layout in the code. This is bad, and should be avoided. It is, however, the topic of another tutorial, so for now, we’ll include the formatting in our content:

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

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

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

```php
<?php
function onthisdate_all() {
...
    // check to see if there was any content before setting up the block
    if ($page_content == '') {
        // no content from a week ago, let the user know
        print theme("page",
            "No events occurred on this site on this date in history.");
        return;
    }
    print theme("page", $page_content);
}
?>
```

**Letting Drupal know about the new function**

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

```php
<?php
function onthisdate_link($type, $node=0) {
}
```
There are many different types, but we’re going to use only ‘system’ in this tutorial.

```
<?php
function onthisdate_link($type, $node=0) {
    if (($type == "system")) {
        // URL, page title, func called for page content, arg, 1 = don’t disp menu
        menu("onthisdate", t("On This Date"), "onthisdate_all", 1, 1);
    }
}
?>
```

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

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

**Adding a more link and showing all entries**

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

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

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

This will add the more link.

**Conclusion**

We now have a working module. It created a block and a page. You should now have enough to get started writing your own modules. We recommend you start with a block module of your own and move onto a node module. Alternately, you can write a filter or theme.
As is, this tutorial’s module isn’t very useful. However, with a few enhancements, it can be entertaining. Try modifying the select query statement to select only nodes of type ‘blog’ and see what you get. Alternatively, you could get only a particular user’s content for a specific week. Instead of using the block function, consider expanding the menu and page functions, adding menus to specific entries or dates, or using the menu callback arguments to adjust what year you look at the content from.

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

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

**How to build up a _help hook**

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

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

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

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

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

If you also want this same text displayed for an individual help link in your menu area. You have this kind of tree:
+ Administration
| -> Your area
| | |
| -> Your configuration
| | -> help
| |
-> Overall admin help.

Change the function line to this:

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

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

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

### How to convert a _system hook

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>$field == &quot;name&quot;</td>
<td>The module name</td>
</tr>
<tr>
<td>$field == &quot;description&quot;</td>
<td>The description placed in the module list</td>
</tr>
<tr>
<td>$field == &quot;admin-help&quot;</td>
<td>The help text placed at the TOP of this module’s individual configuration area.</td>
</tr>
</tbody>
</table>

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

- If `$system is $system["name"]` then the case is case 'admin/system/modules#name'
- If `$system is $system["description"]` then case is case 'admin/system/modules#description'
- If `$system is $system["admin-help"]` then the case is case 'admin/system/modules/<modulename>'
Now remove the _system function and you are done.

An example:

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

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

How to convert an _auth_help hook

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

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

1. The text inside the _auth_help hook needs to be moved inside the _help hook under the section user/help#<modulename> and
2. You have to change the _page hook, which normally displays that help text, to find your text in a new location by changing the function call <modulename>_auth_help() to
See, it is not THAT terrible.

An example:

```php
function exampleda_page() {
    theme("header");
    theme("box", "Example DA", exampleda_auth_help());
    theme("footer");
}
function exampleda_auth_help() {
    $site = variable_get("site_name", "this web site");
    $html_output = "
        <p>This is my example Distributed Auth help. Using this example you cannot login to <i>%s</i> because it has no _auth hook.&</p>
        <p><u>BUT</u> you should still use Drupal since it is a <b>GREAT</b> CMS and is only getting better.</p>
        <p>To learn about about Drupal you can <a href="www.drupal.org">visit the site</a></p>
    
    return sprintf(t($html_output), $site);
}
```
Converting 4.3 modules to 4.4

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

Menu system

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

The flow of page generation now proceeds as follows:

1. The _link hook in all modules is called, so that modules can use menu() to add items to the menu. For example, a module could define:
   ```php
   function example_link($type) {
       if ($type == "system") {
           menu("example", t("example"), "example_page");
           menu("example/foo", t("foo"), "example_foo");
       }
   }
   ```
2. The menu system examines the current URL, and finds the "best fit" for the URL in the menu. For example, if the current URL is example/foo/bar/12, the above menu() calls would cause example_foo("bar", 12) to get invoked.
3. The callback may set the title or breadcrumb trail if the defaults are not satisfactory (more on this later).
4. The callback is responsible for printing the requested page. This will usually involve preparing the content, and then printing the return value of theme("page"). For example:
   ```php
   function example_foo($theString, $theNumber) {
       $output = $theString . " - " . $theNumber;
       print theme("page", $output);
   }
   ```

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

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

The trailing MENU_HIDE argument in this call makes the menu item hidden, so the callback functions but the module does not clutter the user menu.

- Old administrative callbacks returned their content. In the new system, administrative and normal callbacks alike are responsible for printing the entire page.
- The title of the page is printed by the theme system, so page content does not need to be wrapped in a theme("box") to get a title printed. If the default title is not satisfactory, it can be changed by calling drupal_set_title($title) before theme("page") gets called, or by passing the title to theme("page") as a parameter.
- The breadcrumb trail is also printed by the theme. If the default one needs to be overridden (to present things like forum hierarchies), this can be done by calling drupal_set_breadcrumb($breadcrumb) before theme("page") gets called, or by passing the breadcrumb to theme("page") as a parameter. $breadcrumb should be a list of links beginning with "Home" and proceeding up to, but not including, the current page.

### Theme system

For full information on theme system changes, see [converting 4.3 themes to CVS]. The following points are directly relevant to module development:

- All theme functions now return their output instead of printing them to the user. Old theme() usage:
  ```php
  <?php
  theme("box", $title, $output);
  ?>
  ```

  New usage:
  ```php
  <?php
  print theme("box", $title, $output);
  ?>
  ```

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

- The naming of theme functions defined by modules has been standardized to theme_&lt;module&gt;_&lt;name&gt;. When using a theme function there is no need to include the theme_part, as theme() will do this automatically. Example:
  ```php
  <?php
  function theme_example_list($list) {
    return implode('<br />', $list);
  }
  print theme('example_list', array(1,2,3));
  ?>
  ```

  Theme functions must always be called using theme() to allow for the active theme to modify the output if necessary.
The theme("header") and theme("footer") functions are not available anymore. Module developers should use the theme("page") function which wraps the content in the site theme. The full syntax of this function is

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

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

**Node system**

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

- The _node() hook has been deprecated. In its place, modules that define nodes should use _node_name() and _help().
- The _node_name() function should return a translated string containing the human-readable name of the node type.
- The _help() function, when called with parameter "node/add#modulename", should return a translated string containing the description of the node type.
- Modules wishing to use the new ability to define multiple node types should see the Doxygen documentation for hook_node_name() and hook_node_types().

**Filter system**

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

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

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

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

If your filter currently performs such a step in the main "process" step, it should be moved into "prepare" instead. If you don’t need any escaping, your filter should simply return $text without processing in this case.

- "process" is the equivalent of the old "filter" hook. Normal filtering is performed here, and the changed $text is returned.
- "settings" is the equivalent of the old "conf_filters" hook. If your filter provides configurable options, you should return them here (using the standard form_* functions).

- The filter handling code has been moved to a new required filter.module, and thus most of the filter function names changed, although none of those should have been called from modules. The check_output() function is still available with the same functionality.
- Node filtering is optimized with the node_prepare() function now, which only runs the body through the filters if the node view page is displayed. Otherwise, only the teaser is filtered.
- The _compose_tips hook (defined by the contrib compose_tips.module) is not supported anymore, but more advanced functionality exists in the core. You can emit extensive compose tips related to the filter you define via the _help hook with the 'filter#long-tip' section identifier. The compose_tips URL is thus changed to filter/tips. The form_allowed_tags_text() function is replaced with filter_tips_short(), which now supports short tips to be placed under textareas. Any module can inject short tips about the filter defined via the _help hook, with the 'filter#short-tip' section identifier.

### Hook changes

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

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

```php
<?php
function example_view($node, $main = 0) {
    if ($main) {
        theme("node", $node, $main);
    } else {
        $breadcrumb[] = l(t("Home"), ", ");
        $breadcrumb[] = l(t("foo"), ", foo");
```
The _form hook used by node modules no longer takes 3 arguments. The second argument $help, typically used to print submission guidelines, has been removed. Instead, the help should be emitted using the module's _help hook. For examples, check the story, forum or blog module.

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

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

See also the description of the _compose_tips hook changes below.

Emitting links

The functions url() and l() take a new $fragment parameter. Calls to url() or l() that have '#' in the $url parameter need to be updated. If you don’t update such calls, Drupal’s path aliasing won’t work for URLs with # in them.

Drupal now emits relative URLs instead of absolute URLs. Contributed modules must be updated whenere an absolute url is required. For example:

- Any module that outputs an RSS feed without using node_feed() should be updated. Note: this is discouraged. please use node_feed() instead. Also modules using node_feed() should provide an absolute link in the 'link' key, if any.
- Any module which send email should be updated so that links in the email have
absolute urls instead of relative urls. You do this using a parameter in your call to \l() or \url().

**Status and error messages**

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

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

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

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

**Converting 4.4 modules to 4.5**

**Menu system**

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

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

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

As an example, the old pattern:

```php
<?php
function blog_link($type, $node = 0, $main) {
    global $user;
    if ($type == 'system') {
        menu('node/add/blog', t('blog entry'), user_access('maintain personal blog')) ? MENU_FALLTHROUGH : MENU_DENIED, 0);
        menu('blog', t('blogs'), user_access('access content') ? 'blog_page' : MENU_DENIED, 0, MENU_HIDE);
        menu('blog/'. $user->uid, t('my blog'), MENU_FALLTHROUGH, 1, MENU_SHOW, MENU_LOCKED);
```
Drupal now distinguishes between 404 (Not Found) pages and 403 (Forbidden) pages. To accommodate this, modules should abandon the practice of not declaring menu items when access is denied to them. Instead, they should set the "access" attribute of their newly-declared menu item to FALSE. This will have the effect of the menu item being hidden, and also preventing the callback from being invoked by typing in the URL. Modules may also want to take advantage of the drupal_access_denied() function, which prints a 403 page (the analogue of drupal_not_found(), which prints a 404).

Path changes

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

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

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

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

become

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

See node access rights in the Doxygen reference.
Filtering changes

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

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

Check_output() changes

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

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

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

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

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

To do this, you must:

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

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

Filter hook

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

```php
<?php
function hook_filter($op, $delta = 0, $format = -1, $text = '') {
    switch ($op) {
        case 'list':
            return array(0 => t('Filter name'));
        case 'description':
            return t("Short description of the filter’s actions.");
    }
}
```
When converting a module to 4.5, you can normally ignore the $delta parameter: it is used to have multiple filters inside one module. The 'prepare', 'process' and 'settings' operations still work the same as before, with only small changes.

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

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

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

Filter tips

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

A module’s filter tips are returned through the filter_tips hook:
<?php
function hook_filter_tips($delta, $format, $long = false) {
    if ($long) {
        return t("Long tip");
    } else {
        return t("Short tip");
    }
}
?>

As in the filter hook you can ignore the $delta parameter if you’re upgrading an existing module. If your filter’s tips depend on its settings, make sure you use $format to retrieve the setting for the current input format. $long tells you whether to return long or short tips.

**Other changes**

In addition to the above mentioned changes:

- hook_user() was changed to allow multiple pages of user profile information. The new syntax of the hook is given in the API reference. Pay particular attention to the "categories", "form", and "view" operations.

- When processing a form submission, you should use drupal_goto() to redirect to the result if the submission was accepted. This prevents a double post when people refresh their browser right after submitting. Messages set with drupal_set_message() will be saved across the redirect. If a submission was rejected, you should not use drupal_goto(), but simply print out the form along with error messages.

**Converting 4.5 modules to 4.6**

**Block system**

Every block now has a configuration page to control block-specific options. Modules which have configurations for their blocks should move those into hook_block().

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

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

The search system got a significant overhaul.

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

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

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

If you are unsure of what you need to do, please refer to the [complete search documentation](#).

Module paths

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

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

All instances of module_get_path should be renamed to drupal_get_path.

Database backend

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

Theme system

The function theme_page() no longer takes $title or $breadcrumb arguments. Set page titles using hook_menu() or, if the title must be dynamically determined, use drupal_set_title(). Set breadcrumb trails first using hook_menu(), which can be overridden with menu_set_location() and drupal_set_breadcrumb().
Watchdog messages

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

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

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

Node markers

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

Control over destination page after form processing

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

Confirmation messages

Confirmations for dangerous actions should now be presented with the theme('confirm') function for consistency. Check the function's documentation or look at some of the core modules for examples.

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

Inter module calls

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

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

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

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

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

Example:

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

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

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

note the 'f' parameter of db_rewrite_sql().

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

Text output

Drupal’s text output was audited and several escaping bugs were found. For more info, see the check_plain patch.
You need to pay attention that all user-submitted plain-text in your module is escaped using check_plain() when you output it into HTML. No escaping should be done on data that is going into the database: only escape when outputting to HTML.

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

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

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

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

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

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

**Converting XML-RPC using modules**

The first Drupal versions to use the new library are 4.5.5, 4.6.3 and 4.7. If you have a custom written Drupal module for an older version, then the following applies.

In hook_xmlrpc

```php
<?php
return array('foaf.getUrl' => array('function' => 'foaf_get_url'));
?>
```

becomes

```php
<?php
return array('foaf.get Url' => 'foaf_get_url');
?>
```
Now let’s see the handler function itself. It’s parameters are regular PHP variables now, there is nothing to process. So now you can write things like function
blogapi_blogger_get_user_info($appkey, $username, $password) instead of doing parameter processing.

Return becomes a lot simpler, too:

```php
return new xmlrpcresp(new xmlrpcval($string, 'string'));
```

becomes

```php
return $string;
```

Client side:

```php
// send an xmlrpc message to the server to get a full foaf url
$message = new xmlrpcmsg('foaf.getUrl', array(new xmlrpcval($name, 'string')));
$client = new xmlrpc_client('/xmlrpc.php', $server, 80);
if ($result && !$result->faultCode()) {
    $value = $result->value();
    $user->foaf_url = $value->scalarval();
}
```

this becomes

```php
$result = xmlrpc($server . '/xmlrpc.php', 'foaf.getUrl', $name);
if ($result !== FALSE) {
    $user->foaf_url = $result;
}
```

Back to method handlers. All the efforts that are made to make the PHP -- XML-RPC conversations transparent can’t handle date and base64 encoding.

So:

```php
$node->created = iso8601_decode($struct['dateCreated'], 1);
```

becomes

```php
$node->created = mktime($struct['dateCreated']->hour, $struct['dateCreated']->minute, $struct['dateCreated']->second, $struct['dateCreated']->month, $struct['dateCreated']->day, $struct['dateCreated']->year);
```
And if you want to return a date (either a Unix timestamp or an ISO 8601 formatted one):

```php
<?php
    return xmlrpc_date($node->created);
?>
```

otherwise you’d get a return type of int for Unix timestamp and string for ISO.

Likewise, to make the distinction between base64 and string, you need to use `xmlrpc_base64($binary_data)` your binary data.

### Converting 4.6 modules to 4.7

**Overview of Drupal API changes in 4.7**

1. new handling of return values from callbacks must know.
2. new node definition system must know.
3. `node_load()` must know.
4. `node_save()` must know.
5. `node_list()` moderately used.
6. `node titles now handled by node modules` must know
7. `node_get_module_name() -> node_get_base()` minor required change.
8. `format_name()` -> `theme('username')` minor required change.
9. `theme_table()` minor required change.
10. `check_output()` -> `check_markup()` infrequently used, security requirement.
11. XML-RPC infrequently used, backported to 4.6.3 for security reasons.
12. `taxonomy_save_vocabulary()`, `taxonomy_save_term()` advanced use, taxonomy specific.
13. `message_access()` removed.
14. Unicode string API must know.
15. `conf_url_rewrite()` rarely used.
16. revisions overhaul important for node modules.
17. Upgrading to forms API
18. `node_delete()` moderately used
19. New order of node hooks
20. `hook_nodeapi('settings', ...) replaced by form api.
21. `hook_nodeapi('form', ...) replaced by form api.
22. file_directory variables moderately used, replaced by functions
23. array2object moderately used: replace by native PHP conversion
24. `user_load()` moderately used: different return value when user not found.
25. UTF-8 SQL conversion required.
26. We no longer use the `<base>` element.
27. `hook_onload` replaced by addLoadEvent()
28. database creation instructions, updates, and other module setup code go in `install files`
29. hook_search_item was replaced by hook_search_page.

New handling of return values from callbacks

Most (menu) callbacks used print theme('page', $output); to return HTML code. To enable the reuse of any callback in blocks, sub-pages, etc callbacks are no longer expected to call theme('page', ...):

- If you return something (return $output;), then print('page', ...); is called.
- If you do not return anything, it will assume you have handled your own output.

In general, there is no longer need to use theme('page', ...); in your module.

- Impacts most modules.
- Importance: recommended.
- Compatibility: old code will still work, but it is deprecated -- you can expect it to break any time.
- Affected code: any use of theme('page', ...).

Example:

```php
<?php
// Drupal 4.6
function mymodule_admin_page(...) {
   $output .= "some content"
   print theme('page', $output);
}
// Drupal 4.7
function mymodule_admin_page(...) {
   $output .= "some content"
   return $output;
}
?>
```

node definition changes

In 4.6, you had hook_node_name and hook_node_types. Now you must implement hook_node_info if you want to create a module which defines node type(s). The implementation of the hook_node_info() needs to:

```php
<?php
   return array($type1 => array('name' => $name1, 'base' => $base1),
                 $type2 => array('name' => $name2, 'base' => $base2));
?>
```

where $type is the node type, $name is the human readable name of the type and $base is
used instead of
<hook>
for
<hook>
_load()
,
<hook>
_view()
, etc.

For example, the story module’s node_info hook looks like this:

```php
<?php
function story_node_info() {
    return array('story' => array('name' => t('story'), 'base' => 'story'));
}
?>
```

The page module’s node_info hook looks like:

```php
<?php
function page_node_info() {
    return array('page' => array('name' => t('page'), 'base' => 'page'));
}
?>
```

However, more complex node modules like the project module and the flexinode module can use the 'base' parameter to specify a different base.

The project module implements two node types, projects and issues, so it can do:

```php
<?php
function project_node_info() {
    return array(        array('project_project' => array('name' => t('project'), 'base' => 'project'),        array('project_issue' => array('name' => t('issue'), 'base' => 'project_issue')));
}
?>
```

In the flexinode module’s case there can only be one base.

Now you have `node_get_name($type)` to get the name of a node type and `node_get_base($type)` to get the base of a node type.

- Impacts **most modules**.
- Importance: **absolute must**.
- Compatibility: old code will **not work**
- Affected code: any use of `node_name` hook, `node_types`.
node_load() changes

You should write $node = node_load($nid); instead of $node = node_load(array('nid' => $nid));. The array syntax is for cases when you are not using $nid or need to add more fields.

- Impacts many modules.
- Importance: suggested.
- Compatibility: old code will still work, but is not as efficient: by passing an array, the node_load() cache is not used resulting in lower performance.
- Affected code: any use of node_load(array('nid' => $nid)); can be replaced by node_load($nid).
- Docs: node_load()

The only use of node_load() that should be changed are the ones that only pass the $nid:

```php
<?php
// Drupal 4.6
$node = node_load(array('nid' => $nid));
// Drupal 4.7
$node = node_load($nid);
?>
```

node_save() changes

node_save() now receives the $node parameter by reference, and modifies the object as needed. It has no return value anymore.

node_list() changes

node_list() became node_get_types now returns an associative array about node types. The types are now returned as the keys of this arrays (formerly they were returned as the values). The values are now the relevant human readable names.

- Impacts some modules.
- Importance: required.
- Compatibility: old code will not work as the types are now the keys of the returned array and not the values (and the function name is changed, too).
- Affected code: any use of node_list().
- Docs: node_get_type()

The most typical use is:

```php
<?php
// Drupal 4.6
```
foreach (node_list() as $type) {
    // Drupal 4.7
    foreach (node_get_types() as $type => $name) {
        ?>
        Node titles now handled by node modules
        <p>Formerly hard-coded into the node.module, titles are now handled by the individual node modules.</p>
        In many cases, the only change needed is to add a title field at the beginning of the hook_form() hook, like this:
        <pre><?php
            function modulename_form(&$node) {
                $form['title'] = array('type' => 'textfield',
                                        'title' => t('Subject'),
                                        'default_value' => $node->title,
                                        'size' => 60,
                                        'maxlength' => 128,
                                        'required' => TRUE);
            ...
        ?></pre>
        ● Impacts: all node modules.
        ● Importance: required.
        ● Compatibility: old code will not work.
        ● Affected code: hook_form() need to be edited to add the title field.

module_get_node_name deprecated

Fetching a node type’s module name is now handled by the node_get_base function.

Example:

<?php
    // Drupal 4.6
    $module = node_get_module_name($type);
    // Drupal 4.7
    $module = node_get_base($type);
?>
**format_name() renamed**

For consistency and enabling theming, `format_name()` was renamed to `theme_username()` and should be invoked using the `theme('username', ...)` API. Every use of `format_name()` must be replaced.

- **Impacts:** few modules.
- **Importance:** required.
- **Compatibility:** old code will not work.
- **Affected code:** any use of `format_name(...)` should be replaced by `theme('username', ...)`.  
- **Docs:** `format_name()` (Drupal 4.6) and `theme_username()` (Drupal 4.7).

Example:

```php
<?php
// Drupal 4.6
$output = t('by %username', array('%username' => format_name($node)));  
// Drupal 4.7
$output = t('by %username', array('%username' => theme('username', $node)));  
?>
```

**theme_table() change**

`theme('table', ...)`; sometimes was called with arguments set to NULL or an empty string (``) to indicate that there were either no rows or no header. This is no longer allowed: the `$rows` and `$header` parameters now have to be an array. If you have no rows or no header to pass to `theme('table', ...)`, you have to pass an empty array.

- **Impacts:** some modules.
- **Importance:** required.
- **Compatibility:** old code will not work if you pass NULL or an empty string.
- **Affected code:** any use of `theme('table', ...)` where you pass no rows or no header.
- **Docs:** `theme_table()`

Example:

```php
<?php
// Drupal 4.6
theme('table', '', $rows);  
// Drupal 4.7
theme('table', array(), $rows);  
?>
```
**check_output() change**

Due to a security vulnerability discovered in the filter system, we have tightened security around the `check_output()` function. The format passed to `check_output()` is now checked for access by default. If you don’t want this check, pass `FALSE` for the third parameter, `$check`. `check_output()` was renamed to `check_markup()` to enforce this change. The new syntax is:

```php
function check_markup($text, $format = FILTER_FORMAT_DEFAULT, $check = TRUE) {
    //

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

In general you will want to use `check_markup($text, $format, TRUE)` prior to validating or saving the $text and use `check_markup($text, $format, FALSE)` when viewing the $text.

Please review any use of `check_markup()` carefully!

- **Impacts** few modules.
- **Importance:** required.
- **Compatibility:** old code will not work.
- **Affected code:** any use of `check_output()` should be checked (and double checked!) before replacing it with `check_markup()`.
- **Docs:** [check_output()](Drupal 4.6), [check_markup()](Drupal 4.7) and [filter_access()](Drupal 4.7)

**XML-RPC changes**

As this happened between 4.6.2 and 4.6.3, there is a [separate guide](Drupal 4.7) for this.

- **Impacts** few modules.

**Taxonomy API change**

In order to provide more meaningful messages to the user, now you can provide your own messages when using the taxonomy APIs to create or modify terms and vocabularies. The `taxonomy_save_vocabulary()` and `taxonomy_save_term()` functions now return a status message which can be either `SAVED_NEW`, `SAVED_UPDATED` or `SAVED_DELETED`. The function will return `NULL` when there was an error. Also note that you must call these with an `$edit` array parameter which will be modified, and will include the `’vid’` key.
Impacts few modules.

Importance: required.

Compatibility: old code will not work as expected: no feedback is given to the user anymore. If you used the return value of the functions, you should be aware that this return value has changed.

Affected code: any use of `taxonomy_save_vocabulary()` or `taxonomy_save_term()`.

Docs: `taxonomy_save_vocabulary()` and `taxonomy_save_term()`

For example, this snippet shows you a typical handling of the returned status values:

```php
<?php
// Drupal 4.6
taxonomy_save_vocabulary($edit);
?>

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

**message_access() removed**

The `message_access()` function was removed. Replace all occurrences with a nice case error message that is specific to the error that has been caught.

**Unicode string API**

Drupal now provides some wrappers for doing string handling in the most language-independent fashion. The following functions are available and behave exactly as their PHP counterparts, except that they count and work with Unicode characters in UTF-8 encoding and not literal bytes: `drupal_strlen()`, `drupal_strtolower()`, `drupal_strtoupper()`, `drupal_substr()`, `drupal_ucfirst()`. 
If you use any of the original functions in your own module, you should almost certainly replace them with their Drupal counterparts. Only use the plain PHP string API when you want to work with the literal bytes.

Note that `strpos()` is not mirrored, because it can still be safely used. If you want to chop off a string at a location found by `strpos()`, you should use `substr()` instead of `drupal_substr()`.

For more information, see the API reference.

**conf_url_rewrite() became custom_url_rewrite()**

Watch out for changes in the URL rewriting process if you have a module or settings file, which implements mass URL aliasing. The `conf_url_rewrite()` you are expected to implement if you are about to provide mass URL aliasing in Drupal was renamed to `custom_url_rewrite()`. Now this function is always called on all URL aliasing request, so if you would not like to mangle already aliased URLs, you need to take action yourself.

```php
// Drupal 4.6
function conf_url_rewrite($path, $mode = 'incoming') {
  // $mode is either 'incoming' or 'outgoing' and function
  // is only called if there is no system alias found
}
?>

<?php
// Drupal 4.7
function custom_url_rewrite($type, $path, $original) {
  // $type is either 'alias' or 'source', depending on the
  // desired resulting path type of the operation (to be
  // in line with drupal_lookup_path() operation),
  // $path is possibly already processed by Drupal,
  // $original is the originally passed path
  if ($path == $original) {
    // path is not yet aliased (this is the only case
    // conf_url_alias() was invoked in earlier versions)
  }
}
?>
```

**node_delete(): moderately used**

The confirmation screen has been abstracted from the function. `node_delete` now strictly handles deletion of a node. It no longer takes an array as an argument--instead pass the nid of the node you wish to delete.
Instead of:

node_delete(array('nid' => $node->nid, 'confirm' => 1));

Now use:

node_delete($node->nid);

If a confirmation screen is desired, redirect to node/[nid]/delete, where [nid] is the nid of the node to delete.

**New order of node hooks**

There is a new order in the way node hooks are called in 4.7 vs. 4.6:

**Drupal 4.6**
1. validate
2. form pre
3. form post
4. if the form was submitted then validate (again)
5. insert/update

**Drupal 4.7**
1. prepare
2. form (return an array)
3. validate (you can only validate here, no changes possible)
4. submit (prepare the node for save)
5. insert/update

In short, Drupal 4.7 separates out the individual parts of the validation, rather than combining it together into one function.

**hook_nodeapi(‘settings’, ...) replaced by form api**

Adding elements to the node type settings page is now done exclusively with the form api. Below is an example of the new way to add elements to this form from the node module.

```php
<?php
function node_form_alter($form_id, &$form) {
    if (isset($form['type']) && $form['type']['#value'] . '_node_settings' == $form_id) {
        $form['workflow']['node_options_']. $form['type']['#value'] = array(
            '#type' => 'checkboxes', '#title' => t('Default options'),
            '#default_value' => variable_get('node_options_'),
            $form['type']['#value'], array('status', 'promote'),
            '#options' => array('status' => t('Published'), 'moderate' =>
```
t('In moderation queue'), 'promote' => t('Promoted to front page'),
'sticky' => t('Sticky at top of lists'), 'revision' => t('Create new
revision')),
    '#description' => t('Users with the <em>administer nodes</em>
permission will be able to override these options.'),
    );
}
?>

**hook_nodeapi('form x', ...) replaced by form api**

nodeapi op 'form' is no more. Use hook_form_alter instead. This lets you not just add to the
node form but change at will.

See the core modules for examples. The conversation process is the following: let’s suppose we
have foo.module , and it compiles its nodeapi form in _foo_form which takes one argument,
$node. and the last command is return $form (core actually had such functions). Then we can
rename _foo_form to foo_form_alter and wrap the code as follows:

```php
<?php
function foo_form_alter($form_id, &$form) {
    if (isset($form['type']) && $form['type']['#value'] . '_node_form' == $form_id) {
        $node = $form['#node'];
        // _form_form code follows:
        $form['foo']['myfield'] = array(...);
        // the final return is not needed because $form is a reference.
    }
}
?>
```

**file_directory variables replaced by functions**

variable_get('file_directory_temp', ...) or
variable_get('file_directory_path', ...) should be replaced with calls to
file_directory_temp() and file_directory_path(). No arguments are necessary for
either function.

**array2object replaced by native PHP type conversion**

$my_object = array2object($my_array); should be replaced with$my_object =
(object) $my_array;.
user_load returns FALSE if a user cannot be loaded

user_load now returns FALSE if a user cannot be loaded, instead of an empty object.

MySQL tables are now always UTF-8 encoded

In order to ensure this, two changes need to be made.

First, all MySQL database tables created (e.g. in a my_module.mysql file) need to have a character set attribute appended to each CREATE TABLE statement:

**Drupal 4.6**

create table my_table (  

) type=myisam;

**Drupal 4.7**

create table my_table (  

) type=myisam /*!40100 default character set utf8 */;

Use the snippet as is, including the comment markers. This ensures your .mysql file still works on MySQL 4.0 and below.

The second change is to upgrade all existing tables. Thanks to the upgrade system, this is very easy. Simple create a my_module.install file in the same directory as your my_module.module file, containing:

```php
<?php
function my_module_update_1() {
  return _system_update_utf8(array(‘table1’, ‘table2’, ‘table3’));
}
?>
```

Replace ‘table1’, ‘table2’, ... by the names of your module’s table(s).

We no longer use the <base> element

In versions of Drupal prior to 4.7, we used the HTML BASE tag to indicate the base to which all relative links should be appended to. In Drupal 4.7, however, the BASE tag has been removed entirely. All Drupal functions that specifically return URLs (such as l() or url() or the various theme_add_style() features) have been updated to now return the base_path() prepended to any URLs. If, however, you are manually creating your own URLs and not using one of Drupal’s internal functions, you’ll need to do something like:
BEFORE:
print '<link rel="stylesheet" type="text/css" href="themes/chameleon/common.css" />

AFTER (generic approach):
print '<link rel="stylesheet" type="text/css" href='. base_path() . 
    "themes/chameleon/common.css" />

AFTER (strongest specific approach, using Drupal functions):

// this would only apply to styles, naturally
tHEME('stylesheet_import', base_path() . path_to_theme() . 
    "/common.css"");

**hook_onload replaced by addLoadEvent()**

If you need to add Javascript onLoad events from your modules you now need to use the
Drupal custom Javascript function addLoadEvent(func). This was done to allow for new
[Javascript functions in the core](http://drupal.org/node/42388)

<?php
    // Drupal 4.6
    function hook_onload() {
        return array('my_javascript_function()');
    }
    // Drupal 4.7
    // TO BE USED INSIDE YOUR FUNCTION THAT NEEDS TO LOAD A JAVASCRIPT FUNCTION
    if (isJsEnabled()) {
        addLoadEvent(yourCustomJSFunction);
    }
?>

**hook_search_item replaced by hook_search_page**

This was done to result in cleaner code. Modules must now implement [hook_search_page](http://drupal.org/node/42388) and are expected to provide the necessary themable functions there.

**Extreme long comment on http://drupal.org/node/42388**

I made this into a book page, because eventually this can become a longish description of menu structures which are IMO way too big and rarely needed to stuff into a code comment of Drupal. At original creation, it’s only chx’s understanding of menu system innards.
For easier reference, let $menu = menu_get_menu();

Menu items can be defined at two places: in hook_menu and on the admin/menu interface. The latter items are stored in the database. However, there is the problem of modules changing from time to time and this changes the defined menu items. However, menu items defined on the UI must remain stable, hold their place in the menu tree, so we save some menu information to the menu table to ‘pin it down’. There is another problem here, though -- hook_menu defines a ton more information than menu table store. Unless you want to store the PHP code there is no other choice. So, we should store information from the modules into a table and maintain this relationship.

menu IDs are negative for items defined in hook_menu and positive for those stored in the database. There is a special 0 menu ID which is the root element, it’s children are the menus.

$menu['items'] will contain an associative array, the keys of which are menu IDs. The values are themselves associative arrays, with the following key-value pairs defined:

- pid - the menu ID of the parent of the menu item or 0 for menus.
- path - The Drupal path to the menu item.
- title - The displayed title of the menu or menu item. It will already have been translated by the locale system.
- weight - the weight of the menu or menu item.
- access - whether the current item is accessible to the current user.
- type - typical values are 22 (MENU_NORMAL_ITEM), 4 (MENU_CALLBACK). See the defines in the beginning of menu.inc
- children - A linear list of the menu ID’s of this item’s children. While theoretically this is redundant information as the pid will contain the same information, it’s a huge performance benefit to save these.

$menu['visible'] is already documented, it’s a subset of $menu['items'] actually.

$menu['path index'] is an associative array, the keys are Drupal paths and the values are menu IDs. $menu['callbacks'] is again an associative array, the keys are Drupal paths and the values are an (surprise!) associative array which define the callback. This callback array always contains a ‘callback’ => ‘name_of_callback_function’ pair and optionally a ‘callback arguments’ => array(arg1, arg2, arg3).

Now that we are familiar with the menu structure, let’s see what happens when we save a menu item (menu_edit_item_save): it’s saved into the database, without much consideration about our structure. When you move an item it’s possible that you put it into under an item which is not yet in the database and then we need to fix the situation.

That’s what menu_rebuild does: first it collects all items not yet in the database into the queue. Then it walks the loop and if an element has a valid parent and is not yet in the menu tree then saves it and for new items we fix all its children mid to the new mid. Regardless of the item needed fixing or not, the item is dropped from the queue. As long as there are no recursive menu items, and there can’t be, that’s guaranteed by _menu_build, the queue will empty after a
few loop.

**Revisions overhaul**

This change affects your module if you do direct SQL operations on node body or teaser fields, or if your module provides a node type and stores the node type’s extra information in its own table.

**Fields moved from node table to node_revisions table**

Before Drupal 4.7 the node table included body and teaser fields. As of 4.7, these fields are moved into the node_revisions table. If you have SQL statements referencing these fields, you will have to rewrite your SQL to refer to the node_revisions table. Typically, you’ll do an additional join of node on node_revisions. Here’s an example from blog.module.

The 4.6 version:

```php
<?php
    $result = db_query_range(db_rewrite_sql("SELECT n.nid, n.title, n.teaser, n.created, u.name, u.uid FROM {node} n INNER JOIN {users} u ON n.uid = u.uid WHERE n.type = 'blog' AND u.uid = %d AND n.status = 1 ORDER BY n.created DESC"), $uid, 0, 15);
?>
```

And the 4.7:

```php
<?php
    $result = db_query_range(db_rewrite_sql("SELECT n.nid, n.title, r.teaser, n.created, u.name, u.uid FROM {node} n INNER JOIN {node_revisions} r ON n.vid = r.vid INNER JOIN {users} u ON n.uid = u.uid WHERE n.type = 'blog' AND u.uid = %d AND n.status = 1 ORDER BY n.created DESC"), $uid, 0, variable_get('feed_default_items', 10));
?>
```

**Making your module revisions aware**

This change has no influence on your node modules unless you want the information to be revisions aware.

Until Drupal 4.6 we used the node ID (nid) as a sole reference for any given node. As of Drupal 4.7 we have both a node ID and a version ID (vid). There can be several vids for each nid, one per revision. There is always at least one vid per nid. The vids are unique within one Drupal install.

Let us assume your module is named foo.module and provides the 'foo' node type. The extra information is stored in an extra table:
In order to take advantage of the new revisioning capabilities you need to change it to

The old nid values will become vid values; the actual value will not change. You, of course, need to change any JOINs that you perform in the nodeapi hook and other hooks. Instead of joining on nid you join on vid. The same applies to SELECTs.

If you do not want to take advantage of the revisioning system you can just leave everything as it is and the extra information provided by your module will be shared across all revisions of a given node.

If you already used revisions before this overhaul you will notice that they seem to be lost after you run the upgrade script. This is not the case. The old revisions are stored in the old_revisions table and will be made available again by a forthcoming update.

The development history of this revisions overhaul is found in this issue: [http://drupal.org/node/7582](http://drupal.org/node/7582)

**Join forces**

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

So please consider the following guidelines or ideas:

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

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

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

Drupal database documentation

This place is under construction.

For now, Drupal database documentation can be acquired from http://cvs.drupal.org/viewcvs/drupal/contributions/docs/developer/database/

If you’d like to help, please see http://drupal.org/project/comments/28046

'Status' field values for nodes and comments

Note: we now use CONSTANTS in the code so it isn’t usually necessary to know these values. You only need them if you are querying the DB from a non drupal page.

Just documenting the status field for the following tables

NODES
- 0: not published
- 1: published

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

Module how-to’s

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

How to handle text in a secure fashion

When handling and outputting text in HTML, you need to be careful that proper filtering or escaping is done. Otherwise there might be bugs when users try to use angle brackets or ampersands, or worse you could open up XSS exploits.

When handling data, the golden rule is to store exactly what the user typed. When a user edits a post they created earlier, the form should contain the same things as it did when they first submitted it. This means that conversions are performed when content is output, not when saved to the database.
To help you see where checks are needed, it is handy to mentally ‘color’ in each string depending on which format its data is in. Is it plain-text, HTML, BBcode or Textile? Then, whenever you concatenate two strings, you need to make sure they are both in the same format. If they are not, an appropriate check, conversion or filtering must be applied.

User-supplied arguments to the query should be passed in as separate parameters so that they can be properly escaped to avoid SQL injection attacks. More information about this is available at the [Drupal Documentation db_query page](#).

User-submitted data in Drupal can be divided into two categories:

1. **Plain-text**

   This is simple text without any markup. What the user entered is displayed exactly on screen as is, and is not interpreted in any form. This is generally the format used for single-line text fields.

   When outputting plain-text, you need to pass it through `check_plain()` before it can be put inside HTML. This will convert quotes, ampersands and angle brackets into entities, causing the string to be shown literally on screen in the browser.

   Most themable functions and APIs take HTML for their arguments, but there are a few which already have `check_plain()` in it for convenience:

   - `l()`: the link caption should be passed as plain-text (unless overridden with the `$html` parameter).
   - `menus`: the menu item titles are plain-text.
   - `theme('placeholder')`: the placeholder text is plain-text.

   Some places require HTML which might not be obvious:

   - page titles set through `drupal_set_title()`. The page title is displayed in the HTML, where it makes sense to use tags like `<em>` for clarity. When the page title is displayed in the HTML tag however, all tags will be stripped out.
   - block titles passed in through `hook_block()`. For the same reason as the page title, using HTML here is commonly done.

   Note that functions which logically take ‘data’ and not ‘output’ will almost always take plain-text and require no escaping on your side. A good example is the value passed to `form_` functions, e.g. a plain-text field’s contents. What the user entered is exactly what you should pass to `form_textfield()` as the field’s value. On the other hand, the field’s title or description are passed as HTML so that markup can be used in them.

2. **Rich text**

   This is text which is marked up in some language (HTML, Textile, etc). It is stored in the markup-specific format, and converted to HTML on output using the various filters that are enabled. This is generally the format used for multi-line text fields.
All you need to do is pass the rich text to `check_markup()` and you’ll get HTML returned, safe for outputting. You should also allow the user to choose the input format with a format widget through `filter_form()` and should pass the chosen format along to `check_markup()`.

Note that you must make sure that the author of a post is allowed to use a particular input format. As a safe-guard, `check_markup()` performs this check for the current user by default. However, because content is filtered on output, this is often not the person who originally wrote the content. In that case, you must disable this check by passing `$check = false` to `check_markup()`, and making sure that the format is being checked with `filter_access()` when the content is being submitted.

URLs across Drupal require special handling in two ways:

1. If you wish to put any sort of dynamic data into a URL, you need to pass it through `urlencode()`. If you don’t, characters like ‘#’ or ‘?’ will disrupt the normal URL semantics. `urlencode()` will prevent this by escaping them with `%XX` syntax. Note that Drupal paths (e.g. ‘node/123’) are passed through `urlencode()` as a whole since Drupal 4.7 so you don’t need to urlencode individual parts of it. This convenience does not apply to other parts of the URL like GET query arguments or fragment identifiers.

2. When using user-submitted URLs in a hyperlink, you need to use `check_url()` rather than just `check_plain()`. `check_url()` will call `check_plain()`, but also perform additional XSS checks to ensure the URL is safe for clicking on.

Note that all Drupal functions which return URLs (`url()`, `request_uri()`, etc.) output plain URLs which have not been HTML escaped in any way (in other words, they are plain-text). Remember to use `check_url()` to escape them when outputting HTML (or XML). Don’t use `check_url()` in situations where a real URL is expected, e.g. in the HTTP Location: ... header.

**In practice**

All the rules above can be summed up quite easily: no piece of user-submitted content should ever be placed as-is into HTML. If you are unsure of whether this is the case, you can always test it by submitting a piece of text like `<u>xss</u>` into your module’s fields. If the text comes out underlined or mangles existing tags, you know you have a problem.

Here are some examples of good and bad code. `$title`, `$body` and `$url` are assumed to be user-submitted fields containing a title, a piece of marked up text and a URL respectively. They are fresh from the database and thus contain exactly what the user submitted without any changes.

Bad:
```php
<?php print '<tr><td>$title</td><td>'; ?>
<?php print '<a href="..." title="$title">view node</a>'; ?>
```
Good (the title is plain-text and may not be placed into HTML as is):

```php
<?php
print '<tr><td>'.check_plain($title).'</td></tr>'; ?>
<?php
print '<a href="..." title="'.check_plain($title).'">view node</a>'; ?>
```

Bad:

```php
<?php
print l(check_plain($title), url('node/'. $nid)); ?>
```

Good (l() already contains a check_plain() call by default):

```php
<?php
print l($title, url('node/'. $nid)); ?>
```

Bad:

```php
<?php
print '<a href="$url">'; ?>
<?php
print '<a href="'.check_plain($url).'">'; ?>
```

Good (URLs must be checked with check_url):

```php
<?php
print '<a href="'.check_url($url).'">'; ?>
```

## Writing filters

Writing filters

When writing a filter which translates from another markup language into HTML, you need to ensure you don’t open any holes yourself. Generally, the same rules apply: check URLs with check_url() and ensure no literal HTML can be injected by escaping appropriately using check_plain().

## How to write a node module

How to write a node module

This information is superseded by the Doxygen documentation. In particular, its example node module is a good tutorial.

## How to write automated tests

How to write automated tests

Writing an automated test might be a bit disorienting for some Drupal developers. It isn’t hard, but is different from rest of Drupal in a couple of ways

- Tests use classes instead of functions like rest of Drupal. This is a byproduct of our choice to use the simpletest library. Simpletest is a wonderful piece of software, and this minor nit shouldn’t upset a rational developer.
- To write a test, you have to think like a tester. You have to exercise all the important branches of your code and poke on corner cases. At first, this can feel foreign but please persevere - the rewards are great.

The simpletest.module is using the simpletest php library as framework for our tests. Excellent documentation is available for both general tests and our popular classes, web testing and unit testing.
You need not understand all the pages linked above. Feel free to learn from existing tests in this package and elsewhere in Contrib.

The basic class structure

By using the simpletest framework the drupal tests are defined as classes in object oriented style. Here is a skeleton test:

```php
<?php
 /**
 * Description
 */
 class ModuleFeatureTest extends DrupalTestCase {
     function get_info() {
         return array('name' => 'CoolFeature Test',
                       'desc' => t('Assure that all cool features of your module work.'),
                       'group' => 'Your Module Tests');
     }
     function testYourCoolFeature() {
         /* Test code here */
     }
 }
?>
```

Your new class must implement a method named `get_info`:
It returns an associative array with your test’s name, description, and which ‘group’ it belongs to. You may create a new group or insert your test into an existing group by using the same name.

This would go into a .test file.

The DrupalTestCase features

By extending DrupalTestCase your test inherits all features of both [WebTestCase](#) and [UnitTestCase](#) as well as the Drupal-specific features of DrupalTestCase.

```php
function $this->drupalModuleEnable($name)
function $this->drupalModuleDisable($name)
```

Enables or disables a Drupal module by name.
This is useful for testing modules that are not required or disabling modules that might interfere with your test by adding required input.
The module settings will be restored after your test method has completed.
**ATTENTION**: Make sure you use return instead of exit/die in your test! When you extend the tearDown() / setUp() methods please call the parent:: methods!
Example:

```php
<?php
/* make sure the profile module is disabled to avoid conflicts */
$this->drupalModuleDisable('profile');
?>

function $this->drupalVariableSet($name, $value)

This function sets a Drupal variable (like variable_set), but restores the original value after your test method has completed.

Example:

```php
<?php
/* We first allow every user to login instantly. */
$this->drupalVariableSet('user_register', 1);
?>

function $this->randomName($number = 4, $prefix = 'simpletest_')

Returns a string with $number alphanumerical character(s) prefixed by $prefix. The first character will not be a number.

The internal browser

The DrupalTestCase features an internal browser that can be used to navigate on your test site. Please read the basic documentation of the WebTestCase class for more information.

function $this->drupalPostRequest($path, $edit, $submit, $reporting = TRUE)

This function does a post request on a drupal page. The $path indicates a page containing a form that will be filled with $edit data. Then the button indicated by $submit will be clicked (submit caption will be translated by this method).

It also does assertion that the requests were successful and form fields could be set.

Example:

```php
<?php
$name = $this->randomName();
$mail = "$name@example.com"
$edit = array('name' => $name,
             'mail' => $mail);
$this->drupalPostRequest('user/register', $edit, 'Create new account');
?>
```
function $this->clickLink($label, $index = 0)

Follows a link on the current page by name. Will click the first link found with this link text by default, or a later one if an $index is given. The $label is automatically translated. An assertion is done about the availability of the link and the URL it points to. Also gives some output including current and requested URL.

Example:

```php
<?php
$this->clickLink('log out');
?>
```

function $this->drupalCreateUserRolePerm($permissions = NULL)

This function creates a user and returns the user object with an additional value pass_raw containing the non-hashed password. It also creates a role with the specified $permissions that is assigned to the returned user. The $permissions are specified as an array of strings. If it is omitted or NULL, the default permissions for a registered user will be used:

`'access comments, access content, post comments, post comments without approval'`

An assertion for success is done as well as clean-up on the user and role tables.

function $this->drupalLoginUser($user = NULL)

This function logs a user into your site via the internal browser. You can just hand it a $user object (required is a pass_raw value). If the argument is omitted this function will create a user and role with the standard permissions mentioned above.

After the user is logged in, you can now navigate with the internal browser.

Example:

```php
<?php
/* Prepare a user to do the stuff */
$user = $this->drupalCreateUserRolePerm(array('access content', 'create pages'));
$this->drupalLoginUser($user);
/* now do something with the users */
$this->get(url('node/view/' . $node->nid));
?>
```

This method also does several assertions about the login process from the browsers perspective.
function $this->drupalCreateRolePerm($permissions = NULL)

This function is rarely useful. The $permissions parameter behaves exactly like in drupalCreateUserRolePerm.
The return value is a role-id integer or FALSE on failure.
A success assertion is done, as well as decent clean-up of the role and permission tables.

Implementing hook_simpletest

This hook allows your module to tell the simpletest.module where its test files reside.
Contrib module tests should be placed in a sub-directory called 'tests' in the module directory.
The file extension for your tests should be '.test'.

Implementing this hook is usually just a matter of copying the code below and substituting your own module name for 'example'.

```php
<?php
/**
 * Implementation of hook_simpletest().
 */
function example_simpletest() {
    $dir = drupal_get_path('module', 'example') . '/tests';
    $tests = file_scan_directory($dir, '\.test');
    return array_keys($tests);
}
?>
```

Running tests

You can run finished test classes that are referenced by hook_simpletests from administer > simpletest. The administer unit tests permission is required to run tests. You will get a list of core tests and module tests. Select your test(s) and choose 'Run Selected Tests'.
If the test run outputs green text like the following:

Drupal Unit Tests
1/1 test cases complete: 10 passes, 0 fails and 0 exceptions.
then the test(s) completed successfully. Currently the passes are not reported.

Otherwise you will get a report like the following:

Drupal Unit Tests
Fail: Taxonomy Module -> Test taxonomy’s functions ->
testVocabularyFunctions -> Checking value of nodes at line [45]
Fail: File API Tests -> Upload user picture -> testUploadPicture ->
Checking response on proper image at line [56]
Fail: File API Tests -> Upload user picture -> testUploadPicture ->
Checking response on proper image at line [69]
Fail: File API Tests -> Upload user picture -> testUploadPicture ->
Checking response on proper image at line [106]
Fail: File API Tests -> Upload user picture -> testUploadPicture -> [browser] Setting edit[picture_delete]="1"
2/2 test cases complete: 32 passes, 5 fails and 0 exceptions.

For each test failure, the test framework outputs a detailed trace, including which assertion failed.

**Testing core modules and APIs**

The simpletest.module comes with a set of tests which exercise Drupal’s core modules and APIs. These can be helpful when making patches against Drupal’s core. The best idea is to run the tests before you write a core fix; everything should pass. If you get fails in a HEAD install please file an issue including as much information about your environment as possible (Drupal version, db version, php_info, ...).

After you write your patch, run the same test set again. Any new failures may indicate bugs introduced by your patch.

**Function testing vs. browser testing**

There are many different tools in the Simpletest framework; however we focused on two: browser and function tests. Both are very useful during various stages of application development.

If you want to test a new module which doesn’t contain a user interface yet, function tests are the best option. You can quickly check if our code is valid before the whole application is finished.

After your module has been written, it is essential to test the user experience. At this stage, the best solution is testing with browser tests.

**Function-based tests**

In order to test a function, we simply call the function and check the result. For example, assume we want to test the `user_validate_mail` function (you can see this in user_validation.test).

At the beginning we will look at our examined function:

```php
<?php
function user_validate_mail($mail) {
    if (!$mail) return t('You must enter an e-mail address.‘);
    if (!valid_email_address($mail)) {
        return t('The e-mail address %mail is not valid.’, array('%mail' => theme('placeholder', $mail)));
    }
}
?>
```
As you can see, if $mail contains an invalid e-mail address, an error message will be displayed. Otherwise, the function returns nothing.

Let’s write a test class. We start with get_info(), which is required in every test.

```php
<?php
class OurFirstTest extends DrupalTestCase {
    function get_info() {
        return array(
            'name' => 'User’s email validation',
            'desc' => 'Exercise email address validation.',
            'group' => 'Example tests'
        );
    }
}?>
```

after that, we add:

```php
<?php
function testInvalidMail() {
    $name = 'abc';
    $result = user_validate_mail($name);
    $this->assertNotNull($result, 'Invalid mail');
}
function testValidMail() {
    $name = 'absdsdsdc@dsdsde.com';
    $result = user_validate_mail($name);
    $this->assertNull($result, 'Valid mail');
}
?>
```

As we mentioned before, we check if the variable is set or not using assertNot Null and assertNotNull.

After writing this we should have our test on the admin/simpletest page:

When you run this test you see something like this:

Drupal Unit Tests
1/1 test cases complete: 2 passes, 0 fails and 0 exceptions.

See [image 2] for a screenshot.

**Browser-based tests**

Now it’s time to try a browser test. We don’t have to change too much in our class, because DrupalTestCase class contains tools for both tests.
We would like to check Drupal’s response when specifying an invalid email address in the registration form (user/register). Let’s look into the source:

```php
<?php
<\div class="form-item">
  <label for="edit-name" class="form-required">Username:</label><span>
  <input maxlength="64" class="form-text required" name="edit[name]" id="edit-name" size="30" value="" type="text">
  <div class="description">Your full name or your preferred username; only letters, numbers and spaces are allowed.</div>
</\div>
<\div class="form-item">
  <label for="edit-mail" class="form-required">E-mail address:</label><span>
  <input maxlength="64" class="form-text required" name="edit[mail]" id="edit-mail" size="30" value="" type="text">
  <div class="description">A password and instructions will be sent to this e-mail address, so make sure it is accurate.</div>
</\div>
?>
```

It is important to notice the name of fields in which we want to put data. There are two of them: edit[name] and edit[mail].

We begin with the same thing everytime:

```php
<?php
class OurSecondTest extends DrupalTestCase {
  // we need this function to notify the world about our great test
  function get_info() {
    return array('name' => 'Be a browser', 'desc' => 'This tests the browser’s response on invalid mail input in the registration process.', 'group' => 'Example tests');
  }
}
?>
```

Then we write our test function:

```php
<?php
function testBrowserResponse() {
  // let’s create a random name and email
  $name = $this->randomName(10);
  $mail = $this->randomName(10);
  // try to register
  $edit = array('name' => $name, 'mail' => $mail);
  $this->drupalPostRequest('user/register', $edit, 'Create new
```
account’);

    $expectation = t('The e-mail address %mail is not valid.', array('%mail' => $mail));
    $this->assertWantedText($expectation, 'Checkin response on invalid e-mail address');
}
?>

Let’s look at each line individually:

```php
<?php
    $name = $this->randomName(10);
?>
```

This line creates a 10 character random string, which is by default prefixed with ‘simpletest_’.

**Example:** `simpletest_XK3sKW5lFx`

```php
<?php
    $edit = array('name' => $name, 'mail' => $mail);
    $this->drupalPostRequest('user/register', $edit, 'Create new account');
?>
```

In these two lines, we prepare and send data to our form.

We run `drupalPostRequest` with the following parameters:

1. The path to our form (e.g. ‘user/register’). We don’t use ?q=, or the whole address `http://www.example.com/?q=user/register` because this function creates the proper address for us.
2. Array of sent data. Look at the special construction of this array ( => ). We just use the form field names in brackets from above: ‘name’ and ‘mail’. The rest is done by this function.
3. The name of the button which we have to click to send our form. In our example it is: ‘Create new account’.

```php
<?php
    $expectation = t('The e-mail address %mail is not valid.', array('%mail' => $mail));
?>
```

We prepare our expectation using t() function. We took this from user.module.

```php
<?php
    $this->assertWantedText($expectation, 'Checkin response on invalid e-mail address');
?>
```
At last we check if our expectation appears on the content page.

When we run this test we should receive the following output:

Drupal Unit Tests
1/1 test cases complete: 5 passes, 0 fails and 0 exceptions

See Image 3 for a screenshot.

If you look closely, you notice that we have 5 passes in the result, but in the test code we only have one check (using `assertWantedText`). Where is the rest hidden?

The answer is: `$this->drupalPostRequest('user/register', $edit, 'Create new account');`

If we don’t specify the 4th parameter in the `drupalPostRequest` function we will have additional checking. In our example it is:

1. checking if url is valid
2. checking if data was inserted in form’s fields
3. checking if we clicked our button

and that’s why we have $1 + 4 = 5$ passes. To avoid this, give the 4th parameter the value 0.

```php
<?php
$this->drupalPostRequest('user/register', $edit, 'Create new account', 0);
?>
```

**Tips**

Here is a place where you can find various tips and hints which will help you in writing tests

**Viewing source during a browser test**

Use `$this->showSource()` to output the source that the simpletest browser is receiving. Useful for debugging during development. Comes from simpletest/simpletest/webtester.php.

**How to write database independent code**

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

- When you need to LIMIT your result set to certain number of records, you should use the `db_query_range()` function instead of `db_query()`. The syntax of the two functions is the same, with the addition of two required parameters at the end of `db_query_range()`. Those parameters are `$from` and then `$count`. Usually, `$from` is 0 and `$count` is the maximum number of records you want returned.
- If possible, provide SQL setup scripts for each supported database platform. The differences
between each platform are slight - we hope documentation on these differences will be forthcoming.

- You should test any complex queries for ANSI compatibility using this tool by Mimer
- If you are developing on MySQL, use its ANSI compatibility mode
- If you can install all database servers in your environment, it is helpful to create shell databases in each and then run sample queries in each platform's query dispatch tool. Once your query succeeds in all tools, congratulate yourself.
- Don't use '' when you mean NULL
- Avoid table and field names that might be reserved words on any platform.
- Don't use auto-increment or SERIAL fields. Instead, use an integer field and leverage Drupal's own sequencing wrapper: `db_next_id(<tablename_fieldname>)`

How to write efficient database JOINs

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

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

```
[left table] [INNER | LEFT | RIGHT] JOIN [right table] ON [join criteria]
```

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

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

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

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

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

For example:

Table A

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

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

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

Hope that helps in reading some of the queries.

**How to connect to multiple databases within Drupal**

Drupal can connect to different databases with elegance and ease!

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

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

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

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

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

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

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

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

How to write themable modules

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

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

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

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

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

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

By default, this will call theme_node($node). However, if the currently active theme is
"mytheme", and this theme has defined a function mytheme_node(), then
mytheme_node($node) will be invoked instead.

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

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

This handbook section contains all the Proposals for enhancing Drupal. If you have a big project to overhaul a part of the core, a big infrastructural change, or big changes to a contribution, please write up a structured proposal first. By no means is this required, we strongly encourage everyone to just add patches to the issues queue. Also please discuss any proposals on the development mailinglist first.

A DEP should be structured using the following template:

Title:
Abstract:
Author:
Dependencies:
Repository:
Status:

1. Introduction:
   Keep it short

2. Motivation:

3. Approach:

4. Security considerations: (optional, if it involves security related issues)

5. Excluding: (optional, what will NOT be covered in this DEP)

DEPS in progress

This chapter contains all the Drupal enhancement proposals that are in progress.

Events Improvement

Drupal has a many event organizing modules avaliable which provide a range of functionality including scheduling, invitations, rsvp, and volunteer management. Our goal is to bring together the various module developers, funders, and site builders to work towards better integration and general improvements in event management modules.

The goal is three-fold:

- Develop API’s to integrate modules and create new functionality
- Establish best practices and documentation
- Fundraise to support development efforts
How this project is organized

- We communicate primarily over an email list. To subscribe send an email to events-dev-subscribe@civicspacelabs.org. (This will hopefully be moved to Drupal.org some time soon)
- We hold regular IRC meetings
- We use this book to publish and collaborate.
  - Add descriptions of the events related work you are involved in as a child of the Areas of improvement page
  - Add events related cases to the Use cases and motivation page

Who’s involved

Please add yourself to this list and a brief description of what events related work you have done so far and how you will be involved in this effort. If you don’t have permission leave a comment and we will add you.

- Richard Orris of CivicActions Working on iCal.
- Angela Byron of CivicSpace Labs lead developer of GoJoinGo. Helping coordnate this effort.
- Chad Phillips creator and maintainer of Event Repeat modules, co-maintainer of Signup module. Working on development.
- Boris Mann of Bryght Helping with fundraising.
- Zack Rosen of CivicSpace Helping with fundraising and coordination.
- Derek Wright heavy user of Drupal’s event management modules. Working on design, development, and helping to coordinate.
- Greg Knaddison (greggles) interested in an appointment scheduler system. Helping with requirements and funding.

Areas of improvement

There are a number of different areas of improvement to events and event management that are covered in this proposal. Each area of work will have its own subpage.

Workflow Proposals

We want to be able to provide a seamless user experience across the common functionality of event organizing in addition to tight integration of modules over API’s. To do this we need to do interaction design for event modules across the domain of functionality. Workflow designs for event modules are posted as subpages here.

Workflow for event creation, invite, and rsvp

This workflow proposal was created as part of the "GoJoinGo" Group/Events development effort.
19. RSVP for an event and 20. change RSVP status (a subset of this)

If logged in, Create Account page for trying to do something that requires an account message at top error, then, and give option to log in if it is a user account

If logged in as registered user, user is logged in (remembered via cookie)

If logged in as unregistered user, user is not logged in, use login page

If user is not a registered user, Create Account page

If user is already a member of the event, event detail page

If user is not a member of the event, authorization by email

If not a member, event detail page

21. Find someone I met at a group’s event

Click on a past Event

Event Page

RSVP record shows list of attendees, sometimes not displayable to non-registered

If person RSVP only using email addresses (with no account) then there is no way to contact them, should organizer try to see those email addresses?

22-23 reserved (placeholders for future event tasks)
**CivicSpace: Workflow event creation, searching, invitation, and rsvp**

This diagram is a proposal for interaction design for a Drupal event system that can handle event creation, searching, invite, and rsvp.

- The bubbles represent interface screens
- The arrows represent paths the user can take through the different event components.
- Each bubble links to a mockup of the interface designed in a way to lead the user through the different paths they can take when using the event system.

**Next generation event management based on views and the content creation kit**

This is a stub to discuss next generation event modules based on views and content creation kit.

**Use cases and motivation**

This page will contain information about all the use cases and motivation that’s leading to this effort to improve event management in Drupal. Anyone with their own ideas and examples should create a subpage from here.
**IRC Meetings**

We periodically hold IRC discussions to plan event development work.

- Meetings will be scheduled in advance and notice will be posted to [News and Announcements forum](#), [Developers Mailinglist](#) and [Consultants Mailinglist](#).
- IRC meetings will be held in #drupal-events on irc.freenode.org.
- Meeting notes and action items will be posted afterwards as children of this page.

**3/6/06 IRC Meeting: Re-Thinking Events in Drupal**

Right now, Drupal has a plethora of available event options available which range from modules which handle scheduling, to those which record attendance, to huge mega-modules that intend to do it all. This was also a topic of discussion at DrupalCon. I’d like to setup a meeting on IRC **Monday, March 6 at 12 PM PST** (which is 3PM EST, 8PM GMT) in #drupal-events, to bring funders, developers, and site builders together to improve event management in Drupal. The goal is three-fold:

1. Finding funding
2. Developing APIs and adding functionality
3. Establishing best practices and documentation

There are lots of people thinking, developing, funding, and otherwise contributing in this space, including:

- [Kieran Lal](#) of CivicSpace Labs
- Richard Orris and Owen Barton of CivicActions
- Aaron Welch who did the initial re-write of Event module and maintains RSVP
- Gerhard Killesreiter who maintains the Event module
- Donald Lobo who leads development of CiviCRM
- Angela Byron of CivicSpace Labs lead developer of GoJoinGo.
- Chad Phillips creator and maintainer of SIgn-up and Event Repeat modules.
- Ankur Rishi creator of the Location module and gsearch (location search) modules.
- [Josh Koenig](#) of Trellon original author of Volunteer module
- Michael Haggerty of Trellon author of EventFinder module
- Boris Mann of Bryght

So let’s get together and combine our efforts, and help make event management in Drupal shine!

**Meeting Notes**

Thank you very much to all who attended! The following is a summary of points raised:

- Use the relationships API to keep track of relationships between users and events. This can then be used to track sign-ups, RSVPs, volunteers, possibly resources, and eliminate/reduce the need for these additional modules.
- Capitalize on the work that’s gone into CCK, by making ‘events’ a date/time field in CCK that can then be used to expand any type of node.
Use Views to tie in ‘front-end’ components to those on the ‘back-end’.

Events need to be easier to theme. We also need a way to make add-ons to the event module. Killes welcomes any patches to the event patch queue.

We need to draft a series of use cases, to show what events should do from the user’s perspective. CivicSpace Labs/CivicActions already have done some work in this area.

Timezones are an issue -- a standard approach, such as the one used in `events_timezone.inc`, should be part of Drupal core.

Integration with Send module to handle invitations/responses via e-mail.

Generally, smaller, more focused modules is the desired approach, over huge ‘do-it-all’ modules, but it also needs to be easy for users to get something ‘quick and easy’ to satisfy certain common use-cases.

I’ve also uploaded a full log of the conversation (minus people entering/exiting) here:


And a placeholder for the forth-coming DEP (Drupal Enhancement Proposal) can be found here:

[http://drupal.org/node/52884](http://drupal.org/node/52884)

The next goals are:

1. Defining a DEP to describe our overall goals
2. Creating use cases to cover types of functionality users are looking for
3. Identify code that needs to be written vs. code that can be re-used
4. From the above, create a road-map document to illustrate the ‘grand master plan’ for Drupal events.

We need people to step up and volunteer to "own" some of these items. If you would like to help out, please post back here or feel free to [contact me via my contact page](mailto:). I’ll be contacting those who already offered as well (and thanks!).

We will have a follow-up meeting to check progress on these issues within a couple weeks’ time (I’m thinking around Tuesday, March 21 at a time that enables Australians to attend).

**Events DEP Stub**

Title : Event Improvements

Abstract:

Author: Event Special Interest Group

Dependencies:

Repository:

Status: planning stages

1. Introduction :

Keep it short
2. Motivation:

3. Approach:

4. Security considerations: (optional, if it involves security related issues)

5. Excluding: (optional, what will NOT be covered in this DEP)

**Mapping API Generalization**

Title: Mapping API (map.module)
Abstract: A generalized mapping API
Author: webgeer
Repository: TBA
Status: Planning Stage

1. Introduction:

This will be a new module that will have a generalized mapping API for displaying manipulating and using maps within Drupal. Different types of maps (google maps, yahoo maps, open source mapping projects etc) would be implemented with specific .inc files for each type. Plug-ins will be used for a variety of mapping parsing inputs and outputs to allow gathering map data from other websites or sources, or to provide map feeds for other websites or applications.

2. Motivation:

There are a few separate modules that do some mapping. However, these are not well coordinated and create some conflicts with each other. This will make developing mapping information into drupal sites much easier and have a consistant approach.

3. Approach:

The map module would have generalized calls in the form of map_draw($mapvar) which would then call the applicable function from the include file based on the site administrators preferred map type (i.e. google_map_draw($mapvar) ). Different interfaces would have differences in exactly how it is implemented and what capabilities are available, but in general if the same $mapvar is used for yahoo_map_draw() or google_map_draw(), it should return roughly the same map with the same overlays. The $mapvar would be a standardized variable type. The map.module would also have some generalized map manipulation functions. Details on the specific map.module API functions and the $mapvar definitions can be found on the [link to subpage on mapping API]

The module will also have the ability to parse map data from files or external websites based on formats defined by plug-in files. It will also be able to output data from map information in drupal in various formats. Details on the plug-ins for inputting and outputting map files are available on the [link to subpage on map formats].
5. Excluding:

The base module should not require the installation of any other modules or tables. It will not include functionality for geolocating based on address.

Please email any comments on this to me, or add comments to the appropriate subpage. I will keep this DEP up to date based on comments and suggestions received.

**Finished DEPs**

Drupal enhancement proposals that were finished or closed.
Drupal.org site maintainers

Below is an alphabetical list of users who have additional permissions to help maintain the drupal.org website:

1. adrian
2. adrinux
3. ahoppin
4. aldon@deanspace.org
5. Amazon
6. andremolnar
7. ax
8. BÀ'r Kessels
9. bertboerland@ww...
10. Boris Mann
11. bryan kennedy
12. cel4145
13. chx
14. cshields
15. Cvbge
16. Development Seed
17. dikini
18. Dries
19. DriesK
20. drumm
21. Dublin Drupaller
22. ericgundersen
23. factoryjoe
24. Gerhard Killesreiter
25. Goba
26. greggles
27. Gunnar Langemark
28. handelaar
29. Heine
30. hunmonk
31. jblack
32. JonBob
33. Junyor
34. ivandyk
35. kbahey
36. Kieran Huggins
37. kika
38. killes@www.drop.org
If you have been around for a while, and you want to help maintain Drupal.org, get in touch with Dries.
Site maintainer’s guide

This page lists some guidelines for Site Maintainers on Drupal.org.

Unpublishing vs deleting of content

You should only unpublish a post if you can conceivably imagine it being re-published in the future. This should only be for very rare cases. A good example is an unmaintained project (someone might take over development later): unpublish, don’t delete (*). On the other hand, spam can be deleted immediately.

(*) Actually, old projects are automatically unpublished by the cvs scripts, so this is not something you need to do.

Blocking vs deleting of users

Deleting users is a very destructive action, as it makes all their content inaccessible in most places, even to administrators. It should not be done. If a user is a troublemaker, just block their account (click username -> edit -> status: blocked). Of course, you should not block people just because they say unfavorable things about Drupal. Here are good reasons to block someone:

- Spamming (even once)
- Repetitive flaming
- Repetitive posting of trash content (test posts, inappropriate book pages, ...)

Suggested Workflow

When you spot something out of the ordinary, we suggest these steps:

1. Take a look at the user’s post history on the "user -> track -> track posts" page. This will possibly show more bad posts by the same person.
2. Take a look at the user’s page visit history on the "user -> track -> track page visits" page. That way, you can easily tell if a user just registered to spam or if they made only one bad post in a series of good ones.
3. If the content is spam, you can delete it immediately and block the person’s account. Otherwise, send them a note through their contact tab about it:

   Your post Foobar on [http://drupal.org/node/1234](http://drupal.org/node/1234) was inappropriate because it contained flaming. Please be nice to your fellow visitors on Drupal.org, or your account may be blocked.

4. If you know someone to be a troublemaker who has been warned before, block their account.
Badly formatted posts

If you see a post with bad formatting which messes up the page’s layout, please edit it. A common mistake for newbies is to use two opening tags rather than an opening/closing pair. Tags like bold and italic can ‘bleed through’ beyond the post, while unclosed block-level tags can mess up the positioning of the sidebar.

If someone made a serious mistake while posting a forum topic and posted a correction in a comment below, try to update the original post and delete the correction.
Translator’s guide

This is the Drupal translator’s guide. It will cover most aspects of translating Drupal’s user interface. It will not cover the use of the various programs that can be used to do a translation. These programs are usually quite well documented.

As of version 4.5.0, Drupal includes an extended locale.module that enables you to share translations through the use of PO files. PO files are files containing translations as used by the GNU gettext program.

User contributed PO files for various languages can be found on the download page.

If your language is not present, you might want to start a translation yourself. If this is the case, please download the Drupal POT translation templates. You can get a PO file editor and start translating.

You should translate the individual PO files (per module) rather than one big file. The individual files are automatically packaged into one large file per language in the CVS repository, which is what others will download from this site.

Once you have completed a reasonable part of the translation, create an issue on the Translation templates project and upload your PO files. Some helpful developer will then come by and put them in CVS for you. If you have write access to the contrib CVS you can commit your files yourself. In any case a project for your translation will be created, you will be made the maintainer, and your translation becomes available on the download page.

Programs to use for translation

Recommended PO file editors are (in no particular order):

- XEmacs (with po-mode): runs on Unices with X
- GNU Emacs (with po-mode): runs on Unices
- KBabel: runs on KDE
- poEdit: linux and windows
  poEdit does support multiple plural forms since version 1.3.

For Mac OS X there is AquaEmacs and a port of GNU Emacs available using carbon for OS X: http://www.apple.com/downloads/macosx/unix_open_source/carbonemacspackage.html also see the Emacs wiki for more usage help and tips: http://www.emacswiki.org/cgi-bin/emacs-en/CarbonEmacsPackage po-mode is not included, but is easy to add. get it from the GNU gettext distribution.

Another PO file editor for OS X is LocFactory Editor. The ‘Lite’ version is free to use, the ‘Pro’ version costs USD 49.95. It is a relatively simple editor, without a lot of options, but it is really easy to use and ideal for simple PO file editing.
Be sure to get a recent version for all editors, multiple plural forms are a recent addition to the gettext standard.

**Issues using poEdit**

poEdit for windows, version 1.3.1 (latest at the moment) seems to require some additional steps to recognize plural forms (if you try to edit a term which has plurals, even if you translate it, it doesn’t appear in poedit when you move to an other term, as usual, and even if you save, it doesn’t).

**Plurals Solution #1**

So, if you find a plural term, close poedit, open the file you were translating with a normal text editor (no, not Word...), and search for "plural" in it, you find something similar to this:

```plaintext
#: modules/comment.module:187 modules/node.module:89
msgid "1 comment"
msgid_plural "%count comments"
msgstr[0] "1 commento"
msgstr[1] "%count commenti"
```

simply translate the text in `msgid` (singular form) into `msgstr[0]`, and the text in `msgid_plural` (plural form) into `msgstr[1]`, save the file, close the editor and return to poedit. Even better, you can do this BEFORE start translating the rest of the file with poedit, translating every occurrence of plural in the same way, in every file, and THEN start using poedit: this way, you will find those strings already translated in poedit, and they don’t bother you.

**Plurals Solution #2**

To use plurals in PO edit you can start with the catalog setting for english and then modify to suit. The syntax is:

```plaintext
nplurals=2; plural=(n != 1);
```

which gave me what I needed in Swedish translation of:

```plaintext
#: modules/aggregator.module:100;711;722
msgid "1 item"
msgid_plural "items"
msgstr[0] "1 inlägg"
msgstr[1] "%count inlägg"
```

I tested this in PO Edit 1.3.1 and got the proper GUI response and saved without error.
Plurals Solution #3

The plural forms to use in PO edit under catalog-settings where you see
nplural=INTEGER; plural=EXPRESSION

Only one form:
Some languages only require one single form. There is no
distinction between the singular and plural form. An appropriate
header entry would look like this:

Plural-Forms: nplurals=1; plural=0;

Languages with this property include:

Finno-Ugric family
Hungarian

Asian family
Japanese, Korean

Turkic/Altaic family
Turkish

Two forms, singular used for one only
This is the form used in most existing programs since it is what
English is using. A header entry would look like this:

Plural-Forms: nplurals=2; plural=n != 1;

(Note: this uses the feature of C expressions that boolean
expressions have to value zero or one.)

Languages with this property include:

Germanic family
Danish, Dutch, English, German, Norwegian, Swedish

Finno-Ugric family
Estonian, Finnish

Latin/Greek family
Greek

Semitic family
Hebrew
Romanic family
Italian, Portuguese, Spanish

Artificial
Esperanto

Two forms, singular used for zero and one
Exceptional case in the language family. The header entry would be:

Plural-Forms: nplurals=2; plural=n>1;

Languages with this property include:

Romanic family
French, Brazilian Portuguese

Three forms, special case for zero
The header entry would be:

Plural-Forms: nplurals=3; plural=n%10==1 && n%100!=11 ? 0 : n != 0 ? 1 : 2;

Languages with this property include:

Baltic family
Latvian

Three forms, special cases for one and two
The header entry would be:

Plural-Forms: nplurals=3; plural=n==1 ? 0 : n==2 ? 1 : 2;

Languages with this property include:

Celtic
Gaeilge (Irish)

Three forms, special case for numbers ending in 1[2-9]
The header entry would look like this:

Plural-Forms: nplurals=3; \nplural=n%10==1 && n%100!=11 ? 0 : \nn%10>=2 && (n%10<10 || n%100>=20) ? 1 : 2;

Languages with this property include:
Baltic family
Lithuanian

Three forms, special cases for numbers ending in 1 and 2, 3, 4, except those ending in 1[1-4]
The header entry would look like this:

```
Plural-Forms: nplurals=3; \\
    plural=n%10==1 && n%100!=11 ? 0 : \\
        n%10>=2 && n%10<=4 && (n%100<10 || n%100>=20) ? 1 \\
        : 2;
```

Languages with this property include:

Slavic family
Croatian, Czech, Russian, Slovak, Ukrainian

Three forms, special case for one and some numbers ending in 2, 3, or 4
The header entry would look like this:

```
Plural-Forms: nplurals=3; \\
    plural=n==1 ? 0 : \\
        n%10>=2 && n%10<=4 && (n%100<10 || n%100>=20) ? 1 \\
        : 2;
```

Languages with this property include:

Slavic family
Polish

Four forms, special case for one and all numbers ending in 02, 03, or 04
The header entry would look like this:

```
Plural-Forms: nplurals=4; \\
    plural=n%100==1 ? 0 : n%100==2 ? 1 : n%100==3 || n%100==4 ? 2 : 3;
```

Languages with this property include:

Slavic family
Slovenian

Long plural formulae: Those should not be broken into several lines in the header of the PO file. Drupal expects the formula to be on one line. One could consider this a bug.

I don’t think that you can use line breaks in POedit either. The text is fixed to keep from breaking the site layout. But this:
n plurals=1; plural=ar;

produces an error. The plural form "ar" is not recognized.

**Setting up XEmacs with po-mode on Windows**

XEmacs has been supported on Windows for a long time and can be downloaded from here: [http://www.xemacs.org/Download/win32](http://www.xemacs.org/Download/win32) The po-mode is bundled with XEmacs (no need to get the GNU gettext distribution).

However, you need a MULE-enabled XEmacs binary to edit the UTF-8-encoded PO files and I could not find such a binary for Windows on [www.xemacs.org](http://www.xemacs.org). What I did is this:

1. Install XEmacs 21.4.13 using the Netinstall
2. Replace the files installed in the C:\Program Files\XEmacs\XEmacs-21.4.13 directory with those from
3. Install the MULE packages in the C:\Program Files\XEmacs\mule-packages directory (these can be downloaded from [ftp://ftp.xemacs.org/xemacs/packages/xemacs-all-mule-packages.tar.gz](ftp://ftp.xemacs.org/xemacs/packages/xemacs-all-mule-packages.tar.gz)).
4. Set the environment variable EMACSPACKAGEPATH with this value:
   C:\Program Files\XEmacs\site-packages;C:\Program Files\XEmacs\mule-packages;C:\Program Files\XEmacs\xemacs-packages
5. To ensure automatic Unicode detection when opening files, add these lines to your init file (init.el):
   (require ‘un-define)
   (set-coding-priority-list ‘(utf-8))
   (set-coding-category-system ‘utf-8 ‘utf-8)
6. And finally, add this to automatically enable the po-mode:
   (require ‘po-mode)

Yes, this is a bit complicated... Welcome to the wonderful world of XEmacs! :-) If you never used XEmacs before, prepare yourself for a steep learning curve.

BTW, your installation directory does not have to be C:\Program Files\XEmacs. I used this for simplicity in the above instructions.

**Translated Drupal information**

Some documentation about Drupal (outside of the Drupal interface itself) has also been translated into other languages. Links to such 3rd party translations of external Drupal documentation should live here.
Afrikaans

This page is for the translation of Drupal Core into Afrikaans. The rest of this text will be in Afrikaans, as that is the purpose of this document.

Vir diegene wat betrokke wil raak by die vertaling:

Stuur ’n e-pos aan Kobus en spesifiseer waarmee jy betrokke sou wou raak. Kobus sal dan met jou in verbinding tree indien jy die nodige besonderhede verskaf het.

Om ’n fout met die vertaling te rapporteer:

Besoek asb. die [foutrapporteringsblad](#)

Vir enige ander verwante redes waaroor jy wil kontak:

Stuur ’n e-pos aan Kobus en spesifiseer die presiese rede vir u skakeling.

Russian

Translations and original documentation for Russian Drupal users (still very incomplete, but work is in progress):

- [Read about Drupal features in Russian](#)
- [Drupal Administrator’s Guide - a translation of the original English version](#)
- [Drupal Handbook - a translation of the original English version with some rewrites and additions](#)

Visit [drupal.ru/docs](#) for a complete list of links to Russian documentation.

[Join our efforts to translate Drupal docs into Russian?](#)

Spanish

- [Drupal: Manejador de Contenidos y Comunidad Virtual](#)
- [Drupal: Características](#)
- [Drupal: Manuales](#)

Translation guidelines

To achieve translations that are consistent throughout a whole Drupal site, certain guidelines need to be agreed upon by the translator community for a particular language.
Such guidelines should include a wordlist for words that occur in Drupal's strings. The Ankur Bangla Developer's Guide provides a good example of how this is done on a project unrelated to Drupal. It will be helpful to not set up a new word lists, but re-use existing ones from an existing translation project.

Other areas which need guidelines will differ from language to language. Please add those guidelines as child pages to this book page.

Translation of contributed modules

Translatable strings from contributed modules are not included in the Drupal core POT files. Module authors can use the extractor.php script which comes with the core PO files to generate a POT file on their own. Instructions for running the script can be found in the README that comes with the core POT files. The generated POT file should be named as the module, but with a .pot extension, e.g. event.module gets an event.pot file. This file should be placed in a subdirectory po. Translations should be added to the same directory. E.g. the po subdirectory of event.module currently contains the following files: de.po, es.po, event.pot, he.po, hu.po.

Translators should take care to populate their started translation with the strings from the general.po file for their language using msgmerge. In this way they can avoid using different translations for terms that occur in both files.

Distributing the translation effort

To facilitate easier handling of a community translation effort, the Drupal POT file is split up into small files that do not contain doubly occurring strings.

All strings that occur more than once in the Drupal core distribution are put into the general.pot file. This ensures that those strings are translated to the same string. Also, files that have ten or less translatable strings will not get their own POT file, but those strings will be appended to the general.pot file.

Of course, some coordination among the project members is still needed to ensure the quality of the translation.

If a language has several options on how to translate some strings, then it is possible to create PO files that only change those strings. An example would be German where you can translate you either as Du or Sie depending on the audience of your site.

Status of the translations

The table below presents an overview of the status of each translation project. This page is updated daily by the package script: it was last updated 1 hour 24 sec ago.
# Status overview

<table>
<thead>
<tr>
<th>Language</th>
<th>cvs</th>
<th>4.7.0</th>
<th>4.6.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>af</td>
<td>85% (300 missing)</td>
<td></td>
<td>99% (5 missing)</td>
</tr>
<tr>
<td>ar</td>
<td>93% (145 missing)</td>
<td>97% (47 missing)</td>
<td>43% (1037 missing)</td>
</tr>
<tr>
<td>bg</td>
<td>69% (643 missing)</td>
<td>81% (348 missing)</td>
<td></td>
</tr>
<tr>
<td>bn</td>
<td>2% (2066 missing)</td>
<td></td>
<td>3% (1771 missing)</td>
</tr>
<tr>
<td>ca</td>
<td>69% (658 missing)</td>
<td>66% (698 missing)</td>
<td>96% (58 missing)</td>
</tr>
<tr>
<td>cs</td>
<td>100% (complete)</td>
<td>100% (complete)</td>
<td>98% (23 missing)</td>
</tr>
<tr>
<td>da</td>
<td>96% (65 missing)</td>
<td></td>
<td>100% (complete)</td>
</tr>
<tr>
<td>de</td>
<td>90% (204 missing)</td>
<td>74% (544 missing)</td>
<td>88% (205 missing)</td>
</tr>
<tr>
<td>el</td>
<td>38% (1306 missing)</td>
<td></td>
<td>44% (1011 missing)</td>
</tr>
<tr>
<td>es</td>
<td>96% (79 missing)</td>
<td>97% (55 missing)</td>
<td>100% (complete)</td>
</tr>
<tr>
<td>et</td>
<td>71% (598 missing)</td>
<td>translation broken</td>
<td>69% (566 missing)</td>
</tr>
<tr>
<td>eu</td>
<td>49% (1073 missing)</td>
<td></td>
<td>57% (778 missing)</td>
</tr>
<tr>
<td>fa</td>
<td>translation broken</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fi</td>
<td>72% (581 missing)</td>
<td></td>
<td>84% (286 missing)</td>
</tr>
<tr>
<td>fr</td>
<td>85% (315 missing)</td>
<td></td>
<td>99% (18 missing)</td>
</tr>
<tr>
<td>gl</td>
<td>52% (1018 missing)</td>
<td></td>
<td>60% (723 missing)</td>
</tr>
<tr>
<td>gu</td>
<td>80% (420 missing)</td>
<td></td>
<td>16% (1527 missing)</td>
</tr>
<tr>
<td>he</td>
<td>85% (317 missing)</td>
<td></td>
<td>98% (22 missing)</td>
</tr>
<tr>
<td>hu</td>
<td>84% (322 missing)</td>
<td></td>
<td>98% (25 missing)</td>
</tr>
<tr>
<td>id</td>
<td>70% (625 missing)</td>
<td></td>
<td>81% (330 missing)</td>
</tr>
<tr>
<td>is</td>
<td>84% (333 missing)</td>
<td></td>
<td>98% (29 missing)</td>
</tr>
<tr>
<td>it</td>
<td>98% (23 missing)</td>
<td>85% (308 missing)</td>
<td>93% (115 missing)</td>
</tr>
<tr>
<td>ja</td>
<td>85% (304 missing)</td>
<td></td>
<td>100% (complete)</td>
</tr>
<tr>
<td>kn</td>
<td>6% (1980 missing)</td>
<td></td>
<td>8% (1685 missing)</td>
</tr>
<tr>
<td>ko</td>
<td>65% (737 missing)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lt</td>
<td>53% (996 missing)</td>
<td></td>
<td>61% (701 missing)</td>
</tr>
</tbody>
</table>
Checking your translation status

To see how many of the strings in the PO files you already translated you can try this:

```bash
for i in *.po; do echo -n "$i: " ; msgfmt --statistics $i ; done
```

Some PO editors already include this feature.

Make a single file from the loose .po files from CVS

If you want to make a single po file from a CVS folder containing all the small po files, the following commands will do (*nix only). You should execute this, while being in the folder with the .po files.
Recycling old translations

Drupal users with existing translations might want to add those to the translations download page. To do this they first need to export their translation from the localization manage languages screen (export subtab). Let us assume you have an Italian translation. The above mentioned process will create an it.po file for you. To use this file as a basis for a new translation, you treat it as a PO compendium, i.e. a library of pre-translated strings.

This guide assumes a Unix/Linux environment. If you use Windows, check if your PO editor doesn’t have a function for this.

We will split the single, large PO file into the smaller files that the Drupal translation Project requires.

First, put the small PO files into a subdirectory drupal-pot and your it.po file into another one. Then create an empty directory where you want to keep your new small PO files.

Then go to the empty directory and execute the following command from the command line:

```
for i in /path/to/drupal-pot/*.pot ; do msgmerge --compendium /path/to/it.po -o `basename $i .pot`.po /dev/null $i ; done
```

After a while (yes this will take a few minutes) you should have a directory of small PO files that have the matching strings inserted.

Troubleshooting

When doing translations or importing them, several problems can occur. If you think you found a bug in either a translation or in Drupal’s locale module, please file bug reports against the project in question.

If you have a more general question you can ask it in the translations forum.

Here we collect some of the more common issues found.

Some strings do not translate

Symptom: After importing a translation, some strings on your site are translated, some are not.

Possible causes (and solutions):

$ msgcat --use-first general.po [*g]*.po | msgattrib --no-fuzzy -o nl.po

Off course you should change nl into your own language code.
1. The imported translation is not complete. Consider finishing it and contributing your update by filing an issue against the translation.

2. The actual source strings have been modified after the translation file was created. You might need to update the PO file from a fresh POT file (translation template). The latest core POT files are available for download but you might need to generate fresh POT files yourself.

3. There are end-of-line inconsistencies between the actual source strings and the source strings from the PO file. All Drupal files have Unix-style end-of-lines (\n), but these sometimes get messed up on Windows (which normally uses \r\n). WinZip users beware: Make sure the "TAR file smart CR/LF conversion" option is not checked before extracting any Drupal tarball.

**Weird characters or question marks**

Symptom: After importing a translation you find all kind of weird characters or question marks on your site.

Solution 1: The translator did not use UTF-8. Drupal is fully UTF-8 aware and expects translations to be supplied in that character set as well. You can change the charset of a PO file using GNU msgconv. Please file a bug against the translation in question.

Solution 2: You do not have the correct font installed to display the language in question.
Bazaar-NG

Bazaar-NG is a decentralized revision control system. The key point of decentralized revision control systems is that third parties can branch off of the official drupal branches and perform work under revision control without requiring permission from the core drupal team.

Installation

Bazaar-NG is nearly as portable as CVS, though installation is a little trickier. The general rule of thumb is that the further one gets from a GNU/Linux installation, the more effort the installation.

Installation on Ubuntu & Debian based systems

This platform is by far the easiest system to install Bazaar-NG on. Installing on this platform essentially involves adding a source line for the nightly built packages, updating the packages list and installing Bazar-NG and a couple related tools. Once the sources has been done any distro tool -- whether apt-get, aptitude, or synaptic can be used to install and keep current an installation of Bazaar-NG. This document will use apt, since its command line based and can be easily expressed in book format. Feel free to use the one of your choice and follow along conceptually! I'll also assume that you have used su to become root.

One of the members of the Bazaar-NG is kind enough to make and distributes nightly debs of Bazaar-NG and related packages. You can gain access to these packages via apt or synaptic by adding the following line to /etc/apt/sources.list:

```
deb http://people.ubuntu.com/~jbailey/snapshot/bzr ./
```

Now you'll need to update your packages list. In apt we perform this step by running "apt-get update". This will update the packages that are available on your system.

Finally, perform the actual installation by running the command "apt-get install bzr bzrtools".

Congratulations, you now have Bazaar-NG installed!

Installation on Cygwin

The second most challenging platforms to install bazaar-NG on is Cygwin. Though each step involved in installation isn't difficult, a lot of them exist.

The first thing that you'll want to do is to ensure that python 2.4 installed. You can check this by running "python -V". If the python that you have is earlier than this, or if you don't have python at all, then use the cygwin package manager to install the newer one. Install openssh and rsync as well, while you’re at it.
The second thing that you’ll do is manually download and install a few python packages. No worries; this process is easy. First, download the latest package for each of these modules:
1. cElementTree
2. pyCrypto
3. Paramiko

The third thing you need to do is perform two steps for each downloaded module:
1. unpack the latest version
2. cd into the unpacked directory and run "python setup.py install"

Now, it's time to download a copy of bzr. Since you’ve installed rsync, we’ll use that here:

```
$ cd /usr/local/lib
$ rsync -av bazaar-ng.org::bazaar-ng/bzr/bzr.dev bzr.dev
$ ln -s /usr/local/lib/bzr.dev/bzr /usr/local/bin/bzr
```

Congratulations! You now have Bazaar-NG installed. You’re not done quite yet. As the last part of installation, we’re going to use Bazaar-NG to install a plugin for itself called bzrtools. We do that this way:

```
$ cd ~
$ mkdir -p .bazaar/plugins
$ cd .bazaar/plugins
$ bzr branch http://panoramicfeedback.com/opensource/bzr/bzrtools/ bzrtools
```

## Installation on OS X

While the Apple Developer Tools come with python, you will need upgrade to python 2.4 in order to install the latest version of Bazaar-NG.

You can get the python binaries from [http://undefined.org/python/](http://undefined.org/python/)

If you are running 10.4 (Tiger), you will need to also install the TigerPython24Fix, also available from [http://undefined.org/python/](http://undefined.org/python/)

If you want to do pushes or pulls over SFTP you will need to get and install:

1. pyCrypto
2. Paramiko

Directions are included with the tar.gz files.

Finally, you can install bzr. [Download the source](http://bzrtools.org) and away you go.

The windows installation guide also has a [convenient write up on bzrtools](http://bzrtools.org).
You can also get Bazaar-NG via the [DarwinPorts system](http://www.apple.com/ports/).

As of this writing the [Fink project](http://www.fink-project.org) is a little behind the times for Bazaar-NG.

### Setting up Bazaar-NG

When one commits to a branch in Bazaar-NG several things are saved. Some of these things include the modifications you made and the message that you gave when you committed. Another thing that is saved is the email address of the committer. This way, when somebody wants to chase down the person that hacked on some code, they know who to contact! Bazaar-NG is happy to make a guess, though the guess is usually not as accurate as setting it yourself.

Several methods exist for telling Bazaar-NG what your email address is. All of these methods are listed on the [Setting Email in Bzr](http://www.bazaar-NG.org/bzrja.html) web page. The easiest method among those listed is setting one in a special file in your home directory. This is done by making a directory to hold a Bazaar-NG config file. In pure windows this file is `%APPDATA%azaar\2.0\bazaar.conf`. In cygwin and other unixlike operating systems the file is `~/.bazaar/bazaar.conf`.

Inside of this file you’ll want to add two lines:

```
[DEFAULT]
email=Your Name <name@isp.com>
```

Now, every time you commit, you will be credited with the patches that you made. Now, whenever somebody merges your code, you will get credit for your hard work.

### Getting Drupal Head via Bazaar-NG

Once Bazaar-NG is installed and set up you can use Bazaar-NG to download and then keep updated the current development version of drupal. One downloads a development version of drupal via Bazaar-NG by running the following command:

```
$ bzr branch http://drupal.revisioncontrol.net/core/head [directory you want it in]
```

I’ll assume that you decided to name the directory "drupal.dev". You can now go into the `drupal.dev` directory/folder and verify that this is indeed a full copy of the head version of drupal. You’ll see `upgrade.php`, `databases/` and `LICENSE.txt` among other things. This is your copy of drupal, fit to edit as you see fit.

You can update your copy of drupal whenever you like. If you have never made changes to your drupal then updating is as simple as running:

```
$ bzr pull
```
We’ll cover how to update a drupal branch that you have modified in a later chapter. Don’t worry; the process is almost as easy.

**Hacking your local Drupal**

Bazaar-NG is not like CVS. If you have a copy of a branch in Bazaar-NG then you can commit to it without getting permission from anyone. This kind of makes sense; if the branch is on your hard drive, then you should be able to commit to it!

There are a few rules to follow to making local commits:

* You will want to commit your work from time to time. This is performed with `bzr commit -m"A description of your changes"`
* If you add a new file or directory to your drupal branch then you’ll want to run "bzr add ".
* If you want to take a file or directory out of revision control, then you do so with "bzr remove ". Doing this will not delete the file, but will tell Bazaar-NG to stop tracking it for you.
* You can ignore files by running "bzr ignore "
* Once you have committed to your drupal branch "bzr pull" will no longer work. This is because the branch is no longer strictly a copy of the official development drupal branch. Now, instead of pulling new changes, you will have to merge them.

When you commit to a branch then those changes will stay in your branch until somebody else merges them. This is because Bazaar-NG does not check into a central server like cvs does. This means that if you want somebody to merge your code, that you’ll have to put a copy of a branch in a place where other people can reach it. They in turn can either branch or merge from you. We’ll cover this in a different chapter.

**Updating your Branch with Official Drupal Dev**

A branch that is downloaded from the drupal bzr server is just a copy of a branch. These sorts of branches, as we’ve already covered, are updated simply by running "bzr pull". This only works, though, if your branch is _just_ a copy. If you have committed to a branch, then a slightly different approach needs to be taken. A branch that has been committed to is a "diverged branch". These branches are different, but equally valid, from the branch they came from.

One merges changes between diverged branches with the "bzr merge" command. In your case, your branch will have diverged from the official drupal.dev branch. You can merge from drupal.dev in this way:

```
$ bzr merge
```

This will figure out what changes you are missing from official drupal.dev and apply them to your version of drupal. You’re still not done, however. You should also review the changes in your branch and decide whether or not you want them. You can see what changed by running "bzr st" and how they changed by "bzr diff". Presumably you’ll want to save the changes. You can do this the same way that you’ve always saved changes: by "bzr commit -m’merged from"
Getting Merged

When your drupal branch is ready for merging you’ll have to put it in a public place so that other people can get it. Publically exposing one of your branches involves "pushing the branch". One can push a branch via sftp or rsync if BzrTools is installed.

One pushes a branch in the following manager:

$ cd my-drupal-fix
$ bzr push sftp://hostname.com/public_html/my-drupal-fix

Once you have pushed your branch you’ll want other people to know about it so that they can merge you. You can do this by registering your branch on Drupal Launchpad page. New branches should be marked as "NEW".

The person responsible for merging Drupal branches will merge your code in the same way that you merge Drupal code. He or she will then review your code and will email you. Two possible reactions are likely: either your hacks were accepted and merged, or your hacks need some more work. Don’t get discouraged if you need to do more work; just fix the outstanding problems, commit, push again and ask that your branch is reviewed.

*NOTE* There is currently no person responsible for merging. Please see the next chapter about generating a diff.

Getting a diff against core/head

Another way that one can submit changes to Drupal with Bazaar-NG is by generating a diff. Diffs are a little bit more awkward to work with, but do fit well within the current system.

You can generate a diff against another branch at any time by using the bzr diff command. For example, to generate a diff against the official Drupal development tree, one would run:

$ cd my-drupal-branch
$ bzr diff -r branch:http://drupal.revisioncontrol.net/core/head > ~/mypatch

One important note though! Make sure that you merge from the official drupal branch and commit before you generate your diff! Otherwise, bzr will think that you mean for the changes in official Drupal development should be reverted!

If you find it a pain in the rear to constantly have to type so much whenever you want a diff, then consider putting the following line in your ~/.bashrc. The next time that you run in, you can compare the differences in your branches against official drupal whenever you want by running "ddiff".
alias ddiff='bzr diff -r branch:http://drupal.revisioncontrol.net/core/head'