TYPO3 Core APIs

Extension Key: doc_core_api
Language: en
Keywords: tsref, typoscript, reference, forDevelopers, forAdvanced
Copyright 2000-2011, TYPO3 Core Development Team, <info@typo3.org>

This document is published under the Open Content License
available from http://www.opencontent.org/opl.shtml

The content of this document is related to TYPO3
- a GNU/GPL CMS/Framework available from www.typo3.org

Revised for TYPO3 4.3, March 2010

Table of Contents

Introduction

Overview

TYPO3 is known for its extensibility. However to really benefit from this power, a complete documentation is needed. "Core APIs" and its companion, "Inside TYPO3", aim to provide such information to developers and administrators. Not all areas are covered with the same amount of details, but at least some pointers are provided.

"Inside TYPO3" contains the overall introduction to the architecture of the TYPO3 core. It also contains API descriptions to a certain degree but mostly in the form of examples and short table listings. "Core APIs" goes into much more detail about such APIs and covers subjects more closely related to development.

These documents do *not* contain any significant information about the frontend of TYPO3. Creating templates, setting up TypoScript objects etc. is not the scope of these documents; they are about the *backend* part of the core only.

The TYPO3 Core Team hopes that these two documents, "Inside TYPO3" and "TYPO3 Core API", will form a complete picture of the TYPO3 Core architecture, the backend and be the reference of choice in your work with TYPO3. It took Kasper more than a year to get the first version published and we've tried to maintain it as best we could.

Code examples

Many of the code examples found in this document come from the TYPO3 Core itself.

Quite a few others come from the "examples" extension which is available in the TER. You can install it if you want to try out these examples yourself and use them as a basis for your own stuff.

Yet some other examples just belong to this manual. Some may be moved to the "examples" extension at some later stage.

Dedication

I want to dedicate this document to the people in the TYPO3 community who have the *discipline* to do the boring job of writing documentation for their extensions or contribute to the TYPO3 documentation in general. It's great to have good coders, but it's even more important to have coders with character to carry their work through till the end - even when it means spending days writing good documents. Go for completeness!

- kasper

TYPO3 Extension API

Extension Architecture

Introduction

TYPO3 can be extended in nearly any direction without loosing backwards compatibility. The Extension API provides a powerful framework for easily adding, removing, installing and developing such extensions to TYPO3. This is in particular powered by the Extension Manager (EM) inside TYPO3 and the online TYPO3 Extension Repository (TER) found at typo3.org for easy sharing of extensions.

"*Extensions*" is a general term in TYPO3 which covers many kinds of additions to TYPO3. The main types are:

- **Plugins** which play a role on the website itself, e.g. a discussion board, guestbook, shop, etc. It is normally enclosed in a PHP class and invoked through a USER or USER_INT cObject from TypoScript. A plugin is an extension in the frontend.
- **Modules** are backend applications which have their own entry in the main menu. They require a backend login and work inside the framework of the backend. We might also call something a module if it exploits any connectivity of an existing module, that is if it simply adds itself to the function menu of existing modules. A module is an extension in the backend.
- **Services** are libraries that provide a given service through a clearly defined API. A service may exist both in the frontend and the backend. Please refer to the "TYPO3 Services" manual (doc_core_services) for more information about this type of extension.

Extensions and the Core

Extensions are designed in a way so that extensions can supplement the core seamlessly. This means that a TYPO3 system will appear as "a whole" while actually being composed of the core application *and* a set of extensions providing various features. This philosophy allows TYPO3 to be developed by many individuals without loosing fine control since each developer will have a special area (typically a system extension) of responsibility which is effectively encapsulated.

So, in one end of the spectrum system extensions makes up what is known as "TYPO3" to the outside world. In the other end, extensions can be 100% project specific and carry only files and functionality related to a single implementation.

Files and locations

Files

An extension consists of

1. a directory named by the *extension key* (which is a worldwide unique identification string for the extension unless prefix with "user_")
2. standard files with reserved names for configuration related to TYPO3 (of which most are optional, see list below)
3. any number of additional files for the extension itself.

Reserved filenames

This list of filenames are all reserved filenames in the root directory of extensions. None of them are required but for example you cannot have a TYPO3 extension recognized by TYPO3 without the "ext_emconf.php" file etc. You can read more details like that in the table below.

In general, do not introduce your own files in root directory of extensions with the name prefix "ext_".

Filename	Description
ext_emconf.php	Definition of extension properties. Name, category, status etc. Used by the EM. The content of this file is described in more details below. Note that it is auto-written by EM when extensions are imported from the repository. **Notice:** If this file is *not* present the EM will *not* find the extension.

Filename	Description
ext_localconf.php	Addition to `localconf.php` which is included if found. Should contain additional configuration of $TYPO3_CONF_VARS and may include additional PHP class files. All `ext_localconf.php` files of included extensions are included right **after** the `typo3conf/localconf.php` file has been included and database constants defined. Therefore you cannot setup database name, username, password though, because database constants are defined already at this point. **Notice:** Observe rules for content of these files. See section on caching below.
ext_tables.php	Addition to `tables.php` which is included if found. Should contain configuration of tables, modules, backend styles etc. Everything which can be done in an "extTables" file is allowed here. All `ext_tables.php` files of loaded extensions are included right **after** the `tables.php` file in the order they are defined in the global array `TYPO3_LOADED_EXT` but right before a general "extTables" file (defined with the var `$typo_db_extTableDef_script` in the `typo3conf/localconf.php` file, later set as the constant `TYPO3_extTableDef_script`). Thus a general "extTables" file in typo3conf/ may overrule any settings made by loaded extensions. You should *not* use this file for setting up `$TYPO3_CONF_VARS`. See "ext_localconf.php" above. **Notice:** Observe rules for content of these files. See section below.
ext_tables.sql	SQL definition of database tables. This file should contain a table-structure dump of the tables used by the extension. It is used for evaluation of the database structure and is therefore important to check and update the database when an extension is enabled. If you add additional fields (or depend on certain fields) to existing tables you can also put them here. In that case insert a CREATE TABLE structure for that table, but remove all lines except the ones defining the fields you need. The ext_tables.sql file may not necessarily be "dumpable" directly to MySQL (because of the semi-complete table definitions allowed defining only required fields, see above). But the EM or Install Tool can handle this. The only very important thing is that the syntax of the content is exactly like MySQL made it so that the parsing and analysis of the file is done correctly by the EM.
ext_tables_static+adt.sql	Static SQL tables and their data. If the extension requires static data you can dump it into a sql-file by this name. Example for dumping mysql data from bash (being in the extension directory): `mysqldump --password=[password] [database name] [tablename] --add-drop-table > ./ext_tables_static.sql` `--add-drop-table` will make sure to include a DROP TABLE statement so any data is inserted in a fresh table. You can also drop the table content using the EM in the backend. **Notice:** The table structure of static tables needs to be in the ext_tables.sql file as well - otherwise an installed static table will be reported as being in excess in the EM!
ext_typoscript_constants.txt	Preset TypoScript constants *Deprecated (use static template files instead, see extMgm API description)* Such a file will be included in the constants section of all TypoScript templates.
ext_typoscript_setup.txt	Preset TypoScript setup *Deprecated (use static template files instead, see extMgm API description)* Such a file will be included in the setup section of all TypoScript templates.
ext_typoscript_editorcfg.txt	Preset TypoScript editor configuration *Deprecated (use static template files instead, see extMgm API description)* Such a file will be included in the "Backend Editor Configuration" section of all TypoScript templates.

Filename	Description
ext_conf_template.txt	Extension Configuration template. Configuration code in TypoScript syntax setting up a series of values which can be configured for the extension in the EM. If this file is present the EM provides you with an interface for editing the configuration values defined in the file. The result is written as a serialized array to `localconf.php` file in the variable `$TYPO3_CONF_VARS['EXT']['extConf'][extension_key]` The content of the "res/" folder is used for filelists in configuration forms. If you want to do user processing before the content from the configuration form is saved (or shown for that sake) there is a hook in the EM which is configurable with `$TYPO3_CONF_VARS['SC_OPTIONS']['typo3/mod/tools/em/index.php']['tsStyleConfigForm'][] = "function reference"`
ext_icon.gif	Extension Icon 18x16 gif icon for the extension.
(*/) locallang*.xml	Localization values. The filename `locallang.xml` (or any file matching `locallang*.xml`) is used for traditional definition of language labels in the `$LOCAL_LANG` array. If you use this name consistently those files will be detected by the translation tool! **Notice:** PLEASE DO ONLY put the definition of the variable `$LOCAL_LANG` into this file and don't rely on comments in the file. The file will be automatically updated by the extension repository when translations are applied.
class.ext_update.php	Local Update tool class If this file is found it will install a new menu item, "UPDATE", in the EM when looking at details for the extension. When this menu item is selected the class inside of this file (named "ext_update") will be instantiated and the method "main()" will be called and expected to return HTML content. Also you must add the function "access()" and make it return a boolean value whether or not the menu item should be shown. This feature is meant to let you disable the update tool if you can somehow detect that it has already been run and doesn't need to run again. The point of this file is to give extension developers the possibility to provide an update tool if their extensions in newer versions require some updates to be done.
ext_autoload.php	Since TYPO3 4.3, it is possible to declare classes in this file so that they will be automatically detected by the TYPO3 autoloader. This means that it is not necessary to require the related class files anymore. See the "Autoloading" chapter for more details.
ext_api_php.dat	PHP API data A file containing a serialized PHP array with API information for the PHP classes in the extension. The file is created - and viewed! - with tools found in the extension "extdeveval" (Extension Development Evaluator)
pi*/	Typical folder for a frontend plugin class.
mod*/	Typical folder for a backend module.
sv*/	Typical folder for a service.
res/	Extensions normally consist of other files: Classes, images, html-files etc. Files not related to either a frontend plugin (pi/) or backend module (mod/) might be put in a subfolder of the extension directory named "res/" (for "resources") but you can do it as you like (inside of the extension directory that is). The "res/" folder content will be listed as files you can select in the configuration interface. Files in this folder can also be selected in a selector box if you set up Extension configuration in a "ext_conf_template.txt" file.

System, Global and Local extensions

The files for an extension are located in a folder named by the *extension key*. The location of this folder can be either inside typo3/sysext/, typo3/ext/ or typo3conf/ext/.

The extension *must* be programmed so that it does automatically detect where it is located and can work from all three locations. If it is not possible to make the extension that flexible, it is possible to lock its installation requirement to one of these locations in the emconf.php file (see "lockType")

Type	Path	Description
Local	typo3conf/ext/	This is where to put extensions *which are lo*cal for a particular TYPO3 installation. The typo3conf/ dir is always local, containing local configuration (e.g. *lo*calconf.php), local modules etc. If you put an extension here it will be available for a single TYPO3 installation only. This is a "per-database" way to install an extension. **Notice about symlinking:** Local extension can successfully be symlinked to other local extensions on a server as long as they are running under the same TYPO3 source version (which would typically also be symlinked). This method is useful for maintenance of the same local extension running under several sites on a server.
Global	typo3/ext/	This is a "per-server" way to install an extension; they are global for the TYPO3 source code on the web server. These extensions will be available for any TYPO3 installation sharing the source code. **Notice on distribution:** As of version 4.0, TYPO3 is no longer distributed with a fixed set of global extensions. In previous versions these were distributed for reasons like popularity and sometimes history.
System	typo3/sysext/	This is system default extensions which cannot and should not be updated by the EM. They are distributed with TYPO3 core source code and generally understood to be a part of the core system.

Loading precedence

Local extensions take precedence which means that if an extension exists both in typo3conf/ext/ and typo3/ext/ the one in typo3conf/ext/ is loaded. Likewise *global* extension takes precedence over *system* extensions. This means that extensions are loaded in the order of priority local-global-system.

In effect you can therefore have - say - a "stable" version of an extension installed in the global dir (typo3/ext/) which is used by all your projects on a server sharing source code, but on a single experimental project you can import the same extension in a newer "experimental" version and for that particular project the locally available extension will be used instead.

Extension key

The "extension key" is a string uniquely identifying the extension. The folder where the extension resides is named by this string. The string can contain characters a-z0-9 and underscore. No uppercase characters should be used (keeps folder-,file- and table/field-names in lowercase). Furthermore the name must not start with an "tx" or "u" (this is prefixes used for modules) and because backend modules related to the extension should be named by the extension name *without* underscores, the extension name must still be unique even if underscores are removed (underscores are allowed to make the extension key easily readable).

The naming conventions of extension keys are automatically validated by the registration at the repository, so you have nothing to worry about here.

There are two ways to name an extension:

- **Project specific extensions** (not generally usable or shareable): Select any name you like and prepend it "user_" (which is the only allowed use of a key starting with "u"). This prefix denotes that this extension is a local one which does not come from the central TYPO3 Extension Repository or is ever intended to be shared. Probably this is an "adhoc" extension you have made for some special occasion.
- **General extensions:** Register an extension name online at the TYPO3 Extension Repository. Your extension name will automatically be validated and you are sure to have a unique name returned which nobody else in the world uses. This makes it very easy to share your extension later on with every one else, because it ensures that no conflicts with other extension will happen. But by default a new extension you make is defined "private" which means nobody else but you have access to it until you permit it to be public.
  It's free of charge to register an extension name. By definition all code in the TYPO3 Extension Repository is covered by the GPL license because it interfaces with TYPO3. You should really consider making general extensions!

Suggestion: It is far the easiest to settle for the right extension key from the beginning. Changing it later involves a cascade of name changes to tables, modules, configuration files etc.

About GPL and extensions: Remember that TYPO3 is GPL software and at the same moment you extend TYPO3 your extensions are legally covered by GPL. This does not *force* you to share your extension, but it should *inspire* you to do so and legally you cannot prevent anyone who gets hold of your extension code from using it and further develop it.
The TYPO3 Extension API is designed to make sharing of your work easy as well as using others work easy. Remember TYPO3 is Open Source Software and we rely on each other in the community to develop it further.

Responsibility: It's also your responsibility to make sure that all content of your extensions is legally covered by GPL. The webmaster of TYPO3.org reserves the right to kick out any extension *without notice* that is reported to contain non-GPL material.

Security: You are responsible for security issues in your extensions. People may report security issues either directly to you or to the TYPO3 Security Team. Whatever the case you should get in touch with the Security Team who will validate the security fixes. They will also include information about your (fixed) extension in their next Security bulletin. If you don't respond to requests from the Security Team, your extension will be forcibly removed from the TYPO3 Extension Repository.

More details on the security team's policy on handling security issues can be found at http://typo3.org/teams/security/extension-security-policy/.

Naming conventions

Based on the extension key of an extension these naming conventions should be followed:

	General	Example	User specific	Example
Extension key (Lowercase "alnum" + underscores.)	Assigned by the TYPO3 Extension Repository.	cool_shop	Determined by yourself, but prefixed "user_"	user_my_shop
Database tables and fields	Prefix with "tx_[*key*]_" where key is *without* underscores!	**Prefix:** tx_coolshop_ **Examples:** tx_coolshop_products tx_coolshop_categories	Prefix with "[*key*]_"	**Prefix:** user_my_shop_ **Examples:** user_my_shop_products user_my_shop_categories
Backend module (Names are always *without* underscores!)	Name: The extension key name *without* underscores, prefixed "tx"	txcoolshop	Name: No underscores, prefixed "u"	uMyShop or umyshop or ...
Frontend PHP classes	*(Same as database tables and fields. Prepend class file names "class." though.)*			

You may also want to refer to the TYPO3 Core Coding Guidelines for more on general naming conventions in TYPO3.

Best practice on using underscores

If you study the naming conventions above closely you will find that they are complicated due to varying rules for underscores in key names. Sometimes the underscores are stripped off, sometimes not.

The best practice you can follow is to *avoid using underscores* in your extensions keys at all! That will make the rules simpler. This is highly encouraged.

Note on "old" and default extensions:

Some the "classic" extensions from before the extension structure came about does not comply with these naming conventions. That is an exception made for backwards compatibility. The assignment of new keys from the TYPO3 Extension Repository will make sure that any of these old names are not accidentially reassigned to new extensions.

Further, some of the classic plugins (tt_board, tt_guest etc) users the "user_" prefix for their classes as well.

Import and install of extensions

There are only two (maybe three) simple steps involved in using extensions with TYPO3:

1. You must *import* it.
   This simply means to copy the extensions files into the correct directory in either typo3/ext/ (global) or typo3conf/ext/ (local). More commonly you import an extension directly from the online TYPO3 Extension Repository. When an extension is found located in one of the extension locations, it is *available* to the system.
   The EM should take care of this process, including updates to newer versions if needed.
   Notice that backend modules will have their "conf.php" file modified in the install process depending on whether they are installed locally or globally!

2. You must *install* it.
   An extension is loaded only if its extension key is listed in comma list of the variable $TYPO3_CONF_VARS["EXT"]["extList"]. The list of enabled extensions must be set and modified from inside typo3conf/localconf.php. Extensions are loaded in the order they appear in this list. Any extensions listed in $TYPO3_CONF_VARS["EXT"]["requiredExt"] will be forcibly loaded before any extensions in $TYPO3_CONF_VARS["EXT"]["extList"].
   An enabled extension is always global to the TYPO3 Installation - you cannot disable an extension from being loaded in a particular branch of the page tree.
   The EM takes care enabling extensions. It's highly recommended that the EM is doing this, because the EM will make sure the priorities, dependencies and conflicts are managed according to the extension characteristics, including clearing of the cache-files if any.

3. You *might* need to configure it.
   Certain extensions may allow you to configure some settings. Again the EM is able to handle the configuration of the extensions based on a certain API for this. Any settings - if present - configured for an extension is available as an array in the variable $TYPO3_CONF_VARS["EXT"]["extConf"][*extension key*].

Loaded extensions are registered in a global variable, $TYPO3_LOADED_EXT, available in both frontend and backend of TYPO3. The loading and registration process happens in t3lib/config_default.php. Since TYPO3 4.3, when rendering FE content (TYPO3_MODE = FE), $TYPO3_LOADED_EXT contains only extensions where the $EM_CONF option "doNotLoadInFE" is not set.

This is how the data structure for an extension in this array looks:

```
$TYPO3_LOADED_EXT[extension key] = array(
      “type” =>         S, G, L for system, global or local type of availability.
      “siteRelPath” => Path of extension dir relative to the PATH_site constant
                        e.g. “typo3/ext/my_ext/” or “typo3conf/ext/my_ext/”
      “typo3RelPath” => Path of extension dir relative to the “typo3/” admin folder
                        e.g. “ext/my_ext/” or “../typo3conf/ext/my_ext/”
      “ext_localconf” => Contains absolute path to 'ext_localconf.php' file if present
      “ext_tables” => [same]
      “ext_tables_sql” => [same]
      “ext_tables_static+adt.sql” => [same]
      “ext_typoscript_constants.txt” => [same]
      “ext_typoscript_setup.txt” => [same]
      “ext_typoscript_editorcfg.txt” => [same]
)
```

The order of the registered extensions in this array corresponds to the order they were listed in TYPO3_CONF_VARS["EXT"]["requiredExt"].TYPO3_CONF_VARS["EXT"]["extList"] with duplicates removed of course.

The inclusion of ext_tables.php or ext_localconf.php files are done by traversing (a copy of) the $TYPO3_LOADED_EXT array.

ext_tables.php and ext_localconf.php

These two files are the most important for the execution of extensions to TYPO3. They contain configuration used within the system on almost every request. Therefore they should be optimized for speed.

- ext_localconf.php is always included in global scope of the script, either frontend or backend.
  You *can* put functions and classes into the script, but you should consider doing that in other ways because such classes and functions would *always* be available - and it would be better if they were included only when needed.
  So stick to change values in TYPO3_CONF_VARS only!
- ext_tables.php is *not* always included in global scope on the other hand (in the frontend)
  Don't put functions and classes - or include other files which does - into this script!
- Use the API of the class extMgm for various manipulative tasks such as adding tables, merging information into arrays etc.
- Before the inclusion of any of the two files, the variables $_EXTKEY is set to the extention-key name of the module and $_EXTCONF is set to the configuration from $TYPO3_CONF_VARS["EXT"]["extConf"][*extension key*]
- $TYPO3_LOADED_EXT[*extension key*] contains information about whether the module is loaded as *local, global* or *system* type, including the proper paths you might use, absolute and relative.
- The inclusion can happen in two ways:
  - 1) Either the files are included individually on each request (many file includes) ($TYPO3_CONF_VARS["EXT"]["extCache"]=0;)
  - 2) or (better) the files are automatically imploded into one single temporary file (cached) in typo3conf/ directory (only one file include) ($TYPO3_CONF_VARS["EXT"]["extCache"]=1; [or 2]). This is default (value "1")

  In effect this means:

  - Your ext_tables.php / ext_localconf.php file must be designed so that it can safely be read and subsequently imploded into one single file with all the other configuration scripts!
  - You must NEVER use a "return" statement in the files global scope - that would make the cached script concept break.
  - You should NOT rely on the PHP constant __FILE__ for detection of include path of the script - the configuration might be executed from a cached script and therefore such information should be derived from the $TYPO3_LOADED_EXT[*extension key*] array. E.g. $TYPO3_LOADED_EXT[$_EXTKEY]["siteRelPath"]

ext_emconf.php

This script configures the extension manager. The only thing included is an array, `$EM_CONF[extension_key]` with these associative keys (below in table).

When extensions are imported from the online repository this file is auto-written! So don't put any custom stuff in there - only change values in the `$EM_CONF` array if needed.

Key	Data type	Description
title	string, required	The name of the extension in English.
description	string, required	Short and precise description in English of what the extension does and for whom it might be useful.
category	string	Which category the extension belongs to: • **be** Backend (Generally backend oriented, but not a module) • **module** Backend modules (When something is a module or connects with one) • **fe** Frontend (Generally frontend oriented, but not a "true" plugin) • **plugin** Frontend plugins (Plugins inserted as a "Insert Plugin" content element) • **misc** Miscellaneous stuff (Where not easily placed elsewhere) • **services** Contains TYPO3 services • **templates** Contains website templates • **example** Example extension (Which serves as examples etc.) • **doc** Documentation (e.g. tutorials, FAQ's etc.)
shy	boolean	If set, the extension will normally be hidden in the EM because it might be a default extension or otherwise something which is not so important. Use this flag if an extension is of "rare interest" (which is not the same as unimportant - just an extension not sought for very often...) It does not affect whether or not it's enabled. Only display in EM. Normally "shy" is set for all extensions loaded by default according to TYPO3_CONF_VARS.
dependencies	list of extension-keys	This is a list of other extension keys which this extension depends on being loaded *before* itself. The EM will manage that dependency while writing the extension list to localconf.php. **Deprecated**, use "constraints" instead.
conflicts	list of extension-keys	List of extension keys of extensions with which this extension does *not* work (and so cannot be enabled before those other extensions are un-installed) **Deprecated**, use "constraints" instead.
constraints	array	List of requirements, suggestions or conflicts with other extensions or TYPO3 or PHP version. Here's how a typical setup might look:

```
'constraints' => array(
      'dependencies' => array(
            'typo3' => '0.0.0-4.2.0',
            'php' => '5.0.0-0.0.0'
      ),
      'conflicts' => array(
            'dam' => ''
      ),
      'suggests' => array(
            'tt_news' => '2.5.0-0.0.0'
      )
)
```

"dependencies" lists extensions that this extension depends on. "conflicts" lists extensions which will not work with this extension. "suggests" is just suggestions of extensions that work together or enhance this extension.

In the example above, it is indicated that the extension depends on a version of TYPO3 lower than 4.2 and a PHP version of at least 5.0. It will conflict with the DAM (any version) and it is suggested that it might be worth installing "tt_news" (version at least 2.5.0).

Key	Data type	Description
priority	"top", "bottom"	This tells the EM to try to put the extensions as the very first in the list.
doNotLoadInFE	boolean	If set, the extension will not be included in the list of extensions to be loaded in the frontend (`$TYPO3_CONF_VARS['extListFE']`). New in TYPO3 4.3. **Background:** In TYPO3 versions **before 4.3** the `temp_CACHED*` files in folder `typo3conf/` did always contain the content of the `ext_tables.php` and `ext_localconf.php` files of all installed extensions. However not all installed extensions are needed to render frontend output (e.g. wizard_sortpages, tstemplate_analyzer and others) and so their `$EM_CONF` array has the flag `doNotLoadInFE` set. This will prevent TYPO3 from adding the extension's `ext_localconf.php` and `ext_tables.php` to the `temp_CACHED` files when rendering frontend content. Since 'extListFE' is shorter than the list of all extensions this will result in 2 new `temp_CACHED_FE*` files which are smaller than the files containing all extensions settings. This can save some precious milliseconds when delivering content.
loadOrder		(Not used)
module	list of strings	If any subfolders to an extension contains backend modules, those folder names should be listed here. It allows the EM to know about the existence of the module, which is important because the EM has to update the conf.php file of the module in order to set the correct TYPO3_MOD_PATH constant. **Note:** this is not needed anymore if you use the dispatch mechanism for BE modules (see "Inside TYPO3", chapter "Backend modules using typo3/mod.php").
state	string	Which state is the extension in? • **alpha** Alpha state is used for very initial work, basically the state is has during the very process of creating its foundation. • **beta** Under current development. Beta extensions are functional but not complete in functionality. Most likely beta-extensions will not be reviewed. • **stable** Stable extensions are complete, mature and ready for production environment. You will be approached for a review. Authors of stable extensions carry a responsibility to be maintain and improve them. • **experimental** Experimental state is useful for anything experimental - of course. Nobody knows if this is going anywhere yet... Maybe still just an idea. • **test** Test extension, demonstrates concepts etc. • **obsolete** The extension is obsolete or deprecated. This can be due to other extensions solving the same problem but in a better way or if the extension is not being maintained anymore. • **excludeFromUpdates** This state makes it impossible to update the extension through the extension manager (neither by the Update mechanism, nor by uploading a newer version to the installation). This is very useful if you made local changes to an extension for a specific installation and don't want any admin to overwrite them. *New in TYPO3 4.3.*
internal	boolean	This flag indicates that the core source code is specifically aware of the extension. In other words this flag should convey the message that "this extension could not be written independently of core source code modifications". An extension is not internal just because it uses TYPO3 general classes e.g. those from t3lib/. True non-internal extensions are characterized by the fact that they could be written without making core source code changes, but rely only on existing classes in TYPO3 and/or other extensions, plus its own scripts in the extension folder.
uploadfolder	boolean	If set, then the folder named "uploads/tx_[extKey-with-no-underscore]" should be present!
createDirs	list of strings	Comma list of directories to create upon extension installation.
modify_tables	list of tables	List of table names which are only modified - not fully created - by this extension. Tables from this list found in the ext_tables.sql file of the extension.
lockType	char; L, G or S	Locks the extension to be installed in a specific position of the three posible: • **L** = local (typo3conf/ext/) • **G** = global (typo3/ext/) • **S** = system (typo3/sysext/)

Key	Data type	Description
clearCacheOnLoad	boolean	If set, the EM will request the cache to be cleared when this extension is loaded.
author	string	Author name (Use a-z)
author_email	email address	Author email address
author_company	string	Author company (if any company sponsors the extension).
CGLcompliance	keyword	Compliance level that the extension claims to adhere to. A compliance defines certain coding guidelines, level of documentation, technical requirements (like XHTML, DBAL usage etc). Possible values are: • CGL360 Please see the Project Coding Guidelines for a description of each compliance keyword (and the full allowed list). **Deprecated**
CGLcompliance_note	string	Any remarks to the compliance status. Might describe some minor incompatibilities or other reservations. **Deprecated**
private	boolean	If set, *this version* of the extension is not included in the public list! (Not supported anymore)
download_password	string	If set, this password must additionally be specified if people want to access (import or see details for) this the extension. (Not supported anymore)
version	main.sub.dev	Version of the extension. Automatically managed by EM / TER. Format is [int].[int].[int]

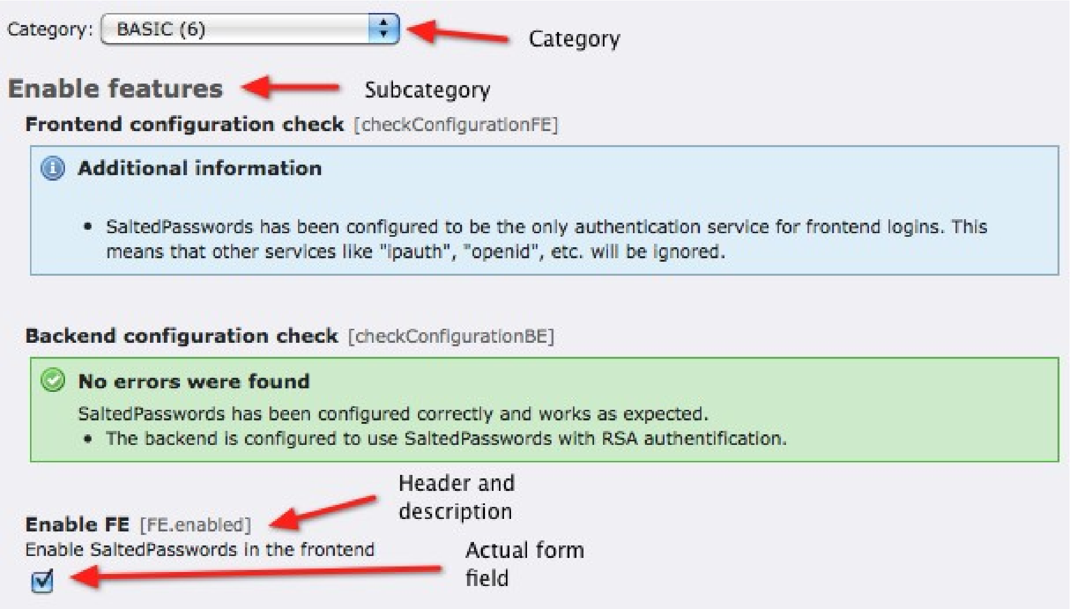
Extension configuration options (ext_conf_template.txt)

In this file configuration options for an extension can be defined. They will be accessible from the TYPO3 BE in the Extension Manager (EM), in the Information view of the extension.

There's a specific syntax to declare these options properly, which is similar to the one used for TypoScript constants (see "Declaring constants for the Constant editor" in "TypoScript Syntax and In-depth Study"). This syntax applies to the comment line that should be placed just before the constant. Consider the following example (taken from system extension "saltedpasswords"):

```
# cat=basic/enable; type=boolean; label=Enable FE: Enable SaltedPasswords in the frontend
FE.enabled = 1
```

First a category (cat) is defined ("basic") with the subcategory "enable". Then a type is given ("boolean") and finally a label, which is itself split (on the colon ":") into a title and a description. The above example will be rendered like this in the EM:



Category: BASIC (6)

Enable features

Frontend configuration check [checkConfigurationFE]

Additional information

- SaltedPasswords has been configured to be the only authentication service for frontend logins. This means that other services like "ipauth", "openid", etc. will be ignored.

Backend configuration check [checkConfigurationBE]

No errors were found

SaltedPasswords has been configured correctly and works as expected.

- The backend is configured to use SaltedPasswords with RSA authentification.

Enable FE [FE.enabled]

Enable SaltedPasswords in the frontend

☑

The options screen displays all options from a single category. A selector is available to switch between categories. Inside an option screen, options are grouped by subcategory. At the bottom of the screenshot, the label – split between header and description – is visible. Then comes the field itself, in this case a checkbox, because the option's type is “boolean”.

Extending “extensions-classes”

A rather exotic thing to do but nevertheless...

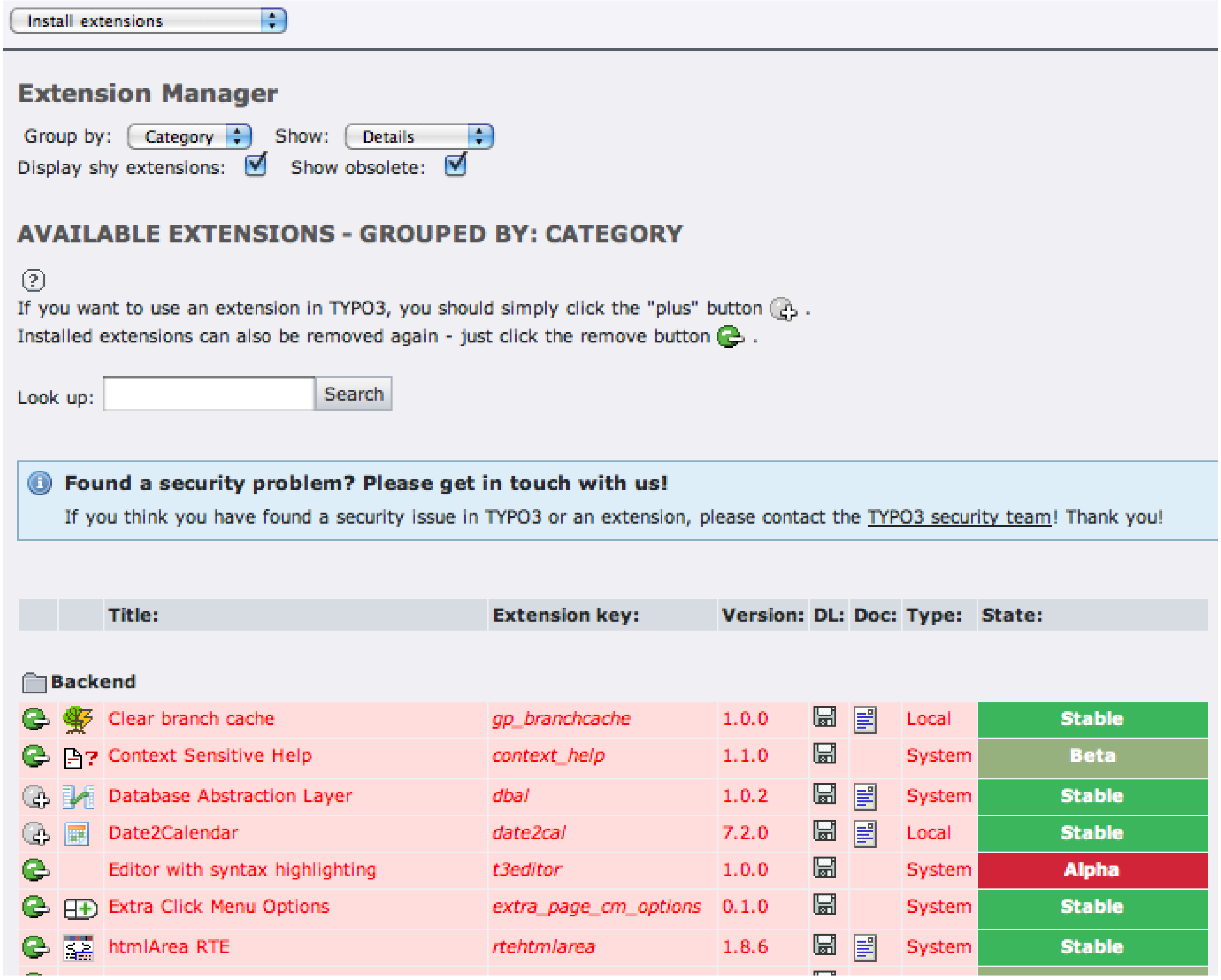
If you are programming extensions yourself you should as a standard procedure include the “class extension code” in the bottom of the class file:

```
if (defined("TYPO3_MODE") && $TYPO3_CONF_VARS[TYPO3_MODE]["XCLASS"]["ext/class.cool_shop.php"])
{
       include_once($TYPO3_CONF_VARS[TYPO3_MODE]["XCLASS"]["ext/class.cool_shop.php"]);
}
```

Normally the key used as example here (“ext/class.cool_shop.php”) would be the full path to the script relative to the PATH_site constant. However because modules are required to work from both typo3/sysext/, typo3/ext/ *and* typo3conf/ext/ it is policy that any path before “ext/” is omitted.

The Extension Manager (EM)

Extensions are managed from the Extension Manager inside TYPO3 by "admin" users. The module is located at "Admin tools > Ext Manager" and offers a menu with options to see loaded extensions (those that are installed or activated), available extensions on the server and the possibility to import extensions from online resources, typically the TER (TYPO3 Extension Repository) located at typo3.org.



Extension Manager

Group by: Category Show: Details

Display shy extensions: ☑ Show obsolete: ☑

AVAILABLE EXTENSIONS - GROUPED BY: CATEGORY

If you want to use an extension in TYPO3, you should simply click the "plus" button .
Installed extensions can also be removed again - just click the remove button .

Look up:

Found a security problem? Please get in touch with us!
If you think you have found a security issue in TYPO3 or an extension, please contact the TYPO3 security team! Thank you!

Title:	Extension key:	Version:	DL:	Doc:	Type:	State:
Backend						
Clear branch cache	*gp_branchcache*	1.0.0	💾	📄	Local	Stable
Context Sensitive Help	*context_help*	1.1.0	💾		System	Beta
Database Abstraction Layer	*dbal*	1.0.2	💾	📄	System	Stable
Date2Calendar	*date2cal*	7.2.0	💾	📄	Local	Stable
Editor with syntax highlighting	*t3editor*	1.0.0	💾		System	Alpha
Extra Click Menu Options	*extra_page_cm_options*	0.1.0	💾		System	Stable
htmlArea RTE	*rtehtmlarea*	1.8.6	💾	📄	System	Stable

The interface looks like this for the list of available extensions:

The interface is really easy to use. You just click the +/- icon to the left of an extension in order to install it and follow the instructions.

Basic framework for a new extension

This document will not describe into details how to create extensions. It only aims to be a reference for the facts regarding the rules of how extensions register with the system.

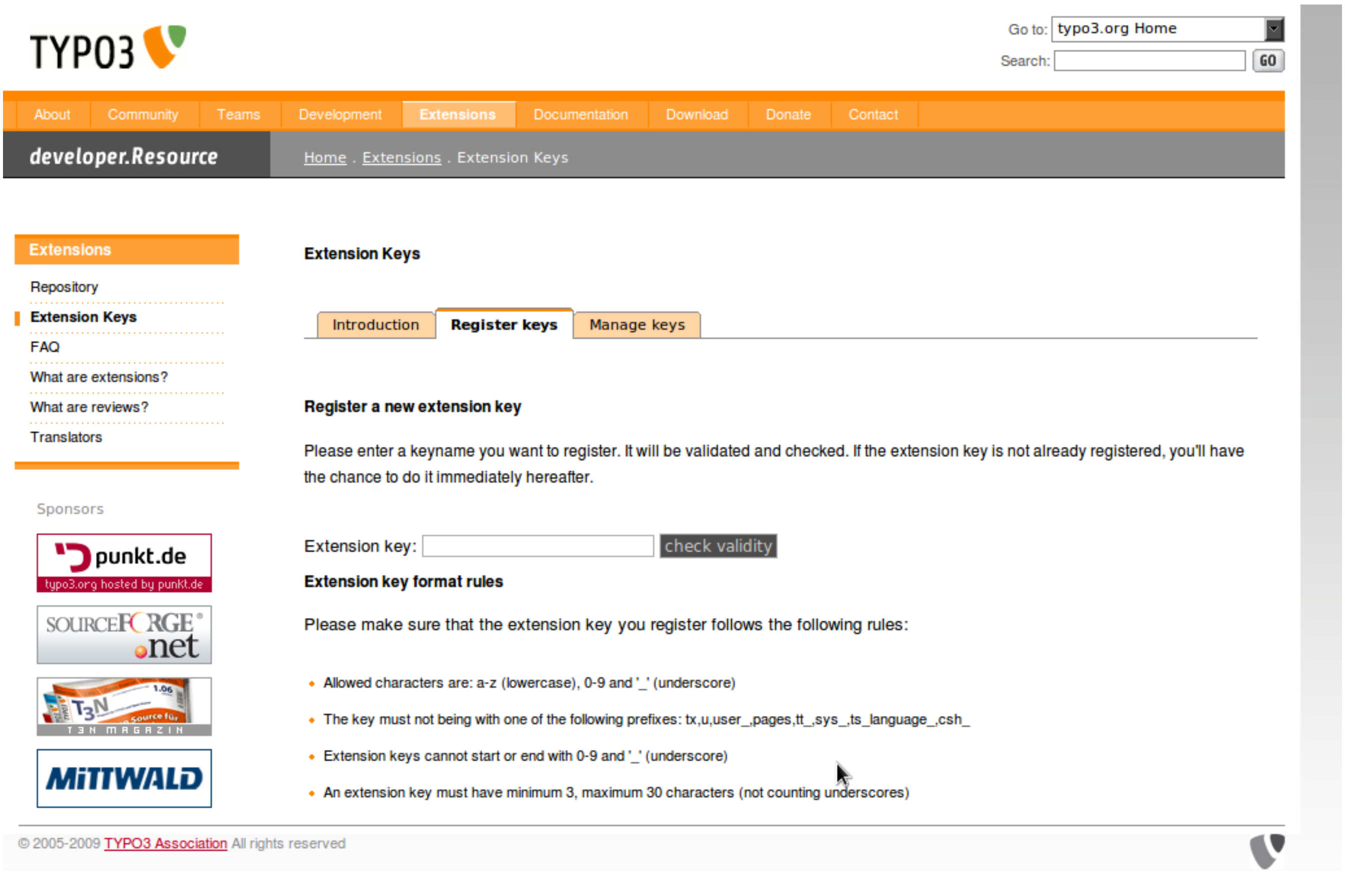
To learn to create extensions you should read one of the extension tutorials that are available. They will take you through the process step by step and explain best-practices for you.

To start up a new extension the most popular tool is the Extension Kickstarter. From a series of menus it allows you to configure a basic set of features you want to get into your extension and a selection of default files will be created. The idea is that you continue to develop these files into your specific application.

Registering an extension key

Before starting a new extension you should register an extension key on typo3.org (unless you plan to make an implementation-specific extension – of course – which it does not make sense to share).

Go to typo3.org, log in with your (pre-created) username / password and go to Extensions > Extension Keys and click on the “Register keys” tab. On that page you can enter the key name you want to register.



Extension Keys

Register a new extension key

Please enter a keyname you want to register. It will be validated and checked. If the extension key is not already registered, you'll have the chance to do it immediately hereafter.

Extension key:

Extension key format rules

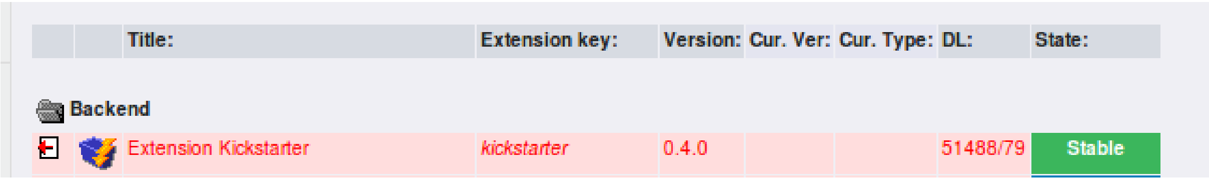
Please make sure that the extension key you register follows the following rules:

- Allowed characters are: a-z (lowercase), 0-9 and '_' (underscore)
- The key must not being with one of the following prefixes: tx,u,user_,pages,tt_,sys_,ts_language_,csh_
- Extension keys cannot start or end with 0-9 and '_' (underscore)
- An extension key must have minimum 3, maximum 30 characters (not counting underscores)

© 2005-2009 TYPO3 Association All rights reserved

Enabling the Extension Kickstarter

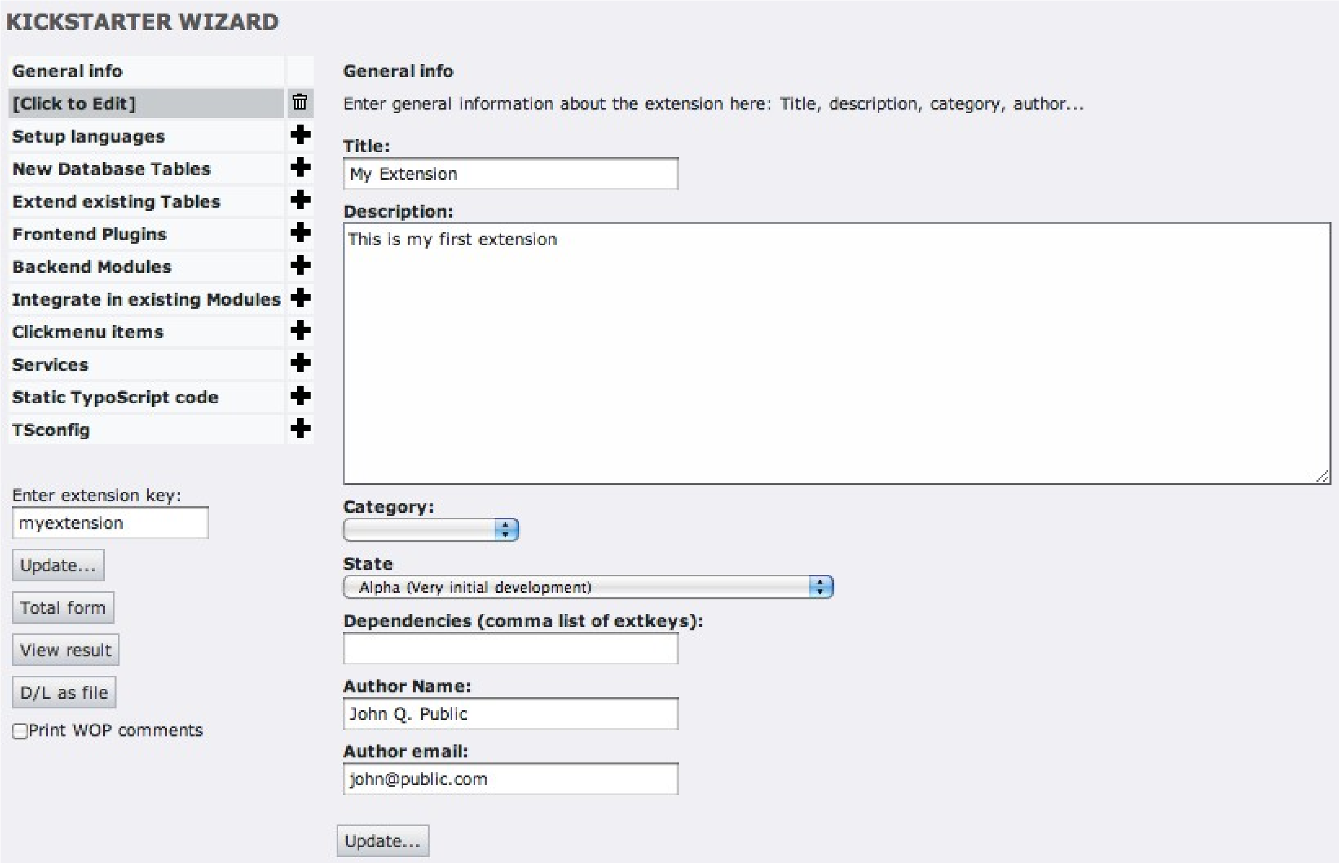
Before you can use the Extension Kickstarter you will have to enable it. The "Kickstarter" is an extension like any other (key: "kickstarter") so it must be installed first:



		Title:	Extension key:	Version:	Cur. Ver:	Cur. Type:	DL:	State:
		Backend						
		Extension Kickstarter	kickstarter	0.4.0			51488/79	Stable

After the installation of the extension you will find a new menu item named "Create new extension" in the selector box menu of the Extension Manager.

Using the Extension Kickstarter



Enter extension key: myextension

General info

Enter general information about the extension here: Title, description, category, author...

Title: My Extension

Description: This is my first extension

Category:

State Alpha (Very initial development)

Dependencies (comma list of extkeys):

Author Name: John Q. Public

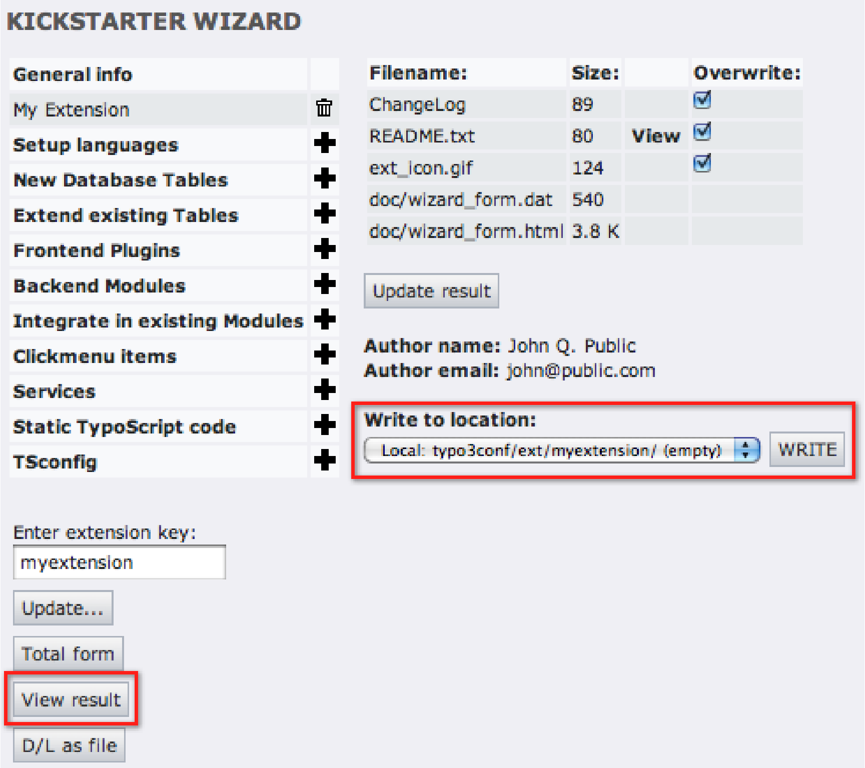
Author email: john@public.com

In the Kickstarter you should always fill in the General Information which includes the title, description, author name etc for the extension. But the most important thing is to enter the extension key as the very first thing!

After entering this information you can begin to create new tables and fields in the database, you can configure backend modules and frontend plugins etc. Basically this is what tutorials will cover in detail.

When you are through with the configuration you click the button to the left called "View result". This will let you preview the content of the files the Kickstarter will write to the server.

It is important that you write the extension to the correct location. Most likely that will be "Local" in your case.



Filename:	Size:		Overwrite:
ChangeLog	89		☑
README.txt	80	View	☑
ext_icon.gif	124		☑
doc/wizard_form.dat	540		
doc/wizard_form.html	3.8 K		

Author name: John Q. Public
Author email: john@public.com

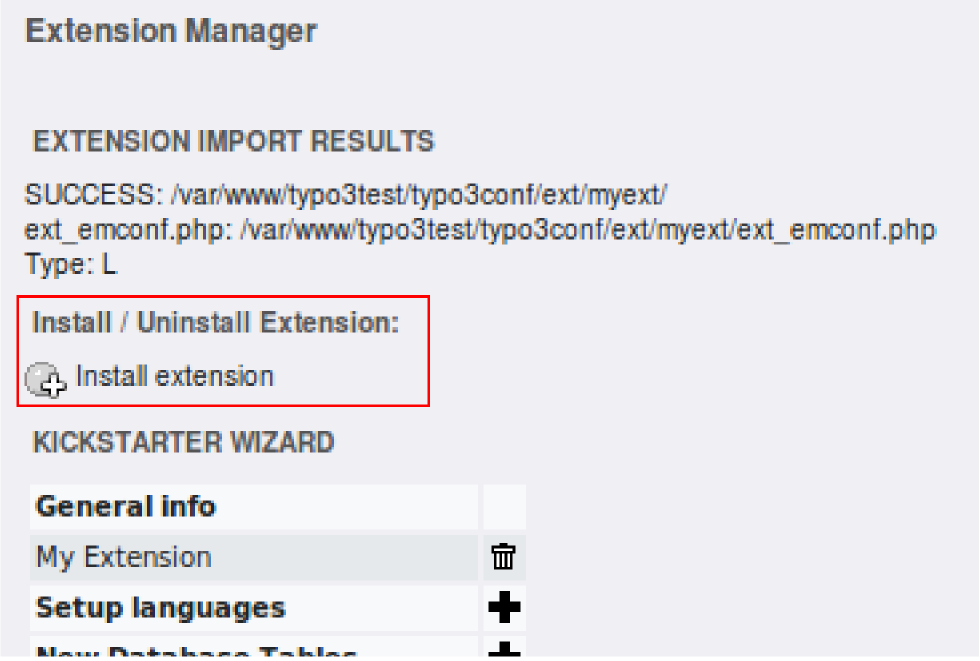
Write to location: Local: typo3conf/ext/myextension/ (empty)

Enter extension key: myextension

Finally, if there already is an extension with the same extension key *every file from that extension will be overwritten with the Kickstarter's output!* Remember: This is *a kickstarter, not an editor!* It is meant to kick you off with the development of your new TYPO3 extension and nothing more! So of course it overwrites all existing files!

Enabling your newly created extension

After the extension is written to the server's disk you should see a message like this, which allows you to install your extension immediately:



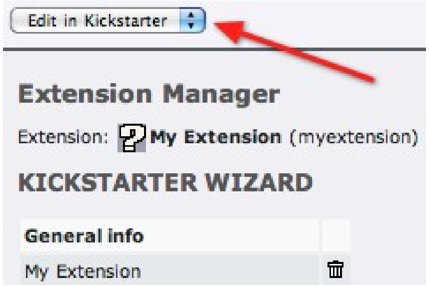
SUCCESS: /var/www/typo3test/typo3conf/ext/myext/
ext_emconf.php: /var/www/typo3test/typo3conf/ext/myext/ext_emconf.php
Type: L

Re-edit the extension

In the process of creating an extension it is rather typical to go back to the Kickstarter a few times to fine tune the base code. Experience suggests that this is especially useful to tuning the configuration of database tables and fields.

The Kickstarter generates 2 files while it is working: `doc/wizard_form.dat` and `doc/wizard_form.html`. As long as these two files are present, the extension can be edited again. Obviously you should remove those files once you are done with the Kickstarter.

If you want to load the Kickstarter with the original configuration used for your extension so you can add or edit features, just click the extension title in the list of loaded/available extensions and select "Edit in Kickstarter" from the menu:



You should be back to the Kickstarter with all the original configuration used (provided the files mentioned above still exist).

Warning about re-editing

This feature is potentially dangerous as it may give the impression that the Kickstarter is an editor. So once more: **theKickstarter is not an editor for your extensions**! Whatever custom changes have been made to the scripts of your new extension will be overwritten when you write back the extension from the Kickstarter.

A good workflow for using the Kickstarter would be like this:

- Start by setting up all the features you need for your extension and write it with the Kickstarter.
- Begin to fill in dummy information in the database tables your extension contains. You will most likely find that you forgot something or misconfigured a database field. Since you have not yet done any changes to the files in the extension you can safely re-load the extension configuration (see above) and add the missing fields or whatever. Your dummy database content will not be affected by this operation.
- When you have tested that your database configuration works for your purpose you can begin to edit the PHP-scripts by hand (i.e. programming the extension itself). This is the "point-of-no-return" where you cannot safely return to the Kickstarter because you have now changed scripts manually.

TYPO3 API overview

Introduction

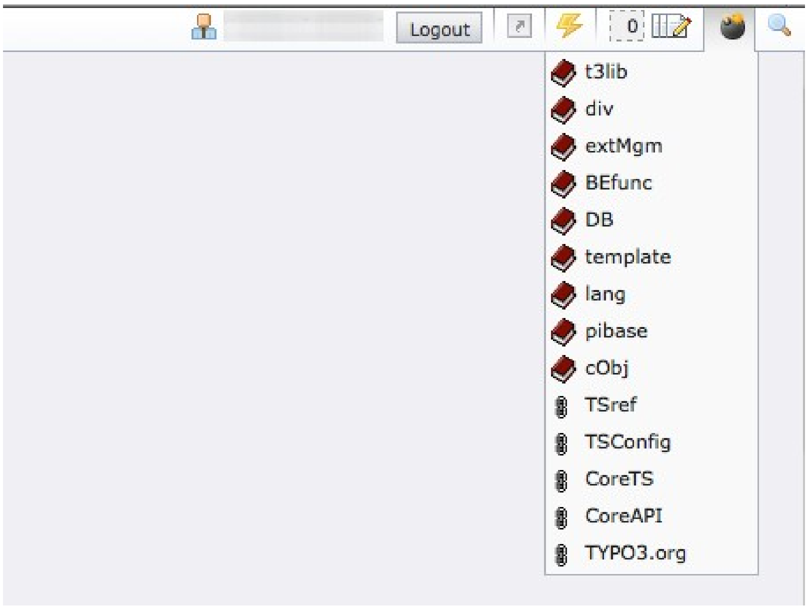
The source is the documentation!

(General wisdom)

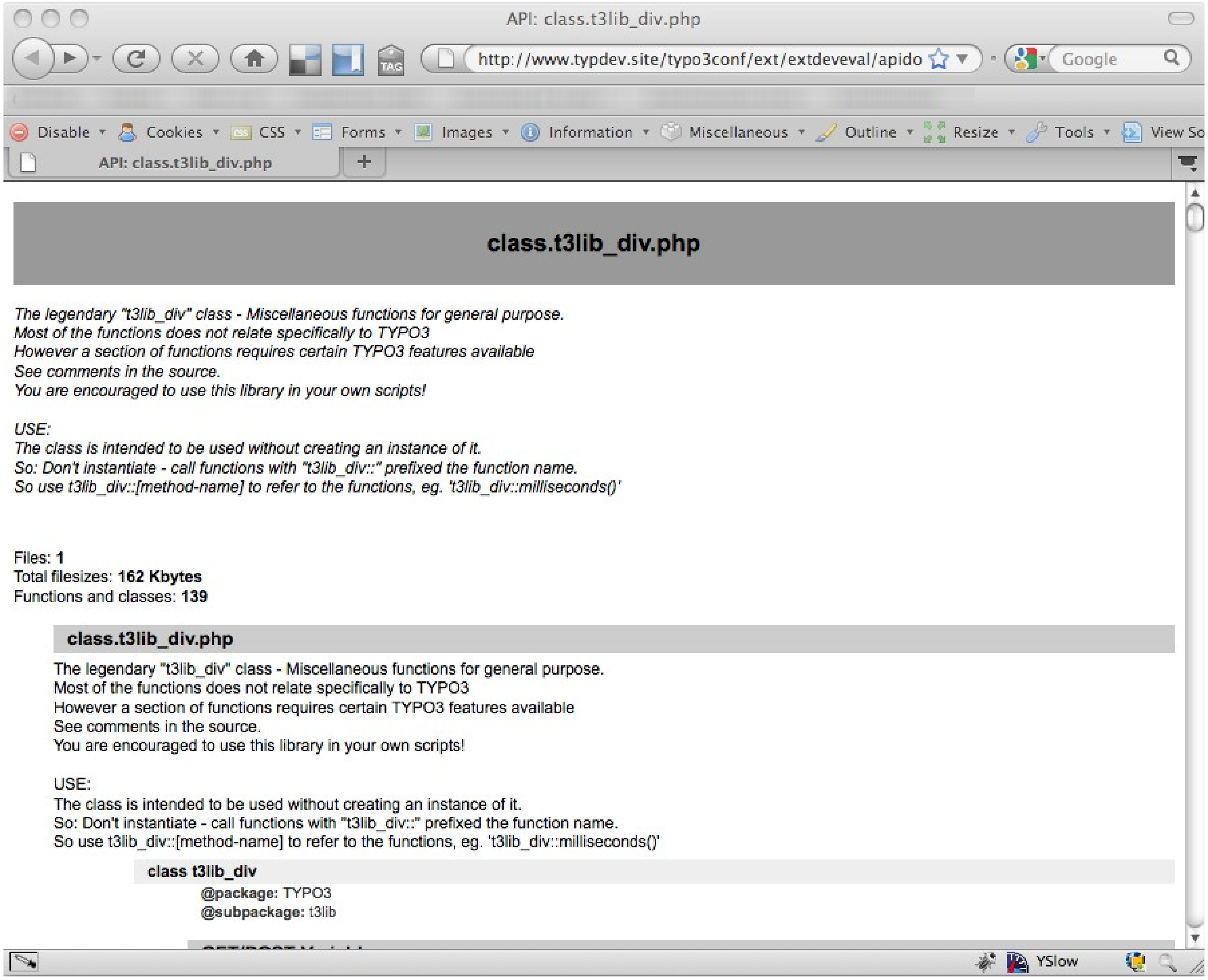
The TYPO3 APIs are first and foremost documented inside of the source scripts. It would be impossible to maintain documentation at more than one location given the fact that things change and sometimes fast. This chapter describes the most important elements of the API. Some other elements have their own chapter further on.

Inline documentation

We have dedicated ourselves to document the classes and methods inside the source scripts (JavaDoc style). This means that you can use any phpDoc compliant documentor program to extract API documentation from the source. You can also install the extension "extdeveval" which provides a menu in the top bar with links to the most important APIs in TYPO3 from within the TYPO3 backend:



Clicking a link like "div" will bring up a new window with the full API of that class:



class.t3lib_div.php

The legendary "t3lib_div" class - Miscellaneous functions for general purpose.
Most of the functions does not relate specifically to TYPO3
However a section of functions requires certain TYPO3 features available
See comments in the source.
You are encouraged to use this library in your own scripts!

USE:
The class is intended to be used without creating an instance of it.
So: Don't instantiate - call functions with "t3lib_div::" prefixed the function name.
So use t3lib_div::[method-name] to refer to the functions, eg. 't3lib_div::milliseconds()'

Files: **1**
Total filesizes: **162 Kbytes**
Functions and classes: **139**

class.t3lib_div.php

The legendary "t3lib_div" class - Miscellaneous functions for general purpose.
Most of the functions does not relate specifically to TYPO3
However a section of functions requires certain TYPO3 features available
See comments in the source.
You are encouraged to use this library in your own scripts!

USE:
The class is intended to be used without creating an instance of it.
So: Don't instantiate - call functions with "t3lib_div::" prefixed the function name.
So use t3lib_div::[method-name] to refer to the functions, eg. 't3lib_div::milliseconds()'

class t3lib_div

@package: TYPO3
@subpackage: t3lib

Pointers in the right direction

The point of this documentation is to help you understand which parts of the API are particularly important or useful for your TYPO3 hacking. The next pages will highlight functions and classes which you should make yourself familiar with.

General functions

There are a few core classes in TYPO3 which contain general functionality. These classes are (typically) just a collection of individual functions you call non-instantiated, like [class name]::[method name].

These are the most important classes to know about in TYPO3:

Class name:	Description:	Usage:
t3lib_DB	**Database Abstraction Base API** All access to the database must go through this object. That is the first step towards DBAL compliance in your code. The class contains MySQL wrapper functions which can almost be search/replaced with your existing calls.	$GLOBALS['TYPO3_DB'] in both frontend and backend
t3lib_cs	**Character Set handling API** Contains native PHP code to handle character set conversion based on charset tables from Unicode.org. It is not certain that you will have to use this class directly but if you need to do charset conversion at any time you should use this class.	In backend, $GLOBALS['LANG']->csConvObj In frontend, $GLOBALS['TSFE']->csConvObj
t3lib_div	**General Purpose Functions** A collection of multi-purpose PHP functions. Some are TYPO3 specific but not all.	t3lib_div:: (Non-instantiated!)

Class name:	Description:	Usage:
t3lib_BEfunc	**Backend Specific Functions** Contains functions specific for the backend of TYPO3. You will typically need these when programming backend modules or other backend functionality. *This class is NOT available in the frontend!*	t3lib_BEfunc:: (Non-instantiated!)
t3lib_extMgm	**Extension API functions** Functions for extensions to interface with the core system. Many of these functions are used in ext_localconf.php and ext_tables.php files of extensions. They let extensions register their features with the system. *See extension programming tutorials for more details.*	t3lib_extMgm:: (Non-instantiated!)
t3lib_iconWorks	**Icons / Part of skinning API** Contains a few functions for getting the right icon for a database table record or the skinned version of any other icon. *This class is NOT available in the frontend!*	t3lib_iconWorks:: (Non-instantiated!)
template	**Backend Template Class** Contains functions for producing the layout of backend modules, setting up HTML headers, wrapping JavaScript sections correctly for XHTML etc.	$GLOBALS['TBE_TEMPLATE'] or $GLOBALS['SOBE'] or $this->doc (inside of Script Classes)

These classes are always included and available in the TYPO3 backend and frontend (except "t3lib_BEfunc" and "t3lib_iconWorks").

The following pages will list methods from these classes in priority of importance. You should at least acquaint yourself with all High-priority functions since these are a part of the Coding Guidelines requirements. In addition you might like to know about other functions which are very often used since they might be very helpful to you (they were to others!).

High priority functions (CGL requirements)

The functions listed in this table is of high priority. Generally they provide APIs to functionality which TYPO3 should always handle in the same way. This will help you to code TYPO3 applications with less bugs and greater compatibility with various system conditions it will run under.

Remember, this list only serves to point out important functions! The real documentation is found in the source scripts. The comments given is only a supplement to that.

Function	Comments
t3lib_div::_GP t3lib_div::_GET t3lib_div::_POST	**Getting values from GET or POST vars** APIs for getting values in GET or POST variables with slashes stripped regardless of PHP environment. Always use these functions instead of direct access to $_GET or $_POST. t3lib_div::_GP($varname) will give you the value of either the GET or POST variable with priority to POST if present. This is useful if you don't know whether a parameter is passed as GET or POST. Many scripts will use this function to read variables in the init function:

```
    // Setting GPvars:
$this->file = t3lib_div::_GP('file');
$this->size = t3lib_div::_GP('size');
```

t3lib_div::_GET() will give you GET vars. For security reasons you should use this if you know your parameters are passed as GET variables. This example gives you the whole $_GET array:

```
$params = t3lib_div::_GET();
```

t3lib_div::POST() will give you POST variables. Works like t3lib_div::_GET(). For security reasons you should use this if you know your parameters are passed as POST variables.
This example gives you the content of the POST variable TSFE_ADMIN_PANEL, for instance it could come from a form field like "<input name="TSFE_ADMIN_PANEL[command]" ..../>

```
$input = t3lib_div::_POST('TSFE_ADMIN_PANEL');
```

Function	Comments
t3lib_div::makeInstance	**Creating objects** Factory API for creating an object instance of a class name. This function makes sure the "XCLASS extension" principle can be used on (almost) any class in TYPO3. You **must** use this function when creating objects in TYPO3. Examples:

```
// Making an instance of class "t3lib_TSparser":
$parseObj = t3lib_div::makeInstance('t3lib_TSparser');

// Make an object with an argument passed to the constructor (TYPO3 4.3+):
$xmlObj = t3lib_div::makeInstance('t3lib_xml', 'typo3_export');
```

NOTE: t3lib_div::makeInstanceClassName() has been deprecated in TYPO3 4.3 and should not be used anymore.

t3lib_div::getIndpEnv	**Environment-safe server and environment variables.** API function for delivery of system and environment variables on any web-server brand and server OS. Always use this API instead of $_ENV/$_SERVER or getenv() if possible. Examples:

```
if (t3lib_div::getIndpEnv('HTTP_ACCEPT_LANGUAGE') == $test)...
if (t3lib_div::cmpIP(t3lib_div::getIndpEnv('REMOTE_ADDR'), $pcs[1]))...
$prefix = t3lib_div::getIndpEnv('TYPO3_REQUEST_URL');
$redirectTo = t3lib_div::getIndpEnv('TYPO3_SITE_URL').$redirectTo;
if (!t3lib_div::getIndpEnv('TYPO3_SSL')) ...
```

t3lib_div::loadTCA	**Loading full table description into $TCA** If you want to access or change any part of the $TCA array for a table except the ['ctrl'] part then you should call this function first. The $TCA might not contain the full configuration for the table (depending on configuration of the table) and to make sure it is loaded if it isn't already you call this function. Examples of PHP code which traverses the ['columns'] part of an unknown table and loads the table before.

```
t3lib_div::loadTCA($this->table);
foreach ($TCA[$this->table]['columns'] as $fN) {
    $fieldListArr[] = $fN;
}
```

t3lib_BEfunc::deleteClause	**Get SQL WHERE-clause filtering "deleted" records** Tables from $TCA might be configured to set an integer flag when deleted instead of being physically deleted from the database. In any case records with the deleted-flag set *must never* be selected in TYPO3. To make sure you never make that mistake always call this function which will pass you a SQL WHERE-clause like " AND deleted=0" if the table given as argument has been configured with a deleted-field. (Notice: In the frontend this is build into the "enableFields()" function.) Example:

```
$res = $GLOBALS['TYPO3_DB']->exec_SELECTquery(
                'pid,uid,title,TSconfig,is_siteroot,storage_pid',
                'pages',
                'uid='.intval($uid).' '.
                    t3lib_BEfunc::deleteClause('pages').' '.
                    $clause
            );
```

Function	Comments
t3lib_extMgm::isLoaded	**Returns true if an extension is loaded (installed)** If you need to check if an extension is loaded in a TYPO3 installation simply use this function to ask for that. Example:

```
  // If the extension "sys_note" is loaded, then...
if (t3lib_extMgm::isLoaded('sys_note'))     ...
  // If the "cms" extension is NOT loaded, return false
if (!t3lib_extMgm::isLoaded('cms'))     return;
  // Check if the "indexed_search" extension is loaded. If not, exit PHP!
t3lib_extMgm::isLoaded('indexed_search', TRUE);
  // Assign value "popup" if extension "tsconfig_help" is loaded
$type = t3lib_extMgm::isLoaded('tsconfig_help') ? 'popup' : '';
```

t3lib_extMgm::extPath t3lib_extMgm::extRelPath t3lib_extMgm::siteRelPath	**Get file path to an extension directory** If you need to get the absolute or relative filepaths to an extension you should use these functions. Extension can be located in three different positions in the filesystem whether they are local, global or system extensions. These functions will always give you the right path. Examples:

```
  // Include a PHP file from the extension "extrep_wizard".
  // t3lib_extMgm::extPath() returns the absolute path to the
  // extension directory.
require_once(
    t3lib_extMgm::extPath('extrep_wizard') .
    'pi/class.tx_extrepwizard.php'
);
  // Get relative path (relative to PATH_typo3) to an icon (backend)
$icon = t3lib_extMgm::extRelPath('tt_rating') . 'rating.gif';
  // Get relative path (relative to PATH_site) to an icon (frontend)
return '<img src="'.
    t3lib_extMgm::siteRelPath('indexed_search') . 'pi/res/locked.gif'
    ... />';
```

Function	Comments
t3lib_div::getFileAbsFileName t3lib_div::validPathStr t3lib_div::isAbsPath t3lib_div::isAllowedAbsPath t3lib_div::fixWindowsFilePath	**Evaluate files and directories for security reasons**

When you allow references to files to be inputted from users there is always the risk that they try to cheat the system to include something else than intended. These functions makes it easy for you to evaluate filenames for validity before reading, writing or including them.

Here the functions are described in order of importance:

t3lib_div::getFileAbsFileName() - Returns the absolute filename of a relative reference, resolves the "EXT:" prefix (way of referring to files inside extensions) and checks that the file is inside the PATH_site of the TYPO3 installation and implies a check with t3lib_div::validPathStr(). Returns false if checks failed. Does not check if the file exists.

```
  // Getting absolute path of a temporary file:
$cacheFile = t3lib_div::getFileAbsFileName('typo3temp/tempfile.tmp');
  // Include file if it exists:
$file = t3lib_div::getFileAbsFileName($fileRef);
if (@is_file($file))    {
    include($file);
}
```

t3lib_div::validPathStr() - Checks for malicious file paths. Returns true if no '//', '..' or '\' is in the $theFile. This should make sure that the path is not pointing 'backwards' and further doesn't contain double/back slashes.

```
    // If the path is true and validates as a valid path string:
if ($path && t3lib_div::validPathStr($path))    ...
```

t3lib_div::isAbsPath() - Checks if the input path is absolute or relative (detecting either '/' or 'x:/' as first part of string) and returns true if so.

```
  // Returns relative filename for icon:
if (t3lib_div::isAbsPath($Ifilename))    {
    $Ifilename = '../' . substr($Ifilename, strlen(PATH_site));
}
```

t3lib_div::isAllowedAbsPath() - Returns true if the path is absolute, without backpath '..' and within the PATH_site OR within the lockRootPath. Contrary to t3lib_div::getFileAbsFileName() this function can also validate files in filemounts outside the web-root of the installation, but this is rarely used!

```
if (@file_exists($path) && t3lib_div::isAllowedAbsPath($path))    {
    $fI = pathinfo($path);
         ....
```

t3lib_div::fixWindowsFilePath() - Fixes a path for Windows-backslashes and reduces double-slashes to single slashes

Function	Comments
t3lib_div::mkdir	**Creates directory**

One would think that creating directories is one thing you can do directly with PHP. Well, it turns out to be quite error-prone if it should be compatible with Windows servers and safe-mode at the same time. So TYPO3 offers a substitution function you should always use.

Example:

```
$root.=$dirParts . '/';
if (!is_dir($extDirPath . $root))    {
    t3lib_div::mkdir($extDirPath . $root);
    if (!@is_dir($extDirPath.$root))    {
        return 'Error: The directory "' .
                $extDirPath.$root.
                '" could not be created...';
    }
}
```

Function	Comments
t3lib_div::upload_to_tempfile t3lib_div::unlink_tempfile t3lib_div::tempnam	**Functions for handling uploads and temporary files**

You need to use these functions for managing uploaded files you want to access as well as creating temporary files within the same session. These functions are safe_mode and open_basedir compatible which is the main point of you using them!

t3lib_div::upload_to_tempfile() - Will move an uploaded file (normally in "/tmp/xxxxx") to a temporary filename in PATH_site."typo3temp/" from where TYPO3 can use it under safe_mode. Remember to use t3lib_div::unlink_tempfile() afterwards - otherwise temp-files will build up! They are *not* automatically deleted in PATH_site."typo3temp/"!

t3lib_div::unlink_tempfile() - Deletes (unlink) a temporary filename in 'PATH_site."typo3temp/"' given as input. The function will check that the file exists, is in PATH_site."typo3temp/" and does not contain back-spaces ("../") so it should be pretty safe. Use this after upload_to_tempfile() or tempnam() from this class!

This example shows how to handle an uploaded file you just want to read and then delete again:

```
    // Read uploaded file:
$uploadedTempFile = t3lib_div::upload_to_tempfile(
    $GLOBALS['HTTP_POST_FILES']['upload_ext_file']['tmp_name']
);
$fileContent = t3lib_div::getUrl($uploadedTempFile);
t3lib_div::unlink_tempfile($uploadedTempFile);
```

t3lib_div::tempnam() - Create temporary filename (creates file with unique file name). This function should be used for getting temporary filenames - will make your applications safe for "open_basedir = on". Remember to delete the temporary files after use! This is done by t3lib_div::unlink_tempfile()
In the following example it is shown how two temporary filenames are created for being processed with an external program (diff) after which they are deleted again:

```
    // Create file 1 and write string
$file1 = t3lib_div::tempnam('diff1_');
t3lib_div::writeFile($file1, $str1);
    // Create file 2 and write string
$file2 = t3lib_div::tempnam('diff2_');
t3lib_div::writeFile($file2, $str2);
    // Perform diff.
$cmd = $GLOBALS['TYPO3_CONF_VARS']['BE']['diff_path'].
           ' '.$this->diffOptions . ' ' . $file1 . ' ' . $file2;
exec($cmd, $res);
unlink($file1);
unlink($file2);
```

Function	Comments
t3lib_div::fixed_lgd_cs	**Truncating a string for visual display, observing the character set (backend only)**

This function allows you to truncate a string from e.g. "Hello World" to "Hello Wo..." so for example very long titles of records etc. will not break the visual appearance of your backend modules.
Since text strings cannot be cropped at any byte if the character set is utf-8 or another multibyte charset this function will process the string assuming the character set to be the one used in the backend.
It is recommended to use $BE_USER->uc['titleLen'] for the length parameter.

```
  // Limits Record title to 30 chars
t3lib_div::fixed_lgd_cs($thisRecTitle, 30);
  // Limits string to title-length configured for backend user:
$title = t3lib_div::fixed_lgd_cs(
             $row['title'],
             $this->BE_USER->uc['titleLen']
);
```

Function	Comments
t3lib_div::formatForTextarea	**Preparing a string for output between <textarea> tags.**

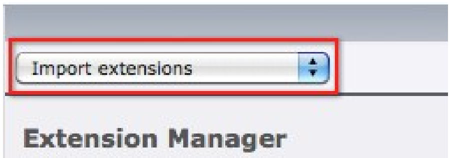
Use this function to prepare content for <textarea> tags. Then you will avoid extra / stripped whitespace when the form is submitted multiple times.

```
    // Create item:
$item = '
    <textarea>' .
    t3lib_div::formatForTextarea($value) .
    '</textarea>';
```

Function	Comments
t3lib_div::locationHeaderUrl	**Preparing a URL for a HTTP location-header** Use this to prepare redirection URLs for location-headers. It will convert the URL to be absolute. This is also useful in other cases where an absolute URL must be used, for example when passing a callback URL to some third-party software. Redirection example:

```
header('Location: ' . t3lib_div::locationHeaderUrl($this->retUrl));
exit;
```

Function	Comments
t3lib_BEfunc::getFuncMenu t3lib_BEfunc::getFuncCheck	**Create "Function menu" in backend modules** Creates a selector box menu or checkbox with states automatically saved in the backend user session. Such a function menu could look like this:



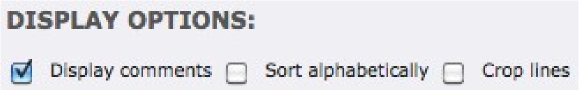
The selector box is made by this function call. It sets the ID variable (zero if not available), the GET var name, "SET[mode]", the current value from MOD_SETTINGS and finally the array of menu options, MOD_MENU['mode']:

```
t3lib_BEfunc::getFuncMenu(
    $this->id,
    'SET[mode]',
    $this->MOD_SETTINGS['mode'],
    $this->MOD_MENU['mode']
)
```

Prior to making the menu it is required that the MOD_MENU array is set up with an array of options. This could look like this (getting some labels from the "locallang" system). In addition the incoming "SET" GET-variable must be registered in the session which is also done in this listing:

```
$this->MOD_MENU = array(
    'mode' => array(
        0 => $LANG->getLL('user_overview'),
        'perms' => $LANG->getLL('permissions')
    )
);
    // Clean up settings:
$this->MOD_SETTINGS = t3lib_BEfunc::getModuleData(
                        $this->MOD_MENU,
                        t3lib_div::_GP('SET'),
                        $this->MCONF['name']
                    );
```

You can have a checkbox as well:



Then the function call looks like this. Notice the fourth argument is gone because a checkbox does not have any information about options like a selector box would have.

```
t3lib_BEfunc::getFuncCheck(
    0,
    'SET[own_member_only]',
    $this->MOD_SETTINGS['own_member_only']
);
```

For checkboxes you must set the key in the MOD_MENU array as well. Otherwise the values are not registered in the user session:

```
'own_member_only' => '',
```

Function	Comments
t3lib_BEfunc::editOnClick	**Create onclick-JavaScript code that links to edit form for a record** Use this function to create a link to the "alt_doc.php" core script which can generate editing forms for any $TCA configured record. The actual editing command is passed to "alt_doc.php" through the GET parameter "&edit". See the section with "Various examples" for detailed examples of this! Example:

```php
$params = '&edit[pages][' . $row['uid'] . ']=edit';
$link = '<a href="#" onclick="' .
            htmlspecialchars(t3lib_BEfunc::editOnClick($params, '', -1)) .
            '">Edit</a>';
```

Function	Comments
t3lib_BEfunc::viewOnClick	**Create onclick-JavaScript code that opens a page in the frontend** It will detect the correct domain name if needed and provide the link with the right back path. Also it will re-use any window already open.

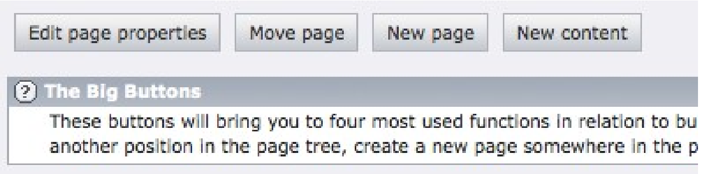
```php
    // "View page" link is added:
$link = '<a href="#" onclick="' .
        htmlspecialchars(t3lib_BEfunc::viewOnClick(
            $pageId,
            $GLOBALS['BACK_PATH'],
            t3lib_BEfunc::BEgetRootLine($pageId)
        )) . '">View page</a>';
```

Function	Comments
$GLOBALS['TBE_TEMPLATE']->issueCommand	**Creates a link to "tce_db.php" (with a command like copy, move,delete for records)** Creates a URL to the TYPO3 Core Engine interface provided from the core script, "tce_db.php". The $params array is filled with date or cmd values. For detailed list of options see section about TCE elsewhere in this document. Example:

```php
    // Delete
$params = '&cmd[tt_content][' . $row['uid'] . '][delete]=1';
$out .= '<a href="' .
    htmlspecialchars($GLOBALS['SOBE']->doc->issueCommand($params)) .
    '" onclick="' .
    htmlspecialchars("return confirm('Want to delete?');").
    '">Delete record</a>';
```

Function	Comments
t3lib_BEfunc::helpTextIcon t3lib_BEfunc::helpText t3lib_BEfunc::cshItem	**Create icon or short description for Context Sensitive Help (CSH)**

You are encouraged to integrate Content Sensitive help in your backend modules and for your database tables. This will help users to use TYPO3 and your TYPO3 applications more easily.
With these functions you can create content sensitive help texts (and links to more details) like this:



The Big Buttons

These buttons will bring you to four most used functions in relation to bu
another position in the page tree, create a new page somewhere in the p

(Note: For the short description to be displayed and not only the icon the user must set up "Field help mode" in the User>Setup module to "Display full text message".)

Examples:

```
  // Setting "table name" to module name with prefix
$tableIdent = '_MOD_' . $this->MCONF['name'];

  // Creating CSH icon and short description:
$HTMLcode .=
    t3lib_BEfunc::helpTextIcon($tableIdent, 'quickEdit', $BACK_PATH).
    t3lib_BEfunc::helpText($tableIdent, 'quickEdit', $BACK_PATH).
    '<br />';
```

Prior to calling helpTextIcon and helpText you might need to load the description table with:

```
if ($BE_USER->uc['edit_showFieldHelp'])     {
    $LANG->loadSingleTableDescription($tableIdent);
}
```

Alternatively you can use t3lib_BEfunc::cshItem(). It's a quicker way of integrating the descriptions since description files are loaded automatically and the text/icon mode is integrated in a single function call. This is recommended for sporadic usage:

```
$HTMLcode .=
t3lib_BEfunc::cshItem($tableIdent,'quickEdit', $BACK_PATH);
```

Function	Comments
t3lib_iconWorks::getIconImage t3lib_iconWorks::getIcon	**Getting correct icon for database table record**

Always use these functions if you need to get the icon for a record. Works only for records from tables which are defined in $TCA

```
  // Getting default icon for the "tt_content" table:
t3lib_iconWorks::getIconImage('tt_content', array(), $this->backPath, '');

  // Getting an icon where record content may define the look:
$icon = t3lib_iconWorks::getIconImage(
            $this->table,
            $row,
            $this->backPath,
            'align="top" class="c-recIcon"'
        );

  // Getting the icon filename only:
$ficon = t3lib_iconWorks::getIcon($table, $row);
```

Function	Comments
t3lib_iconWorks::skinImg	**Processing icons for skin API**

Pass the filename and width/height attributes of all images you use in your backend applications through this function. See Skin API description for more details.

```
$skin_enabled_icon = '<img' .
    t3lib_iconWorks::skinImg(
        $this->doc->backPath,
        'gfx/recordlock_warning3.gif',
        'width="17" height="12"'
    ) .
    ' alt="" />';
```

Function	Comments
$GLOBALS['TYPO3_DB']-> exec_INSERTquery exec_UPDATEquery exec_DELETEquery exec_SELECTquery	**Database Access API** To be compatible with Database Abstraction Layers you should always use the global object $TYPO3_DB for database access. The class "t3lib_db" contains a list of MySQL wrapper functions (sql(), sql_fetch_assoc(), etc...) which you can use almost out of the box as a start. Just search/replace. But it is recommended that you port your application to using the four execution functions directly. These will both build the query for you and execute it. See the Coding Guidelines, t3lib_db API and Inside TYPO3 document for more information.

Inserting a record:
Just fill an array with "fieldname => value" pairs and pass it to exec_INSERTquery() along with the table name in which it should be inserted:

```
$insertFields = array(
    'md5hash' => $md5,
    'tstamp' => time(),
    'type' => 2,
    'params' => $inUrl
);
$GLOBALS['TYPO3_DB']->exec_INSERTquery(
    'cache_md5params',
    $insertFields
);
```

Updating a record:
Create an array of "fieldname => value" pairs before calling exec_UPDATEquery(). The function call is almost like inserting, but you need to add a WHERE clause to target the update to the record you want to update. It is the second argument you set to a value like "uid=???".

```
$fields_values = array(
    'title' => $data['sys_todos'][$key]['title'],
    'deadline' => $data['sys_todos'][$key]['deadline'],
    'description' => $data['sys_todos'][$key]['description'],
    'tstamp' => time()
);
$GLOBALS['TYPO3_DB']->exec_UPDATEquery(
    'sys_todos',
    'uid=' . intval($key),
    $fields_values
);
```

Deleting a record:
Call exec_DELETEquery() with the tablename *and* the WHERE clause selecting the record to delete:

```
$GLOBALS['TYPO3_DB']->exec_DELETEquery(
    'sys_todos',
    'uid=' . intval($key)
);
```

Selecting a record:
Call exec_SELECTquery() with at least the first three arguments (field list to select, table name and WHERE clause). The return value is a result pointer (or object) which should be passed to ->sql_fetch_assoc() in a loop in order to traverse the result rows.

```
$res = $GLOBALS['TYPO3_DB']->exec_SELECTquery(
    '*',
    $theTable,
    $theField . '="' .
        $GLOBALS['TYPO3_DB']->quoteStr($theValue, $theTable) . '"' .
        $this->deleteClause($theTable) . ' ' .
        $whereClause,
    $groupBy,
    $orderBy,
    $limit
);
$rows = array();
while(($row = $GLOBALS['TYPO3_DB']->sql_fetch_assoc($res))) {
    $rows[] = $row;
}
$GLOBALS['TYPO3_DB']->sql_free_result($res);
if (count($rows))    return $rows;
```

Function	Comments
$GLOBALS['BE_USER']->isAdmin	**Return true if current backend user is "admin"** Use this if you need to restrict a user from doing something unless he is "admin".
$GLOBALS['BE_USER']->getPagePermsClause	**Return WHERE clause for filtering pages which permission mismatch for current user** The most typical usage of this is to call the function with the value "1". Then the WHERE clause returned will filter away all pages to which the user has no read-access.

TYPO3 Coding Guidelines

You should also refer to the TYPO3 Core Coding Guidelines (CGL) document which is the authoritative source to know about which coding practices are required for TYPO3 core and extension programming. It also contains some more information about recommended API usage.

Functions typically used and nice to know

These functions are generally just nice to know. They provide functionality which you will often need in TYPO3 applications and therefore they will save you time and make your applications easier for others to understand as well since you use commonly known functions.

Please take time to learn these functions!

Function	Comments
t3lib_div::inList	Check if an item exists in a comma-separated list of items. `if (t3lib_div::inList('gif,jpg,png', $ext)) {`
t3lib_div::intInRange	Forces the input variable (integer) into the boundaries of $min and $max. `t3lib_div::intInRange($row['priority'], 1, 5);`
t3lib_div::isFirstPartOfStr	Returns true if the first part of input string matches the second argument. `t3lib_div::isFirstPartOfStr($path, PATH_site);`
t3lib_div::testInt	Tests if the input is an integer.

t3lib_div::shortMD5
t3lib_div::md5int
	Creates partial/truncated MD5 hashes. Useful when a 32 byte hash is too long or you rather work with an integer than a string.
	t3lib_div::shortMD5() - Creates a 10 byte short MD5 hash of input string
	`$addQueryParams.= '&myHash=' . t3lib_div::shortMD5(serialize($myArguments));`
	t3lib_div::md5int() - Creates an integer from the first 7 hex chars of the MD5 hash string
	`'mpvar_hash' => t3lib_div::md5int($GLOBALS['TSFE']->MP),`

t3lib_div::deHSCentities t3lib_div::htmlspecialchars_decode	Reverse conversions of htmlspecialchars() **t3lib_div::deHSCentities()** - Re-converts HTML entities if they have been converted by htmlspecialchars(). For instance "&amp;amp;" which should stay "&amp;". Or "&amp;#1234;" to "&#1234;". Or "&amp;#x1b;" to "&#x1b;" `$value = t3lib_div::deHSCentities(htmlspecialchars($value));` **t3lib_div::htmlspecialchars_decode()** - Inverse version of htmlspecialchars()

t3lib_div::modifyHTMLColor
t3lib_div::modifyHTMLColorAll
	Modifies the RGB values of an 6-digit HTML hex color by adding/subtracting. Useful for increasing or decreasing brightness of colors.
	`t3lib_div::modifyHTMLColor('#cca823', +10, +10, +10)`
	`t3lib_div::modifyHTMLColorAll($this->doc->bgColor, -20);`

t3lib_div::formatSize	Formats a number of bytes as Kb/Mb/Gb for visual output. `$size = ' (' . t3lib_div::formatSize(filesize($v)) . 'bytes)';`
t3lib_div::validEmail	Evaluates a string as an email address. `if ($email && t3lib_div::validEmail($email)) {`

Function	Comments

t3lib_div::trimExplode
t3lib_div::intExplode
t3lib_div::revExplode
	Various flavors of exploding a string by a token.
	t3lib_div::trimExplode() - Explodes a string by a token and trims the whitespace away around each item. Optionally any zero-length elements are removed. Very often used to explode strings from configuration, user input etc. where whitespace can be expected between values but is insignificant.
	`array_unique(t3lib_div::trimExplode(',', $rawExtList, 1));`
	`t3lib_div::trimExplode(chr(10), $content);`
	t3lib_div::intExplode() - Explodes a by a token and converts each item to an integer value. Very useful to force integer values out of a value list, for instance for an SQL query.
	`// Make integer list`
	`implode(t3lib_div::intExplode(',', $row['subgroup']), ',');`
	t3lib_div::revExplode() - Reverse explode() which allows you to explode a string into X parts but from the back of the string instead.
	`$p = t3lib_div::revExplode('/', $path, 2);`

t3lib_div::array_merge_recursive_overrule t3lib_div::array_merge	Merging arrays with fixes for "PHP-bugs" **t3lib_div::array_merge_recursive_overrule()** - Merges two arrays recursively and "binary safe" (integer keys are overridden as well), overruling similar the values in the first array ($arr0) with the values of the second array ($arr1). In case of identical keys, i.e. keeping the values of the second. **t3lib_div::array_merge()** - An array_merge function where the keys are NOT renumbered as they happen to be with the real php-array_merge function. It is "binary safe" in the sense that integer keys are overridden as well.
t3lib_div::array2xml_cs t3lib_div::xml2array	Serialization of PHP variables into XML. These functions are made to serialize and unserialize PHParrays to XML files. They are used for the FlexForms content in TYPO3, Data Structure definitions etc. The XML output is optimized for readability since associative keys are used as tagnames. This also means that only alphanumeric characters are allowed in the tag names and only keys *not* starting with numbers (so watch your usage of keys!). However there are options you can set to avoid this problem. Numeric keys are stored with the default tagname "numIndex" but can be overridden to other formats). The function handles input values from the PHP array in a binary-safe way; All characters below 32 (except 9,10,13) will trigger the content to be converted to a base64-string. The PHP variable type of the data is preserved as long as the types are strings, arrays, integers and booleans. Strings are the default type unless the "type" attribute is set. **t3lib_div::array2xml_cs()** - Converts a PHP array into an XML string. `t3lib_div::array2xml_cs($this->FORMCFG['c'],'T3FormWizard');` **t3lib_div::xml2array()** - Converts an XML string to a PHP array. This is the reverse function of array2xml()

```
if ($this->xmlStorage)    {
    $cfgArr = t3lib_div::xml2array($row[$this->P['field']]);
}
```

Function	Comments
	NOTE: t3lib_div::array2xml() is deprecated since TYPO3 4.3. Use t3lib_div::array2xml_cs() which takes care of proper character set conversion.

t3lib_div::getURL
t3lib_div::writeFile
	Reading / Writing files
	t3lib_div::getURL() - Reads the full content of a file or URL. Used throughout the TYPO3 sources.
	`$templateCode = t3lib_div::getURL($templateFile);`
	t3lib_div::writeFile() - Writes a string into an absolute filename.
	`t3lib_div::writeFile($extDirPath.$theFile,$fileData['content']);`

t3lib_div::split_fileref	Splits a reference to a file in 5 parts. Alternative to "path_info" and fixes some "PHP-bugs" which makes page_info() unattractive at times.

Function	Comments
t3lib_div::get_dirs t3lib_div::getFilesInDir t3lib_div::getAllFilesAndFoldersInPath t3lib_div::removePrefixPathFromList	Read content of file system directories.

t3lib_div::get_dirs() - Returns an array with the names of folders in a specific path

```
if (@is_dir($path))    {
    $directories = t3lib_div::get_dirs($path);
    if (is_array($directories))    {
        foreach($directories as $dirName)    {
            ...
        }
    }
}
```

t3lib_div::getFilesInDir() - Returns an array with the names of files in a specific path

```
$sFiles = t3lib_div::getFilesInDir(PATH_typo3conf ,'', 1, 1);
$files = t3lib_div::getFilesInDir($dir, 'png,jpg,gif');
```

t3lib_div::getAllFilesAndFoldersInPath() - Recursively gather all files and folders of a path.
t3lib_div::removePrefixPathFromList() - Removes the absolute part of all files/folders in fileArr (useful for post processing of content from t3lib_div::getAllFilesAndFoldersInPath())

```
    // Get all files with absolute paths prefixed:
$fileList_abs =
    t3lib_div::getAllFilesAndFoldersInPath(array(), $absPath, 'php,inc');

    // Traverse files and remove abs path from each (becomes relative)
$fileList_rel =
    t3lib_div::removePrefixPathFromList($fileList_abs, $absPath);
```

t3lib_div::implodeArrayForUrl	Implodes a multidimensional array into GET-parameters (e.g. &param[key][key2]=value2&param[key][key3]=value3) `$pString = t3lib_div::implodeArrayForUrl('', $params);`

t3lib_div::get_tag_attributes
t3lib_div::implodeAttributes
	Works on HTML tag attributes
	t3lib_div::get_tag_attributes() - Returns an array with all attributes of the input HTML tag as key/value pairs. Attributes are only lowercase a-z
	`$attribs = t3lib_div::get_tag_attributes('<' . $subparts[0] . '>');`
	t3lib_div::implodeAttributes() - Implodes attributes in the array $arr for an attribute list in e.g. and HTML tag (with quotes)
	`$tag = '<img ' . t3lib_div::implodeAttributes($attribs, 1) . ' />';`

t3lib_div::resolveBackPath	Resolves "../" sections in the input path string. For example "fileadmin/directory/../other_directory/" will be resolved to "fileadmin/other_directory/"

Function	Comments
t3lib_div::callUserFunction t3lib_div::getUserObj	General purpose functions for calling user functions (creating hooks). See the chapter about Hooks in this document for detailed description of these functions. **t3lib_div::callUserFunction()** - Calls a user-defined function/method in class. Such a function/method should look like this: "function proc(&$params, &$ref) {...}"

```
function procItems($items,$iArray,$config,$table,$row,$field) {
    global $TCA;
    $params=array();
    $params['items'] = &$items;
    $params['config'] = $config;
    $params['TSconfig'] = $iArray;
    $params['table'] = $table;
    $params['row'] = $row;
    $params['field'] = $field;

    t3lib_div::callUserFunction(
        $config['itemsProcFunc'],
        $params,
        $this
    );
    return $items;
}
```

t3lib_div::getUserObj() - Creates and returns reference to a user defined object.

```
$_procObj = &t3lib_div::getUserObj($_classRef);
$_procObj->pObj = &$this;
$value = $_procObj->transform_rte($value,$this);
```

Function	Comments
t3lib_div::linkThisScript	Returns the URL to the current script. You can pass an array with associative keys corresponding to the GET-vars you wish to add to the URL. If you set them empty, they will remove existing GET-vars from the current URL.
t3lib_div::plainMailEncoded t3lib_div::quoted_printable	Mail sending functions **t3lib_div::plainMailEncoded()** - Simple substitute for the PHP function mail() which allows you to specify encoding and character set. **t3lib_div::quoted_printable()** - Implementation of quoted-printable encode.
t3lib_BEfunc::getRecord t3lib_BEfunc::getRecordsByField	Functions for selecting records by uid or field value. **t3lib_BEfunc::getRecord()** - Gets record with uid=$uid from $table

```
  // Getting array with title field from a page:
t3lib_BEfunc::getRecord('pages', intval($row['shortcut']), 'title');

  // Getting a full record with permission WHERE clause
$pageinfo = t3lib_BEfunc::getRecord(
        'pages',
        $id,
        '*',
        ($perms_clause ? ' AND ' . $perms_clause : '')
    );
```

t3lib_BEfunc::getRecordsByField() - Returns records from table, $theTable, where a field ($theField) equals the value, $theValue

```
    // Checking if the id-parameter is an alias.
if (!t3lib_div::testInt($id))    {
    list($idPartR) =
        t3lib_BEfunc::getRecordsByField('pages', 'alias', $id);
    $id = intval($idPartR['uid']);
}
```

Function	Comments
t3lib_BEfunc::getRecordPath	Returns the path (visually) of a page $uid, fx. "/First page/Second page/Another subpage"

```
$label = t3lib_BEfunc::getRecordPath(
        intval($row['shortcut']),
        $perms_clause,
        20
    );
```

Function	Comments

t3lib_BEfunc::readPageAccess	Returns a page record (of page with $id) with an extra field "_thePath" set to the record path *if* the WHERE clause, $perms_clause, selects the record. Thus is works as an access check that returns a page record if access was granted, otherwise not.
	`$perms_clause = $GLOBALS['BE_USER']->getPagePermsClause(1);`
	`$pageinfo = t3lib_BEfunc::readPageAccess($id, $perms_clause);`

t3lib_BEfunc::date
t3lib_BEfunc::datetime
t3lib_BEfunc::calcAge
	Date/Time formatting functions using date/time format from TYPO3_CONF_VARS.
	t3lib_BEfunc::date() - Returns $tstamp formatted as "ddmmyy" (According to $TYPO3_CONF_VARS['SYS']['ddmmyy'])
	`t3lib_BEfunc::datetime($row['crdate'])`
	t3lib_BEfunc::datetime() - Returns $tstamp formatted as "ddmmyy hhmm" (According to $TYPO3_CONF_VARS['SYS']['ddmmyy'] AND $TYPO3_CONF_VARS['SYS']['hhmm'])
	`t3lib_BEfunc::datetime($row['item_mtime'])`
	t3lib_BEfunc::calcAge() - Returns the "age" in minutes / hours / days / years of the number of $seconds inputted.
	`$agePrefixes = ' min| hrs| days| yrs';`
	`t3lib_BEfunc::calcAge(time()-$row['crdate'], $agePrefixes);`

t3lib_BEfunc::titleAttribForPages	Returns title-attribute information for a page-record informing about id, alias, doktype, hidden, starttime, endtime, fe_group etc.
	`$out = t3lib_BEfunc::titleAttribForPages($row, '', 0);`
	`$out = t3lib_BEfunc::titleAttribForPages($row, '1=1 ' . $this->clause, 0);`

t3lib_BEfunc::thumbCode t3lib_BEfunc::getThumbNail	Returns image tags for thumbnails **t3lib_BEfunc::thumbCode()** - Returns a linked image-tag for thumbnail(s)/fileicons/truetype-font-previews from a database row with a list of image files in a field. Slightly advanced. It's more likely you will need t3lib_BEfunc::getThumbNail() to do the job. **t3lib_BEfunc::getThumbNail()** - Returns single image tag to thumbnail using a thumbnail script (like thumbs.php)
t3lib_BEfunc::storeHash t3lib_BEfunc::getHash	Get/Set cache values. **t3lib_BEfunc::storeHash()** - Stores the string value $data in the 'cache_hash' table with the hash key, $hash, and visual/symbolic identification, $ident **t3lib_BEfunc::getHash()** - Retrieves the string content stored with hash key, $hash, in cache_hash Example of how both functions are used together; first getHash() to fetch any possible content and if nothing was found how the content is generated and stored in the cache:

t3lib_BEfunc::getThumbNail

```
t3lib_BEfunc::getThumbNail(
    $this->doc->backPath . 'thumbs.php',
    $filepath,
    'hspace="5" vspace="5" border="1"'
);
```

t3lib_BEfunc::storeHash

```
    // Parsing the user TS (or getting from cache)
$userTS = implode($TSdataArray,chr(10) . '[GLOBAL]' . chr(10));
$hash = md5('pageTS:' . $userTS);
$cachedContent = t3lib_BEfunc::getHash($hash, 0);
$TSconfig = array();
if (isset($cachedContent))     {
    $TSconfig = unserialize($cachedContent);
} else {
    $parseObj = t3lib_div::makeInstance('t3lib_TSparser');
    $parseObj->parse($userTS);
    $TSconfig = $parseObj->setup;
    t3lib_BEfunc::storeHash($hash,serialize($TSconfig), 'IDENT');
}
```

Function	Comments
t3lib_BEfunc::getRecordTitle t3lib_BEfunc::getProcessedValue	Get processed / output prepared value from record

t3lib_BEfunc::getRecordTitle() - Returns the "title" value from the input records field content.

```
$line.= t3lib_BEfunc::getRecordTitle('tt_content', $row, 1);
```

t3lib_BEfunc::getProcessedValue() - Returns a human readable output of a value from a record. For instance a database record relation would be looked up to display the title-value of that record. A checkbox with a "1" value would be "Yes", etc.

```
$outputValue = nl2br(
    htmlspecialchars(
        trim(
            t3lib_div::fixed_lgd_cs(
                t3lib_BEfunc::getProcessedValue(
                    $table,
                    $fieldName,
                    $row[$fieldName]
                ),
                250
            )
        )
    )
);
```

Function	Comments
t3lib_BEfunc::getFileIcon	Returns file icon name (from $FILEICONS) for the file extension $ext

```
$fI = pathinfo($filePath);
$fileIcon = t3lib_BEfunc::getFileIcon(strtolower($fI['extension']));
$fileIcon = '<img' .
    t3lib_iconWorks::skinImg(
        $this->backPath,
        'gfx/fileicons/' . $fileIcon,
        'width="18" height="16"'
    ) . ' alt="" />';
```

Function	Comments
t3lib_BEfunc::getPagesTSconfig	Returns the Page TSconfig for page with id, $id. This example shows how an object path, "mod.web_list" is extracted from the Page TSconfig for page $id:

```
$modTSconfig = $GLOBALS['BE_USER']->getTSConfig(
    'mod.web_list',
    t3lib_BEfunc::getPagesTSconfig($id)
);
```

Function	Comments
t3lib_extMgm::addTCAcolumns	Adding fields to an existing table definition in $TCA For usage in "ext_tables.php" files

```
    // tt_address modified
t3lib_div::loadTCA('tt_address');
t3lib_extMgm::addTCAcolumns('tt_address', array(
         'module_sys_dmail_category' =>
            array('config' => array('type' => 'passthrough')),
        'module_sys_dmail_html' =>
            array('config' => array('type' => 'passthrough'))
));
```

Function	Comments
t3lib_extMgm::addToAllTCAtypes	Makes fields visible in the TCEforms, adding them to the end of (all) "types"-configurations For usage in "ext_tables.php" files

```
t3lib_extMgm::addToAllTCAtypes(
    'fe_users',
    'tx_myext_newfield;;;;1-1-1, tx_myext_another_field'
);
```

Function	Comments
t3lib_extMgm::allowTableOnStandardPages	Add table name to default list of allowed tables on pages (in $PAGES_TYPES) For usage in "ext_tables.php" files

```
t3lib_extMgm::allowTableOnStandardPages('tt_board');
```

Function	Comments
t3lib_extMgm::addModule	Adds a module (main or sub) to the backend interface For usage in "ext_tables.php" files
t3lib_extMgm::insertModuleFunction	Adds a "Function menu module" ('third level module') to an existing function menu for some other backend module For usage in "ext_tables.php" files
t3lib_extMgm::addPlugin	Adds an entry to the list of plugins in content elements of type "Insert plugin" For usage in "ext_tables.php" files
t3lib_extMgm::addPItoST43	Add PlugIn to Static Template #43 When adding a frontend plugin you will have to add both an entry to the TCA definition of tt_content table AND to the TypoScript template which must initiate the rendering. Since the static template with uid 43 is the "content.default" and practically always used for rendering the content elements it's very useful to have this function automatically adding the necessary TypoScript for calling your plugin. It will also work for the extension "css_styled_content" For usage in "ext_localconf.php" files

t3lib_extMgm::addModule

```
t3lib_extMgm::addModule(
    'user',
    'setup',
    'after:task',
    t3lib_extMgm::extPath($_EXTKEY) . 'mod/'
);

t3lib_extMgm::addModule(
    'tools',
    'txcoreunittestM1',
    '',
    t3lib_extMgm::extPath($_EXTKEY) . 'mod1/'
);
```

t3lib_extMgm::insertModuleFunction

```
t3lib_extMgm::insertModuleFunction(
    'web_func',
    'tx_cmsplaintextimport_webfunc',
    t3lib_extMgm::extPath($_EXTKEY) .
        'class.tx_cmsplaintextimport_webfunc.php',
    'LLL:EXT:cms_plaintext_import/locallang.php:menu_1'
);
```

t3lib_extMgm::addPlugin

```
t3lib_extMgm::addPlugin(
    array(
        'LLL:EXT:newloginbox/locallang_db.php:tt_content.list_type1',
        $_EXTKEY . '_pi1'
    ),
    'list_type'
);
```

t3lib_extMgm::addPItoST43

```
t3lib_extMgm::addPItoST43($_EXTKEY);
```

Programming with workspaces in mind

The concept of workspaces needs attention from extension programmers. The implementation of workspaces is however made so that no critical problems can appear with old extensions;

- First of all the "Live workspace" is no different from how TYPO3 has been working for years so that will be supported out of the box (except placeholder records must be filtered out in the frontend with "t3ver_state!=" , see below).
- Secondly, all permission related issues are implemented in TCEmain so the worst your users can experience is an error message.

However, you probably want to update your extension so that in the backend the current workspace is reflected in the records shown and the preview of content in the frontend works as well. Therefore this chapter has been written with instructions and insight into the issues you are facing.

Frontend challenges in general

For the frontend the challenges are mostly related to creating correct previews of content in workspaces. For most extensions this will work transparently as long as they use the API functions in TYPO3 to request records from the system.

The most basic form of a preview is when a live record is selected and you lookup a future version of that record belonging to the current workspace of the logged in backend user. This is very easy as long as a record is selected based on its "uid" or "pid" fields which are not subject to versioning; You simply call "sys_page->versionOL()" after record selection.

However, when other fields are involved in the where clause it gets dirty. This happens all the time! For instance, all records displayed in the frontend must be selected with respect to "enableFields" configuration! What if the future version is hidden and the live version is not? Since the live version is selected first (not hidden) and then overlaid with the content of the future version (hidden) the effect of the hidden field we wanted to preview is lost unless we also check the overlaid record for its hidden field (->versionOL() actually does this). But what about the opposite; if the live record was hidden and the future version not? Since the live version is never selected the future version will never have a chance to display itself! So we must first select the live records with no regard to the hidden state, then overlay the future version and eventually check if it is hidden and if so exclude it. The same problem applies to all other "enableFields", future versions with "delete" flags and current versions which are invisible placeholders for future records. Anyway, all that is handled by TYPO3s t3lib_page class which includes functions for "enableFields" and "deleted" so it will work out of the box for you. But as soon as you do selection based on other fields like email, username, alias etc. it will fail.

Summary:

Challenge: How to preview elements which are disabled by "enableFields" in the live version but not necessarily in the offline version. Also, how to filter out new live records with "t3ver_state" set to 1 (placeholder for new elements) but only when not previewed.

Solution: Disable check for "enableFields"/"where_del_hidden" on live records and check for them in versionOL on input record.

Frontend implementation guidelines

- Any place where enableFields() are not used for selecting in the frontend you must at least check that "t3ver_state!=1" so placeholders for new records are not displayed.
- Make sure never to select any record with pid = -1! (offline records - related to versioning).
- If you need to detect preview mode for versioning and workspaces you can read these variables:
  - $GLOBALS['TSFE']->sys_page->versioningPreview: If true, you are allowed to display previews of other record versions.
  - $GLOBALS['TSFE']->sys_page->versioningWorkspaceId: Will tell you the id of the workspace of the current backend user. Used for preview of workspaces.
- Use these API functions for support of version previews in the backend:

Function	Description
$GLOBALS['TSFE']->sys_page->versionOL($table,&$row, $unsetMovePointers=FALSE)	Versioning Preview Overlay. Generally ALWAYS used when records are selected based on uid or pid. If records are selected on other fields than uid or pid (e.g. "email = ...") then usage might produce undesired results and that should be evaluated on individual basis. Principle; Record online! => Find offline? **Example:** This is how simple it is to use this record in your frontend plugins when you do queries directly (not using API functions already using them):

```
$res = $GLOBALS['TYPO3_DB']->exec_SELECTquery(...);
while (($row = $GLOBALS['TYPO3_DB']-
>sql_fetch_assoc($res))) {
        $GLOBALS['TSFE']->sys_page->versionOL($table,
$row);

        if (is_array($row)) {


...
```

When the live record is selected, call ->versionOL() and make sure to check if the input row (passed by reference) is still an array.

The third argument, $unsetMovePointers=FALSE, can be set to TRUE when selecting records for display ordered by their position in the page tree. Difficult to explain easily, so only use this option if you don't get a correct preview of records that has been moved in a workspace (only for "element" type versioning)

Function	Description
$GLOBALS['TSFE']->sys_page->fixVersioningPid()	Finding online PID for offline version record. Will look if the "pid" value of the input record is -1 (it is an offline version) and if the table supports versioning; if so, it will translate the -1 PID into the PID of the original record Used whenever you are tracking something back, like making the root line. In fact, it is currently only used by the root line function and chances are that you will not need this function often. Principle; Record offline! => Find online?

Frontend scenarios impossible to preview

These issues are not planned to be supported for preview:

- Lookups and searching for records based on other fields than uid,pid,"enableFields" will never reflect workspace content since overlays happen to online records *after* they are selected.
  - This problem can largely be avoided for *versions of new records* because versions of a "New"-placeholder can mirror certain fields down onto the placeholder record. For the "tt_content" table this is configured as 'shadowColumnsForNewPlaceholders' => 'sys_language_uid,l18n_parent,colPos,header', so that these fields used for column position, language and header title are also updated in the placeholder thus creating a correct preview in the frontend.
  - For *versions of existing records* the problem is in reality reduced a lot because normally you don't change the column or language fields after the record is first created anyways! But in theory the preview can fail.
  - When changing the type of a page (e.g. from "Standard" to "External URL") the preview might fail in cases where a look up is done on the "doktype" field of the live record.
    - Page shortcuts might not work properly in preview.
    - Mount Points might not work properly in preview.
- It is impossible to preview the value of "count(*)" selections since we would have to traverse all records and pass them through ->versionOL() before we would have a reliable result!
- In tslib_fe::getPageShortcut() sys_page->getMenu() is called with an additional WHERE clause which will not respect if those fields are changed for a future version. This could be the case other places where getmenu() is used (but a search shows it is not a big problem). In this case we will for now accept that a wrong shortcut destination can be experienced during previews.

Backend challenges

The main challenge in the backend is to reflect how the system will look when the workspace gets published. To create a transparent experience for backend users we have to overlay almost every selected record with any possible new version it might have. Also when we are tracking records back to the page tree root point we will have to correct pid-values. All issues related to selecting on fields other than pid and uid also relates to the backend as they did for the frontend.

Workspace related API functions for backend modules

Function	Description

t3lib_BEfunc::workspaceOL()	Overlaying record with workspace version if any. Works like ->sys_page->versionOL() does, but for the backend. Input record must have fields only from the table (no pseudo fields) and the record is passed by reference.
	
	Example:
	`$result = $GLOBALS['TYPO3_DB']->exec_SELECTquery('*', 'pages',`
	`'uid=' . intval($id) . $delClause);`
	`$row = $GLOBALS['TYPO3_DB']->sql_fetch_assoc($result);`
	`t3lib_BEfunc::workspaceOL('pages', $row);`

t3lib_BEfunc::getRecordWSOL()	Gets record from table and overlays the record with workspace version if any.
	
	Example:
	`$row = t3lib_BEfunc::getRecordWSOL($table, $uid);`
	
	`// This is the same as:`
	`$row = t3lib_BEfunc::getRecord($table, $uid);`
	`t3lib_BEfunc::workspaceOL($table, $row);`

Function	Description
t3lib_BEfunc::fixVersioningPid()	Translating versioning PID -1 to the pid of the live record. Same as sys_page->fixVersioningPid() but for the backend.
t3lib_BEfunc::isPidInVersionizedBranch()	Will fetch the rootline for the pid, then check if anywhere in the rootline there is a branch point. Returns either "branchpoint" (if branch) or "first" (if page) or false if nothing. Alternatively, it returns the value of "t3ver_stage" for the branchpoint (if any)
t3lib_BEfunc::getWorkspaceVersionOfRecord()	Returns offline workspace version of a record, if found.
t3lib_BEfunc::getLiveVersionOfRecord()	Returns live version of workspace version.
t3lib_BEfunc::versioningPlaceholderClause()	Returns a WHERE-clause which will deselect placeholder records from other workspaces. This should be implemented almost everywhere records are selected based on other fields than uid and where t3lib_BEfunc::deleteClause() is used. **Example:**

```
$res = $GLOBALS['TYPO3_DB']->exec_SELECTquery(
   'count(*)',
   $this->table,
   $this->parentField . '=' . $GLOBALS['TYPO3_DB']-
>fullQuoteStr($uid, $this->table) .
   t3lib_BEfunc::deleteClause($this->table) .
   t3lib_BEfunc::versioningPlaceholderClause($this->table) .
   $this->clause
);
```

Function	Description
$BE_USER->workspaceCannotEditRecord()	Checking if editing of an existing record is allowed in current workspace if that is offline.
$BE_USER->workspaceCannotEditOfflineVersion()	Like $BE_USER->workspaceCannotEditRecord() but also requires version to be offline (draft)
$BE_USER->workspaceCreateNewRecord()	Checks if new records can be created in a certain page (according to workspace restrictions).
$BE_USER->workspacePublishAccess($wsid)	Returns true if user has access to publish in workspace.
$BE_USER->workspaceSwapAccess()	Returns true if user has access to swap versions.
$BE_USER->checkWorkspace()	Checks how the users access is for a specific workspace.
$BE_USER->checkWorkspaceCurrent()	Like ->checkWorkspace() but returns status for the current workspace.
$BE_USER->setWorkspace()	Setting another workspace for backend user.
$BE_USER->setWorkspacePreview()	Setting frontend preview state.

Backend module access

You can restrict access to backend modules by using $MCONF['workspaces'] in the conf.php files. The variable is a list of keywords defining where the module is available:

```
$MCONF['workspaces'] = online,offline,custom
```

You can also restrict function menu items to certain workspaces if you like. This is done by an argument sent to the function t3lib_extMgm::insertModuleFunction(). See that file for more details.

Detecting current workspace

You can always check what the current workspace of the backend user is by reading $GLOBALS['BE_USER']->workspace. If the workspace is a custom workspace you will find its record loaded in $GLOBALS['BE_USER']->workspaceRec.

The values for workspaces are as following:

- workspace 0 = online (live)
- workspace -1 = offline (draft)
- workspace > 0 = custom (projects)
- workspace -99 = none selected at all (ERROR!)

Using TCEmain with workspaces

Since admin-users are also restricted by the workspace it is not possible to save any live records when in a workspace. However for very special occasions you might need to bypass this and to do so, you can set the instance variable t3lib_tcemain::bypassWorkspaceRestrictions to TRUE. An example of this is when users are updating their user profile using the User > Setup module; that actually allows them to save to a live record (their user record) while in a draft workspace.

Moving in workspaces

TYPO3 4.2 and beyond supports moving for "Element" type versions in workspaces. Technically this works by creating a new online placeholder record (like for new elements in a workspace) in the target location with "t3ver_state" = 3 (move-to placeholder) and a field, "t3ver_move_id", holding the uid of the record to move (source record) upon publishing. In addition, a new version of the source record is made and has "t3ver_state" = 4 (move-to pointer). This version is simply necessary in order for the versioning system to have something to publish for the move operation.

So in summary, two records are created for a move operation in a workspace: The placeholder (online, with state=3 and t3ver_move_id set) and a new version (state=4) of the online source record (the one being moved).

When the version of the source is published a look up will be made to see if a placeholder exists for a move operation and if so the record will take over the pid / "sortby" value upon publishing.

Preview of move operations is almost fully functional through the t3lib_page::versionOL() and t3lib_BEfunc::workspaceOL() functions. When the online placeholder is selected it simply looks up the source record, overlays any version on top and displays it. When the source record is selected it should simply be discarded in case shown in context where ordering or position matters (like in menus or column based page content). This is done in the appropriate places.

TYPO3 Core Engine (TCE)

Introduction

Database

The TYPO3 Core Engine is the class that handles *all **data** writing to database tables configured in $TCA*. In addition the class handles **commands** such as copy, move, delete. It will handle undo/history and versioning of records as well and everything will be logged to the sys_log. And it will make sure that write permissions are evaluated correctly for the user trying to write to the database. Generally, any processing specific option in the $TCA array is handled by TCE.

Using TCE for manipulation of the database content in the TCA configured tables guarantees that the data integrity of TYPO3 is respected. This cannot be safely guaranteed if you write to $TCA configured database tables directly. It will also manage the relations to files and other records.

TCE requires a backend login to work. This is due to the fact that permissions are observed (of course) and thus TCE needs a backend user to evaluate against. This means you cannot use TCEmain from the frontend scope. Thus writing to tables (such as a guestbook) will have to be done from the frontend *without* TCEmain.

The features of the $TCA (Table Configuration Array) array are discussed in the end of this document.

Files

TCE also has a part for handling files. The file operations are normally performed in the File > List module where you can manage a directory on the server by copying, moving, deleting and editing files and directories. The file operations are managed by two core classes, t3lib_basicFileFunc and t3lib_extFileFunc.

Database: t3lib_TCEmain basics

When you are using TCE from your backend applications you need to prepare two arrays of information which contain the instructions to TCEmain of what actions to perform. They fall into two categories: Data and Commands.

"Data" is when you want to write information to a database table or create a new record.

"Commands" is when you want to move, copy or delete a record in the system.

The data and commands are created as multidimensional arrays and to understand the API of TCEmain you simply need to understand the hierarchy of these two arrays.

Commands Array ($cmd):

Syntax:

```
$cmd[ tablename ][ uid ][ command ] = value
```

Description of keywords in syntax:

Key	Data type	Description
tablename	string	Name of the database table. Must be configured in $TCA array, otherwise it cannot be processed.

Key	Data type	Description
uid	integer	The UID of the record that is manipulated. This is always an integer.
command	string (command keyword)	The command type you want to execute. **Notice:** Only *one* command can be executed at a time for each record! The first command in the array will be taken. *See table below for command keywords and values*
value	mixed	The value for the command *See table below for command keywords and values*

Command keywords and values:

Command	Data type	Value
copy	integer	The significance of the value depends on whether it is positive or negative: ● Positive value: The value points to a page UID. A copy of the record (and possibly child elements/tree below) will be inserted inside that page as the first element. ● Negative value: The (absolute) value points to another record from the same table as the record being copied. The new record will be inserted on the same page as that record and if $TCA[...]['ctrl']['sortby'] is set, then it will be positioned *after*. ● Zero value: Record is inserted on tree root level
move	integer	Works like "copy" but moves the record instead of making a copy.
delete	"1"	Value should always be "1" This action will delete the record (or mark the record "deleted" if configured in $TCA)
undelete	"1"	Value should always be "1". This action will set the deleted-flag back to 0.
localize	integer	Pointer to a "sys_language" uid to localize the record into. Basically a localization of a record is making a copy of the record (possibly excluding certain fields defined with "l10n_mode") but changing relevant fields to point to the right sys language / original language record. Requirements for a successful localization is this: ● [ctrl] options "languageField" and "transOrigPointerField" must be defined for the table ● A "sys_language" record with the given "sys_language_uid" must exist. ● The record to be localized by currently be set to "Default" language and not have any value set for the "transOrigPointerField" either. ● There cannot exist another localization to the given language for the record (looking in the original record PID). Apart from this ordinary permissions apply as if the user wants to make a copy of the record on the same page.

Command	Data type	Value
version	array	Versioning action. **Keys:** ● [action] : Keyword determining the versioning action. Options are: ● "new" : Indicates that a new version of the record should be created. Additional keys, specific for "new" action: ● [treeLevels] : *(Only pages)* Integer, -1 to 4, indicating the number of levels of the page tree to version together with a page. This is also referred to as the versioning type: -1 ("element") means only the page record gets versioned (default) 0 ("page") means the page + content tables (defined by ctrl-flag "versioning_followPages") >0 ("branch") means the the whole branch is versioned (*full copy* of all tables), down to the level indicated by the value (1= 1 level down, 2= 2 levels down, etc.) The treeLevel is recorded in the field "t3ver_swapmode" and will be observed when the record is swapped during publishing. ● [label] : Indicates the version label to apply. If not given, a standard label including version number and date is added. ● "swap" : Indicates that the current online version should be swapped with another. Additional keys, specific for "swap" action: ● [swapWith] : Indicates the uid of the record to swap current version with! ● [swapIntoWS]: Boolean, indicates that when a version is published it should be swapped into the workspace of the offline record. ● "clearWSID" : Indicates that the workspace of the record should be set to zero (0). This removes versions out of workspaces without publishing them. ● "flush" : Completely deletes a version without publishing it. ● "setStage" : Sets the stage of an element. *Special feature: The id-key in the array can be a comma list of ids in order to perform the stageChange over a number of records. Also, the internal variable ->generalComment (also available through tce_db.php as "&generalComment") can be used to set a default comment for all stage changes of an instance of tcemain.* Additional keys for this action is: ● [stageId] : Values are: -1 (rejected), 0 (editing, default), 1 (review), 10 (publish) ● [comment] : Comment string that goes into the log.

Examples of commands:

```
$cmd['tt_content'][54]['delete'] = 1;    // Deletes tt_content record with uid=54
$cmd['pages'][1203]['copy'] = -303;   //Copies page id=1203 to the position after page 303
$cmd['pages'][1203]['move'] = 303;  // Moves page id=1203 to the first position in page 303
```

Data Array ($data):

Syntax:

```
$data[ tablename ][ uid ][ fieldname ] = value
```

Description of keywords in syntax:

Key	Data type	Description
tablename	string	Name of the database table. Must be configured in $TCA array, otherwise it cannot be processed.
uid	mixed	The UID of the record that is modified. If the record already exists, this is an integer. If you're creating new records, use a random string prefixed with "NEW", e.g. "NEW7342abc5e6d".
fieldname	string	Name of the database field you want to set a value for. Must be configure in $TCA[*tablename*]['columns']
value	string	Value for "fieldname". (Always make sure $this->stripslashes_values is false before using TCEmain.)

Notice: For FlexForms the data array of the FlexForm field is deeper than three levels. The number of possible levels for FlexForms is infinite and defined by the data structure of the FlexForm. But FlexForm fields always end with a "regular value" of course.

Examples of Data submission:

This creates a new page titled "The page title" as the first page inside page id 45:

```
$data['pages']['NEW9823be87'] = array(
    'title' => 'The page title',
    'subtitle' => 'Other title stuff',
    'pid' => '45'
);
```

This creates a new page titled "The page title" right after page id 45 in the tree:

```
$data['pages']['NEW9823be87'] = array(
    'title' => 'The page title',
    'subtitle' => 'Other title stuff',
    'pid' => '-45'
);
```

This creates two new pages right after each other, located right after the page id 45:

```
$data['pages']['NEW9823be87'] = array(
    'title' => 'Page 1',
    'pid' => '-45'
);
$data['pages']['NEWbe68s587'] = array(
    'title' => 'Page 2',
    'pid' => '-NEW9823be87'
);
```

Notice how the second "pid" value points to the "NEW..." id placeholder of the first record. This works because the new id of the first record can be accessed by the second record. However it works only when the order in the array is as above since the processing happens in that order!

This updates the page with uid=9834 to a new title, "New title for this page", and no_cache checked:

```
$data['pages'][9834] = array(
    'title' => 'New title for this page',
    'no_cache' => '1'
);
```

Clear cache

TCE also has an API for clearing the cache tables of TYPO3:

Syntax:

```
$tce->clear_cacheCmd($cacheCmd);
```

$cacheCmd values	Description
[integer]	Clear the cache for the page id given.
"all"	Clears all cache tables (cache_pages, cache_pagesection, cache_hash). Only available for admin-users unless explicitly allowed by User TSconfig "options.clearCache.all"
"pages"	Clears all pages from cache_pages. Only available for admin-users unless explicitly allowed by User TSconfig "options.clearCache.pages"
"temp_CACHED"	Clears the temp_CACHED files in typo3conf/

Hook for cache post-processing

You can configure cache post-processing with a user defined PHP function. Configuration of the hook can be done from (ext_)localconf.php. An example might look like:

```
$TYPO3_CONF_VARS['SC_OPTIONS']['t3lib/class.t3lib_tcemain.php']['clearCachePostProc'][] =
'myext_cacheProc->proc';
require_once(t3lib_extMgm::extPath('myext') . 'class.myext_cacheProc.php');
```

Flags in TCEmain

There are a few internal variables you can set prior to executing commands or data submission. These are the most significant:

Internal variable	Data type	Description
->deleteTree	Boolean	Sets whether a page tree branch can be recursively deleted. If this is set, then a page is deleted by deleting the whole branch under it (user must have delete permissions to it all). If not set, then the page is deleted *only* if it has no branch. Default is false.

Internal variable	Data type	Description
->copyTree	Integer	Sets the number of branches on a page tree to copy. If 0 then branch is *not* copied. If 1 then pages on the 1st level is copied. If 2 then pages on the second level is copied ... and so on. Default is zero.
->reverseOrder	Boolean	If set, the data array is reversed in the order, which is a nice thing if you're creating a whole bunch of new records. Default is zero.
->copyWhichTables	list of strings (tables)	This list of tables decides which tables will be copied. If empty then none will. If "*" then all will (that the user has permission to of course). Default is "*"
->stripslashes_values	boolean	If set, then all values will be passed through stripslashes(). This has been the default since the birth of TYPO3 in times when input from POST forms were always escaped an needed to be unescaped. Today this is deprecated and values should be passed around without escaped characters. **It is highly recommended to set this value to zero every time the class is used!** If you set this value to false you can pass values as-is to the class and it is most like that this is what you want. Otherwise you would have to pass all values through addslashes() first. Default is (currently) "1" (true) but *might be changed in the future!*

Using t3lib_TCEmain in scripts

It's really easy to use the class "t3lib_TCEmain" in your own scripts. All you need to do is include the class, build a $data/ $cmd array you want to pass to the class and call a few methods.

However please mind that these scripts have to be run in the **backend scope**! There must be a global $BE_USER object.

In your script you simply insert this line to include the class:

```
require_once (PATH_t3lib . 'class.t3lib_tcemain.php');
```

When that is done you can create an instance of t3lib_TCEmain. Here follows a few code listings with comments which will provide you with enough knowledge to get started. It is assumed that you have populated the $data and $cmd arrays correctly prior to these chunks of code. The syntax for these two arrays is explained on the previous pages.

Example: Submitting data

This is the most basic example of how to submit data into the database. It is four lines. Line 1 instantiates the class, line 2 defines that values will be provided without escaped characters (recommended!), line 3 registers the $data array inside the class and initializes the class internally! Finally line 4 will execute the data submission.

```
1: $tce = t3lib_div::makeInstance('t3lib_TCEmain');
2: $tce->stripslashes_values = 0;
3: $tce->start($data, array());
4: $tce->process_datamap();
```

Example: Executing commands

The most basic way of executing commands. Line 1 creates the object, line 2 defines that values will be provided without escaped characters (recommended), line 3 registers the $cmd array inside the class and initializes the class internally! Finally line 4 will execute the commands.

```
1: $tce = t3lib_div::makeInstance('t3lib_TCEmain');
2: $tce->stripslashes_values=0;
3: $tce->start(array(), $cmd);
4: $tce->process_cmdmap();
```

Example: Clearing cache

In this example the clear-cache API is used. No data is submitted, no commands executed. Still you will have to initialize the class by calling the start() method (which will initialize internal variables).

Notice: Clearing "all" cache will be possible only for users that are "admin" or for users with specific permissions to do so.

```
1: $tce = t3lib_div::makeInstance('t3lib_TCEmain');
2: $tce->start(array(), array());
3: $tce->clear_cacheCmd('all');
```

Example: Complex data submission

Imagine the $data array something like this:

```
$data = array(
    'pages' => array(
        'NEW_1' => array(
            'pid' => 456,
            'title' => 'Title for page 1',
        ),
        'NEW_2' => array(
            'pid' => 456,
            'title' => 'Title for page 2',
        ),
    )
);
```

This aims to create two new pages in the page with uid "456". In the follow code this is submitted to the database. Notice how line 3 reverses the order of the array. This is done because otherwise "page 1" is created first, then "page 2" in the *same* PID meaning that "page 2" will end up above "page 1" in the order. Reversing the array will create "page 2" first and then "page 1" so the "expected order" is preserved.

Apart from this line 6 will send a "signal" that the page tree should be updated at the earliest occasion possible. Finally, the cache for all pages is cleared in line 7.

```
1: $tce = t3lib_div::makeInstance('t3lib_TCEmain');
2: $tce->stripslashes_values = 0;
3: $tce->reverseOrder = 1;
4: $tce->start($data, array());
5: $tce->process_datamap();
6: t3lib_BEfunc::getSetUpdateSignal('updatePageTree');
7: $tce->clear_cacheCmd('pages');
```

Example: Both data and commands executed with alternative user object

In this case it is shown how you can use the same object instance to submit both data and execute commands if you like. The order will depend on the order of line 4 and 5.

In line 3 the start() method is called, but this time with the third possible argument which is an alternative BE_USER object. This allows you to force another backend user account to create stuff in the database. This may be useful in certain special cases. Normally you should not set this argument since you want TCE to use the global $BE_USER.

```
1: $tce = t3lib_div::makeInstance('t3lib_TCEmain');
2: $tce->stripslashes_values = 0;
3: $tce->start($data, $cmd, $alternative_BE_USER);
4: $tce->process_datamap();
5: $tce->process_cmdmap();
```

The "tce_db.php" API

This script is a gateway for POST forms to class.t3lib_TCEmain. It has historically been *the* script to which data was posted when you wanted to update something in the database.

Today it is used for editing by only a few scripts, actually only the "Quick Edit" module in "Web>Page" (frontend). The standard forms you find in TYPO3 are normally rendered and handled by "alt_doc.php" which includes t3lib_TCEmain on its own.

For commands it is still used from various locations.

You can send data to this file either as GET or POST vars where POST takes precedence. The variable names you can use are:

GP var name:	Data type	Description
data	array	Data array on the form [tablename][uid][fieldname] = value Typically it comes from a POST form which submits a form field like <input name="data[tt_content][123][header]" value="This is the headline" />
cmd	array	Command array on the form [tablename][uid][command] = value. This array may get additional data set internally based on clipboard commands send in CB var! Typically this comes from GET vars passed to the script like "&cmd[tt_content][123][delete]=1" which will delete Content Element with UID 123
cacheCmd	string	Cache command sent to ->clear_cacheCmd
redirect	string	Redirect URL. Script will redirect to this location after performing operations (unless errors has occurred)

GP var name:	Data type	Description
flags	array	Accepts options to be set in TCE object. Currently it supports "reverseOrder" (boolean).
mirror	array	Example: [mirror][table][11] = '22,33' will look for content in [data][table][11] and copy it to [data][table][22] and [data][table][33]
prErr	boolean	If set, errors will be printed on screen instead of redirection. Should always be used, otherwise you will see no errors if they happen.
CB	array	Clipboard command array. May trigger changes in "cmd"
vC	string	Verification code
uPT	string	Update Page Tree Trigger. If set and the manipulated records are pages then the update page tree signal will be set.

Files: t3lib_extFileFunctions basics

File operations in TCE is handled by the class "t3lib_extFileFunctions" which extends "t3lib_basicFileFunctions". The instructions for file manipulation is passed to this class as a multidimensional array.

Files Array ($file):

Syntax:

```
$file[ command ][ index ][ key ] = value
```

Description of keywords in syntax:

Key	Data type	Description
command	string (command keyword)	The command type you want to execute. *See table below for command keywords, keys and values*
index	integer	Integer index in the array which separates multiple commands of the same type.
key	string	Depending on the command type. The keys will carry the information needed to perform the action. Typically a "target" key is used to point to the target directory or file while a "data" key carries the data. *See table below for command keywords, keys and values*
value	string	The value for the command *See table below for command keywords, keys and values*

Command keywords and values:

Command	Keys	Value
delete	"data"	"data" = Absolute path to the file/folder to delete
copy	"data" "target" "altName"	"data" = Absolute path to the file/folder to copy "target" = Absolute path to the folder to copy to (destination) "altName" = (boolean): If set, a new filename is made by appending numbers/unique-string in case the target already exists.
move	"data" "target" "altName"	(Exactly like copy, just replace the word "copy" with "move")
rename	"data" "target"	"data" = New name, max 30 characters alphanumeric "target" = Absolute path to the target file/folder
newfolder	"data" "target"	"data" = Folder name, max 30 characters alphanumeric "target" = Absolute path to the folder where to create it
newfile	"data" "target"	"data" = New filename "target" = Absolute path to the folder where to create it
editfile	"data" "target"	"data" = The new content "target" = Absolute path to the target file
upload	"data" "target" upload_$id	"data" = ID-number (points to the global var that holds the filename-ref ($GLOBALS["HTTP_POST_FILES"]["upload_".$id]["name"]) "target" = Absolute path to the target folder (destination) upload_$id = File reference. $id must equal value of file[upload][...][data]! See t3lib_t3lib_extFileFunctions::func_upload()

Command	Keys	Value
unzip	"data" "target"	"data" = Absolute path to the zip-file. (fileextension must be "zip") "target" = The absolute path to the target folder (destination) (if not set, default is the same as the zip-file)

It is unlikely that you will need to use this internally in your scripts like you will need t3lib_TCEmain. It is fairly uncommon to need the file manipulations in own scripts unless you make a special application. Therefore the most typical usage of this API is from tce_file.php and the core scripts that are activated by the "File > List" module.

However, if you need it this is an example (taken from tce_file.php) of how to initialize the usage.

```
1:     // Initializing:
2: $this->fileProcessor = t3lib_div::makeInstance('t3lib_extFileFunctions');
3: $this->fileProcessor->init($FILEMOUNTS, $TYPO3_CONF_VARS['BE']['fileExtensions']);
4: $this->fileProcessor->init_actionPerms($BE_USER->user['fileoper_perms']);
5:
6: $this->fileProcessor->start($this->file);
7: $this->fileProcessor->processData();
```

Line 2 makes an instance of the class and line 3 initializes the object with the filemounts of the current user and the array of allow/deny file extensions in web-space and ftp-space (see below). Then the file operation permissions are loaded from the user object in line 4. Finally, the file command array is loaded in line 6 (and internally additional configuration takes place from $TYPO3_CONF_VARS!). In line 7 the command map is executed.

Web-space, FTP-space and $TYPO3_CONF_VARS['BE']['fileExtensions']

The control of file extensions goes in two categories. Webspace and ftpspace. Webspace is folders accessible from a web browser (below TYPO3_DOCUMENT_ROOT) and ftpspace is everything else.

The control is done like this: if an extension matches 'allow' then the check returns true. If not and an extension matches 'deny' then the check return false. If no match at all, returns true.

You list extensions comma-separated. If the value is a '*' every extension is matched. If no file extension, true is returned if 'allow' is '*', false if 'deny' is '*' and true if none of these matches. This (default) configuration below accepts everything in ftpspace and everything in webspace except php files:

```
$TYPO3_CONF_VARS['BE']['fileExtensions'] = array (
    'webspace' => array('allow' => '', 'deny' => 'php'),
    'ftpspace' => array('allow' => '*', 'deny' => '')
);
```

The "tce_file.php" API

This script serves as the file administration part of the TYPO3 Core Engine. It's a gateway for TCE (TYPO3 Core Engine) file-handling through POST forms. It uses "t3lib_extfilefunc" for the manipulation of the files.

This script is used from the File > List module where you can rename, create, delete etc. files and directories on the server.

You can send data to this file either as GET or POST vars where POST takes precedence. The variable names you can use are:

GP var name:	Data type	Description
file	array	Array of file operations. See previous information about "t3lib_extFileFunctions" This could typically be a GET var like "&file[delete][0][data]=[absolute file path]" or a POST form field like "<input type="text" name="file[newfolder][0][data]" value=""/><input type="hidden" name="file[newfolder][0][target]" value="[absolute path to folder to create in]"/>"
redirect	string	Redirect URL. Script will redirect to this location after performing operations.
CB	array	Clipboard command array. May trigger changes in "file"
vC	string	Verification code
overwriteExistingFiles	boolean	If existing files should be overridden.

Hooks

The concept of "hooks"

Hooks are basically places in the source code where a user function will be called for processing if such has been configured. Hooks provide a way to extend functionality of TYPO3 and extensions easily and without blocking for others to do the same.

Hooks vs. XCLASS extensions

Hooks are the recommended way of extending TYPO3 compared to extending the PHP classes with a child class (see "XCLASS extensions"). It is so because only one extension of a PHP class can exist at a time while hooks may allow many different user-designed processing functions to occur. On the other hand hooks have to be implemented in the core before you can use them while extending a PHP class via the XCLASS method allows you to extend anything spontaneously.

Proposing hooks

If you need to extend something which has no hook yet, then you should suggest implementing a hook. Normally that is rather easily done by the author of the source you want to extend.

How a hook looks

The two lines of code below are an example of how a hook is used for clear-cache post-processing. The objective of this need could be to perform additional actions whenever the cache is cleared for a specific page.

```
require_once(t3lib_extMgm::extPath('myext') . 'class.myext_cacheProc.php');
$TYPO3_CONF_VARS['SC_OPTIONS']['t3lib/class.t3lib_tcemain.php']['clearCachePostProc'][] =
'myext_cacheProc->proc';
```

Line 1 includes a class which contains the user-defined PHP code to be called by the hook.

Line 2 registers the class/method name from the included file with a hook inside of "t3lib_TCEmain". The hook will call the user function after the clear-cache command has been executed. The user function will receive parameters which allows it to see what clear-cache action was performed and typically also an object reference to the parent object. Then the user function can take additional actions as needed.

If we take a look inside of t3lib_TCEmain we find the hook to be activated like this:

```
   1:     // Call post processing function for clear-cache:
   2: if (is_array($TYPO3_CONF_VARS['SC_OPTIONS']['t3lib/class.t3lib_tcemain.php']
['clearCachePostProc']))    {
   3:     $_params = array('cacheCmd' => $cacheCmd);
   4:     foreach($TYPO3_CONF_VARS['SC_OPTIONS']['t3lib/class.t3lib_tcemain.php']['clearCachePostProc']
as $_funcRef)    {
   5:         t3lib_div::callUserFunction($_funcRef, $_params, $this);
   6:     }
   7: }
```

This is how hooks are typically constructed. The main action happens in line 5 where the function "t3lib_div::callUserFunction()" is called. The user function is called with two arguments, an array with variable parameters and the parent object.

In line 3 the contents of the parameter array is prepared. This is of high interest to you because this is where you see what data is passed to you and what data might possibly be passed by reference and thereby possible to manipulate from your hook function.

Finally, notice how the array $TYPO3_CONF_VARS['SC_OPTIONS']['t3lib/class.t3lib_tcemain.php']['clearCachePostProc'] is traversed and for each entry the value is expected to be a function reference which will be called. This allows many hooks to be called at the same place. The hooks can even rearrange the calling order if they dare.

The syntax of a function reference (or object reference if t3lib_div::getUserObj is used in the hook instead) can be seen in the API documentation of t3lib_div.

Hook configuration

There is no complete index of hooks in the core. But they are easy to search for and find. And typically it comes quite naturally since you will find the hooks in the code you want to extend - if they exist.

This index will list the main variable spaces for configuration of hooks. By the names of these you can easily scan the source code to find which hooks are available or might be interesting for you.

The index below also includes some variable spaces which do not only carry hook configuration but might be used for other purposes as well.

$TYPO3_CONF_VARS['EXTCONF']

Configuration space for extensions.

This will contain all kinds of configuration options for specific extensions including possible hooks in them! What options are available to you will depend on a search in the documentation for that particular extension.

`$TYPO3_CONF_VARS['EXTCONF'][ extension_key ][ sub_key ] = value`

- **extension_key :** The unique extension key
- **sub_key :** Whatever the script defines. Typically it identifies the context of the hook
- **value :** It is up to the extension what the values mean, if they are mere configuration options or hooks or whatever and how deep the arrays go. Read the source code where the options are implemented to see. Or the documentation of the extension, if available.

Notice: $TYPO3_CONF_VARS['EXTCONF'] is the recommended place to put hook configuration that are available inside your extensions!

Here is an example of how the EXTCONF array is used inside an extension. Notice, this example is *not* a hook (sorry, couldn't find a better example) but it is based on the same principles. It is just an example of configuration of additional "root line fields" that can be used during indexing (line 8-12). It shows the versatility of the EXTCONF array:

```
   1: function getRootLineFields(&$fieldArr)    {
   2:     $rl = $this->rootLevel;
   3:
   4:     $fieldArr['rl0'] = intval($rl[0]['uid']);
   5:     $fieldArr['rl1'] = intval($rl[1]['uid']);
   6:     $fieldArr['rl2'] = intval($rl[2]['uid']);
   7:
   8:     if (is_array($GLOBALS['TYPO3_CONF_VARS']['EXTCONF']['indexed_search']
['addRootLineFields']))    {
   9:         foreach($GLOBALS['TYPO3_CONF_VARS']['EXTCONF']['indexed_search']['addRootLineFields'] as
$fieldName => $rootLineLevel)    {
  10:             $fieldArr[$fieldName] = intval($rl[$rootLineLevel]['uid']);
  11:         }
  12:     }
  13: }
```

$TYPO3_CONF_VARS['SC_OPTIONS']

Configuration space for core scripts.

This array is created as an ad hoc space for creating hooks from any script. This will typically be used from the core scripts of TYPO3 which do not have a natural identifier like extensions have their extension keys.

`$TYPO3_CONF_VARS['SC_OPTIONS'][ main_key ][ sub_key ][ index ] = function_reference`

- **main_key :** The relative path of a script (for output scripts it should be the "script ID" as found in a comment in the HTML header)
- **sub_key :** Whatever the script defines. Typically it identifies the context of the hook.
- **index :** Integer index typically. Can be unique string if you have a reason to use that. Normally it has no greater significance since the value of the key is not used. The hooks normally traverse over the array and uses only the value (function reference)
- **function_reference :** A function reference using the syntax of t3lib_div::callUserFunction() or t3lib_div::getUserObj() depending on implementation of the hook.

The above syntax is how a hook is typically defined but it might differ and it might not be a hook at all, but just configuration. Depends on implementation in any case.

The following example shows a hook from tslib_fe. In this case the function t3lib_div::getUserObj() is used for the hook. The function_reference is referring to the class name only since the function returns an object instance of that class. The method name to call is predefined by the hook, in this case "sendFormmail_preProcessVariables()". This method allows to pass any number of variables along instead of the limited $params and $pObj variables from t3lib_div::callUserFunction().

```
    // Hook for preprocessing of the content for formmails:
if (is_array($this->TYPO3_CONF_VARS['SC_OPTIONS']['tslib/class.tslib_fe.php']['sendFormmail-
PreProcClass'])) {
    foreach($this->TYPO3_CONF_VARS['SC_OPTIONS']['tslib/class.tslib_fe.php']['sendFormmail-PreProcClass'] as $_classRef) {
        $_procObj = &t3lib_div::getUserObj($_classRef);
        $EMAIL_VARS = $_procObj->sendFormmail_preProcessVariables($EMAIL_VARS, $this);
    }
}
```

In this example we are looking at a special hook, namely the one for RTE transformations. Well, maybe this is not a "hook" in the normal sense, but the same principles are used. In this case the "index" key is defined to be the transformation key name, not a random integer since we do not iterate over the array as usual. In this case t3lib_div::getUserObj() is also used.

```
if ($_classRef = $GLOBALS['TYPO3_CONF_VARS']['SC_OPTIONS']['t3lib/class.t3lib_parsehtml_proc.php']['transformation'][$cmd]) {
    $_procObj = &t3lib_div::getUserObj($_classRef);
    $_procObj->pObj = &$this;
    $_procObj->transformationKey = $cmd;
    $value = $_procObj->transform_db($value, $this);
}
```

A classic hook also from tslib_fe. This is also based on t3lib_div::callUserFunction() and it passes a reference to $this along to the function via $_params. In the user defined function $_params['pObj']->content is meant to be manipulated in some way. The return value is insignificant - everything works by the reference to the parent object.

```
    // Hook for post-processing of page content cached/non-cached:
if (is_array($this->TYPO3_CONF_VARS['SC_OPTIONS']['tslib/class.tslib_fe.php']['contentPostProc-all'])) {
    $_params = array('pObj' => &$this);
    foreach($this->TYPO3_CONF_VARS['SC_OPTIONS']['tslib/class.tslib_fe.php']['contentPostProc-all'] as $_funcRef) {
        t3lib_div::callUserFunction($_funcRef, $_params, $this);
    }
}
```

$TYPO3_CONF_VARS['TBE_MODULES_EXT']

Configuration space for backend modules.

Among these configuration options you might find entry points for hooks in the backend. This somehow overlaps the intention of "SC_OPTIONS" above but this array is an older invention and slightly outdated.

`$TBE_MODULES_EXT[ backend_module_key ][ sub_key ] = value`

- **backend_module_key :** The backend module key for which the configuration is used.
- **sub_key :** Whatever the backend module defines.
- **value :** Whatever the backend module defines.

The following example shows TBE_MODULES_EXT being used for adding items to the Context Sensitive Menus (Clickmenu) in the backend. The hook value is an array with a key pointing to a file reference to class file to include. Later each class is instantiated and a fixed method inside is called to do processing on the array of menu items. This kind of hook is non-standard in the way it is made.

```
    // Setting internal array of classes for extending the clickmenu:
$this->extClassArray = $GLOBALS['TBE_MODULES_EXT']['xMOD_alt_clickmenu']['extendCMclasses'];

    // Traversing that array and setting files for inclusion:
if (is_array($this->extClassArray)) {
    foreach($this->extClassArray as $extClassConf) {
        if ($extClassConf['path'])    $this->include_once[]=$extClassConf['path'];
    }
}
```

The following code listings works in the same way. First, a list of class files to include is registered. Then in the second code listing the same array is traversed and each class is instantiated and a fixed function name is called for processing.

```
    // Setting class files to include:
if (is_array($TBE_MODULES_EXT['xMOD_db_new_content_el']['addElClasses'])) {
    $this->include_once = array_merge($this->include_once, $TBE_MODULES_EXT['xMOD_db_new_content_el']['addElClasses']);
}
```

```
    // PLUG-INS:
if (is_array($TBE_MODULES_EXT['xMOD_db_new_content_el']['addElClasses'])) {
    reset($TBE_MODULES_EXT['xMOD_db_new_content_el']['addElClasses']);
    while(list($class,$path)=each($TBE_MODULES_EXT['xMOD_db_new_content_el']['addElClasses'])) {
        $modObj = t3lib_div::makeInstance($class);
        $wizardItems = $modObj->proc($wizardItems);
    }
}
```

Creating hooks

You are encouraged to create hooks in your extensions if they seem meaningful. Typically someone would request a hook somewhere. Before you implement it, consider if it is the right place to put it etc. On the one hand we want to have many hooks but not more than needed. Redundant hooks or hooks which are implemented in the wrong context is just confusing. So put a little thought into it first, but be generous.

There are two main methods of calling a user defined function in TYPO3.

- t3lib_div::callUserFunction() - The classic way. Takes a file/class/method reference as value and calls that function. The argument list is fixed to a parameter array and a parent object. So this is the limitation. The freedom is that the reference defines the function name to call. This method is mostly useful for small-scale hooks in the sources.
- t3lib_div::getUserObject() - Create an object from a user defined file/class. The method called in the object is fixed by the hook, so this is the non-flexible part. But it is cleaner in other ways, in particular that you can even call many methods in the object and you can pass an arbitrary argument list which makes the API more beautiful. You can also define the objects to be singletons, instantiated only once in the global scope.

Here follows some examples.

Hook made with t3lib_div::getUserObj()

```
    // Hook for preprocessing of the content for formmails:
if (is_array($this->TYPO3_CONF_VARS['SC_OPTIONS']['tslib/class.tslib_fe.php']['sendFormmail-
PreProcClass'])) {
    foreach($this->TYPO3_CONF_VARS['SC_OPTIONS']['tslib/class.tslib_fe.php']['sendFormmail-
PreProcClass'] as $_classRef) {
        $_procObj = &t3lib_div::getUserObj($_classRef);
        $EMAIL_VARS = $_procObj->sendFormmail_preProcessVariables($EMAIL_VARS, $this);
    }
}
```

Hook made with t3lib_div::callUserFunction()

```
    // Call post processing function for constructor:
if (is_array($this->TYPO3_CONF_VARS['SC_OPTIONS']['tslib/class.tslib_fe.php']['tslib_fe-PostProc'])) {
    $_params = array('pObj' => &$this);
    foreach($this->TYPO3_CONF_VARS['SC_OPTIONS']['tslib/class.tslib_fe.php']['tslib_fe-PostProc'] as
$_funcRef) {
        t3lib_div::callUserFunction($_funcRef,$_params, $this);
    }
}
```

Variables and Constants

After init.php has been included in the backend there is a set of variables, constants and classes available to the parent script.

The column "Avail. in FE" is an indicator that tells you if the constant, variable or class mentioned is also available to scripts running under the frontend of the "cms" extension.

Constants

Constants normally define paths and database information. These values are global and cannot be changed when they are first defined. This is why constants are used for such vital information.

These constants are defined by either init.php or scripts included from that script.

Notice: Constants in italics *may* be set in a script prior to inclusion of init.php so they are optional.

Constant	Defined in	Description	Avail. in FE
TYPO3_OS	init.php	Operating systen; Windows = "WIN", other = "" (presumed to be some sort of Unix)	YES

Constant	Defined in	Description	Avail. in FE
TYPO3_MODE	init.php	Mode of TYPO3: Set to either "FE" or "BE" depending on frontend or backend execution. So in "init.php" and "thumbs.php" this value is "BE"	YES value = "FE"
PATH_thisScript	init.php	Abs. path to current script.	YES
TYPO3_mainDir	init.php	This is the directory of the backend administration for the sites of this TYPO3 installation. Hardcoded to "typo3/". Must be a subdirectory to the website. See elsewhere for descriptions on how to change the default admin directory, "typo3/", to something else.	YES
PATH_typo3	init.php	Abs. path of the TYPO3 admin dir (PATH_site + TYPO3_mainDir).	-
PATH_typo3_mod	init.php	Relative path (from the PATH_typo3) to a properly configured module. Based on TYPO3_MOD_PATH.	-
PATH_site	init.php	Abs. path to directory with the frontend (one directory above PATH_typo3)	YES
PATH_t3lib	init.php	Abs. path to "t3lib/" (general TYPO3 library) within the TYPO3 admin dir	YES
PATH_typo3conf	init.php	Abs. TYPO3 configuration path (local, not part of source) Must be defined in order for "t3lib/config_default.php" to return!	YES
TYPO3_db	config_default.php	Name of the database, for example "t3_coreinstall". Is defined after the inclusion of "typo3conf/localconf.php" (same for the other TYPO3_* constants below	YES
TYPO3_db_username	config_default.php	Database username	YES
TYPO3_db_password	config_default.php	Database password	YES
TYPO3_db_host	config_default.php	Database hostname, e.g. "localhost"	YES
TYPO3_tables_script	config_default.php	By default "t3lib/stddb/tables.php" is included as the main table definition file. Alternatively this constant can be set to the filename of an alternative "tables.php" file. Must be located in "typo3conf/" **Deprecated**. Make Extensions instead.	YES
TYPO3_extTableDef_script	config_default.php	Name of a php-include script found in "typo3conf/" that contains php-code that further modifies the variables set by "t3lib/stddb/tables.php" **Deprecated.** Make Extensions instead.	YES
TYPO3_languages	config_default.php	Defines the system language keys in TYPO3s backend.	YES
TYPO3_DLOG	config_default.php	If true, calls to t3lib_div::devLog() can be made in both frontend and backend; This is event logging which can help to track debugging in general.	YES
TYPO3_MOD_PATH	[prior to init.php]	Path to module relative to PATH_typo3 (as defined in the module configuration). Must be defined prior to "init.php".	-
TYPO3_enterInstallScript	[prior to init.php]	If defined and set true the Install Tool is activated and the script exits after that. Used in "typo3/install/index.php": **Example:** `define('TYPO3_enterInstallScript', '1');`	-

TYPO3_PROCEED_IF_NO_USER	[prior to init.php]	If defined and set true the "init.php" script will return to the parent script *even if no backend user was authenticated!*
		This constant is set by for instance the "index.php" script so it can include "init.php" and still show the login form:
		`define("TYPO3_PROCEED_IF_NO_USER", 1);`
		`require ("init.php");`
		Please be very careful with this feature - use it only when you have total control of what you are doing!
			-

TYPO3_cliMode	[prior to init.php]	Initiates CLI (Command Line Interface) mode. This is used when you want a shell executable PHP script to initialize a TYPO3 backend. For more details see section about "Initialize TYPO3 backend in a PHP shell script" in "Inside TYPO3"	
TYPO3_version	config_default.php	The TYPO3 version: x.x.x for released versions, x.x.x-dev for development versions leading up to releases x.x.x-bx for beta-versions	YES

Global variables

Notice: Variables in italics *may* be set in a script prior to inclusion of "init.php" so they are optional.

Notice: The variables from "t3lib/stddb/tables.php" are only available in the frontend occasionally or partly. Please read more in the documentation of the "cms" extension on this issue.

Global variable	Defined in	Description	Avail. in FE
$TYPO3_CONF_VARS	config_default.php	TYPO3 configuration array. Please refer to the source code of "t3lib/config_default.php" where each option is described in detail as comments. The same comments are also available in the Install Tool under the menu "All Configuration"	YES
$TYPO3_LOADED_EXT	config_default.php	Array with all loaded extensions listed with a set of paths. You can check if an extension is loaded by the function t3lib_extMgm::isLoaded($key) where $key is the extension key of the module.	YES
$TYPO3_DB	init.php	An instance of the TYPO3 DB wrapper class, t3lib_db. You have to use this object for all interaction with the database. t3lib_db contains mysql wrapper functions so you easily swap all hardcoded MySQL calls with function calls to $GLOBALS['TYPO3_DB']->	YES
$EXEC_TIME	config_default.php	Is set to "time()" so that the rest of the script has a common value for the script execution time.	YES
$SIM_EXEC_TIME	config_default.php	Is set to $EXEC_TIME but can be altered later in the script if we want to simulate another execution-time when selecting from e.g. a database (used in the frontend for preview of future and past dates)	YES
$TYPO_VERSION	config_default.php	*Deprecated - used constant "TYPO3_version" instead!*	YES
$TYPO3_AJAX	ajax.php	Set to true to indicate that an AJAX call is being processed	-
$CLIENT	init.php	Array with browser information (based on HTTP_USER_AGENT). Array keys: "BROWSER" = msie,net,opera or blank, "VERSION" = browser version as double, "SYSTEM" = win,mac,unix	YES
$PARSETIME_START	init.php	Time in milliseconds right after inclusion of the configuration.	-
$PAGES_TYPES	t3lib/stddb/tables.php	See section on $TCA	(occastionally)
$ICON_TYPES	t3lib/stddb/tables.php	See section on $TCA	(occastionally)
$LANG_GENERAL_LABELS	t3lib/stddb/tables.php	See section on $TCA	(occastionally)
$TCA	t3lib/stddb/tables.php	See section on $TCA	YES, partly
$TBE_MODULES	t3lib/stddb/tables.php	The backend main/sub module structure. See section elsewhere plus sourcecode of "class.t3lib_loadmodules.php" which also includes some examples.	(occastionally)
$TBE_STYLES	t3lib/stddb/tables.php		(occastionally)
$T3_SERVICES	t3lib/stddb/tables.php	Global registration of services.	
$T3_VAR	config_default.php	Space for various internal global data storage in TYPO3. Each key in this array is a data space for an application. Keys currently defined for use is: ['callUserFunction'] + ['callUserFunction_classPool']: Used by t3lib_div::callUserFunction to store persistent objects. ['getUserObj'] : User by t3lib_div::getUserObj to store persistent objects. ['RTEobj'] : Used to hold the current RTE object if any. See t3lib_BEfunc. ['ext'][*extension-key*] : Free space for extensions.	
$FILEICONS	t3lib/stddb/tables.php	Assoc. array; keys are the type (e.g. "tif") and values are the filename (without path)	(occastionally)
$WEBMOUNTS	init.php	Array of uid's to be mounted in the page-tree	(depends)
$FILEMOUNTS	init.php	Array of filepaths on the server to be mountet in the directory tree	(depends)
$BE_USER	init.php	Backend user object	(depends)
$temp_*	-	Various temporary variables are allowed to use global variables prefixed $temp_	-

Global variable	Defined in	Description	Avail. in FE
$typo_db*	[config_default.php but N/A!]	Variables used inside of "typo3conf/localconf.php" to configure the database. **Notice:** These values are unset again by "config_default.php".	-
$TBE_MODULES_EXT	[In ext_tables.php files of extensions]	Used to store information about modules from extensions that should be included in "function menus" of real modules. See the Extension API for details. Unset in "config_default.php"	(occasionally)
$TCA_DESCR	[tables.php files]	Can be set to contain file references to local lang files containing TCA_DESCR labels. See section about Context Sensitive Help. Unset in "config_default.php"	

Backend User Object

Checking user access for $BE_USER from PHP

The backend user of a session is always available to the backend scripts as the global variable $BE_USER. The object is created in init.php and is an instance of the class "t3lib_beUserAuth" (which extends "t3lib_userAuthGroup" which extends "t3lib_userAuth").

In addition to $BE_USER two other global variables are of interest - $WEBMOUNTS and $FILEMOUNTS, each holding an array with the DB mounts and File mounts of the $BE_USER.

In order to introduce how the $BE_USER object can be helpful to your backend scripts/modules, this is a few examples:

Checking access to current backend module

$MCONF is module configuration and the key $MCONF["access"] determines the access scope for the module. This function call will check if the $BE_USER is allowed to access the module and if not, the function will exit with an error message.

```
$BE_USER->modAccess($MCONF, 1);
```

Checking access to any backend module

If you know the module key you can check if the module is included in the access list by this function call:

```
$BE_USER->check('modules', 'web_list');
```

Here access to the module "Web>List" is checked.

Access to tables and fields?

The same function ->check() can actually check all the ->groupLists inside $BE_USER. For instance:

Checking modify access to the table "pages":

```
$BE_USER->check('tables_modify', 'pages');
```

Checking selecting access to the table "tt_content":

```
$BE_USER->check('tables_select', 'tt_content');
```

Checking if a table/field pair is allowed explicitly through the "Allowed Excludefields":

```
$BE_USER->check('non_exclude_fields', $table . ':' . $field);
```

Is "admin"?

If you want to know if a user is an "admin" user (has complete access), just call this method:

```
$BE_USER->isAdmin();
```

Read access to a page?

This function call will return true if the user has read access to a page (represented by its database record, $pageRec):

```
$BE_USER->doesUserHaveAccess($pageRec, 1);
```

Changing the "1" for other values will check other permissions. For example "2" will check id the user may edit the page and "4" will check if the page can be deleted.

Is a page inside a DB mount?

Access to a page should not be checked only based on page permissions but also if a page is found within a DB mount for ther user. This can be checked by this function call ($id is the page uid):

```
$BE_USER->isInWebMount($id)
```

Selecting readable pages from database?

If you wish to make a SQL statement which selects pages from the database and you want it to be only pages that the user has read access to, you can have a proper WHERE clause returned by this function call:

```
$BE_USER->getPagePermsClause(1);
```

Again the number "1" represents the "read" permission; "2" would represent "edit" permission and "4" would be delete permission and so on. The result from the above query could be this string:

```
((pages.perms_everybody & 1 = 1)OR(pages.perms_userid = 2 AND pages.perms_user & 1 =
1)OR(pages.perms_groupid in (1) AND pages.perms_group & 1 = 1))
```

Saving module data

This stores the input variable $compareFlags (an array!) with the key "tools_beuser/index.php/compare"

```
$compareFlags = t3lib_div::GPvar('compareFlags');
$BE_USER->pushModuleData('tools_beuser/index.php/compare', $compareFlags);
```

Getting module data

This gets the module data with the key "tools_beuser/index.php/compare" (lasting only for the session)

```
$compareFlags = $BE_USER->getModuleData('tools_beuser/index.php/compare', 'ses');
```

Returning object script from TSconfig

This function can return a value from the "User TSconfig" structure of the user. In this case the value for "options.clipboardNumberPads":

```
$BE_USER->getTSConfigVal('options.clipboardNumberPads');
```

Getting the username

The full "be_users" record of a authenticated user is available in $BE_USER->user as an array. This will return the "username":

```
$BE_USER->user['username']
```

Get User Configuration value

The internal ->uc array contains options which are managed by the User>Setup module (extensions "setup"). These values are accessible in the $BE_USER->uc array. This will return the current state of "Condensed mode" for the user:

```
$BE_USER->uc['condensedMode']
```

Using the system log

The log table (sys_log)

Writing to the system log is done using the backend user object:

```
$this->BE_USER->writelog($type, $action, $error, $details_nr, $details, $data, $table, $recuid, $recpid,
$event_pid, $NEWid);
```

Here are description of the arguments to this function call:

Field	Type	Var	Description
type	tinyint	$type	Value telling which module in TYPO3 set the log entry. The type values are paired with an action-integer which is telling in more detail what the event was. Here type and action values are arranged hierarchically (type on first level, action on second level): ● 1 : t3lib_TCEmain ("TYPO3 Core Engine" where database records are manipulated) ● Action values are for new, updated, copy, move, delete etc. ● 2 : "tce_file" (File handling in fileadmin/ and absolute filemounts) ● Action values are for various file handling types like upload, rename, edit etc. ● 3 : System (e.g. sys_history save) ● 4 : Modules: This is the mode you may use for extensions having backend module functionality. Probably you would like to use BE_USER->simplelog() for your extensions. ● 254 : Personal settings changed ● 255 : Login or Logout action ● 1=login ● 2=logout ● 3=failed login (+ errorcode 3) ● 4=failure_warning_email sent
action	tinyint	$action	*See "type" above* When not available, use value "0"
error	tinyint	$error	Error level: ● 0 = message, a notice of an action that happened. ● 1 = error, typically a permission problem for the user ● 2 = System Error, something which should not happen for technical reasons. ● 3 = Security notice, like login failures
details_nr	tinyint	$details_nr	Number of "detail" message. This number should be unique for the combination of type/action -1 is a temporary detail number you can use while developing and error messages are not fixed yet. 0 is a value that means the message is not supposed to be translated >=1 means the message is fixed and ready for translation.
details	tinytext	$details	The log message text (in english). By identification through type/action/details_nr this can be translated through the localization system. If you insert "%s" markers in the details message and set $data to an array the first 5 entries (keys 0-4) from $data will substitute the markers sequentially (using sprintf)
log_data	tinyblob	$data	Data that follows the log entry. Can be an array. See "details" for more info.
tablename	varchar(40)	$table	Table name. Special field used by tce_main.php.
recuid	int	$recuid	Record UID. Special field used by tce_main.php.
recpid	int	$recpid	Record PID. Special field used by tce_main.php. [OBSOLETE; not used anymore.]
event_pid	int	$event_pid	The page ID (pid) where the event occurred. Used to select log-content for specific pages.
NEWid	varchar(20)	$NEWid	Special field used by tce_main.php. NEWid string of newly created records.
tstamp	int	-	EXEC_TIME of event, UNIX time in seconds.
uid	int	-	Unique ID for log entry, automatically inserted
userid	int	-	User ID of backend user, automatically set for you
IP	varchar(39)	-	REMOTE_ADDR of client
workspace	int	-	Workspace ID

Making logging simple

While it is nice to have log message categorized and numbered during development and sometimes beyond that point a simpler logging API is necessary. Therefore you can also call this function:

```
BE_USER->simplelog($message, $extKey='', $error=0);
```

All you need is to set $message to store a log message. If you call it from an extension it is good practice to also supply the extension key. Finally you can add the error number (according to the table above) if you need to signal an error.

Using the system registry

The purpose of the registry (introduced in TYPO3 4.3) is to hold key-value pairs of information. You can actually think of it being an equivalent to the Windows registry (just not as complicated).

You might use the registry to store information that your script needs to store across sessions or request.

An example would be a setting that needs to be altered by a PHP script, which currently is not possible with TypoScript.

Another example: The scheduler system extension stores when it ran the last time. The reports system extension then checks that value, in case it determines that the scheduler hasn't run for a while it issues a warning. While this might not be of great use to anyone with an actual cron job set up for the scheduler, it is of use for users that have to run the scheduler tasks by hand due to missing access to a cron job.

The registry is not meant to store things that are supposed to go into a session or a cache, use the appropriate API for these instead.

The registry table (sys_registry)

Here's a description of the fields found in the sys_registry table:

Field	Type	Description
uid	int	Primary key, needed for replication and also usfull as an index.
entry_namespace	varchar (128)	Represents an entry's namespace. In general the namespace is an extension key starting with "tx_", a user script's prefix "user_", or "core" for entries that belong to the core. The point of namespaces is that entries with the same key can exist inside different namespaces.
entry_key	varchar (255)	The entry's key. Together with the namespace the key is unique for the whole table. The key can be any string to identify the entry. It's recommended to use dots as dividers if necessary. This way the naming is similar to the already known syntax in TypoScript.
entry_value	blob	The entry's actual value. The value is stored as a serialized string, thus you can even store arrays or objects in a registry entry – it's not recommended though. Using phpMyAdmin's Show BLOB option you can check the value in that field although being stored as a binary.

The registry API

To use the registry, there's an easy to use API. Simply use `t3lib_div::makeInstance('t3lib_Registry')` to retrieve an instance of the registry. The instance returned will always be the same as the registry is a singleton:

```
$registry = t3lib_div::makeInstance('t3lib_Registry');
```

After retrieving an instance of the registry you can access the registry values through its get() method. The get() method offers an interesting third parameter to specify a default value, that value is returned in case the requested entry was not found in the registry. That happens when accessing an entry for the first time for example. Setting a value is easy as well by using the set() method.

Method	Parameters	Description
set	**$namespace**: namespace in which to set the value **$key**: the key of the value to set **$value**: the value to store	Represents an entry's namespace. In general the namespace is an extension key starting with "tx_", a user script's prefix "user_", or "core" for entries that belong to the core.
get	**$namespace**: namespace to get the value from **$key**: the key of the value to retrieve **$defaultValue**: a default value if the key was not found in the given namespace	Used to get a value from the registry.
remove	**$namespace**: namespace to remove the value from **$key**: the key of the value to remove	Remove an entry from a given namespace.
removeAllByNamespace	**$namespace**: namespace to empty	Deletes all value for a given namespace.

Note that you should not store binary data into the registry, it's not designed to do that. Use the filesystem instead, if you have such needs.

Examples

Here's an example taken from the Scheduler system extension:

```
$registry = t3lib_div::makeInstance('t3lib_Registry');
$runInformation = array('start' => $GLOBALS['EXEC_TIME'], 'end' => time(), 'type' => $type);
$registry->set('tx_scheduler', 'lastRun', $runInformation);
```

It is retrieved later using:

```
$registry = t3lib_div::makeInstance('t3lib_Registry');
$lastRun = $registry->get('tx_scheduler', 'lastRun');
```

PHP Class Extension

Introduction

Practically all important scripts have their main code encapsulated in a class (typically named SC_[scriptname] and instantiated as the global object $SOBE) and almost all library classes used in TYPO3 - both frontend and backend - can be extended by user defined classes. Extension of TYPO3 PHP classes may also be referred to as an "XCLASS extension".

Extending TYPO3s PHP classes is recommended mostly for special needs in individual projects. This is due to the limitation that a class can only be extended once. Thus, if many extensions try to extend the same class, only one of them will succeed and in turn the others will not function correctly.

So, extending classes is a great option for individual projects where special "hacks" are needed. But generally it is a poor way of programming TYPO3 extensions in which case you should look for a system hook or request a system hook to be made for your purpose if generally meaningful.

Configuring user-classes works like this:

1. In (ext_)localconf.php you configure for either frontend or backend that you wish to include a file with the extension of the class. This inclusion is usually done in the end of the class-file itself based on a lookup in TYPO3_CONF_VARS.
2. Whenever the class is instantiated as an object, the source code checks if a user-extension of that class exists. If so, then *that* class (or an extension of the extended class) is instantiated and not the "normal" (parent) class.
   Getting the correct instance of a class is done by using the function t3lib_div::makeInstance() instead of "new ..." when an object is created.

Example

Say you wish to make an addition to the stdWrap method found in the class "tslib_cObj" (found in the class file `typo3/sysext/cms/tslib/class.tslib_content.php`).

The first thing to do is to create the extension class. So you create a file in the typo3conf/ directory named "class.ux_tslib_content.php". "ux" is a prefix meaning "user-extension". This file may look like this:

```
<?php
/**
* User-Extension of tslib_cObj class.
*
* @author    Kasper Skårhøj <kasper@typo3.com>
*/

class ux_tslib_cObj extends tslib_cObj {
    function stdWrap($content,$conf) {
            // Call the real stdWrap function in the parent class:
        $content = parent::stdWrap($content,$conf);
            // Process according to my user-defined property:
        if ($conf['userDefined_wrapInRed']) {
            $content='<font color="red">' . $content . '</font>';
        }
        return $content;
    }
}
?>
```

The next thing is to configure TYPO3 to include this class file as well after the original file tslib/class.tslib_content.php:

```
$TYPO3_CONF_VARS['FE']['XCLASS']['tslib/class.tslib_content.php']=
                              PATH_typo3conf . 'class.ux_tslib_content.php';
```

So when the file "tslib/class.tslib_content.php" is included inside of class.tslib_pagegen.php, the extension class is included immediately from inside the "tslib/class.tslib_content.php" file (this is from the bottom of the file):

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

The last thing which remains is to instantiate the class ux_tslib_cObj instead of tslib_cObj. This is done automatically, because everywhere tslib_cObj is instantiated, it is first examined if ux_tslib_cObj exists and if so, that class is instantiated instead!

This is done by instantiating the object with "t3lib_div::makeInstance()":

```
$cObj = t3lib_div::makeInstance('tslib_cObj');
```

Originally it looked like this:

```
$cObj = new tslib_cObj;
```

Internally "t3lib_div::makeInstance()" does this:

```
$cObj = class_exists('ux_tslib_cObj') ? new ux_tslib_cObj : new tslib_cObj;
```

IMPORTANT

When setting up the file to include, in particular from t3lib/, notice the difference between $TYPO3_CONF_VARS["BE"]["XCLASS"][...] and $TYPO3_CONF_VARS["FE"]["XCLASS"][...]. The key "FE" is used when the class is included by a front-end script (those initialized by tslib/index_ts.php and tslib/showpic.php - both also known as index.php and showpic.php in the root of the website), "BE" is used by backend scripts (those initialized by typo3/init.php or typo3/thumbs.php). This feature allows you to include a different extension when the (t3lib/-) class is used in the frontend and in the backend.

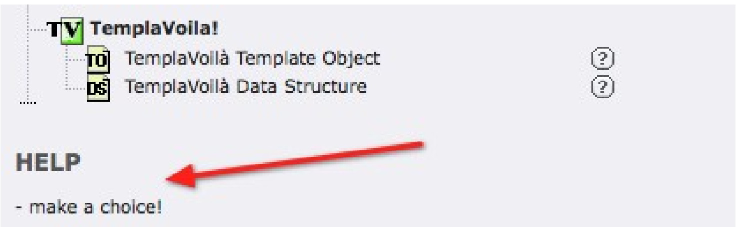
Which classes?

Most code in TYPO3 resides in classes and therefore anything in the system can be extended. So you should rather say to yourself: In which script (and thereby which class) is it that I'm going to extend/change something. When you know which script, you simply open it, look inside and somewhere you'll find the lines of code which are responsible for the inclusion of the extension, typically in the bottom of the script.

The exceptions to this rule is classes like "t3lib_div", "t3lib_extMgm" or "t3lib_BEfunc". These classes contain methods which are designed to be call non-instantiated, like "t3lib_div::fixed_lgd_cs()". Whether a class works on this basis is normally noted in the header of the class file. When methods in a class is called non-instantiated there is no way you can extend that method/class.

Example - Adding a small feature in the interface

Say you wish to add a little section with help text in the bottom of the "New" dialog:



So this is what you do:

1. Find out that the script in question is "typo3/db_new.php" (right-click frame, select "Properties" and look at URL...:-)
2. Then examine the scripts for its classes and methods. In this case you'll find two classes in the file; "localPageTree" (extends t3lib_pageTree) and "SC_db_new". The class "SC_db_new" is the so called "Script Class" - this will hold the code specifically for this script.
   You also find that the only code executed in the global scope is this:

   ```
   $SOBE = t3lib_div::makeInstance('SC_db_new');
   $SOBE->init();
   $SOBE->main();
   $SOBE->printContent();
   ```

3. When you examine the SC_db_new class you find that the main() method is the one you would like to extend.
4. Finally you find that immediately after the definition of the two classes there is three lines of code which will provide you with the final piece of knowledge you need:

   ```
   // Include extension?
   ```

```
if (defined('TYPO3_MODE') && $TYPO3_CONF_VARS[TYPO3_MODE]['XCLASS']['typo3/db_new.php'])    {
    include_once($TYPO3_CONF_VARS[TYPO3_MODE]['XCLASS']['typo3/db_new.php']);
}
```

So now you know that the key to use is "typo3/db_new.php" when you wish to define a script which should be included as the extension.

Let's see what happens then in the extension "examples":

1. First we have a class that extends the SC_db_new (xclasses/class.tx_examples_scdbnew.php):

```
function regularNew() {
    parent::regularNew();
    $this->code .= $this->doc->section($GLOBALS['LANG']->sL('LLL:EXT:examples/locallang.xml:help'),
$GLOBALS['LANG']->sL('LLL:EXT:examples/locallang.xml:make_choice'), 0, 1);
}
```

2. The XCLASS is then registered in the extension's ext_localconf.php file:

```
$TYPO3_CONF_VARS['BE']['XCLASS']['typo3/db_new.php'] = t3lib_extMgm::extPath($_EXTKEY,
'xclasses/class.tx_examples_scdbnew.php');
```

There is no "table of extendable classes" in this document because 1) all classes are extendable and 2) the number of classes will grow as TYPO3 is further developed and extensions are made and 3) finally you cannot extend a class unless you know it exists and have analyzed some of its internal structure (methods / variables) - so you'll have to dig into the source anyway!

Therefore, if you wish to extend something, follow this suggestion for an analysis of the situation and you'll end up with the knowledge needed in order to extend that class and thereby extend TYPO3 *without* loosing backwards compatibility with future updates. Great.

Notes on SC_* classes (script classes)

There is one more thing to note about especially the SC_* classes in the backend:

1. **Global vars:** They use a lot of variables from the global scope. This is due to historical reasons; The code formerly resided in the global scope and a quick conversion into classes demanded this approach. Future policy is to keep as many variables internal as possible and if any of these SC_* classes are developed further in the future, some of the globals might on that occasion be internalized.

2. **Large methods:** There are typically a init(), main() and printContent() method in the SC-classes. Each of these, in particular the main() method may grow large. Processing stuff in the start and end of the methods is easy - you just call parent::[methodname]() from your extension. But if you want to extend or process something in the middle of one of these methods, it would be necessary to call a dummy method at that point in the parent class. Such a dummy method would then be used for processing in *your* class, but would not affect the general use of the parent class. Such dummy-method calls are not widely included yet, but will be as suggestions for them appears. And you are very welcome to give in such suggestions.

   I'll just give an example to illustrate what I mean:

```
class SC_example {
    function main() {
        $number = 100;
        echo 'The number is ' . $number;
    }
}
```

   This class prints the text "The number is 100". If you wish to do some calculations to the $number-variable before it is printed, you are forced to simply include the whole original main-method in your extension script. Here it would be no problem because the method is 2 lines long. But it could be 200 lines! So what you do is that you suggest to the TYPO3 development to call a "harmless" dummy method in the main() method...

```
class SC_example {
    function main() {
        $number = 100;
        $number = $this->processNumber($number);
        echo 'The number is ' . $number;
    }
    function processNumber($theNumber) {
        return $theNumber;
    }
}
```

   ... and then you extend the class as follows:

```
class ux_SC_example extends SC_example {
    function processNumber($theNumber)    {
        return ($theNumber < 100) ? 'less than 100' : 'greater than 100';
    }
}
```

... and now the main() method would print "The number is greater than 100" instead.

Notice that you'll have to make such suggestions for dummy method calls because we will include them only as people need them.

Extending methods

When extending a method like in the case above with stdWrap() please observe that such a method might gain new parameters in the future without further notice. For instance stdWrap is currently defined like this:

```
function stdWrap($content, $conf) {
```

... but maybe some day this method will have another parameter added, eg:

```
function stdWrap($content, $conf, $yet_a_parameter=0) {
```

This means if you want to override stdWrap(), but still call the parent class' method, you must extend your own method call from...:

```
function stdWrap($content, $conf) {
        // Call the real stdWrap method in the parent class:
    $content = parent::stdWrap($content, $conf);
         ...
```

... to:

```
function stdWrap($content, $conf, $yet_a_parameter=0) {
        // Call the real stdWrap method in the parent class:
    $content = parent::stdWrap($content, $conf,$ yet_a_parameter);
  ...
```

Also be aware of constructors. If you have a constructor in your extension class you must observe if there is a constructor in the parent class which you should call first / after. In case, you can do it by "parent::[original class name]"

For instance the class tslib_fe is instantiated into the global object $TSFE. This class has a constructor looking like this:

```
/**
 * Class constructor
 */
function tslib_fe($TYPO3_CONF_VARS, $id, $type, $no_cache='', $cHash='', $jumpurl='') {
        // Setting some variables:
    $this->id = $id;
    $this->type = $type;
    $this->no_cache = $no_cache ? 1 : 0;
    $this->cHash = $cHash;
    $this->jumpurl = $jumpurl;
    $this->TYPO3_CONF_VARS = $TYPO3_CONF_VARS;
    $this->clientInfo = t3lib_div::clientInfo();
    $this->uniqueString=md5(microtime());
    $this->makeCacheHash();
}
```

So as you see, you probably want to call this method. But let's also say you wish to make sure the $no_cache parameter is always set to 1 (for some strange reason...). So you make an extension class like this with a new constructor, ux_tslib_fe(), overriding the $no_cache variable and then calling the parent class constructor:

```
class ux_tslib_fe extends tslib_fe {
 function ux_tslib_fe($TYPO3_CONF_VARS, $id, $type, $no_cache='', $cHash='', $jumpurl='') {
     $no_cache=1;
     parent::tslib_fe($TYPO3_CONF_VARS, $id, $type, $no_cache, $cHash, $jumpurl);
 }
}
```

User defined methods in classes

Prefix user defined methods and internal variables with "ux_"! Thus you don't risk to choose a method name which may later be added to the parent class in the TYPO3 distribution!

Example, continued from above:

```
class ux_tslib_fe extends tslib_fe {
    var $ux_fLPmode = 1;     // If you "feelLuckyPunk" this is the no_cache value

  function ux_tslib_fe($TYPO3_CONF_VARS, $id, $type, $no_cache='', $cHash='', $jumpurl='') {
            // setting no_cache?
        $no_cache=$this->ux_settingNoCache();
            // Calling parent constructor:
        parent::tslib_fe($TYPO3_CONF_VARS, $id, $type, $no_cache, $cHash, $jumpurl);
  }
  /**
   * Setting the no_cache value based on user-input in GET/POST var, feelLuckyPunk
   */
  function ux_settingNoCache() {
      return t3lib_div::GPvar('feelLuckyPunk') ? $this->ux_fLPmode : 0;
  }
}
```

(User defined methods and variables are in purple)

A few examples of extending the backend classes

The concept of extending classes in the backend can come in handy in many cases. First of all it's a brilliant way to make your own project specific extensions to TYPO3 without spoiling the compatibility with the distribution! This is a very important point! Stated another way: By making an "XCLASS extension" you can change one method in a TYPO3 class and next time you update TYPO3, your method is still there - but all the other TYPO3 code has been updated! Great!

Also for development and experimental situations is great. Generally the concept offers you quite a lot of freedom, because you are seriously able to take action if you need something solved here and now which cannot be fixed in the main distribution at the moment.

Anyway, here's a few simple examples:

1) Say you wish to have the backend user time out after 30 seconds instead of the default 6000.

1. In your extension's (named "test") ext_localconf.php fiel, insert:
   $TYPO3_CONF_VARS['BE']['XCLASS']['t3lib/class.t3lib_beuserauth.php']
   = t3lib_extMgm::extPath('test') . 'class.ux_myBackendUserExtension.php';

2. Create the file "class.ux_myBackendUserExtension.php" in your extension's folder and put this content in:

```
<?php

class ux_t3lib_beUserAuth extends t3lib_beUserAuth {
    var $auth_timeout_field = 30;
}
?>
```

Of course you need to know why it's the variable `auth_timeout_field` which must be set, but you are a bright person, so of course you go directly to the file t3lib/class.t3lib_beuserauth.php, open it and find that `var $auth_timeout_field = 6000;` there!

You could also easily insert an IP-filter (which is already present though...). Here you have to take a little adventure a bit further. As you see in "class.t3lib_beuserauth.php" extends "t3lib_userAuthGroup" which extends "t3lib_userAuth" the method start() is the place where the users are authenticated. This could quickly be exploited to make this IP filter for the backend:

```
<?php

class ux_t3lib_beUserAuth extends t3lib_beUserAuth {
    var $auth_timeout_field = 30;

    function start() {
        if (!t3lib_div::cmpIP(getenv('REMOTE_ADDR'), '192.168.*.*'))     {
            die('Wrong IP, you cannot be authenticated!');
        } else {
            return parent::start();
        }
    }
}
?>
```

So now only users with client IP numbers in the 192.168.*.* series will gain access to the backend. If that is the case, notice how the parent start() method is called and any result is returned. Thus your overriding method is a wrapped for the original. Brilliant, right!

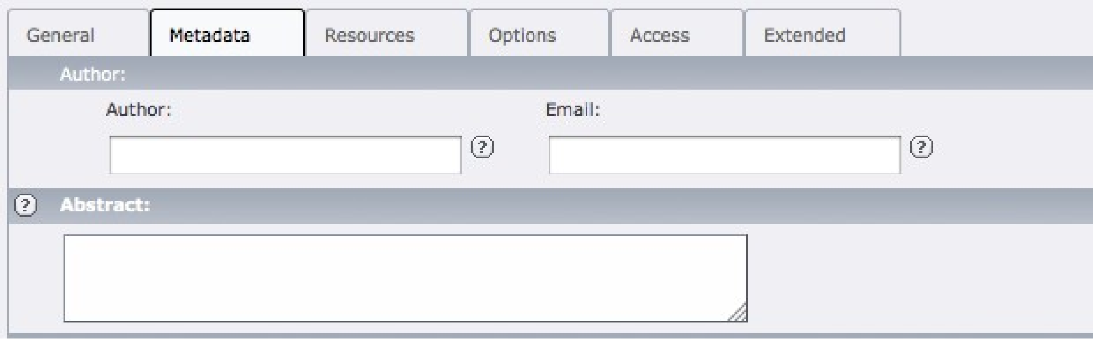
2) Here's another one (from the "examples" extension, file "xclasses/class.tx_examples_tceforms.php"):

```
function formWidth($size = 48, $textarea = FALSE) {
      $size = round($size * 1.5);
      return parent::formWidth($size, $textarea);
}

function printPalette($palArr) {
            // Change all field labels in the palette to uppercase
      foreach ($palArr as $key => $palette) {
            $palArr[$key]['NAME'] = strtoupper($palArr[$key]['NAME']);
      }
      return parent::printPalette($palArr);
}
```

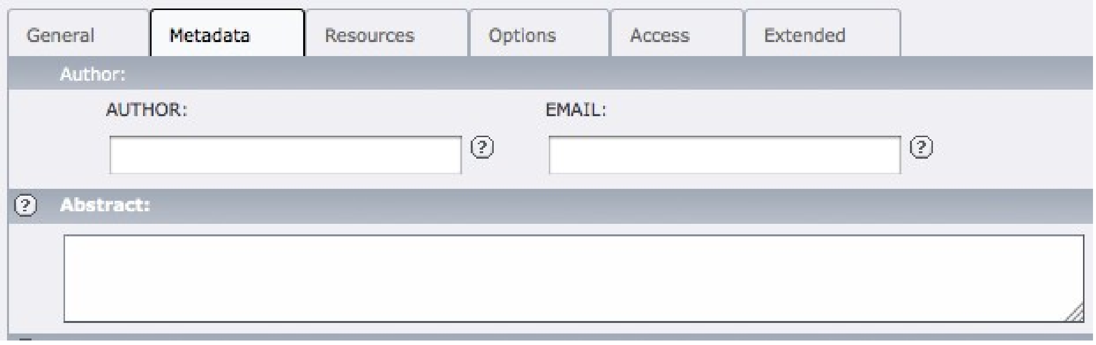
... and configured in ext_localconf.php as this:

```
$TYPO3_CONF_VARS['BE']['XCLASS']['t3lib/class.t3lib_tceforms.php'] = t3lib_extMgm::extPath($_EXTKEY,
'xclasses/class.tx_examples_tceforms.php');
```

What is the result? A typical part of the backend form for a page will look like this:



But the extension does two things: 1) textarea form fields have their width multiplied with 1.5 so they are wider (other field types are unaffected due to different rendering mechanisms), 2) the titles of the palette-fields are converted for uppercase. The result looks like this:



So as you see you can do really stupid details - in fact almost any extension.

Warnings

There are a few warnings about using XCLASS extensions:

- **Avoid using XCLASS extensions in your (public) extensions!**
  A PHP class can only be extended by *one* extension class at a time. Thus, having two extension classes set up, only the latter one will be enabled. There is no way to work around this technologically in PHP. However "t3lib_div::makeInstance()" supports "cascaded" extension classes, meaning that you can do "ux_ux_someclass" which will extend "ux_someclass" but this requires an internal awareness of the extension class "ux_someclass" in the first place. The conclusion is that XCLASS extensions are best suited for project development where you need a quick hack of something in the core which should still stay backwards compatible with TYPO3 core upgrades.
- **Check if child classes are instantiated**
  Quite often people have been confused about extending for instance the "tslib_menu" class when they want to add a feature for "TMENU". But actually the class to extend is "tslib_tmenu" which is an extension of "tslib_menu". So make sure you are extending the *right* class name (and always make sure your extension class is included also).
- **Strange opcode caching behaviors when you upgrade TYPO3 core**
  When you upgrade the TYPO3 core and you have an extension which extends a core class, the upgraded core underneath might not be detected by opcode caches. In particular PHP-Accelerator is known for this behavior producing "undefined function...." errors. The solution is: Always clear "/tmp/php_a_*" files and restart your web server after upgrading source.

AJAX in the TYPO3 Backend

In TYPO3 4.2 a new model for writing AJAX code in the TYPO3 Backend was introduced. Although there were some parts in the TYPO3 Backend that used AJAX already, they are now unified into a single interface that handles errors and dispatches the different calls to their final locations. This way it is ensured that e.g. a BE user is logged in and all TYPO3 variables are loaded.

The whole architecture builds on top of successful techniques developers already know. It's a mixture between the eID concept from the TYPO3 Frontend, the hooking idea we know from other places in TYPO3, piped through a single small AJAX file "typo3/ajax.php" that creates a PHP AJAX object to see if an error occurred or not. If something went wrong, the X-JSON header is set to false and the client-side AJAX request (an AJAX responder in the Prototype Javascript framework) will know that there is an error.

In-depth presentation

Client-Side programming

On the client-side we are using the Prototype JS library (located in typo3/contrib/prototype/prototype.js). If you have used it already, you know that you can make AJAX calls with AJAX.Request, AJAX.Updater and AJAX.PeriodicalUpdater. We extended the library and hooked in these objects, or better: in the callbacks users can define. If an AJAX request is made to our server-side component (typo3/ajax.php), everything developers need to do is to call this URL and add a unique, already registered parameter for their ajaxID. Their defined "onComplete" and "onSuccess" are only rendered if the X-JSON header is set to true by the server-side script. If the X-JSON header is set to false, the Responder checks if there is a callback function named "onT3Error" and executes it instead of the "onComplete" method. If the "onT3Error" method is not defined, the default TYPO3 error handler will be displaying the error in the TYPO3 backend. If the X-JSON header is set to false, the "onSuccess" callback will not be executed as well as but an error message will be shown in the notification area. This behaviour is done automatically with every AJAX call to "ajax.php" made through Prototype's AJAX classes. This responder is also only active if "typo3/js/common.js" is added to the base script.

Server-side programming

If you look into "typo3/ajax.php", it is only a small dispatcher script. It checks for an ajaxID in the TYPO3_CONF_VARS['BE']['AJAX'] array and tries to execute the function pointer. The function has two parameters, where the first (an array) is not used yet. The second parameter is the TYPO3 AJAX Object (located in typo3/classes/typo3ajax.php) that is used to add the content that should be returned as the server-response to the Javascript part, or the error message that should be displayed. The X-JSON header will be set depending on whether "setError()" was called on this AJAX object. You can also specify if the object should return the result in a valid XML object tree, as text/html (default) or as a JSON object, see below.

The "ajaxID" is a unique identifier and can be used to override the existing AJAX calls. Therefore you can extend existing AJAX calls that already exist in the backend by redirecting it to your function. But be aware of the side-effects of this feature: Other extensions could overwrite this function as well (similar problem as with XCLASSing or single inheritance in OOP).

Also, for every TYPO3 request, you will now have a TYPO3_REQUESTTYPE variable that can be used for bitwise comparison. You can now check if you're in Backend or Frontend or in an valid AJAX request with

```
if (TYPO3_REQUESTTYPE & TYPO3_REQUESTTYPE_AJAX)
```

to see if you're calling through the new AJAX interface.

Different Content Formats

As with every AJAX response you can send it in different response formats.

- text/html - plain text
- text/xml - strict XML formatting
- application/json - JSON notation

You can also specify the contentFormat in the AJAX object like this:

```
$ajaxObj->setContentFormat('json');
```

For the keyword you can choose between "plain" (default), "xml" and "json", "jsonbody" and "jsonhead".

Here are the specifics for each format.

Plain Text

The content array in the backend will be concatenated and returned uninterpreted.

The result will be available in the transport object as a string through "xhr.responseText".

XML

The content needs to be valid XML and will be available in javascript as "xhr.responseXML".

JSON

The content is put through the service in "typo3/contrib/json/json.php" and is then available in JSON notation through the second parameter in the onComplete / onSuccess methods, and additionally in the "responseText" part of the transport object ("xhr.responseText"). If it is set to "jsonbody", only the latter variable is filled, if "jsonhead" is set, it is only in the second parameter. This is useful to save traffic and you can use it with whatever format you like.

Developing with AJAX in the TYPO3 Backend

This is a small guide for you to use this feature in your AJAX scripts.

How to choose the right ajaxID

The ajaxID consists of two parts, the class name and the action name, delimited by "::" ("<class>::<action>").

Please note that the ajaxID is just a system-wide identifier to register your AJAX handler. Even if it looks like a static function call, it won't be executed and has no technical dependencies. But it is required for all developers to use a common naming scheme as described above, since it might prevent from using identical names in different extensions.

Some good examples for an ajaxID:

- SC_alt_db_navframe::expandCollapse
- t3lib_TCEforms_inline::processAjaxRequest
- tx_myext_module1::executeSomething

Some bad examples for an ajaxID:

- search::findRecordByTitle
- core::reloadReferences
- inline::processAjaxRequest
- updateRecordList

Server-Side

1) Register your unique ajaxID, at best prepended with your extension ID, in the TYPO3 backend by adding the following line to your "ext_localconf.php" (this way you can also overwrite existing AJAX calls):

```
$TYPO3_CONF_VARS['BE']['AJAX']['tx_myext::ajaxID'] = 'filename:object->method';
```

A working example would be:

```
$TYPO3_CONF_VARS['BE']['AJAX']['SC_alt_db_navframe::expandCollapse'] =
'typo3/alt_db_navframe.php:SC_alt_db_navframe->ajaxExpandCollapse';
```

2) Create a target method to do the logic on the server-side: similarly to the existing hooking mechanism, as with every "callUserFunction", the target method (here: ajaxExpandCollapse) will have two function parameters. The first one is an array of params (not used right now), the second is the TYPO3 AJAX Object (see typo3/classes/class.typo3ajax.php for all available methods). You should use this to set the content you want to output or to write the error message if something went wrong.

```
public function ajaxExpandCollapse($params, &$ajaxObj) {
    $this->init();
    // do the logic...
    if (empty($this->tree)) {
        $ajaxObj->setError('An error occurred');
    } else  {
        // the content is an array that can be set through $key / $value pairs as parameter
        $ajaxObj->addContent('tree', 'The tree works...');
    }
}
```

So, the server-side part is now complete. Let's move over to the client-side.

Client-part

3) In order for you to use client-side AJAX Javascript you have to add these two lines of PHP code to your PHP script (available in template.php, your template document):

```
$this->doc->loadJavascriptLib('contrib/prototype/prototype.js');
$this->doc->loadJavascriptLib('js/common.js');
```

4) With prototype on the client side, it's quite simple to add a new AJAX request in Javascript:

```
new Ajax.Request('ajax.php', {
    method: 'get',
    parameters: 'ajaxID=SC_alt_db_navframe::expandCollapse',
    onComplete: function(xhr, json) {
        // display results, should be "The tree works"
    }.bind(this),
    onT3Error: function(xhr, json) {
        // display error message, will be "An error occurred" if an error occurred
    }.bind(this)
});
```

You can see, that it's almost the same, except that we introduce a new callback function "onT3Error", which is optional. It is called if the method "setError" on the server-side was called. Otherwise "onComplete" will be called. If you do not define "onT3Error", then the error message will be shown in the TYPO3 notification area in the backend. Our AJAX responders also work with the "onSuccess" callback, but the onT3Error is only executed with "onComplete".

This should be all. Please note that our TYPO3 BE AJAX mechanism works best with the prototype JS library. If you want to create similar approaches for other JS frameworks, have a look at "typo3/js/common.js".

Using third-party JavaScript libraries in the TYPO3 Backend

Since TYPO3 4.3, it is very easy to use third-party JavaScript libraries in the TYPO3 BE. A simple API is provided by template class (which all modules use). Third-party JS libraries are those which are available in the “typo3/contrib” folder. Note that use of Prototype/Scriptaculous is now discouraged. Use ExtJS instead. Prototype/Scriptaculous support may be removed in future versions of TYPO3.

Inside a module, the instance of template.php is normally available as

```
$this->doc
```

Using Prototype

To load the Prototype library use this syntax:

```
$this->doc->loadPrototype();
```

That's all, prototype will be added to the head of the document, any double inclusion will be prevented.

Using Scriptaculous

To load the Scriptaculous library use this syntax:

```
$this->doc->loadScriptaculous();
```

To load modules, just pass some module's name to the above call. Possible modules are: “builder”, “effects”, “dragdrop”, “controls”, “slider”.

Example:

```
$this->doc->loadScriptaculous('slider');
$this->doc->loadScriptaculous('effects,dragdrop');
```

If a module depends on other modules, those will automatically be added. The calling order doesn't matter. Proper load order is taken care of by the API.

To load all available modules, use the keyword “all”:

```
$this->doc->loadScriptaculous('all');
```

When loading Scriptaculous, Prototype is automatically added.

Using ExtJS

To load the ExtJS library use this syntax:

```
$this->doc->loadExtJS();
```

There are 2 optional parameters in this call:

```
$this->doc->loadExtJS($css = TRUE, $theme = TRUE);
```

- The first parameter is a boolean. If set to true, “ext-all.css” is added automatically.
- The second parameter is also a boolean. If set to true, theme-grey is added automatically.

Additionally the function takes care of:

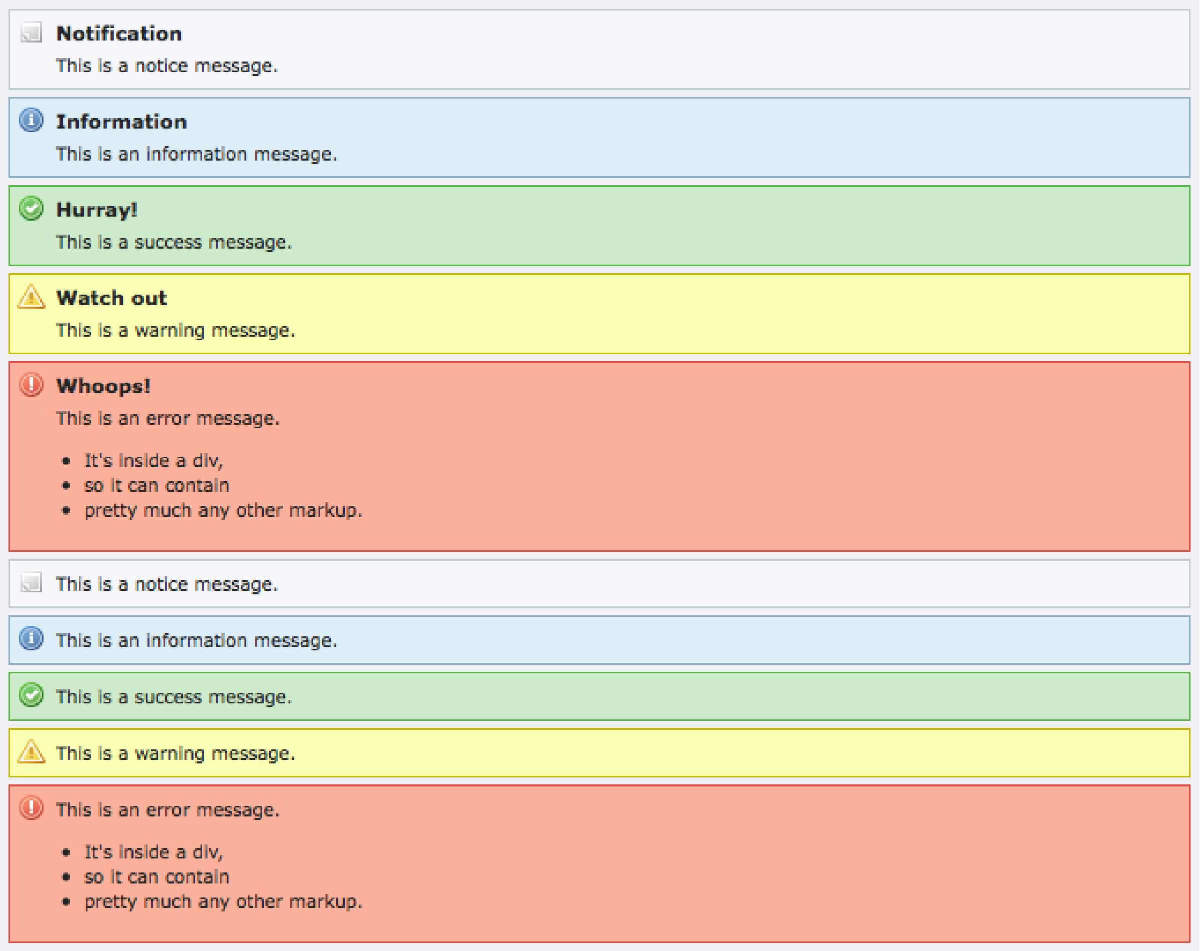
- adding the correct adapter
- adding the localization file of BE-User language
- adding Ext.BLANK_IMAGE_URL

To do debugging in the ExtJS library, use the following call to force the debug variant to be loaded:

```
$this->doc->setExtJSdebug();
```

Flash messages

Since TYPO3 4.3 there is a generic system to show users that an action was performed successfully, or more importantly, failed. This system is known as “flash messages”. The screenshot below shows the various severity levels of messages that can be emitted. It also shows that flash messages can include a header or not.



Notification
This is a notice message.

Information
This is an information message.

Hurray!
This is a success message.

Watch out
This is a warning message.

Whoops!
This is an error message.

- It's inside a div,
- so it can contain
- pretty much any other markup.

This is a notice message.

This is an information message.

This is a success message.

This is a warning message.

This is an error message.

- It's inside a div,
- so it can contain
- pretty much any other markup.

The different severity levels are described below:

- Notifications are used to show very low severity information. Such information usually is so unimportant that it can be left out, unless running in some kind of debug mode.
- Information messages are to give the user some information that might be good to know.

- OK messages are to signal a user about a successfully executed action
- Warning messages show a user that some action might be dangerous, cause trouble or might have partially failed.
- Error messages are to signal failed actions, security issues, errors and the like.

Flash messages API

Creating a flash message is achieved by simply instantiating an object of class t3lib_FlashMessage:

```
$message = t3lib_div::makeInstance('t3lib_FlashMessage',
      'My message text',
      'Message Header', // the header is optional
      t3lib_FlashMessage::WARNING, // the severity is optional as well and defaults to
t3lib_FlashMessage::OK
      TRUE // optional, whether the message should be stored in the session or only in the
t3lib_MessageQueue object (default is FALSE)
);
```

The severity is defined by using class constants provided by t3lib_FlashMessage:

- t3lib_FlashMessage::NOTICE for notifications
- t3lib_FlashMessage::INFO for information messages
- t3lib_FlashMessage::OK for success messages
- t3lib_FlashMessage::WARNING for warnings
- t3lib_FlashMessage::ERROR for errors

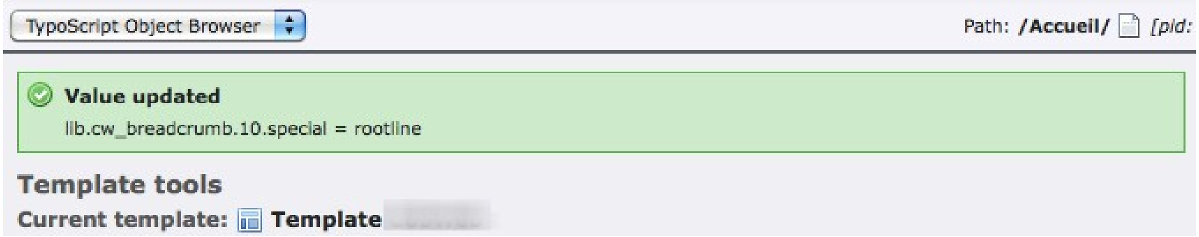
The fourth parameter passed to the constructor is a flag that indicates whether the message should be stored in session or not (the default is not). Storage in session should be used if you need the message to be still present after a redirection.

In backend modules you can then make that message appear on top of the module after a page refresh / the rendering of the next page request or render it on your own where ever you want.

This example adds the flash message at the top of modules when rendering the next request:

```
t3lib_FlashMessageQueue::addMessage($message);
```

The message is added to the queue and then the template class calls t3lib_FlashMessageQueue::renderFlashMessages() which renders all messages from the queue. Here's how such a message looks like in a module:



Value updated
lib.cw_breadcrumb.10.special = rootline

Template tools
Current template: **Template**

By default flash messages are shown atop the content of a module. However, if needed, you can change where the messages are shown by manipulating a module's template and inserting the ###FLASHMESSAGES### marker. Messages will then replace that marker instead of appearing at the top of the module.

It is also possible to render a single message directly, instead of adding it to the queue. This makes it possible to display flash messages absolutely anywhere. Here's how this is achieved:

```
$message->render();
```

Various examples

Introduction

The following pages present some examples of how you can use the APIs of core libraries. Remember, ultimately the source is the documentation and the only point here is to show examples. Whenever you would like to use core features that are not shown here you should search in the core and system extensions for implementations that can work as an example for you.

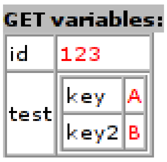
Debugging with debug()

A very common tool used by TYPO3 developers is the debug() function. It basically prints out the content of a variable in a nicely formatted table. There are extensions available which extends the view from the debug() function to something more fancy. Here I will just present the basic version.

Use the debug() function whenever you want to look "inside" an array or parameters passed to a user processing function. Usually it makes it very easy to understand the parameters. For instance, lets say you call a script with the GET parameter string "?id=123&test[key]=A&test[key2]=B". How will the GET vars look to your application inside? Well, using the debug function makes that easy:

```
debug(t3lib_div::_GET(),'GET variables:');
```

The output in the browser will look like:



GET variables:

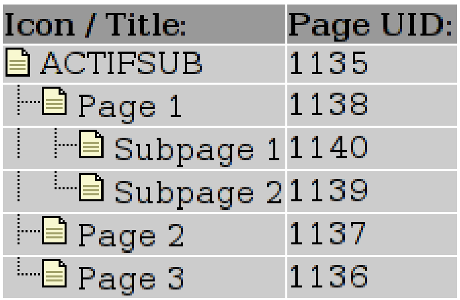
id	123
test	key A key2 B

Notice that the debug() function is a wrapper for t3lib_div::debug() and the difference is that debug() (defined in "t3lib/config_default.php") will only output information if your IP address is within a certain range typical for internal networks.

Rendering page trees

In your backend modules you might like to show information or perform processing for a part of the page tree. There is a whole family of libraries in the core for making trees from records, static page trees or page trees that can be browsed (open/close nodes).

In this simple example I will show how to get the HTML for a static page tree, using the class "t3lib_pageTree" (child of "t3lib_treeView"). The output will look like this (missing the normal TYPO3 styles though):



Icon / Title:	Page UID:
ACTIFSUB	1135
Page 1	1138
Subpage 1	1140
Subpage 2	1139
Page 2	1137
Page 3	1136

The PHP code that generates this looks like:

```
require_once(PATH_t3lib . 'class.t3lib_pagetree.php');

    // Initialize starting point of page tree:
$treeStartingPoint = 1135;
$treeStartingRecord = t3lib_BEfunc::getRecord('pages', $treeStartingPoint);
$depth = 2;

    // Initialize tree object:
$tree = t3lib_div::makeInstance('t3lib_pageTree');
$tree->init('AND ' . $GLOBALS['BE_USER']->getPagePermsClause(1));

    // Creating top icon; the current page
$HTML = t3lib_iconWorks::getIconImage('pages', $treeStartingRecord, $GLOBALS['BACK_PATH'],
'align="top"');
$tree->tree[] = array(
    'row' => $treeStartingRecord,
    'HTML'=>$HTML
);

    // Create the tree from starting point:
$tree->getTree($treeStartingPoint, $depth, '');
#debug($tree->tree);

    // Put together the tree HTML:
$output = '
    <tr  bgcolor="#999999">
        <td><b>Icon / Title:</b></td>
```

```
27:         <td><b>Page UID:</b></td>
28:     </tr>';
29: foreach($tree->tree as $data)    {
30:     $output.='
31:         <tr bgcolor="#cccccc">
32:             <td nowrap="nowrap">' . $data['HTML'] . htmlspecialchars($data['row']['title']) .
'</td>
33:             <td>' . htmlspecialchars($data['row']['uid']) . '</td>
34:         </tr>';
35: }
36:
37: $output = '<table border="0" cellspacing="1" cellpadding="0">' . $output . '</table>';
```

- In line 1 the class is included.
  Notice how the constant "PATH_t3lib" is used to set the path for "t3lib/".
- Line 4-5 sets up the starting point. You need a page id for that and additionally you must select that page record.
  Notice how another important API function, t3lib_BEfunc::getRecord(), is used to get the record array for the page!
- Line 6 defines that the page tree will go 2 levels down from the starting point.
- Line 9-10 initializes the class.
  Notice how the BE_USER object is called to get an SQL where clause that will ensure that only pages that are accessible for the user will be shown in the tree!
  Notice how t3lib_div::makeInstance() is used to create the object. This is required by the TYPO3 CGL.
- Line 13-17 sets up the starting point page in the tree. This must be done externally if you would like your tree to include the root page (which is not always the case).
  Notice how line 13 calls the function t3lib_iconWorks::getIconImage() to get the correct icon image for the pages table record! Also, $GLOBALS['BACK_PATH'] is used to make sure the icon has a correct "back-path" to the location where the icon is on the server.
- Line 20 renders the page tree from the starting point and $depth levels down (at least 1 level)
- The rendered page tree is stored in a data array inside of the tree object. We need to traverse the tree data to create the tree in HTML. This gives us the chance to organize the tree in a table for instance. That is very useful if you need to show additional information for each page.
  - Lines 24-28 renders a table row with headings for the tree.
  - Lines 29-35 traverses the tree data and for each element a table row will be rendered with the icon/title and an additional cell containing the uid.
  - Line 37 wraps the table rows in a table tag.

Local extensions of the page tree classes

If you search in the source for other places where this class is used you will often find that the class is extended locally in those scripts. This is because it is possible to override certain functions that generate for instance the icon or wraps the title in some way.

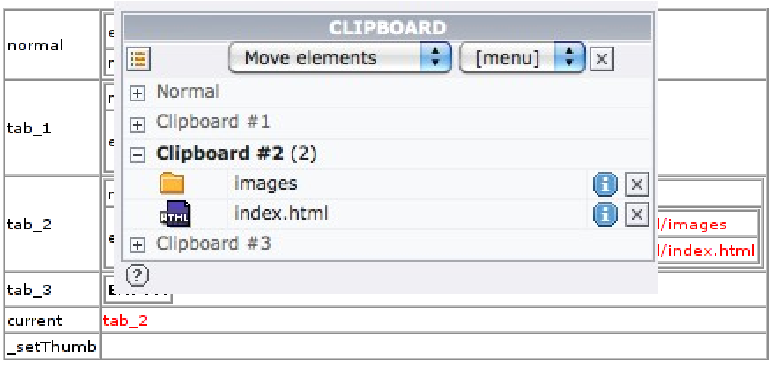
Accessing the clipboard

You can easily access the internal clipboard in TYPO3 from your backend modules.

```
 1: require_once(PATH_t3lib . 'class.t3lib_clipboard.php');
 2:
 3:     // Clipboard is initialized:
 4: $clipObj = t3lib_div::makeInstance('t3lib_clipboard');        // Start clipboard
 5: $clipObj->initializeClipboard();    // Initialize - reads the clipboard content from the user
session
 6: debug($clipObj->clipData);
```

- Line 1 includes the clipboard library
- Line 4-5 initializes it.
- Line 6 outputs the content of the internal variables, ->clipData. That will look like what you see below:

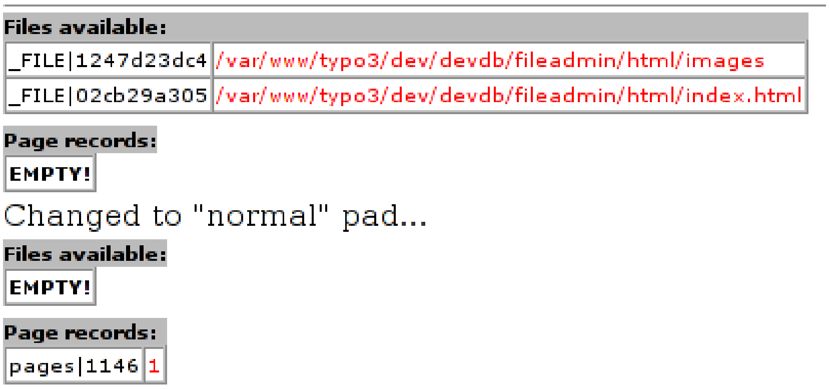
This tells us what objects are registered on the "normal" tab (page record with id 1146 in "copy" mode) and the numeric tabs (can contain more than one element). The current clipboard (Pad 2 active) looks like this:



The correct way of accessing clipboard content is to the method, elFromTable(), in the clipboard object.

```
    debug($clipObj->elFromTable('_FILE'), 'Files available:');
    debug($clipObj->elFromTable('pages'), 'Page records:');
    $clipObj->setCurrentPad('normal');
    echo 'Changed to "normal" pad...';
    debug($clipObj->elFromTable('_FILE'), 'Files available:');
    debug($clipObj->elFromTable('pages'), 'Page records:');
```

Here we first try to get all files and then all page records on the current pad (which is pad 2). Then we change to the "Normal" pad, call the elFromTable() method again and output the results. The output shows that in the first attempt we get the list of files but no page records while in the second attempt after having changed to the normal pad we will get no files but the page record on the normal pad in return:



```
Files available:
_FILE|1247d23dc4 /var/www/typo3/dev/devdb/fileadmin/html/images
_FILE|02cb29a305 /var/www/typo3/dev/devdb/fileadmin/html/index.html

Page records:
EMPTY!
Changed to "normal" pad...
Files available:
EMPTY!

Page records:
pages|1146 1
```

Setting elements on the clipboard

This is too complicated to describe in detail. The following codelisting is from the Web > List module where selections for the clipboard is posted from a form and registered.

```
    // Clipboard actions are handled:
$CB = t3lib_div::_GET('CB');     // CB is the clipboard command array
if ($this->cmd=='setCB') {
        // CBH is all the fields selected for the clipboard, CBC is the checkbox fields which were
checked. By merging we get a full array of checked/unchecked elements
        // This is set to the 'el' array of the CB after being parsed so only the table in question is
registered.
    $CB['el'] = $dblist->clipObj->cleanUpCBC(array_merge(t3lib_div::_POST('CBH'),
t3lib_div::_POST('CBC')), $this->cmd_table);
}
if (!$this->MOD_SETTINGS['clipBoard']) {
        $CB['setP'] = 'normal';    // If the clipboard is NOT shown, set the pad to 'normal'.
}
$dblist->clipObj->setCmd($CB);         // Execute commands.
$dblist->clipObj->cleanCurrent();     // Clean up pad
$dblist->clipObj->endClipboard();     // Save the clipboard content
```

Adding Context Sensitive Menu items

When the CSM is being generated in the "alt_clickmenu.php" script an array with the elements is created. Before the array is passed over to the final rendering function that will create the menu HTML, the array will be passed in turns to external processing scripts. These scripts are configured in this global array:

```
$GLOBALS['TBE_MODULES_EXT']['xMOD_alt_clickmenu']['extendCMclasses'];
```

Each script will then have a chance to manipulate the content of the array and add/remove items as the script wants. This is what makes it possible to add custom options to CSM.

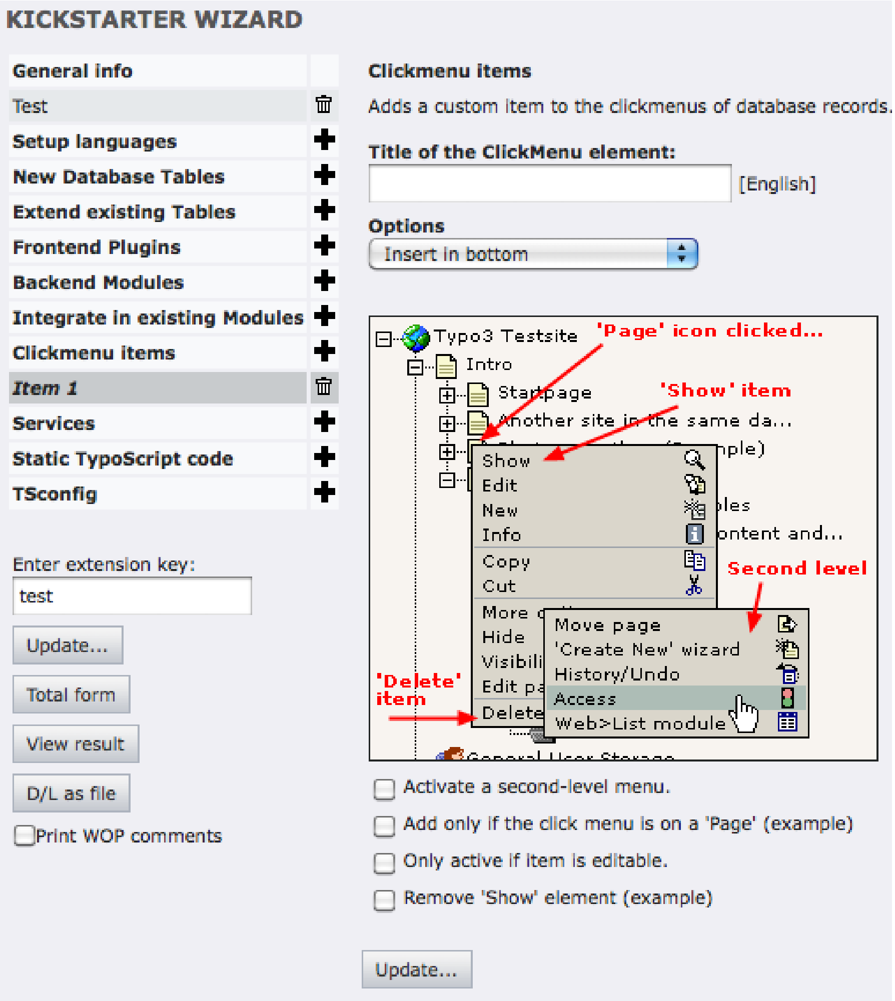
The extensions "extra_page_cm_options" adds a lot of CSM options. The extension has an "ext_tables.php" file and it contains code that adds an entry in the array mentioned above:

```
<?php
if (!defined ('TYPO3_MODE'))     die ('Access denied.');
if (TYPO3_MODE=='BE') {
    $GLOBALS['TBE_MODULES_EXT']['xMOD_alt_clickmenu']['extendCMclasses'][] = array(
        'name' => 'tx_extrapagecmoptions',
        'path' => t3lib_extMgm::extPath($_EXTKEY) . 'class.tx_extrapagecmoptions.php'
    );
}
?>
```

The value of the "path" key is pointed to the absolute path of the class file that contains code for manipulation of the CSM array. This file must contain a class by the name of "name" and inside that class a "main()" method that will be called for manipulation. The basic skeleton looks like this:

```
/**
 * Class, adding extra context menu options
 *
 * @author    Kasper Skaarhoj <kasper@typo3.com>
 * @package TYPO3
 * @subpackage tx_extrapagecmoptions
 */
class tx_extrapagecmoptions {
    /**
     * Adding various standard options to the context menu.
     * This includes both first and second level.
     *
     * @param    object        The calling object. Value by reference.
     * @param    array        Array with the currently collected menu items to show.
     * @param    string        Table name of clicked item.
     * @param    integer        UID of clicked item.
     * @return    array        Modified $menuItems array
     */
    function main(&$backRef, $menuItems, $table, $uid) {
        global $BE_USER,$TCA,$LANG;
        $localItems = array();    // Accumulation of local items.
        ...
            $menuItems = array_merge($menuItems, $localItems);
            return $menuItems;
        }
    }
}
```

The "extra_page_cm_options" is a slightly special since it produces additional CSM elements by calls back to the parent object where rendering functions exists. This is due to historical reasons. Better examples of handcrafted menu items can be found in extensions such as "templavoila" (1st level additions for specific table) and "impexp" (2nd level addition). Finally, the best way to initiate adding elements is using the Kickstarter Wizard which contains an options for creating CSMs:



Enter extension key: test

☐ Print WOP comments

Clickmenu items

Adds a custom item to the clickmenus of database records.

Title of the ClickMenu element: [English]

Options: Insert in bottom

☐ Activate a second-level menu.

☐ Add only if the click menu is on a 'Page' (example)

☐ Only active if item is editable.

☐ Remove 'Show' element (example)

Implementing Context Sensitive Menus

If you want to implement a CSM for an element in your own backend modules you have to do two things:

- Include standard JavaScript and HTML code in the HTML document for all CSM instances.
- Wrap the icon / element title with a link that opens the CSM.

The standard JavaScript and HTML can be fetched from the backend document template object. In a typical backend module environment this object is available as $this->doc and these four lines will do the trick:

```
1:          // Setting up the context sensitive menu:
2:        $CMparts = $this->doc->getContextMenuCode();
3:        $this->doc->bodyTagAdditions = $CMparts[1];
4:        $this->doc->JScode .= $CMparts[0];
5:        $this->doc->postCode.= $CMparts[2];
```

These lines must be executed *before* calling "$this->doc->startPage()".

- Line 2 asks the template object to generate the standard content. It is returned in an array.
- Line 3 adds event handlers for the <body> tag:

  ```
  onmousemove="GL_getMouse(event);" onload="initLayer();"
  ```

- Line 4 adds JavaScript functions in the <head> of the HTML output
- Line 5 adds the <div> layers in the bottom of the page:

  ```
  <div id="contentMenu0" style="z-index:1; position:absolute;visibility:hidden"></div>
  <div id="contentMenu1" style="z-index:2; position:absolute;visibility:hidden"></div>
  ```

CSM for database elements

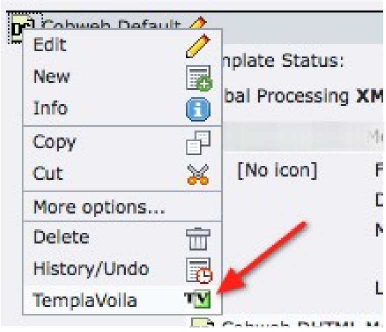
Linking icons to open the CSM is easy:

```
    // Get icon with CSM:
$icon = t3lib_iconworks::getIconImage('tx_templavoila_datastructure', $row, $GLOBALS['BACK_PATH'],
'align="top"');
$content .= $this->doc->wrapClickMenuOnIcon($icon, 'tx_templavoila_datastructure', $row['uid'], 1);
```

In this example the first line creates an <img> tag with the icon of a record from the table "tx_templavoila_datastructure". The variable $row must be the record array of an element from this database table.

The second line wraps the icon ($icon) in a link that will open the CSM over it. This is done by calling "template::wrapClickMenuOnIcon()" with $icon HTML, table name and element uid. The fourth argument is a boolean you should set if your script is shown in the list frame of the backend. This will tell "alt_clickmenu.php" which generates the HTML content that it should be written back to the list frame and not the navigation frame for instance.

Result:



CSM for files

Activating a CSM for a file is also easy. As for database elements it requires that the standard content is added to the HTML document. From that point you just call the same function, "template::wrapClickMenuOnIcon()" but set the second argument to the absolute path of the file (and keep the third argument, the uid, blank).

```
$GLOBALS['SOBE']->doc->wrapClickMenuOnIcon($theIcon,$path);
```

Notice, that in this case the document template object used is the global variable $SOBE which is normally available in backend modules as well. You might also use the default instance found in $TBE_TEMPLATE.

For more information see the inline documentation of the function wrapClickMenuOnIcon(). It is found in the file "template.php" in the typo3/ folder.

Parsing HTML: t3lib_parsehtml

This class is very handy for various processing needs of HTML. In the future it might be obsolete if the "tidy" extension becomes standard in PHP but for now there are no native features in PHP which lets us parse HTML.

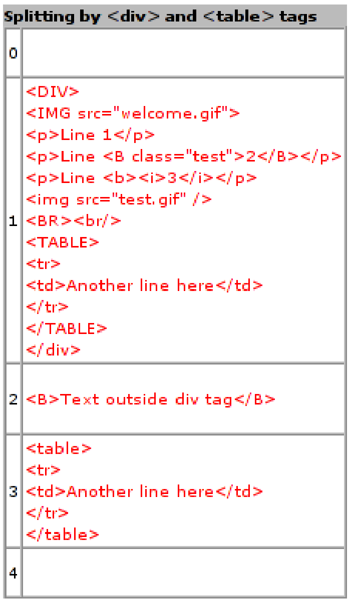
Extracting blocks from an HTML document

In the first example it is shown how we can extract parts of an HTML document.

```
require_once(PATH_t3lib . 'class.t3lib_parsehtml.php');

$testHTML = '
    <DIV>
        <IMG src="welcome.gif">
        <p>Line 1</p>
        <p>Line <B class="test">2</B></p>
        <p>Line <b><i>3</i></p>
        <img src="test.gif" />
        <BR><br/>
        <TABLE>
            <tr>
                <td>Another line here</td>
            </tr>
        </TABLE>
    </div>
    <B>Text outside div tag</B>
    <table>
```

```
19:         <tr>
20:             <td>Another line here</td>
21:         </tr>
22:     </table>
23: ';
24:
25:     // Splitting HTML into blocks defined by <div> and <table> tags
26: $parseObj = t3lib_div::makeInstance('t3lib_parsehtml');
27: $result = $parseObj->splitIntoBlock('div,table', $testHTML);
28: debug($result, 'Splitting by <div> and <table> tags');
29:
```

- Line 1 includes the library.
- Line 3-23 loads the HTML sample code into a variable.
- Line 36 creates an instance of the parser-class.
  Notice how t3lib_div::makeInstance() is used (required).
- Line 27 splits the HTML content into an array dividing it by <div> or <table> tags.
- Line 28 outputs the result array with the debug() function:



```
Splitting by <div> and <table> tags
0
1 <DIV>
  <IMG src="welcome.gif">
  <p>Line 1</p>
  <p>Line <B class="test">2</B></p>
  <p>Line <b><i>3</i></p>
  <img src="test.gif" />
  <BR><br/>
  <TABLE>
  <tr>
  <td>Another line here</td>
  </tr>
  </TABLE>
  </div>
2 <B>Text outside div tag</B>
3 <table>
  <tr>
  <td>Another line here</td>
  </tr>
  </table>
4
```

As you can see the HTML source has been divided so the <div> section and the <table> is found in key 1 and 3. The keys of the extracted content is always the odd keys while the even keys are the "outside" content.

Notice that the table *inside* of the <div> section was not "found". So when you split content like this you get only elements on the same block-level in the source. You have to traverse the content recursively to find all tables - or just split on <table> only (which will not give you tables nested inside of tables though).

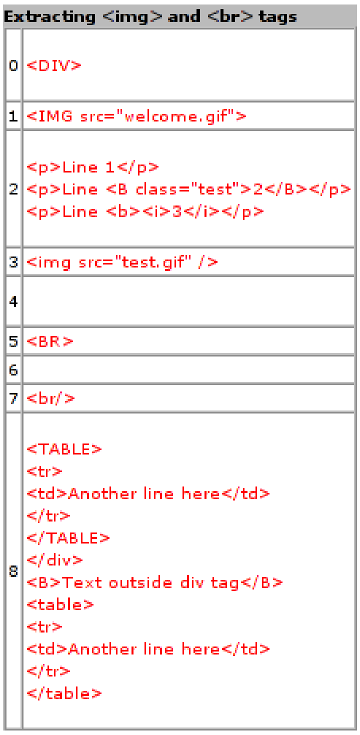
Extracting single tags

You can split the content by tag as well. This is done in the next example. Here all <img> and <br> tags are found:

```
30:     // Splitting HTML into blocks defined by <img> and <br> tags
31: $result = $parseObj->splitTags('img,br', $testHTML);
32: debug($result,'Extracting <img> and <br> tags');
33:
```

Line 31 performs the splitting operation. This is the output:

Again, all the odd keys in the array contains the tags that were found. If you wanted to do processing on this content you just traverse the array, process all odd keys and implode the array again. A code listing for that might look like this:



Extracting <img> and <br> tags

0	<DIV>
1	<IMG src="welcome.gif">
2	<p>Line 1</p> <p>Line <B class="test">2</B></p> <p>Line <b><i>3</i></p>
3	<img src="test.gif" />
4	
5	<BR>
6	
7	<br/>
8	<TABLE> <tr> <td>Another line here</td> </tr> </TABLE> </div> <B>Text outside div tag</B> <table> <tr> <td>Another line here</td> </tr> </table>

```
foreach($result as $intKey => $HTMLvalue)    {
        // Find all ODD keys:
    if ($intKey%2)    {
        $result[$intKey] = '--'.$result[$intKey].'--';
    }
}
$newContent = implode('',$result);
```

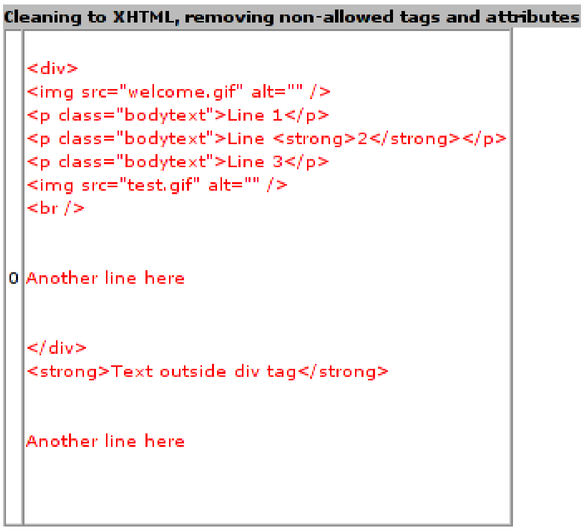
Cleaning HTML content

You can also do processing on the HTML content by the HTMLcleaner() method. This code listings shows a basic example of how you can configure it. There are a lot of features hidden in the $tagCfg array and you should refer to the inline documentation of the method in the class.

```
34:     // Cleaning HTML:
35: $tagCfg = array_flip(explode(',', 'b,img,div,br,p'));
36: $tagCfg['b'] = array(
37:     'nesting' => 1,
38:     'remap' => 'strong',
39:     'allowedAttribs' => 0
40: );
41: $tagCfg['p'] = array(
42:     'fixAttrib' => array(
43:         'class' => array(
44:             'set' => 'bodytext'
45:         )
46:     )
47: );
48: $result = $parseObj->HTMLcleaner($testHTML, $tagCfg, FALSE, FALSE, array('xhtml' => 1));
49: debug(array($result), 'Cleaning to XHTML, removing non-allowed tags and attributes');
```

- Line 35 initializes the $tagCfg array by setting the five allowed tags as keys. Only these tag names are allowed! All others are removed (HTMLcleaner() can be configured to keep all unknown tags though).
- Line 36-40 configures additional options for the "b" tag. First of all correct nesting is required. This means that the single <b> tag in one of the paragraphs will be removed. Then the "remap" key is set which means that all occurencies of <b> tags will be substituted with <strong> tags instead. Finally the allowed attributes are set to false which means that any attributes set for <b> tags are removed.
- Line 41-47 configures additional options for the "p" tag. In this case it just hardcodes that the attribute "class" must exist and it must have the value "bodytext".
- Line 48 calls the HTMLcleaner() method - and notice the extra options being set where "xhtml" cleaning is enabled. This will convert all tag an attribute names to lowercase and "close" tags like <img> and <br> to <img.../> and <br />

This is the output:



Cleaning to XHTML, removing non-allowed tags and attributes

```
  <div>
  <img src="welcome.gif" alt="" />
  <p class="bodytext">Line 1</p>
  <p class="bodytext">Line <strong>2</strong></p>
  <p class="bodytext">Line 3</p>
  <img src="test.gif" alt="" />
  <br />


0 Another line here


  </div>
  <strong>Text outside div tag</strong>


  Another line here
```

Advanced call back processing

This code listing shows how you can register call back functions for recursive processing of an HTML source:

```
class user_processing {
    function process($str) {
        $this->parseObj = t3lib_div::makeInstance('t3lib_parsehtml_proc');

        $outStr = $this->parseObj->splitIntoBlockRecursiveProc(
            'div|table|blockquote|caption|tr|td|th|h1|h2|h3|h4|h5|h6|ol|ul',
            $str,
            $this,
            'callBackContent',
            'callBackTags'
        );

        return $outStr;
    }

    function callBackContent($str, $level) {
        if (trim($str)) {

                // Fixing <P>
            $pSections = $this->parseObj->splitTags('p', $str);
            foreach($pSections as $k => $v)    {
                $pSections[$k] = trim(ereg_replace('[[:space:]]+', ' ', $pSections[$k]));
                if (!($k%2)) {

                    if ($k && !strstr(strtolower($pSections[$k]), '</p>')) {
                        $pSections[$k] = trim($pSections[$k]) . '</p>';
                    }

                    $pSections[$k].=chr(10);
                }
            }
            $str = implode('',$pSections);
        }

        if (trim($str)) {
            $str = $this->parseObj->indentLines(trim($str),$level) . chr(10);
        } else {
            $str = trim($str);
        }

        return $str;
    }

    function callBackTags($tags,$level) {

        if (substr($tags['tag_name'],0,1) == 'h') {
            $tags['tag_end'] .= chr(10);
            $tags['content'] = trim($tags['content']);
                // Removing the <hx> tags if they content nothing when tags are stripped:
            if (!strlen(trim(strip_tags($tags['content'])))) {
                $tags['tag_start'] = $tags['tag_end'] = '';
                $tags['add_level'] = 0;
                $tags['content'] = '';
                return $tags;
            }
        } elseif ($tags['tag_name'] == 'div' || $tags['tag_name'] == 'blockquote') {
            $tags['tag_start'] = $tags['tag_end'] = '';
            $tags['add_level'] = 0;
        } else {
            $tags['tag_start'] = $this->parseObj->indentLines(trim($tags['tag_start']),$level) .
chr(10);
            $tags['tag_end'] = $this->parseObj->indentLines(trim($tags['tag_end']),$level) .
chr(10);
        }
        return $tags;
    }
}
```

In the method "process()" processing is started. Like when splitting HTML content you define a list of tags to split by. Each of these will be processed by the call back functions "callBackContent" and "callBackTags" for processing of both the content between the splitted tags and the tags themselves.

Notice how it is all within the same class which is a requirement for the call back functions.

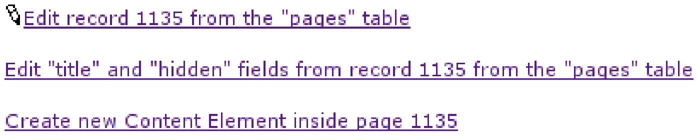
I'll not explain this listing in further detail. Explore it yourself if you are interested in call back processing of HTML sources.

Links to edit records

Quite often in your backend modules you might like to create a link to edit a record. This is easily done with an API function call to t3lib_BEfunc::editOnClick(). This script will create an onclick-JavaScript event linking you to the "alt_doc.php" script in the "PATH_typo3" directory.

All you need to do is prepare GET parameters for the "alt_doc.php" script. Please look inside of "alt_doc.php" for more details of possible GET vars you can use and what they mean. In this example I have shown the most typical options.

The result of the code listing will be three links like these:



Edit record 1135 from the "pages" table

Edit "title" and "hidden" fields from record 1135 from the "pages" table

Create new Content Element inside page 1135

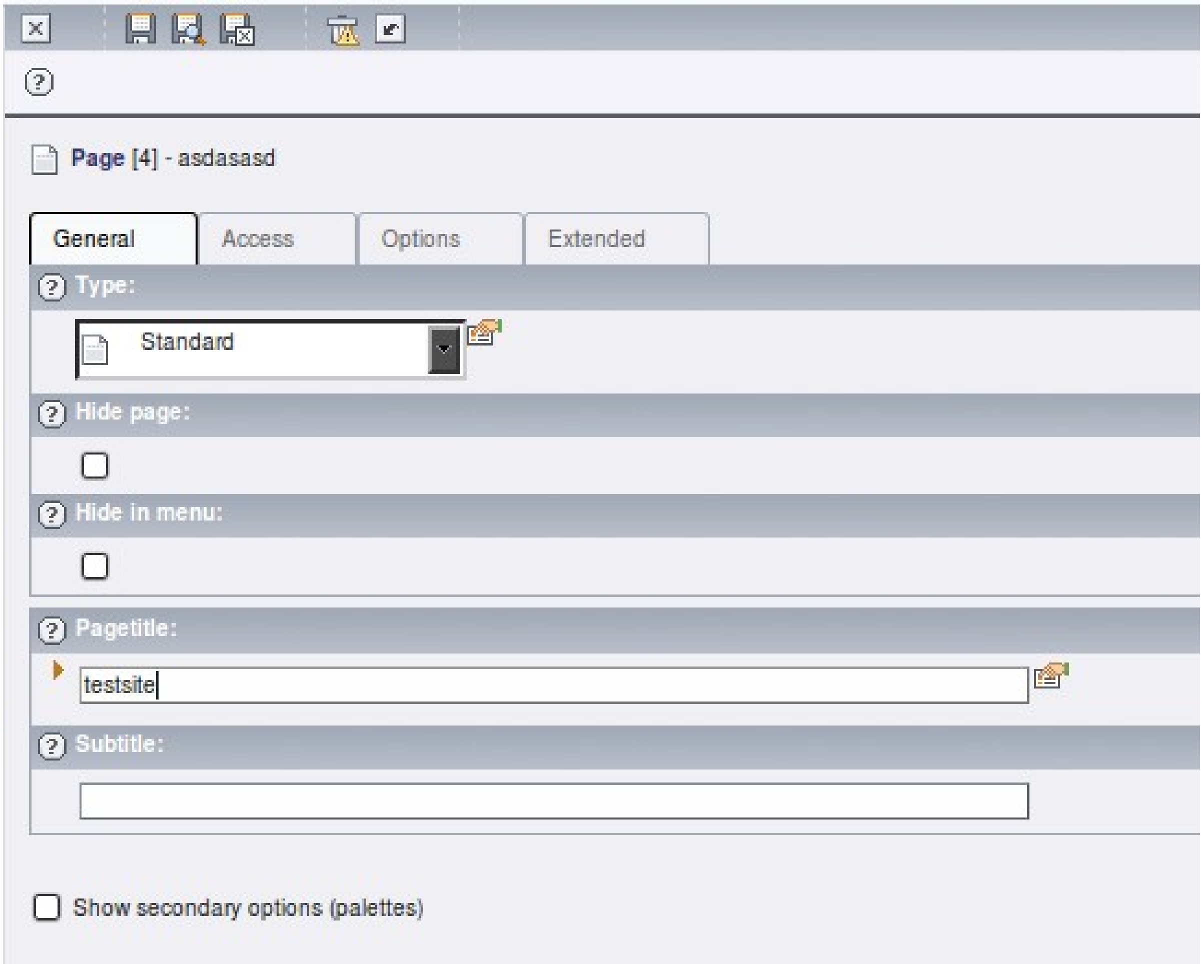
The code listing looks like this:

```
 1: $editUid = 1135;
 2: $editTable = 'pages';
 3:
 4:     // Edit whole record:
 5: $params = '&edit[' . $editTable . '][' . $editUid . ']=edit';
 6: $output.= '<a href="#" onclick="' . htmlspecialchars(t3lib_BEfunc::editOnClick($params,
$GLOBALS['BACK_PATH'])) . '">' .
 7:         '<img'.t3lib_iconWorks::skinImg($GLOBALS['BACK_PATH'], 'gfx/edit2.gif', 'width="11"
height="12"') . ' title="Edit me" border="0" alt="" />'.
 8:         'Edit record ' . $editUid . ' from the "' . $editTable . '" table' .
 9:         '</a><br/><br/>';
10:
11:     // Edit only "title" and "hidden" fields from record:
12: $params = '&edit[' . $editTable . '][' . $editUid.']=edit&columnsOnly=title,hidden';
13: $output .= '<a href="#" onclick="' . htmlspecialchars(t3lib_BEfunc::editOnClick($params,
$GLOBALS['BACK_PATH'])) . '">'.
14:         'Edit "title" and "hidden" fields from record ' . $editUid . ' from the "' . $editTable .
'" table' .
15:         '</a><br/><br/>';
16:
17:     // Create new "Content Element" record in PID 1135
18: $params = '&edit[tt_content][' . $editUid . ']=new&defVals[tt_content][header]=New%20Element';
19: $output .= '<a href="#" onclick="' . htmlspecialchars(t3lib_BEfunc::editOnClick($params,
$GLOBALS['BACK_PATH'])) . '">' .
20:         'Create new Content Element inside page ' . $editUid.
21:         '</a><br/>';
```

Editing a record

In line 5 you see the basic GET parameter you need to set up to edit a record. You need to know the database table name, record uid in advance. The syntax is "&edit[*tablename*][*uid*]=edit". You can specify as many tables and uids you like and you will get them all in one single form! The "uid" variable can even be a comma list of uids (short way of editing more records from the same table at once).

The lines 5-9 produces a link which shows this form:



Page [4] - asdasasd

Type: Standard

Hide page: ☐

Hide in menu: ☐

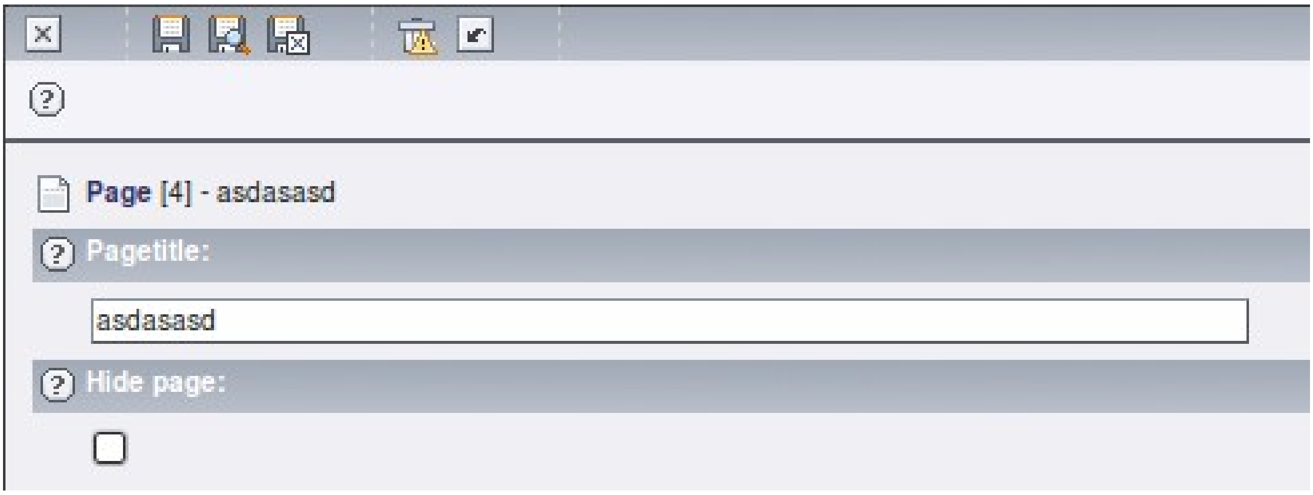
Pagetitle: testsite

Subtitle:

☐ Show secondary options (palettes)

Editing only a few fields from a record

Lines 11-15 creates the same link but with additional information that only the field names "title" and "hidden" should be edited! That is done by adding the GET parameters "&columnsOnly=title,hidden". This means the form will look like this:



Page [4] - asdasasd

Pagetitle: asdasasd

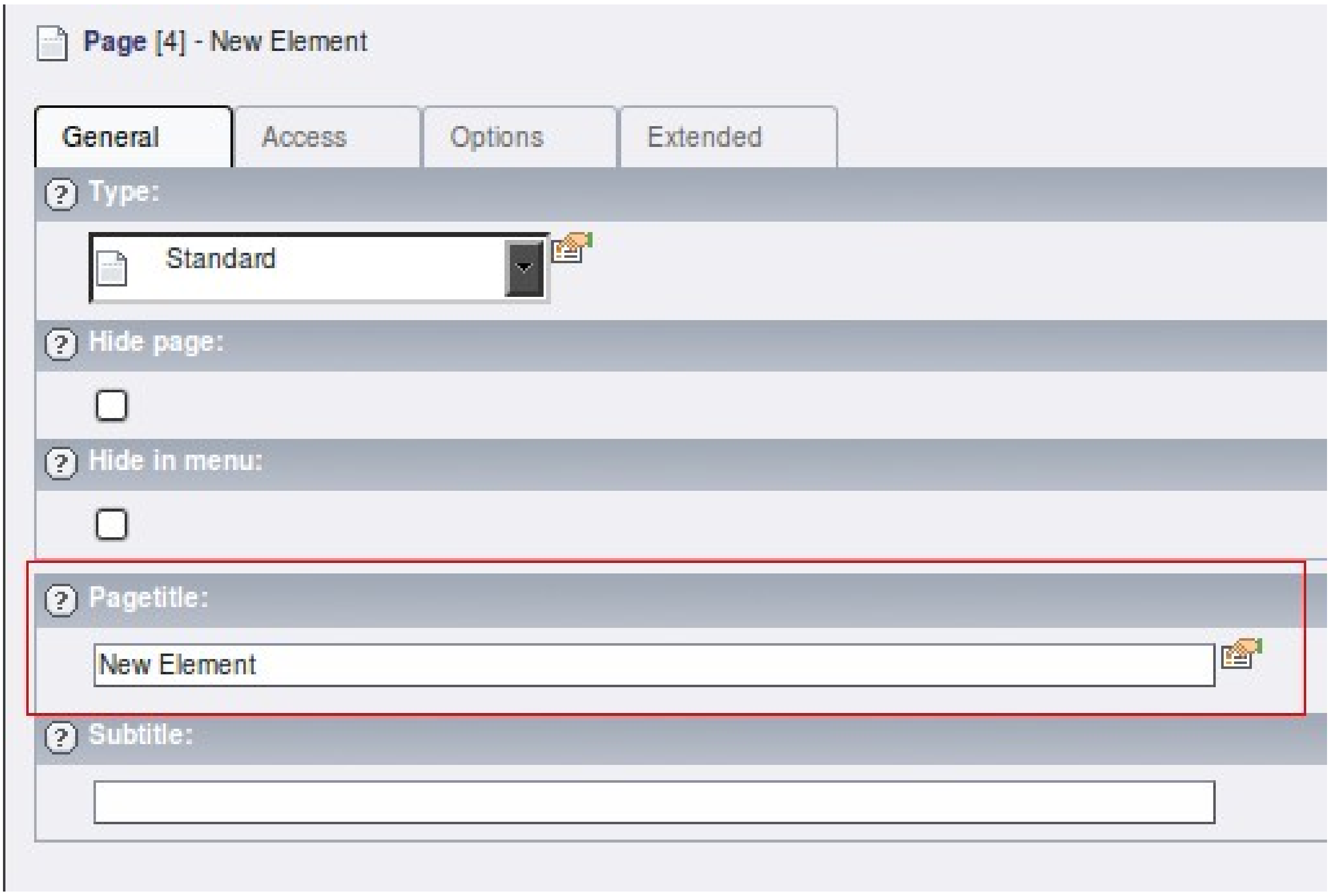
Hide page: ☐

Creating a form for new elements

Lines 17-21 creates a link which will make a new content element inside the page with "pid" 1135. The syntax for creating new records is "&edit[*table_name_of_new_record*][*pid_reference*]=new". The pid reference is special: If it is a negative value it points to another record from the same table *after which* the new record should be created. If it is positive or zero it just points to the page id where the record should be created (as the top element).

Another feature is that a custom default value for the header field is automatically passed along. This is done by the additional GET parameter "&defVals[tt_content][header]=New%20Element" and you can see how the Header field is pre-filled with this value below.

The result of the "create new" will be this form.



Page [4] - New Element

Type: Standard

Hide page: ☐

Hide in menu: ☐

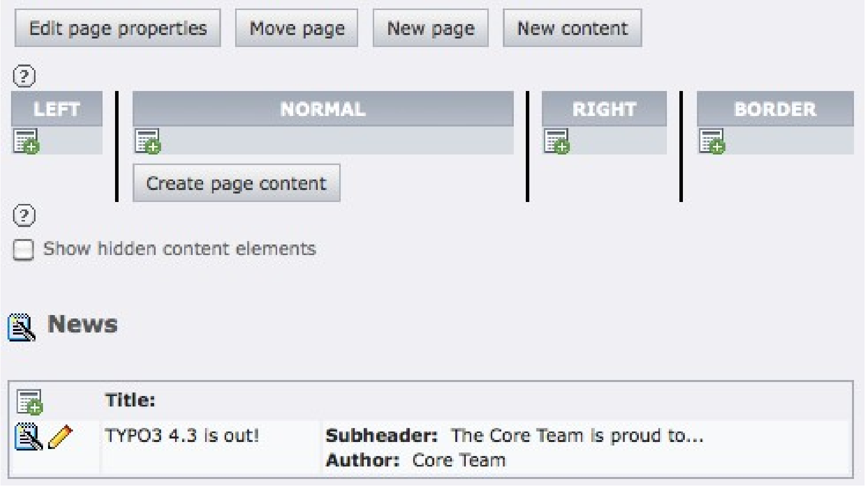
Pagetitle: New Element

Subtitle:

Support for custom tables in the Page module

In the Web > Page module you can have listings of other records than Content Elements and guest book items. If you want your custom table to be listed there you can configure it using the $TYPO3_CONF_VARS["EXTCONF"]['cms'] array. This is a configuration option offered from within the Page module.

In this example the tt_news extension is configured for listing in the Page module. It would look like this:



☐ Show hidden content elements

News

Title:	
TYPO3 4.3 is out!	**Subheader:** The Core Team is proud to... **Author:** Core Team

The configuration required is as simple as this, put into (ext_)localconf.php:

```
$TYPO3_CONF_VARS['EXTCONF']['cms']['db_layout']['addTables']['tt_news'][0] = array(
    'fList' => 'title,short;author',
    'icon' => TRUE
);
```

The "fList" key value is a list of field names separated first by comma and then ";" (semi-colon). The comma separates table columns while the semi-colon allows you to list more than one field to be displayed inside a single column.

Adding elements to the Content Element Wizard

Since TYPO3 4.3 the new content element wizard can be fully configured using TSConfig. The example below describes the base method for adding a plugin to the list of plugins in the wizard.

Adding elements under the “Plugins” header

If you want to add elements in the wizard under the plugins header there is native support in the script for this.

Basically, what you do is to set content in the global variable $TBE_MODULES_EXT['xMOD_db_new_content_el']['addElClasses']. The keys in this array must be class names and the values is the absolute path of the class. When the script is run the class files will be included during initialization. Then, during the building of the array of wizard elements the default wizard array is passed to the class you have configured through the method proc() in your class.

For details the most easy thing will be to look into the script in the function wizardArray() - this will make it clear to you how it works.

Example

As an example of how this works from an extension you can take a look at the extension tt_guest. This extension adds itself in the plugin category by inserting these lines in its ext_tables.php file:

```
if (TYPO3_MODE=='BE')    {
    $TBE_MODULES_EXT['xMOD_db_new_content_el']['addElClasses']['tx_ttguest_wizicon'] =
        t3lib_extMgm::extPath($_EXTKEY) . 'class.tx_ttguest_wizicon.php';
}
```

In the file class.tx_ttguest_wizicon.php you will find a class looking like this:

```
/**
 * Class, containing function for adding an element to the content element wizard.
 *
 * @author    Kasper Skaarhoj <kasper@typo3.com>
 * @package TYPO3
 * @subpackage tx_ttguest
 */
class tx_ttguest_wizicon {

    /**
     * Processing the wizard-item array from db_new_content_el.php
     *
     * @param    array        Wizard item array
     * @return    array        Wizard item array, processed (adding a plugin for tt_guest extension)
     */
    function proc($wizardItems) {
        global $LANG;

            // Include the locallang information.
        $LL = $this->includeLocalLang();

            // Adding the item:
        $wizardItems['plugins_ttguest'] = array(
            'icon' => t3lib_extMgm::extRelPath('tt_guest') . 'guestbook.gif',
            'title' => $LANG->getLLL('plugins_title', $LL),
            'description' => $LANG->getLLL('plugins_description', $LL),
            'params' => '&defVals[tt_content][CType]=list&defVals[tt_content]
[list_type]=3&defVals[tt_content][select_key]=' . rawurlencode('GUESTBOOK, POSTFORM')
        );

        return $wizardItems;
    }

    /**
     * Include locallang file for the tt_guest book extension (containing the description and title for
the element)
     *
     * @return    array        LOCAL_LANG array
     */
    function includeLocalLang()    {
        include(t3lib_extMgm::extPath('tt_guest') . 'locallang.xml');
        return $LOCAL_LANG;
    }
}
```

As you can see this class modifies the wizard array with an additional item. This is how you can also add / modify elements in the array using this API.

Using custom permission options

TYPO3 (3.7.0+) offers extension developers to register their own permission options to be automatically managed by TYPO3s user group access lists. The options can be grouped in categories. A custom permission option is always a checkbox (on/off). The scope of such options is for use in the backend of TYPO3 only.

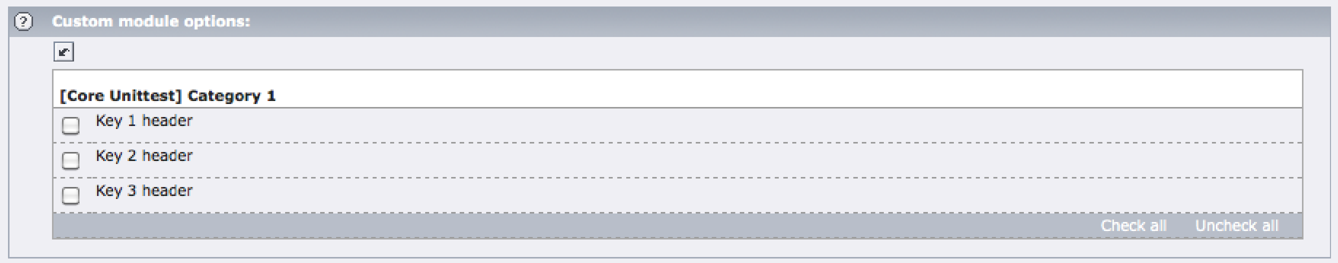
Registering a header and options

You configure options in the global variable $TYPO3_CONF_VARS['BE']['customPermOptions']. You can read the comment inside "config_default.php" regarding the syntax of the array.

This example shows how three options are registered under a new category:

```
$TYPO3_CONF_VARS['BE']['customPermOptions'] = array(
            'tx_coreunittest_cat1' => array(
                'header' => '[Core Unittest] Category 1',
                'items' => array(
                    'key1' => array('Key 1 header'),
                    'key2' => array('Key 2 header'),
                    'key3' => array('Key 3 header'),
                )
            )
        );
```

The result is that these options appear in the group access lists like this:



[Core Unittest] Category 1

☐ Key 1 header

☐ Key 2 header

☐ Key 3 header

You can also add icons, a description and use references to locallang values. Such a detailed configuration could look like this (also just an example):

```
...
'tx_coreunittest_cat2' => array(
    'header' => 'LLL:EXT:coreunittest/locallang_test.php:test_header',
    'items' => array(
        'keyA' => array('Key a header', 'icon_ok.gif', 'This is a description....'),
        'keyB' => array('LLL:EXT:coreunittest/locallang_test.php:test_item',
'../typo3/gfx/icon_ok2.gif', 'LLL:EXT:coreunittest/locallang_test.php:test_description'),
        'key3' => array('Key 3 header', 'EXT:coreunittest/ext_icon.gif'),
    )
)
...
```

Evaluating the options

Checking if a custom permission option is set you simply call this API function in the user object:

```
$BE_USER->check('custom_options', $catKey . ':' . $itemKey);
```

$catKey is the category in which the option resides. From the example above this would be "tx_coreunittest_cat1"

$itemKey is the key of the item in the category you are evaluating. From the example above this could be "key1", "key2" or "key3" depending on which one of them you want to evaluated.

The function returns true if the option is set, otherwise false.

Keys in the array

It is good practice to use the extension keys prefixed with "tx_" on the first level of the array. This will help to make sure you do not pick a key which someone else picked as well!

Also you should never pick a key containing any of the characters ",:|" since they are reserved delimiter characters.

Table Control Array, $TCA

The reference to the $TCA (Table Control Array) was extracted from here and moved to its own manual, called "TCA Reference" (doc_core_tca). Please refere to http://typo3.org/documentation/document-library/core-documentation/doc_core_tca/current/.

Page types

$PAGES_TYPES

$PAGES_TYPES defines the various types of pages (field: doktype) the system can handle and what restrictions may apply to them. Here you can set the icon and especially you can define which tables are allowed on a certain page type (doktype).

NOTE: The "default" entry in the $PAGES_TYPES-array is the "base" for all types, and for every type the entries simply overrides the entries in the "default" type!!

This is the default array as set in t3lib/stddb/tables.php:

```
$PAGES_TYPES = array(
      '254' => array(        //  Doktype 254 is a 'sysFolder' - a general purpose storage folder
            'type' => 'sys',
            'icon' => 'sysf.gif',
            'allowedTables' => '*'
      ),
      '255' => array(        // Doktype 255 is a recycle-bin.
            'type' => 'sys',
            'icon' => 'recycler.gif',
            'allowedTables' => '*'
      ),
      'default' => array(
            'type' => 'web',
            'icon' => 'pages.gif',
            'allowedTables' => 'pages',
            'onlyAllowedTables' => '0'
      )
);
```

Each array has the following options available:

Key	Description
type	Can be "sys" or "web"
icon	Alternative icon. The file reference is on the same format "iconfile" in [ctrl] section of TCA
allowedTables	The tables that may reside on pages with that "doktype". Comma-separated list of tables allowed on this page doktype. "*" = all
onlyAllowedTables	Boolean. If set, the tce_main class will not allow a shift of doktype if unallowed records are on the page.

Notice: *All four options* must be set for the default type while the rest can choose as they like.

User Settings Configuration

Introduction

The User settings module has the most complex form in the TYPO3 BE not to be driven by TCA/TCEforms. As such it has its own configuration array that is quite similar to the $TCA, but with less options.

Reference

$TYPO3_USER_SETTINGS['ctrl']

Just like with the $TCA, the “ctrl” section contains some general options that affect the global rendering of the form.

Key	Datatype	Description	Default
dividers2tabs	int	Render user setup with or without tabs. Possible values are: • 0 = no tabs, • 1 = tabs, empty tabs are hidden • 2 = tabs, empty tabs are disabled	1

$TYPO3_USER_SETTINGS['columns'][fieldname]

This contains the configuration array for single fields in the user settings. This array allows the following configurations:

Key	Datatype	Description
type	string	Defines the type of the input field If type=user you need to define userFunc too. **Example:**

```
'installToolEnableFile' => array(
	'type' => 'user',
	'label' =>
'LLL:EXT:setup/mod/locallang.xml:InstallToolEnableFileButton',
	'userFunc' => 'SC_mod_user_setup_index-
>renderInstallToolEnableFileButton',
	'access' => 'admin',
)
```

Key	Datatype	Description
		Allowed values: text, password, check, select, user
label	string	Label for the input field
csh	string	CSH key for the input field
access	string	Access control. At the moment only a admin-check is implemented Allowed values: admin
table	string	If the user setting is saved in a DB table, this property sets the table. At the moment only be_users is implemented. Allowed values: be_users
eval	string	Evaluates a field with a given function. Currently only md5 is implemented, for password field. Allowed values: md5
items	array	Array of key-value pair for select items Only used by type=select.
itemsProcFunc	string	Defines an external method for rendering items of select-type fields. Contrary to what is done with the TCA you have to render the <select> tag too. Only used by type=select. Use the usual class->method syntax.

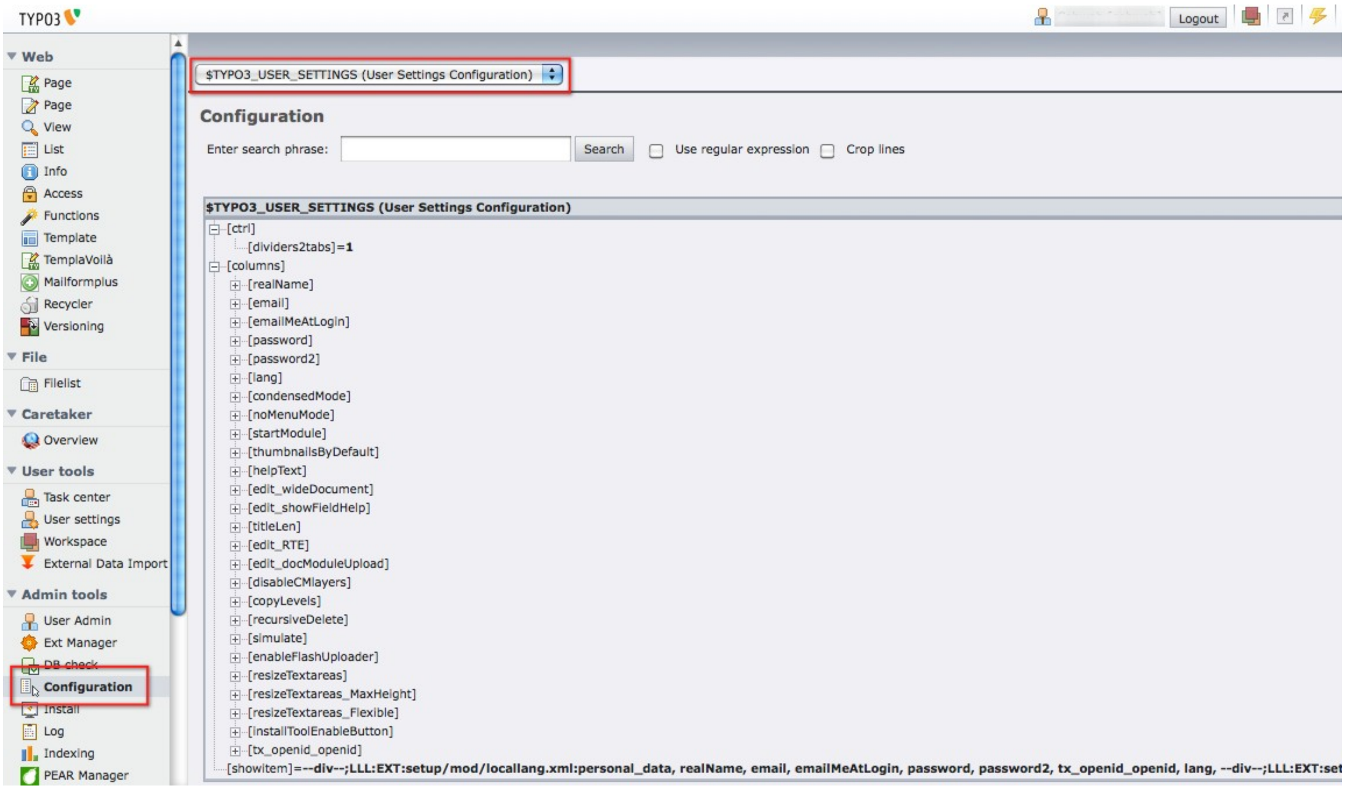
$TYPO3_USER_SETTINGS['showitem']

This string is used for rendering the form in the user setup module. Fields are rendered in the order of this string containing a comma-separated list of field names.

To use a tab insert a "--div--;LABEL" item in the list.

Checking the User Settings Configuration

It is possible to check the configuration via the Admin Tools > Configuration module, just like for the $TCA.



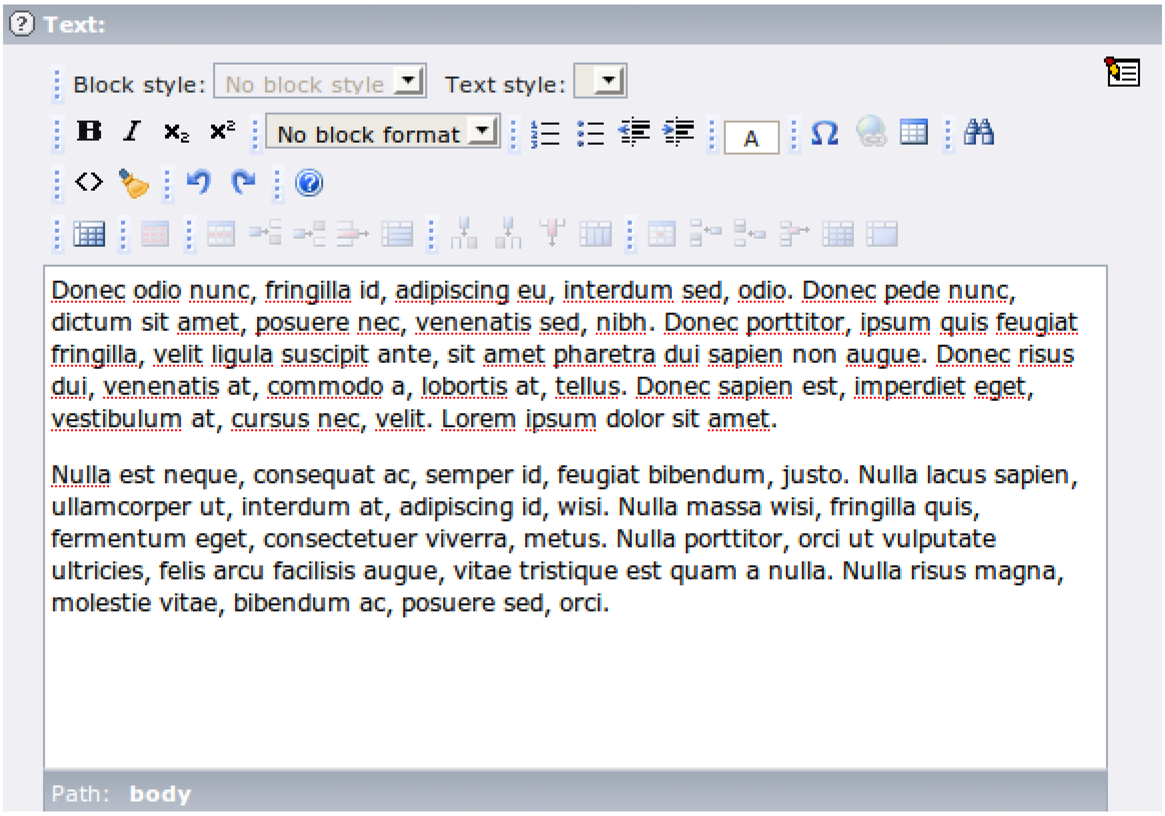
```
$TYPO3_USER_SETTINGS (User Settings Configuration)
[ctrl]
    [dividers2tabs]=1
[columns]
    [realName]
    [email]
    [emailMeAtLogin]
    [password]
    [password2]
    [lang]
    [condensedMode]
    [noMenuMode]
    [startModule]
    [thumbnailsByDefault]
    [helpText]
    [edit_wideDocument]
    [edit_showFieldHelp]
    [titleLen]
    [edit_RTE]
    [edit_docModuleUpload]
    [disableCMlayers]
    [copyLevels]
    [recursiveDelete]
    [simulate]
    [enableFlashUploader]
    [resizeTextareas]
    [resizeTextareas_MaxHeight]
    [resizeTextareas_Flexible]
    [installToolEnableButton]
    [tx_openid_openid]
[showitem]=--div--;LLL:EXT:setup/mod/locallang.xml:personal_data, realName, email, emailMeAtLogin, password, password2, tx_openid_openid, lang, --div--;LLL:EXT:set
```

RTE API

Rich Text Editors in the TYPO3 backend

Introduction

When you configure a table in $TCA and add a field of the type "text" which is edited by a <textarea> you can choose to use a Rich Text Editor (RTE) instead of the <textarea> field. A RTE enables the users to use visual formatting aids to create bold, italic, paragraphs, tables etc. In other words, it gives normal text processing features in the web browser.



Donec odio nunc, fringilla id, adipiscing eu, interdum sed, odio. Donec pede nunc, dictum sit amet, posuere nec, venenatis sed, nibh. Donec porttitor, ipsum quis feugiat fringilla, velit ligula suscipit ante, sit amet pharetra dui sapien non augue. Donec risus dui, venenatis at, commodo a, lobortis at, tellus. Donec sapien est, imperdiet eget, vestibulum at, cursus nec, velit. Lorem ipsum dolor sit amet.

Nulla est neque, consequat ac, semper id, feugiat bibendum, justo. Nulla lacus sapien, ullamcorper ut, interdum at, adipiscing id, wisi. Nulla massa wisi, fringilla quis, fermentum eget, consectetuer viverra, metus. Nulla porttitor, orci ut vulputate ultricies, felis arcu facilisis augue, vitae tristique est quam a nulla. Nulla risus magna, molestie vitae, bibendum ac, posuere sed, orci.

It is not within the scope of this chapter to describe how you set up a text field to use an RTE. As was discussed before, the quickest way is to add the key "defaultExtras" to the configuration of the column and add the string "richtext[]" as value:

```
'poem' => array(
      'exclude' => 0,
      'label' => 'LLL:EXT:examples/locallang_db.xml:tx_examples_haiku.poem',
      'config' => array(
            'type' => 'text',
            'cols' => 40,
            'rows' => 6
      ),
      'defaultExtras' => 'richtext[]'
),
```

This works for FlexForms too.

RTEs in Extensions

TYPO3 supports any Rich Text Editor for which someone might write a connector to the RTE API. This means that you can freely choose whatever RTE you want to use among those available from the Extension Repository on typo3.org.

TYPO3 comes with a built-in RTE called "rtehtmlarea", but other RTEs are available in the TYPO3 Extension Repository.

You can enable more than one RTE if you like but only one will be active at a time. Since Rich Text Editors often depend on browser versions, operating systems etc. each RTE must have a method in the API class which reports back to the system if the RTE is available in the current environment. The Rich Text Editor available to the backend user will be the *first loaded* RTE which reports back to TYPO3 that it *is available* in the environment. If the RTE is not available, the next RTE Extension loaded will be asked.

For example the RTE "rtehtmlarea" is available under Windows and Linux and under both MSIE and Mozilla. Opposite the "rte" extension is only available under MSIE / Windows. If the "rtehtmlarea" extension is loaded before the "rte" extension then the "rtehtmlarea" RTE is always used. But if "rte" is loaded first then it is also asked for availability first; the result is that under Windows / MSIE the "rte" (the "traditional" RTE in TYPO3) is used while "rtehtmlarea" will be used in other environments.

API for Rich Text Editors

Connecting an RTE in an extension to TYPO3 is easy.

- Create a class file in your extensions, named "class.tx_[extensionkey minus underscores]_base.php". Make the class inside an extension of the system class, "t3lib_rteapi" (which you should include first of course) and override functions from the parent class to the degree needed.
- In the "ext_localconf.php" file you put an entry in $TYPO3_CONF_VARS['BE']['RTE_reg'] which registers the new RTE with the system. For example;
  $TYPO3_CONF_VARS['BE']['RTE_reg']['myrte'] = array('objRef' =>
  'EXT:myrte/class.tx_myrte_base.php:&tx_myrte_base');

The object reference in "objRef" consists of a filename reference (for the class file) and then the name of the class prefixed with "&" which ensures that you get the same instance (global) of the object each time you ask for it. "myrte" is the extension key of your RTE extension (with underscores stripped).

class.t3lib_rteapi.php

In the base class for the RTE API there are three main methods of interest:

- **function isAvailable()**
  This method is asked for the availability of the RTE; This is where you should check for environmental requirements that is needed for your RTE. Basically the method must return TRUE if the RTE is available. If it is not, the RTE can put text entries in the internal array ->errorLog which is used to report back the reason why it was not available.
- **function drawRTE(&$pObj,$table,$field,$row,$PA,$specConf,$thisConfig,$RTEtypeVal,$RTErelPath,$thePidValue)**
  This method draws the content for the editing form of the RTE. It is called from the "t3lib_TCEforms" class which also passes a reference to itself in $pObj. For details on the arguments in the method call, please see inside "class.t3lib_rteapi.php".
- **function transformContent($dirRTE,$value,$table,$field,$row,$specConf,$thisConfig,$RTErelPath,$pid)**
  This method is used both from ->drawRTE() and from t3lib_tcemain to transform the content between the database and RTE. When content is loaded from the database to the RTE (and vice versa) it may need some degree of transformation. For instance references to links and images in the database might have to be relative while the RTE requires absolute references. This is just a simple example of what "transformations" can do for you and why you need them. There are plenty of details on this topic later.

Example: The "rte" extension

The "rte" extension has a "ext_localconf.php" file which looks like this:

```
if (!defined ('TYPO3_MODE'))     die ('Access denied.');

$TYPO3_CONF_VARS['BE']['RTE_reg']['rte'] = array('objRef' =>
'EXT:rte/class.tx_rte_base.php:&tx_rte_base');
```

As you can see it registers the API class to the system. In the class "tx_rte_base" the three methods from the list above is available.

The file "class.tx_rte_base.php" looks like this:

```
require_once(PATH_t3lib.'class.t3lib_rteapi.php');
/**
 * RTE base class (Traditional RTE for MSIE 5+ on windows only!)
 *
 * @author    Kasper Skaarhoj <kasper@typo3.com>
 * @package TYPO3
 * @subpackage tx_rte
 */
class tx_rte_base extends t3lib_rteapi {

        // External:
    var $RTEdivStyle;                // Alternative style for RTE <div> tag.

        // Internal, static:
    var $ID = 'rte';                // Identifies the RTE ...
    var $debugMode = FALSE;            // Debug mode


    /**
     * Returns true if the RTE is available. Here you check if the browser requirements are met.
     * If there are reasons why the RTE cannot be displayed you simply enter them as text in
->errorLog
     *
     * @return    boolean        TRUE if this RTE object offers an RTE
     */
    function isAvailable()    {
        global $CLIENT;

        if (TYPO3_DLOG)    t3lib_div::devLog('Checking for availability...','rte');

        $this->errorLog = array();
        if (!$this->debugMode)    {    // If debug-mode, let any browser through
            if ($CLIENT['BROWSER']!='msie')     $this->errorLog[] = '"rte": Browser is not MSIE';
            if ($CLIENT['SYSTEM']!='win')        $this->errorLog[] = '"rte": Client system is not
Windows';
            if ($CLIENT['VERSION']<5)            $this->errorLog[] = '"rte": Browser version
below 5';
        }
        if (!count($this->errorLog))    return TRUE;
    }

    /**
     * Draws the RTE as an iframe for MSIE 5+
     *
...
     * @return    string        HTML code for RTE!
     */
    function drawRTE(&$pObj,$table,$field,$row,$PA,$specConf,$thisConfig,$RTEtypeVal,$RTErelPath,
$thePidValue)    {

            // Draw form element:
        if ($this->debugMode)    {    // Draws regular text area (debug mode)
            $item = parent::drawRTE($pObj,$table,$field,$row,$PA,$specConf,$thisConfig,
$RTEtypeVal,$RTErelPath,$thePidValue);
        } else {    // Draw real RTE (MSIE 5+ only)

                // Adding needed code in top:
            $pObj->additionalJS_pre['rte_loader_function'] = $this->loaderFunc($pObj->formName);
            $pObj->additionalJS_submit[] = "
                            if(TBE_RTE_WINDOWS['".$PA['itemFormElName']."'])    { document.".
$pObj->formName."['".$PA['itemFormElName']."'].value = TBE_RTE_WINDOWS['".
$PA['itemFormElName']."'].getHTML(); } else { OK=0; }";

...

                // Transform value:
            $value = $this->transformContent('rte',$PA['itemFormElValue'],$table,$field,$row,
$specConf,$thisConfig,$RTErelPath,$thePidValue);

                // Register RTE windows:
            $pObj->RTEwindows[] = $PA['itemFormElName'];
            $item = '
                '.$this->triggerField($PA['itemFormElName']).'
                <input type="hidden" name="'.htmlspecialchars($PA['itemFormElName']).'"
value="'.htmlspecialchars($value).'" />
                <div id="cdiv'.count($pObj->RTEwindows).'"
style="'.htmlspecialchars($RTEdivStyle).'">
                <iframe
                    src="'.htmlspecialchars($rteURL).'"
                    id="'.$PA['itemFormElName'].'_RTE"
                    style="visibility:visible; position:absolute; left:0px; top:0px; height:100%;
width:100%;"></iframe>
                </div>';
        }

            // Return form item:
```

```
100:            return $item;
101:        }
```

Here follows some comments:

- Line 28-40 detects the browser. Only if the browser is MSIE on Windows and a version higher than or equal to 5, then will the RTE be available for the user. Notice how error messages are set in ->errorLog so the system can give the user a hint as to why the RTE didn't show up.
- Line 57 starts the method "drawRTE" which creates the RTE as HTML. This RTE is in fact created by another script inside an <iframe>. The content of the field is stored in a hidden field and the script in the IFRAME loads the content by JavaScript from this field.
  Basically, the content submitted from the RTE is *in this hidden field!* In other words, the RTE has to load and save back content to this field. Other RTEs might integrate this differently. For instance a Java RTE would also communicate the content to and from a hidden field while the "rtehtmlarea" extension uses a normal <textarea> field but somehow overlays it with visual formatting.
  In all cases, the call to triggerField() is important (line 89); This returns a hidden field with the same field name as the main field but prefixed "_TRANSFORM_" and having the value "RTE". This hidden field triggers the transformation process from RTE content to database (DB) in TCEmain and therefore you have to add it!
- Notice how line 84 calls the "transformContent" method in the class to create the $value to put into the RTE. In the case of the "rte" extension the "transformContent" method is used from the parent class, but if you need special transformations you can easily do so by overriding the function in you child class.

Transformations

Introduction

Transformation of content between the database and an RTE is needed if the format of the content in the database is different than the format understood by an RTE. A simple example could be that bold-tags in the database <b> should be converted to <strong> tags in the RTE or that references to images in <img> tags in the database should be relative while absolute in the RTE. In such cases a transformation is needed to do the conversion both ways; From database (DB) to RTE and from RTE to DB.

Generally transformations are needed for two reasons:

- **Data Formats;** If the agreed format of the stored content in TYPO3 is different from the HTML format the RTE produces. This could be issues like XHTML, banning of certain tags or maybe a hybrid format in the database. (See section 3 in the illustration some pages ahead)
- **RTE specifics**; If the RTE has special requirements to the content before it can be edited and if that format is different from what we want to store in the database. For instance an RTE could require a full HTML document with <html>, <head> and <body> - obviously we don't want that in the database and likewise we will have to wrap content in such a dummy-body before it can be edited. (This is the case with "rteekit", see section 4 in the illustration some pages ahead).

Hybrid modes

The traditional challenge of incorporating an RTE in TYPO3 has been that the RTE was available only to a limited set of browsers, typically MSIE on Windows. Therefore if an RTE was supported it had to be backwards compatible with situations where content was to be edited from regular <textarea>'s with no visual formatting.

Among the transformations in TYPO3 there are two modes, "ts_transform" and "css_transform", which are trying to maintain a data format that is as human readable as possible while still offering an RTE for editing if applicable.

To know the details of those transformations, please refer to the tables in the next section. More historical background can also be obtained later in this document. But here is a short example of a hybrid mode:

In Database:

This is how the content in the database could look for a hybrid mode (such as "css_transform"). As you can see the TYPO3-specific tag, "<link>" is used for the link to page 123. This tag is designed to be easy for editors to insert. It is of course converted to a real <a> tag when the page is rendered in the frontend. Further line 2 shows bold text. In line 3 the situation is that the paragraph should be centered - and there seems to be no other way than wrapping the line in a <p> tag with the "align" attribute. Not so human readable but we can do no better without an RTE. Line 4 is just plain.

Generally this content will be processed before output on a page of course. Typically the rule will be this: "Wrap each line in a <p> tag which is not already wrapped in a <p> tag and convert all TYPO3-specific <link>-tags to real <a> tags." and thus the final result will be valid HTML.

```
This is line number 1 with a <link 123>link</link> inside
This is line number 2 with a <b>bold part</b> in the text
<p align="center">This line is centered.</p>
This line is just plain
```

In RTE:

The content in the database can easily be edited as plain text thanks to the "hybrid-mode" used to store the content. But when the content above from the database has to go into the RTE it *will not* work if every line is not wrapped in a <p> tag! The same is true for the <link> tag; it has to be converted so the RTE understands it:

```
<p>This is line number 1 with a <a href="index.php?id=123">link</a> inside</p>
<p>This is line number 2 with a <strong>bold part</strong> in the text</p>
<p align="center">This line is centered.</p>
<p>This line is just plain</p>
```

This process of conversion from the one format to the other is what transformations do!

Configuration

Transformations are mainly defined in the "Special Configuration" of the $TCA "types"-configuration. There is detailed description of this in the $TCA section of this document.

In addition transformations can be fine-tuned by Page TSconfig which means that RTE behaviour can be determined even on page branch level! Details about this are found later in this chapter about the RTE API.

Where transformations are performed

The transformations you can do with TYPO3 is done in the class "t3lib_parsehtml_proc". There are typically a function for each direction; From DB to RTE (suffixed "_rte") and from RTE to DB (suffixed "_db").

The transformations are invoked in two cases:

- **Before content enters the editing form**
  This is done by the RTE API itself, calling the method t3lib_rteapi::transformContent(). See examples of this in the extensions "rte", "rtehtmlarea" and "rteekit". In particular "rteekit" is interesting because it not only calls the system transformations but also does some Ekit-specific processing since a whole HTML document has to be used in "Ekit" Java RTE which means that the HTML document body must be wrapped/stripped off as a part of the transformation process.
- **Before content is saved in the database**
  This is done in t3lib_tcemain class and the transformation is triggered by a pseudo-field from the submitted form! This field is added by the RTE API (calling t3lib_rteapi::triggerField()). Lets say the fieldname is "data[tt_content][456][bodytext]" then the trigger field is named "data[tt_content][456][_TRANSFORM_bodytext]" and in t3lib_tcemain this pseudo-field will be detected and used to trigger the transformation process from RTE to DB. Of course the pseudo field will never go into the database (since it is not found in $TCA).

The concept of transformations is discussed in more detail a few pages ahead ("Historical perspective on RTE transformations").

Process illustration

The following illustration shows the process of transformations graphically.

Part 1: The RTE Applications

This is the various possible RTE applications. They can be based on DHTML, Active-X, Java, Flash or whatever.

Part 2: The RTE Specific Transformation

Some RTEs might need to apply additional transformation of the content in addition to the general transformation. An example is "rteekit" which requires a full HTML document for editing (and which will return a full document). In that case the RTE specific transformation must add/remove this html-document wrapper.

RTE specific transformations is normally programmed directly into the rte-api extension class. In the case of "rteekit" that is "tx_rteekit_base" which extends "t3lib_rteapi"

Part 3: The Main Transformation

The main transformation of content between browser format for RTEs and the database storage format. This is general for all RTEs. Normally consists of converting links and image references from absolute to relative and further HTML processing as needed. *This is the kind of transformation specifically described on the coming pages*!

The main transformations is done with "t3lib_parsehtml_proc".

Part 4: The Database

The database where the content is stored for use in both backend and frontend.

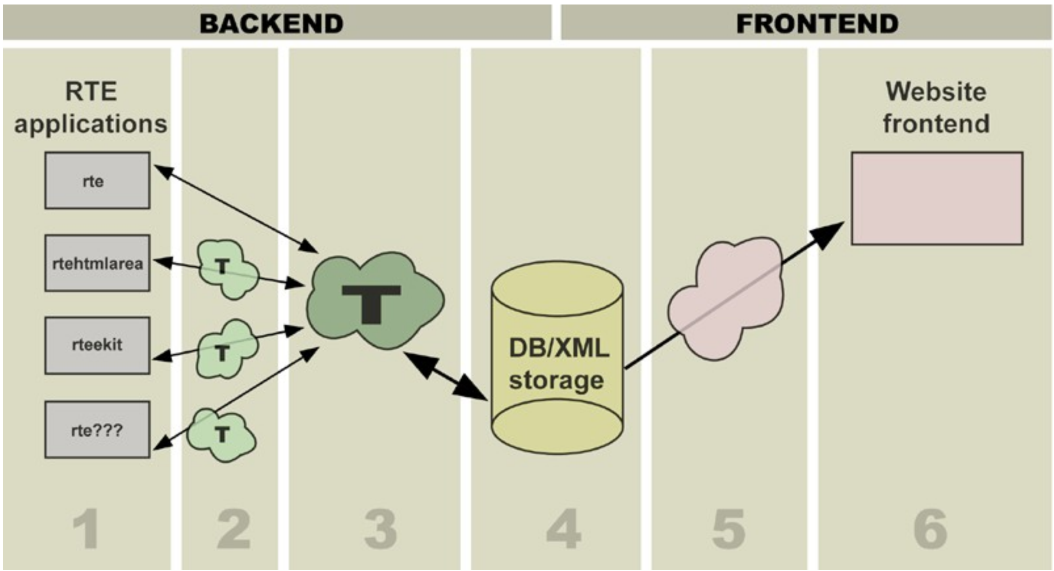
Part 5: Rendering the website

Content from the database is processed for display on the website. Depending on the storage format this might also involve "transformation" of content. For instance the internal "<link>" tag has to be converted into an HTML <a> tag.

The processing normally takes place with TypoScript Templates, the "CSS Styled Content" extension (TS object path "lib.parseFunc_RTE")

Part 6: The Website

The website made with TYPO3.



Content Examples

This table gives some examples of how content will look in the RTE, in the database and on the final website.

Notice: This is only examples! It might not happen exactly like that in real life since it depends on which exact transformations you apply. But it illustrates the point that the content needs to be in different states whether in the RTE, Database or Website frontend.

RTE (#1)	Database (#4)	Website (#6)	Comment
<p>Hello World</p>	Hello World	<p>Hello World</p>	<p> omitted in DB to make it plain-text editable.
<p align="right">Right aligned text</p>	<p align="right">Right aligned text</p>	<p align="right">Right aligned text</p>	Had to keep <p> tag in DB because align attribute was found.
<table ...>....</table>	[stripped out]	-	Tables were not allowed, so stripped.

RTE (#1)	Database (#4)	Website (#6)	Comment
<a href="http://localhost/.../index.php?id=123">	<link 123>	<a href="Contact_us.123.html">	Links are stored with the <link>-tag and needs processing for both frontend and backend.
<img src="http://localhost/fileadmin/image.jpg">	<img src="fileadmin/image.jpg">	<img src="fileadmin/image.jpg">	References to images must usually be absolute paths in RTEs while relative in database.

Transformation overview

The transformation of the content can be configured by listing which *transformation filters* to pass it through. The order of the list is the order in which the transformations are performed when saved to the database. The order is reversed when the content is loaded into the RTE again.

Transformation filter:	Description:
ts_transform	Transforms the content with regard to most of the issues related to content elements types 'Text' and 'Text w/Image'. The mode is optimized for the content rendering of the static template "content (default)" which uses old <font> tag style rendering. The mode is a "hybrid" mode which tries to save only the necessary HTML in the database so that content might still be easily edited without the RTE. For instance a text paragraph will be encapsulated in <p> tags while in the database it will just be a single line ended by a line break character. (Supports the "cms" extension)
css_transform	Like "ts_transform", but headers and bulletlists are preserved as <Hx> tags and <OL> / <UL> (TYPOLIST and TYPOHEAD are still converted to Hx and OL/UL, but not reversely...) and tables are preserved (PROC.preserveTables is disabled). The mode is optimized for the content rendering done by "css_styled_content" or similar.
ts_preserve	Converts the list of preserved tags - if any - to <SPAN>-tags with a custom parameter 'specialtag' which holds the value of the original tag. Deprecated.
ts_images	Checks if any images on the page is from external URLs and if so they are fetched and stored in the uploads/ folder. In addition 'magic' images are evaluated to see if their size has changed and if so the image is recalculated on the server. Finally absolute URLs are converted to relative URLs for all local images.
ts_links	Converts the absolute URLs of links to the TypoScript specific <LINK>-tag. This process is designed to make links in concordance with the typolink function in the TypoScript frontend.
ts_reglinks	Converts the absolute URLs of links to relative. Keeping the <A>-tag.
Meta transformation:	**Description:**
ts	Meta-mode which is basically a substitute for this list: ts_transform,ts_preserve,ts_images,ts_links. This is the one used specifically for the two 'Text'-types of the content elements ("cms" extension).
ts_css	Like "ts", a meta-mode which is a substitute for the list: css_transform,ts_images,ts_links. It is designed to be the new, modern transformation used by most RTE cases, because it converts links between <A> and <LINK> but preserves all other content while still making it as human readable as possible (that means simple <P>-tags are resolved into simple lines.)

In addition, custom transformations can be created. This allows you to create your own tailor made transformations with a PHP class where you can program how content is processed to and from the database. See section later.

Transformation details

The transformations offered by the TYPO3 core are performed by the class "t3lib_parsehtml_proc". Here follows a technical and detailed description of the transformation filters available:

DB -> RTE	RTE -> DB
ts_transform, css_transform	
function t3lib_parseHTML::TS_transform_rte()	function t3lib_parseHTML::TS_transform_db()

DB -> RTE	RTE -> DB
● Sections by the tags TABLE,PRE,UL,OL,H1,H2,H3,H4,H5,H6 are not processed and thus just passed on to the RTE. ● The content of <BLOCKQUOTE> sections are sent recursively through the ts_transform filter. The tag remains. ● <TYPOLIST> sections are converted to <OL> or <UL> sections, the latter is the case if the type parameter is set to 1. The conversion of TYPOLIST-tags can be disabled by setting the 'proc.typolist' option. See later. ● <TYPOHEAD> sections are converted to <Hx>-tags. The type parameter ranging from 1-5 determines which H-tag will be used. If no type parameter is set, H6 is used. The conversion of TYPOHEAD-tags can be disabled by setting the 'proc.typohead' option. See later. ● All content outside the tags already mentioned are now processed as follows: ● Every line is wrapped in <P>-tags (configurable to DIV), if a line is empty a &nbsp; is set and if the line happens to be wrapped in DIV/P-tags already, it's not wrapped again (this might be the case if align or class parameters has been set). ● Then <B> tags are mapped to <STRONG> tags and <I> tags are mapped to <EM> tags (This is how the RTE prefers it). ● All content between the P/DIV tags outside of other allowed HTML-tags are htmlspecialchar()'ed. Thus only allowed HTML code is preserved and other "pseudo tags" are mapped to real text.	● Sections by the tag PRE are not processed and thus just passed on to the DB. ● <TABLE>-sections are dissolved so only the text of the table cells remains. Every cell represents a new line. The reason for this action basically is that tables are not wanted in the 'Text'-types and they may also be nice to get rid of in case you have transferred content from other websites. (This can be disabled.) (Does NOT apply to "css_transform") ● The content of <BLOCKQUOTE> sections are sent recursively through the ts_transform filter. The tag remains. ● <OL> and <UL> sections are converted to <TYPOLIST> sections. If the bulletlist is <OL> (ordered list with numbers) the type parameter of the typolist is set to 1. Bulletlists in multiple levels are not supported. The conversion of TYPOLIST-tags can be disabled by setting the 'proc.typolist' option. See later. (Does NOT apply to "css_transform") ● <Hx> sections are converted to <TYPOHEAD>-tags. The number of the Hx-tag ranging from 1-5 is set as the type-number of the TYPOHEAD tag. <H6> is equal to type=0 (default). Also the align parameter is preserved as well as the class parameter if set. The conversion of TYPOHEAD-tags can be disabled by setting the 'proc.typohead' option. In that case the tag is preserved with the parameters align and class. See later. (Does NOT apply to "css_transform") ● All content outside these block are now processed as follows: ● All <DIV> and <P> sections are dissolved into lines (unless align and/or class parameters are set). ● <BR> tags are as well converted into newlines (configurable since this will resolve "soft linebreaks" into paragraphs!). ● Then <STRONG> and <EM> tags are remapped to <B> and <I> tags. (This is more human readable. Configurable). ● The list of allowed tags (configurable) is preserved - all other tags discarded (thus junk-tags from pasted content will not survive into the database!). ● The content outside the allowed tags is de-htmlspecialchar()'ed - thus converted back to human-readable text. Furthermore the nesting of tags inside of P/DIV sections is preserved. For instance this: <P>One <U><B>two</B> three</P></U> will be converted to <P>One <B>two</B> three</P>. That is the U-tags being removed, because they were falsely nested with the <P> tags.
ts_preserve (deprecated)	
function t3lib_parseHTML::TS_preserve_rte()	function t3lib_parseHTML::TS_preserve_db()
● If 'proc.preserveTags' are configured those tags are converted to <SPAN specialtag="...(the preserved tag rawurlencoded)...">-sections. Those are supposed to be let alone by the RTE.	● If 'proc.preserveTags' are configured <SPAN>-tags with the custom 'specialtag' parameter set are converted back to the tag value contained in the specialtag-parameter.
ts_images	
function t3lib_parseHTML::TS_images_rte()	function t3lib_parseHTML::TS_images_db()
● All <IMG>-tags are processed and if the value of the src-parameter happens *not* to start with 'http' it's expected to be a relative URL and the current site URL is prefixed so the reference is absolute in the RTE as the RTE requires.	● All <IMG>-tags are processed and if the first part of the src-parameter is not the same as the current site URL, the image must be a reference to an external image. In that case the image is read from that URL and stored as a 'magic' image in the upload/ folder (can be disabled). ● All magic images (that is images stored in the uploads/ folder (configured by TYPO3_CONF_VARS["BE"]["RTE_imageStorageDir"], filenames prefixed with 'RTEmagicC_' (child=actual image) and 'RTEmagicP_' (parent=original image))) are processed to see if the physical dimensions of the image on the server matches the dimensions set in the img-tag. If this is not the case, the user must have changed the dimensions and the image must be re-scaled accordingly. ● Finally the absolute reference to the image is converted to a proper relative reference if the image URL is local.

DB -> RTE	RTE -> DB
ts_links	
function t3lib_parseHTML::TS_links_rte()	function t3lib_parseHTML::TS_links_db()
● All <LINK>-tags (TypoScript specific) are converted to proper <A>-tags. The parameters of the <LINK>-tags are separated by space. The first parameter is the link reference (see typolink function in TSref for details on the syntax), second is the target if given (if '-' the target is not set), the third parameter is the class (if '-' the class is not set) and the fourth parameter is the title.	● All <A>-tags are converted to <LINK> tags, however only if they do not contain any parameters other than href, target and class. These are the only three parameters which can be represented by the TypoScript specific <LINK>-tag.
ts_reglinks	
function t3lib_parseHTML::TS_reglinks()	function t3lib_parseHTML::TS_reglinks()
● All A-tags have URLs converted to absolute URLs if they are relative	● All A-tags have their absolute URLs converted to relative if possible (that is the URL is within the current domain).

Page TSconfig

The RTEs can be configured by Page TSconfig. There is a top level object name, "RTE", that is used for this. The main object paths looks like this:

Property:	Data type:	Description:
default.[...] config.[*tablename*].[*field*].[...] config.[*tablename*].[*field*].types.[*type*].[...]	->RTEconf	These objects contain the actual configuration of the RTE interface. For the properties available, refer to the table below. This is a description of how you can customize in general and override for specific fields/types. 'RTE.default' configures the RTE for all tables/fields/types 'RTE.config.[*tablename*].[*field*]' configures a specific field. The values inherit the values from 'RTE.default' in fact this is overriding values. 'RTE.config.[*tablename*].[*field*].types.[*type*]' configures a specific field in case the 'type'-value of the field matches *type*. Again this overrides the former settings.
[individual RTE options]	-	There are other options to set for the RTE toplevel object. These depends on the individual RTEs though! So there can be no further reference in this table to these properties. Generally the "rte" (classic MSIE RTE) will set the standard for configuration options, so you can refer to the documentation for that RTE for more details. On the top level of the RTE object you will normally find that general collections of classes, styles etc. are configured.

[page:RTE]

Configuration examples

This configuration in "Page TSconfig" will disable the RTE altogether:

```
RTE.default.disabled = 1
```

In the case below the RTE is still disabled generally, but this is overridden specifically for the table "tt_content" where the RTE is used in the field "bodytext"; The "disabled" flag is set to false again which means that for Content Elements the RTE will be available.

```
RTE.default.disabled = 1
RTE.config.tt_content.bodytext.disabled = 0
```

In this example the RTE is still enabled for content elements in generally but if the Content Element type is set to "Text" (text) then the RTE will be disabled again!

```
RTE.default.disabled = 1
RTE.config.tt_content.bodytext.disabled = 0
RTE.config.tt_content.bodytext.types.text.disabled = 1
```

The RTE object in Page TSconfig

The RTE object contains configuration of the RTE application. There are a few properties which are used externally from the RTE. The property "disabled" will simply disable the rendering of the RTE and "proc" is reserved to contain additional configuration of transformations.

Property:	Data type:	Description:
disabled	boolean	If set, the editor is disabled. This option is evaluated in t3lib_TCEforms where it determines if the RTE is rendered or not.
proc	->PROC	Customization of the server processing of the content - also called 'transformations'. See table below. The transformations are only initialized, if they are configured ("rte_transform" must be set for the field in the types-definition in TCA.) The "->PROC" object is processed in "t3lib_parsehtml_proc" and is *independant* of the particular RTE used (like transformations generally is!)
[individual RTE options]	-	Each RTE may use additional properties for the RTE. Typically such properties relates to the features of the RTE application. For instance you could configure which tool bar buttons are available etc.

[page:->RTEconf]

Configuration examples

```
 0: RTE.default >
 1: RTE.default {
 2:   mainStyle_font = Arial, sans-serif
 3:   mainStyle_size = 12
 4:   mainStyle_color = black
 5:   classesParagraph = redText
 6:   classesCharacter = redText
 7:   showButtons = cut,copy,fontstyle,fontsize, textcolor,table,bgcolor
 8:   proc.preserveTables = 1
 9:
10:   proc.entryHTMLparser_db = 1
11:   proc.entryHTMLparser_db {
12:     keepNonMatchedTags = 1
13:     xhtml_cleaning = 1
14:   }
15:
16:   mainStyleOverride_add {
17:     P =  font-family:Arial, sans-serif; font-size:12;
18:     H1 =  font-family:Arial, sans-serif; font-size:16;  font-weight:bold; margin-top:0;margin-
bottom:10;
19:     H2 =  font-family:Arial, sans-serif; font-size:12;  font-weight:bold; color:navy; margin-
top:0;margin-bottom:10;
20:     H3 =  font-family:Arial, sans-serif; font-size:18;  font-weight:bold;
21:     H4 =  font-family:Arial, sans-serif; font-size:24;
22:     H5 =  font-family:Arial, sans-serif; font-size:20;  color:navy; font-weight:normal;  margin-
top:0;margin-bottom:10;
23:     H6 =  font-family:Arial, sans-serif; font-size:16;  font-weight:bold;
24:   }
25:   disablePCexamples = 0
26: }
```

In this example all the configuration except line 8-14 ("proc." configuration) is defining the RTE applications internal features. These options will vary depending on the RTE used. In this case the configuration is for the classic MSIE Active-X RTE in the extension "rte".

The ->PROC object

This object contains configuration of the transformations used. These options are *universal for all RTEs* and used inside the class "t3lib_parsehtml_proc".

The main objective of these options is to allow for minor configuration of the transformations. For instance you may disable the mapping between <B>-<STRONG> and <I>-<EM> tags which is done by the 'ts_transform' transformation. Or you could disable the default transfer of images from external URL to the local server. This is all possible through the options.

Notice how many properties relates to specific transformations only! Also notice that the meta-transformations "ts" and "ts_css" implies other transformations like "ts_transform" and "css_transform" which means that options limited to "ts_transform" will also work for "ts" of course.

Property:	Data type:	Description:
overruleMode	List of RTE transformations	This can overrule the RTE transformation set from TCA. Notice, this is a *comma list* of transformation keys. (Not a "dash-list" like in $TCA).

Property:	Data type:	Description:
typolist	boolean	*(Applies for "ts_transform" only)* This enables/disables the conversion between <TYPOLIST> and <UL> sections. Default (if unset) is that "typolist" is enabled. **Example that disables "typolist":** typolist = 0
typohead	boolean	*(Applies for "ts_transform" only)* This enables/disables the conversion between <TYPOHEAD> and <Hx> sections. **Example that disables "typohead":** typohead = 0
preserveTags	list of tags	(DEPRECATED) Here you may specify a list of tags - possibly user-defined pseudo tags - which you wish to preserve from being removed by the RTE. See the information about preservation in the description of transformations. **Example:** In the default TypoScript configuration of content rendering the tags typotags <LINK>, <TYPOLIST> and <TYPOHEAD> are the most widely used. However the <TYPOCODE>-tag is also configured to let you define a section being formatted in monospace. Lets also imaging, you have defined a custom tag, <MYTAG>. In order to preserve these tag from removal by the RTE, you should configure like this.

```
RTE.default.proc {
  preserveTags = TYPOCODE, MYTAG
}
```

Relates to the transformation 'ts_preserve'

Property:	Data type:	Description:
dontConvBRtoParagraph	boolean	*(Applies for "ts_transform" and "css_transform" only (function divideIntoLines))* By default <BR> tags in the content are converted to paragraphs. Setting this value will *prevent* the convertion of <BR>-tags to new-lines (chr(10))
internalizeFontTags	boolean	*(Applies for "ts_transform" and "css_transform" only (function divideIntoLines))* This splits the content into font-tag chunks. If there are any <P>/<DIV> sections inside of them, the font-tag is wrapped AROUND the content INSIDE of the P/DIV sections and the outer font-tag is removed. This functions seems to be a good choice for pre-processing content if it has been pasted into the RTE from e.g. star-office. In that case the font-tags is normally on the OUTSIDE of the sections.
allowTagsOutside	commalist of strings	*(Applies for "ts_transform" and "css_transform" only (function divideIntoLines))* Enter tags which are allowed outside of <P> and <DIV> sections when converted back to database. Default is "img" Example: IMG,HR
allowTagsInTypolists	commalist of strings	*(Applies for "ts_transform" only)* Enter tags which are allowed inside of <typolist> tags when content is sent to the database. Default is "br,font,b,i,u,a,img,span"
allowTags	commalist of strings	*(Applies for "ts_transform" and "css_transform" only (function getKeepTags))* Tags to allow. Notice, this list is *added* to the default list, which you see here: b,i,u,a,img,br,div,center,pre,font,hr,sub,sup,p,strong,em,li,ul,ol,blockquote,strike,span If you wish to deny some tags, see below.
denyTags	commalist of strings	*(Applies for "ts_transform" and "css_transform" only (function getKeepTags))* Tags from above list to disallow.

Property:	Data type:	Description:
transformBoldAndItalicTags	boolean	*(Applies for "ts_transform" and "css_transform" only (function getKeepTags))* Default is to convert b and i tags to strong and em tags respectively in the direction of the database, and to convert back strong and em tags to b and i tags in the direction of the RTE. This transformation may be disabled by setting this property to 0.
HTMLparser_rte HTMLparser_db	->HTMLparser	*(Applies for "ts_transform" and "css_transform" only (function getKeepTags))* This is additional options to the HTML-parser calls which strips of tags when the content is prepared for the RTE and DB respectively. You can configure additional rules, like which other tags to preserve, which attributes to preserve, which values are allowed as attributes of a certain tag etc. .nestingGlobal for HTMLparser_db is set by default to "b,i,u,a,center,font,sub,sup,strong,em,strike,span" unless another value is set. Also B/I tags are mapped to STRONG/EM tags in the RTE direction and vise versa. This parsing is done on a per-line basis, so you cannot expect the paragraph tags (P or DIV) to be included. **Notice** the ->HTMLparser options, "keepNonMatchedTags" and "htmlSpecialChars" is NOT observed. They are preset internally
dontRemoveUnknownTags_db	boolean	*(Applies for "ts_transform" and "css_transform" only (function HTMLcleaner_db))* Direction: To database Default is to remove all unknown tags in the content going to the database. (See HTMLparser_db above for default tags). Generally this is a very usefull thing, because all kinds of bogus tags from pasted content like that from Word etc. will be removed to have clean content in the database. However this disables that and allows all tags, that are not in the HTMLparser_db-list.
dontUndoHSC_db	boolean	*(Applies for "ts_transform" and "css_transform" only (function HTMLcleaner_db))* Direction: To database Default is to re-convert literals to characters (that is &lt; to <) outside of HTML-tags. This is disabled by this boolean. (HSC means HtmlSpecialChars - which is a PHP function)
dontProtectUnknownTags_rte	boolean	*(Applies for "ts_transform" and "css_transform" only (function setDivTags))* Direction: To RTE Default is that tags unknown to HTMLparser_rte is "protected" when sent to the RTE. This means they are converted from eg <MYTAG> to &lt;MYTAG&gt;. This is normally very fine, because it can be edited plainly by the editor and when returned to the database the tag is (by default, disabled by .dontUndoHSC_db) converted back. Setting this option will prevent unknown tags from becoming protected.
dontHSC_rte	boolean	*(Applies for "ts_transform" and "css_transform" only (function setDivTags))* Direction: To RTE Default is that all content outside of HTML-tags is passed through htmlspecialchars(). This will disable that. (opposite to .dontUndoHSC_db) This option disables the default htmlspecialchars() conversion.
dontConvAmpInNBSP_rte	boolean	*(Applies for "ts_transform" and "css_transform" only (function setDivTags))* Direction: To RTE By default all &nbsp; codes are NOT converted to &amp;nbsp; which they naturally word (unless .dontHSC_rte is set). You can disable that by this flag.
allowedFontColors	list of HTMLcolors	*(Applies for "ts_transform" and "css_transform" only (function getKeepTags))* Direction: To DB If set, this is the only colors which will be allowed in font-tags! Case insensitive.
allowedClasses	list of strings	*(Applies for "ts_transform" and "css_transform" only (function getKeepTags))* Direction: To DB Allowed general classnames when content is stored in database. Could be a list matching the number of defined classes you have. Case-insensitive. This might be a really good idea to do, because when pasting in content from MS word for instance there are a lot of <SPAN> and <P> tags which may have class-names in. So by setting a list of allowed classes, such foreign classnames are removed. If a classname is not found in this list, the default is to remove the class-attribute.

Property:	**Data type:**	**Description:**
skipAlign skipClass	boolean	*(Applies for "ts_transform" and "css_transform" only (function divideIntoLines))* If set, then the align and class attributes of <P>/<DIV> sections (respectively) will be ignored. Normally <P>/<DIV> tags are preserved if one or both of these attributes are present in the tag. Otherwise it's removed.
keepPDIVattribs	list of tag attributes (strings)	*(Applies for "ts_transform" and "css_transform" only (function divideIntoLines))* "align" and "class" are the only attributes preserved for <P>/<DIV> tags. Here you can specify a list of other attributes to preserve.
remapParagraphTag	string / boolean	*(Applies for "ts_transform" and "css_transform" only (function divideIntoLines))* When <P>/<DIV> sections are converted to be put into the database, the tag - P or DIV - is preserved. However setting this options to either P or DIV will force the section to be converted to the one or the other. If the value is set true (1), then it works as a general disable-flag for the whole section-convertion stuff here and the result will be no tags preserved what so ever. Just removed.
useDIVasParagraphTagForRTE	string	*(Applies for "ts_transform" only and "css_transform" (function TS_transform_rte))* Use <DIV>-tags for sections when converting lines from database to RTE. Default is <P>. Applies only to lines which has NO tag wrapped around already.
preserveDIVSections	boolean	*(Applies for "ts_transform" and "css_transform" only)* If set, div sections will be treated just like blockquotes. They will be treated recursively as external blocks.
preserveTables	boolean	*(Applies for "ts_transform")* If set, tables are preserved
dontFetchExtPictures	boolean	*(Applies for "ts_images")* If set, images from external urls are not fetched for the page if content is pasted from external sources. Normally this process of copying is done.
plainImageMode	boolean/string	*(Applies for "ts_images")* If set, all "plain" local images (those that are not magic images) will be cleaned up in some way. If the value is just set, then the style attribute will be removed after detecting any special width/height CSS attributes (which is what the RTE will set if you scale the image manually) and the border attribute is set to zero. You can also configure with special keywords. So setting "plainImageMode" to any of the value below will perform special processing: "lockDimensions" : This will read the real dimensions of the image file and force these values into the <img> tag. Thus this option will prevent any user applied scaling in the image! "lockRatio" : This will allow users to scale the image but will automatically correct the height dimension so the aspect ratio from the original image file is preserved. "lockRatioWhenSmaller" : Like "lockRatio", but will not allow any scaling larger than the original size of the image.
exitHTMLparser_rte exitHTMLparser_db entryHTMLparser_rte entryHTMLparser_db	boolean/->HTMLparser	*(Applies for all kinds of processing)* Allows you to enable/disable the HTMLparser for the content before (entry) and after (exit) the content is processed with the predefined processors (e.g. ts_images or ts_transform). There is no default values set.
disableUnifyLineBreaks	boolean	*(Applies for all kinds of processing)* When entering the processor all \r\n linebreaks are converted to \n (13-10 to 10). When leaving the processor all \n is reconverted to \r\n (10 to 13-10). This options disables that processing...
usertrans.[user-defined transformation key]	-	Custom option-space for userdefined transformations. See example from section about custom transformations.

[page:->PROC]

Custom transformations API

Instead of using the built-in transformations of TYPO3 you can program your own. This is done by creating a PHP class with two methods for transformation. Additionally you have to define a key (like "css_transform") for your transformation so you can refer to it in the configuration of Rich Text Editors.

Custom transformation key

You should pick a custom transformation key which is prefixed with either "tx_" or "user_". Use "tx_[extension key]_[suffix]" if you deliver your transformation inside an extension.

Notice: If you pick one of the default transformation keys (except the meta-transformations) you will simply *override it* and your transformation will be called instead!

Registering the transformation key in the system

In "ext_localconf.php" you simply set a $TYPO3_CONF_VARS variable to point to the class which contains the transformation methods:

```
$TYPO3_CONF_VARS['SC_OPTIONS']['t3lib/class.t3lib_parsehtml_proc.php']['transformation']['tx_myext']
    = 'EXT:myext/custom_transformation.php:user_transformation';
```

Here the *transformation key* is defined to be "tx_myext" (assuming the extension has the extension key "myext") and the value points to a file inside the transformation which will contain the class "user_transformation" (instantiated by t3lib_div::getUserObj())

This class must contain two methods, "transform_db" and "transform_rte" for each transformation direction.

Code listing of "user_transformation"

This code listing shows a simple transformation. When content is delivered to the RTE it will add a <hr/> tag to the end of the content. When the content is stored in the database any <hr/> tag in the end of the content will be removed and substituted with whitespace. This is of totally useless but nevertheless shows the concept of transformations between RTE and DB.

```
/**
 * Custom RTE transformation
 */
class user_transformation {

        // object; Reference to the parent object, t3lib_parsehtml_proc
    var $pObj;

        // Transformation key of self.
    var $transformationKey = 'tx_myext';

        // Will contain transformation configuration if found:
    var $conf;


    /**
     * Setting specific configuration for this transformation
     *
     * @return    void
     */
    function initConfig()    {
        $this->conf = $this->pObj->procOptions['usertrans.'][$this->transformationKey.'.'];
    }

    /**
     * Reserved method name, called when content is transformed for DB storage
     * If "proc.usertrans.tx_myext.addHrulerInRTE = 1" then a horizontal ruler in the
     * end of the content will be removed (if found)
     *
     * @param    string        RTE HTML to clean for database storage
     * @return    string        Processed input string.
     */
    function transform_db($value)    {
        $this->initConfig();

        if ($this->conf['addHrulerInRTE'])    {
```

```
36:             $value = eregi_replace('<hr[[:space:]]*[\/]>[[:space:]]*$','',$value);
37:         }
38: 
39:         return $value;
40:     }
41: 
42:     /**
43:      * Reserved method name, called when content is transformed for RTE display
44:      * If "proc.usertrans.tx_myext.addHrulerInRTE = 1" then a horizontal ruler
45:      * will be added in the end of the content.
46:      *
47:      * @param    string        Database content to transform to RTE ready HTML
48:      * @return    string        Processed input string.
49:      */
50:     function transform_rte($value)    {
51:         $this->initConfig();
52: 
53:         if ($this->conf['addHrulerInRTE'])    {
54:             $value.='<hr/>';
55:         }
56: 
57:         return $value;
58:     }
59: }
```

Comments to code listing

- The transformation methods "transform_rte" and "transform_db" takes a single argument which is the value to transform. They have to return that value again.
- The internal variable $pObj is set to be a reference to the parent object which is an instance of "t3lib_parsehtml_proc". Inside of this object you can access the default transformation functions if you need to and in particular you can read out configuration settings.
- The internal variable $transformationKey is automatically set to the transformation key that is active.
- Notice that both transformation functions call initConfig() (line 33 and 51) which reads custom configuration.

Using the transformation

In order to use the transformation you simply use it in the list of transformations in Special Configuration. Here is an example that works:

```
'TEST01' => Array (
    'label' => 'TEST01: Text field',
    'config' => Array (
        'type' => 'text',
    ),
    'defaultExtras' => 'richtext[*]:rte_transform[mode=tx_myext-css_transform]'
),
```

The order is important. The order in this list is the order of calling when the direction is "db". If the order is reversed the <hr/> tag will come out as regular text in the RTE because "css_transform" protects all non-allowed tags with htmlspecialchars().

Now the transformations should be called correctly. Before the <hr/> will be added/removed we also have to configure through Page TSconfig (because we programmed our transformation to look for this configuration option):

```
RTE.default.proc.usertrans.tx_myext.addHrulerInRTE = 1
```

That's all!

Historical perspective on RTE transformations

Introduction

The next sections describe in more details the necessity of RTE transformations. The text is written at the birth of transformations and might therefore be slightly oldfashioned. However it checked out generally OK and may help you to further understand why these issues exist. The argumentation is still valid.

Properties and 'transformations'

The RTE applications typically expect to be fed with content formatted as HTML. In effect an RTE will discard content it doesn't like, for instance fictitious HTML tags and line breaks. Also the HTML content created by the RTE editor is not necessarily as 'clean' as you might like.

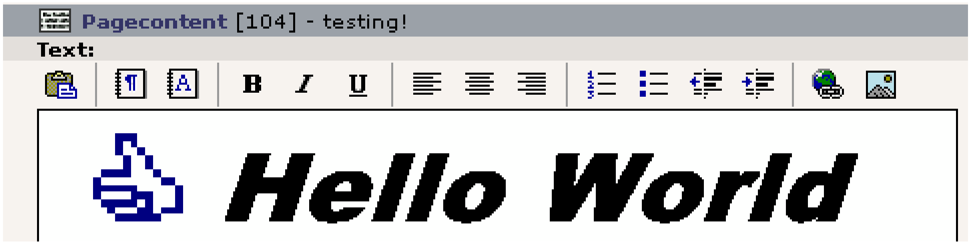
The editor has the ability to paste in formatted content copied/cut from other websites (in which case images are included!) or from text processing applications like MS Word or Star Office. This is a great feature and may solve the issue of transferring formatted content from e.g. Word into TYPO3.

However these inherent features - good or bad - raises the issue how to handle content in a field which we do not wish to 'pollute' with unnecessary HTML-junk. One perspective is the fact that we might like to edit the content with Netscape later (for which the RTE cannot be used, see above) and therefore would like it to be 'human readable'. Another perspective is if we might like to use only Bold and Italics but not the alignment options. Although you can configure the editor to display *only* the bold and italics buttons, this does *not* prevent users from pasting in HTML-content copied from other websites or from Microsoft Word which *does* contain tables, images, headlines etc.

The answer to this problem is a so called 'transformation' which you can configure in the $TCA (global, authoritative configuration) and which you may further customize through Page TSconfig (local configuration for specific branches of the website). The issue of transformations is best explained by the following example from the table, tt_content (the content elements).

RTE transformations in Content Elements

The RTE is used in the bodytext field of the content elements, configured for the types 'Text' and 'Text w/Image'.



This is how the toolbar looks if the type of the content element is not 'Rich Text' but 'Text'.

The configuration of the two 'Text'-types are the same: The toolbar includes only a subset of the total available buttons. The reason is that the text content of these types, 'Text' and 'Text w/Image' is *traditionally* not meant to be filled up with HTML-codes. But more important is the fact that the content is usually (by the standard TypoScript content rendering used on the vast majority of TYPO3 websites!) parsed through a number of routines.

In order to understand this, here is an outline of what typically happens with the content of the two Text-types when rendered by TypoScript for frontend display:

1. All line breaks are converted to <br /> codes.

   (Doing this enables us to edit the text in the field rather naturally in the backend because line breaks in the edit field comes out as line breaks on the page!)

2. All instances of 'http://...' and 'mailto:....' are converted to links.

   (This is a quick way to insert links to URLs and email address)

3. The text is parsed for special tags, so called 'typotags', configured in TypoScript. The default typotags tags are <LINK> (making links), <TYPOLIST> (making bulletlists), <TYPOHEAD> (making headlines) and <TYPOCODE> (making monospaced formatting).

   (The <LINK> tag is used to create links between pages inside TYPO3. Target and additional parameters are automatically added which makes it a very easy way to make sure, links are correct. <TYPOLIST> renders each line between the start and end tag as a line in a bulletlist, formatted like the content element type 'Bulletlist' would be. This would typically result in a bulletlist placed in a table and not using the bullet-list tags from HTML. <TYPOHEAD> would display the tag content as a headline. The type-parameter allows to select between the five default layout types of content element headlines. This might include graphical headers. <TYPOCODE> is not converted).

4. All other 'tags' found in the content are converted to regular text (with htmlspecialchars) unless the tag is found in the 'allowTags' list.

   (This list includes tags like 'b' (bold) and 'i' (italics) and so these tags may be used and will be outputted. However tags like 'table', 'tr' and 'td' is not in this list by default, so table-html code inserted will be outputted as text and not as a table!)

5. Constants and search-words - if set - will be highlighted or inserted.

   (This feature will mark up any found search words on the pages if the page is linked to from a search result page.)

6. And finally the result of this processing may be wrapped in <font>-tags, <p>-tags or whatever is configured. This depends on whether a stylesheet is used or not. If a stylesheet is used the individual sections between the typotags are usually wrapped separately.

Now lets see how this behaviour challenges the use of the RTE. This describes how the situation is handled regarding the two Text-types as mentioned above. (Numbers refer to the previous bulletlist):

1. Line breaks: The RTE removes all line breaks and makes line breaks itself by either inserting a <P>...</P> section or <DIV>...</DIV>. This means we'll have to convert existing lines to <P>...</P> before passing the content to the RTE and further we need to revert the <DIV> and <P> sections in addition to the <BR>-tags to line breaks when the content is returned to the database from the RTE.

   The greatest challenge here is however what to do if a <DIV> or <P> tag has parameters like 'class' or 'align'. In that case we can't just discard the tag. So the tag is preserved.

2. The substitution of http:// and mailto: does not represent any problems here.

3. "Typotags": The typotags are not real HTML tags so they would be removed by the RTE. Therefore those tags must be converted into something else. This is actually an opportunity and the solution to the problem is that all <LINK>-tags are converted into regular <A>-tags, all <TYPOLIST> tags are converted into <OL> or <UL> sections (ordered/unordered lists, type depends on the type set for the <TYPOLIST> tag!), <TYPOHEAD>-tags are converted to <Hx> tags where the number is determined by the type-parameter set for the <TYPOHEAD>-tag. The align/class-parameter - if set - is also preserved. When the HTML-tags are returned to the database they need to be reverted to the specific typotags.

   Other typotags (non-standard) can be preserved by being converted to a <SPAN>-section and back. This must be configured through Page TSconfig.

   (Update: With "css_styled_content" and the transformation "ts_css" only the <link> typotag is left. The <typolist> and <typohead> tags are obsolete and regular HTML is used instead)

4. Allowed tags: As not all tags are allowed in the display on the webpage, the RTE should also reflect this situation. The greatest problem is tables which are (currently) not allowed with the Text-types. The reason for this goes back to the philosophy that the field content should be human readable and tables are not very 'readable'.

   (Update: With "css_styled_content" and the transformation "ts_css" tables are allowed)

5. Constants and search words are no problem.

6. Global wrapping does not represent a problem either. But this issue is related more closely to the line break-issue in bullet 1.

Finally images inserted are processed very intelligently because the 'magic' type images are automatically post-processed to the correct size and proportions after being changed by the RTE in size.

Also if images are inserted by a copy/paste operation from another website, the image inserted will be automatically transferred to the server when saved.

In addition all URLs for images and links are inserted as absolute URLs and must be converted to relative URLs if they are within the current domain.

Conclusion:

These actions are done by so called *transformations* which are configured in the $TCA. Basically these transformations are admittedly very customized to the default behavior of the TYPO3 frontend. And they are by nature “fragile” constructions because the content is transformed back and forth for each interaction between the RTE and the database and may so be erroneously processed. However they serve to keep the content stored in the database 'clean' and human readable so it may continuously be edited by non-RTE browsers and users. And furthermore it allows us to insert TYPO3-bulletlists and headers (especially graphical headers) visually by the editor while still having TYPO3 controlling the output.

Skinning API

$TBE_STYLES

When you make skins for TYPO3 you basically set up values in the global array $TBE_STYLES which will make the system use alternative icons, stylesheets, frame widths etc.

You change values in $TBE_STYLES through an extension, setting the alternative values in the "ext_tables.php" file of the extension. For an example, see the extension "skin360".

$TBE_STYLES API

The $TBE_STYLES array contains these keys

When the values are references to files (icons, logos etc) the path must be *relative* to the TYPO3 backend dir.

Key	Subkeys	Description
colorschemes	[0-x]	*Related to TCEforms. See other section about visual style of TCEforms.*
styleschemes	[0-x]	*Related to TCEforms. See other section about visual style of TCEforms.*
borderschemes	[0-x]	*Related to TCEforms. See other section about visual style of TCEforms.*

Key	Subkeys	Description
mainColors	bgColor bgColor2 bgColor3 bgColor4 bgColor5 bgColor6 hoverColor	Main colorscheme in interface. Notice that these colors are redundantly set in the stylesheet and you have to keep them in sync. Setting the colors here is still necessary but secondary in priority to the stylesheet settings. Always use #xxxxxx color definitions! Here is a description of the colors. • bgColor *Light page background color* • bgColor2 *Alternative header background (steel blue)* • bgColor3 *Color for "documents" - concept which is now removed. Anyways, light color)* • bgColor4 *For table content cells (light tablerow background, brownish)* • bgColor5 *For table header cells in sections (light tablerow background, greenish)* • bgColor6 *For backend module section headers (light H2 background, yellowish. Light)* • hoverColor *Link hover color* **Example:**

```
$TBE_STYLES['mainColors'] = array(
    'bgColor' => '#EDF4EB',
    'bgColor2' => '#7C8591',
    'bgColor3' => '#E4E8F2',
    'bgColor4' => '#92AA8B',
    'bgColor5' => '#A5B7C1',
    'bgColor6' => '#C7BF81',
    'hoverColor' => '#800000'
);
```

Corresponding stylesheet values:
Here is an example of the stylesheet values corresponding to the "mainColors" values shown above. Notice how they share the same name - but with some variations. For instance "bgColor-10" and "bgColor-20" is based on "bgColor" but darkend approx. 10% and 20%. Such variations are available for usage when you want alternating values in a listing.

```
.bgColor {background-color: #F7F3EF;}
.bgColor-10 {background-color: #ede9e5;}
.bgColor-20 {background-color: #e3dfdb;}
.bgColor2 {background-color: #9BA1A8;}
.bgColor3 {background-color: #F6F2E6;}
.bgColor3-20 {background-color: #e2ded2;}
.bgColor4 {background-color: #D9D5C9;}
.bgColor4-20 {background-color: #c5c1b5;}
.bgColor5 {background-color: #ABBBB4;}
.bgColor6 {background-color: #E7DBA8;}
```

(From file typo3/stylesheet.css)

background	-	Background image generally in the backend *Deprecated - use the $TBE_STYLES['skinImg'] feature instead!*
logo	-	Logo in alternative backend, top left: 129x32 pixels *Deprecated - use the $TBE_STYLES['skinImg'] feature instead!*
logo_login	-	Login-logo: 333x63 pixels *Deprecated - use the $TBE_STYLES['skinImg'] feature instead!*
loginBoxImage_rotationFolder	-	Setting login box image rotation folder. From this folder images are selected randomly for display in the login box.
stylesheet	-	Alternative stylesheet to the default "typo3/stylesheet.css" stylesheet.
stylesheet2	-	Additional stylesheet (not used by default). Set BEFORE any in-document styles
styleSheetFile_post	-	Additional stylesheet. Set AFTER any in-document styles
inDocStyles_TBEstyle	-	Additional default in-document styles.

<table>
<tr><th>Key</th><th>Subkeys</th><th>Description</th></tr>
<tr><td>dims</td><td>leftMenuFrameW
topFrameH
shortcutFrameH
selMenuFrame
navFrameWidth</td><td>Setting of alternative dimensions of framesets in TYPO3:

Description of subkeys:

- FrameW
  Left menu frame width
- topFrameH
  Top frame heigth
- shortcutFrameH
  Shortcut frame height
- selMenuFrame
  Width of the selector box menu frame
- navFrameWidth
  Default navigation frame width

Example:

<pre>
    // Alternative dimensions for frameset sizes:
$TBE_STYLES['dims']['leftMenuFrameW']=165;
$TBE_STYLES['dims']['topFrameH']=35;
$TBE_STYLES['dims']['shortcutFrameH']=35;
$TBE_STYLES['dims']['selMenuFrame']=180;
$TBE_STYLES['dims']['navFrameWidth']=350;
</pre></td></tr>
<tr><td>scriptIDindex</td><td>[script-id]</td><td>All scripts in TYPO3s backend calculates an automatic “script-id”. This id can be found in the HTML source:

<pre>
<html>
<head>
      <!-- TYPO3 Script ID: typo3/mod/web/perm/index.php -->
...
</pre>

With the “scriptIDindex” feature you can override *any* $TBE_STYLES setting on a per-script basis as long as you know the script ID.

An example is in the “skin360” extension where the rollover color of the Context Sensitive Menus is defined by $TBE_STYLES['mainColors']['bgColor5']. However the color should be different from the general “bgColor5”. This can be done by the PHP line below - because the script ID 'typo3/alt_clickmenu.php' simply configures the bgColor5 value differently when the alt_clickmenu.php script requests it!

<pre>
$TBE_STYLES['scriptIDindex']['typo3/alt_clickmenu.php']
['mainColors']['bgColor5']='#E0E7C7';
</pre></td></tr>
</table>

Key	Subkeys	Description
skinImgAutoCfg	absDir relDir forceFileExtension scaleFactor	Configures automatic detection of alternative icons. This works by setting up a directory inside of which TYPO3 looks to find a file with the same filename as the one requested - and if found, the icon is used instead. • absDir *Absolute path to the directory with the icons (needed so icons can be read by getimagesize)* • relDir *Relative path to the directory with the icons (needed for making the <img> tag.)* • forceFileExtension *This can allow you to specify an alternative file extension to look for. For instance most icons in TYPO3 are gif-files. By setting this value to "png" all filenames looked for will be the gif-filename body but with a ".png" extension.* • scaleFactor *Allows you to enter a value between 0-1 by which to scale the icons. Thus you can size-down all icons from the skin.* ***Notice:** Backend Module icons are not affected by this scaling factor* **Example code listing:**

```
    // Setting up auto detection of alternative icons:
$TBE_STYLES['skinImgAutoCfg']=array(
    'absDir' => t3lib_extMgm::extPath($_EXTKEY).'icons/',
    'relDir' => t3lib_extMgm::extRelPath($_EXTKEY).'icons/',
    'forceFileExtension' => 'png',
    'scaleFactor' => 2/3,
);
```

Key	Subkeys	Description
skinImg	[icon reference]	Manual configuration of icon alternatives. This is needed especially for backend module icons since they are not possible to skin with the feature "skinImgAutoCfg" which is otherwise recommended instead of manual configuration. Generally each subkey is a reference to the icon, relative to TYPO3 main dir (e.g. "gfx/ol/blank.gif") or if from an extension, relative to "ext/[extension key]/" folder. For modules the key is special. It is prefixed "MOD:" and then the module key. For example "MOD:web/website.gif" or "MOD:web_uphotomarathon/tab_icon.gif" For examples, see code listing below.
border		Path to an alternative HTML file instead of the default "typo3/border.html" which is displayed between the page tree and the right frame.

Here is an example code listing for how most of these values can be set up in a "ext_tables.php" file for an extension:

```


if (TYPO3_MODE=='BE')     {

    $presetSkinImgs = is_array($TBE_STYLES['skinImg']) ? $TBE_STYLES['skinImg'] : array();     // Means, support for other extensions to add own icons...

    $TBE_STYLES['mainColors'] = array(
        'bgColor' => '#EDF4EB',
        'bgColor2' => '#7C8591',
        'bgColor3' => '#E4E8F2',
        'bgColor4' => '#92AA8B',
        'bgColor5' => '#A5B7C1',
        'bgColor6' => '#C7BF81',
        'hoverColor' => '#800000'
    );

        // Setting the relative path to the extension in temp. variable:
    $temp_eP = t3lib_extMgm::extRelPath($_EXTKEY);

        // Setting login box image rotation folder:
    $TBE_STYLES['loginBoxImage_rotationFolder'] = $temp_eP.'loginimages/';

        // Setting up stylesheets (See template() constructor!)
    $TBE_STYLES['styleSheetFile_post'] = $temp_eP.'stylesheet_post.css';     // Additional stylesheet. Set AFTER any in-document styles
```

```

        // Alternative dimensions for frameset sizes:
    $TBE_STYLES['dims']['leftMenuFrameW']=165;        // Left menu frame width
    $TBE_STYLES['dims']['topFrameH']=35;            // Top frame heigth
    $TBE_STYLES['dims']['shortcutFrameH']=35;        // Shortcut frame height
    $TBE_STYLES['dims']['selMenuFrame']=180;        // Width of the selector box menu frame
    $TBE_STYLES['dims']['navFrameWidth']=350;        // Default navigation frame width

        // Setting roll-over background color for click menus:
        // Notice, this line uses the the 'scriptIDindex' feature to override another value in this array (namely $TBE_STYLES['mainColors']['bgColor5']), for a specific script "typo3/alt_clickmenu.php"
    $TBE_STYLES['scriptIDindex']['typo3/alt_clickmenu.php']['mainColors']['bgColor5']='#E0E7C7';

        // Setting up auto detection of alternative icons:
    $TBE_STYLES['skinImgAutoCfg']=array(
        'absDir' => t3lib_extMgm::extPath($_EXTKEY).'icons/',
        'relDir' => t3lib_extMgm::extRelPath($_EXTKEY).'icons/',
        'forceFileExtension' => 'png',    // Force to look for PNG alternatives...
    );

        // Manual setting up of alternative icons. This is mainly for module icons which has a special prefix:
    $TBE_STYLES['skinImg'] = array_merge($presetSkinImgs, array(
        'gfx/ol/blank.gif' => array('clear.gif','width="27" height="24"'),

        'MOD:web/website.gif'  => array($temp_eP.'icons/module_web.png','width="24" height="24"'),
        'MOD:web_layout/layout.gif'  => array($temp_eP.'icons/module_web_layout.png','width="24" height="24"'),
        'MOD:web_view/view.gif'  => array($temp_eP.'icons/module_web_view.png','width="23" height="24"'),
        'MOD:web_list/list.gif'  => array($temp_eP.'icons/module_web_list.png','width="24" height="24"'),
        'MOD:web_info/info.gif'  => array($temp_eP.'icons/module_web_info.png','width="24" height="24"'),
        'MOD:web_perm/perm.gif'  => array($temp_eP.'icons/module_web_perms.png','width="24" height="24"'),
        'MOD:web_func/func.gif'  => array($temp_eP.'icons/module_web_func.png','width="24" height="24"'),
        'MOD:web_ts/ts1.gif'  => array($temp_eP.'icons/module_web_ts.png','width="24" height="24"'),
        'MOD:web_modules/modules.gif' => array($temp_eP.'icons/module_web_modules.png','width="24" height="24"'),
        'MOD:file/file.gif'  => array($temp_eP.'icons/module_file.png','width="24" height="24"'),
        'MOD:file_list/list.gif'  => array($temp_eP.'icons/module_file_list.png','width="24" height="24"'),
        'MOD:file_images/images.gif'  => array($temp_eP.'icons/module_file_images.png','width="24" height="24"'),
        'MOD:doc/document.gif'  => array($temp_eP.'icons/module_doc.png','width="24" height="24"'),
        'MOD:user/user.gif'  => array($temp_eP.'icons/module_user.png','width="24" height="24"'),
        'MOD:user_task/task.gif'  => array($temp_eP.'icons/module_user_taskcenter.png','width="24" height="24"'),
        'MOD:user_setup/setup.gif'  => array($temp_eP.'icons/module_user_setup.png','width="24" height="24"'),
        'MOD:tools/tool.gif'  => array($temp_eP.'icons/module_tools.png','width="25" height="24"'),
        'MOD:tools_beuser/beuser.gif'  => array($temp_eP.'icons/module_tools_user.png','width="24" height="24"'),
        'MOD:tools_em/em.gif'  => array($temp_eP.'icons/module_tools_em.png','width="24" height="24"'),
        'MOD:tools_dbint/db.gif'  => array($temp_eP.'icons/module_tools_dbint.png','width="25" height="24"'),
    'MOD:tools_config/config.gif'  => array($temp_eP.'icons/module_tools_config.png','width="24" height="24"'),
    'MOD:tools_install/install.gif'  => array($temp_eP.'icons/module_tools_install.png','width="24" height="24"'),
        'MOD:tools_log/log.gif'  => array($temp_eP.'icons/module_tools_log.png','width="24" height="24"'),
    'MOD:tools_txphpmyadmin/thirdparty_db.gif'  => array($temp_eP.'icons/module_tools_phpmyadmin.png','width="24" height="24"'),
    'MOD:tools_isearch/isearch.gif' => array($temp_eP.'icons/module_tools_isearch.png','width="24" height="24"'),
        'MOD:help/help.gif'  => array($temp_eP.'icons/module_help.png','width="23" height="24"'),
        'MOD:help_about/info.gif'  => array($temp_eP.'icons/module_help_about.png','width="25"
height="24"'),
  74:
          'MOD:help_aboutmodules/aboutmodules.gif'  => array($temp_eP.'icons/module_help_aboutmodules.png'
,'width="24" height="24"'),
  75:     ));
  76:
  77:         // Adding icon for photomarathon extensions' backend module, if enabled:
  78:     if (t3lib_extMgm::isloaded('user_photomarathon'))    {
  79:         $TBE_STYLES['skinImg']
['MOD:web_uphotomarathon/tab_icon.gif'] = array($temp_eP.'icons/ext/user_photomarathon/tab_icon.png','wi
dth="24" height="24"');
  80:     }
  81:         // Adding icon for templavoila extensions' backend module, if enabled:
  82:     if (t3lib_extMgm::isloaded('templavoila'))    {
  83:         $TBE_STYLES['skinImg']
['MOD:web_txtemplavoilaM1/moduleicon.gif'] = array($temp_eP.'icons/ext/templavoila/mod1/moduleicon.png',
'width="24" height="24"');
  84:     }
  85: }
```

Notice the last lines from 77-84; they configures alternative icons two extensions, "user_photomarathon" (see testsite package) and "templavoila". Thus the skin can include skinning information for other extensions.

When talking about skinning across extensions another way of making sure that a skin also includes other extensions is shown in line 4 where any values set in $TBE_STYLES['skinImg'] prior to this extension is preserved. Thus other extensions can also autonomously provide support for popular skins by themselves!

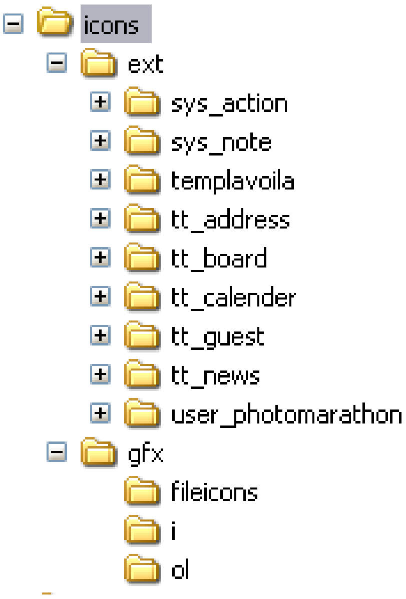
Directory structure for "skinImgAutoCfg" feature

In the example above the directory "icons/" inside the extension is configured to contain the alternative icons which are automatically detected.

Inside of this directory the structure must reflect the *icon reference* of the "skinImg" feature which would have otherwise addressed the icon.

Looking at this screenshot makes it easy to understand. If you want to skin the icon "gfx/closedok.gif" then just put a file with the *same* name (possible as "png" if "forceFileExtension" was set to "png") in the folder "icons/gfx/".

If you have an extension, say, "sys_action" and wants to skin the Action database record icon (sys_action.gif) simply put an alternative file for "sys_action.gif" into the folder "ext/sys_action/"



If you look in the "icons" folder of the "skin360" extension you can also see that all the module icons are located there - but notice that they are *manually* referenced in the "skinImg" key!

How to make your extensions compatible with skinning

Basically, your extensions backend modules will be skinnable by TYPO3 as long as you use the template class to create output. This is the case in well-made extensions so by default you should expect no problems.

However your usage of *icons* is another story. Here you have to pass all icon filenames and sizes to a function, t3lib_iconWorks::skinImg(), which will either return the input value *or* any alternative values should an alternative icon have been configured by a skin extension.

There are two types of icons you can encounter:

- Database record icons
- Any other icon for your interfaces.

Database record icons are not a problem. For a long time the consensus has been that if you want to create an icon for a database record you do like this:

```
$iconImg = t3lib_iconWorks::getIconImage('sys_note',$row,$GLOBALS['BACK_PATH'],' title="This is my
icon"');
```

As long as you keep using the t3lib_iconWorks::getIconImage() function the icons will be skinned.

Any other icon you might use - either from inside the extension or e.g. typo3/gfx/ - should now be created like this:

```
$iconImg = '<img'.t3lib_iconWorks::skinImg($GLOBALS['BACK_PATH'],'gfx/edit2.gif','width="11"
height="12"').' title="My Icon" alt="" />';
```

This is contrary to the non-skinned state which would look like this:

```
$iconImg = '<img src="gfx/edit2.gif" width="11" height="12" title="My Icon" alt="" />';
```

So as you can see it is the src, width and height attributes which are affected!

Skinning support for local extension icons

If you want to add skinning support for icons found inside the extension itself, then use this method:

```
$iconImg = '<img'.t3lib_iconWorks::skinImg($GLOBALS['BACK_PATH'],t3lib_extMgm::extRelPath('templavoila')
.'mod1/greenled.gif','').' title="Rule applies" border="0" alt="" align="absmiddle" />';
```

The main thing to notice is that the relative path to the extension is prefixed the icon name:

```
t3lib_extMgm::extRelPath('templavoila').'mod1/greenled.gif'
```

Finding CSS selectors for the backend documents

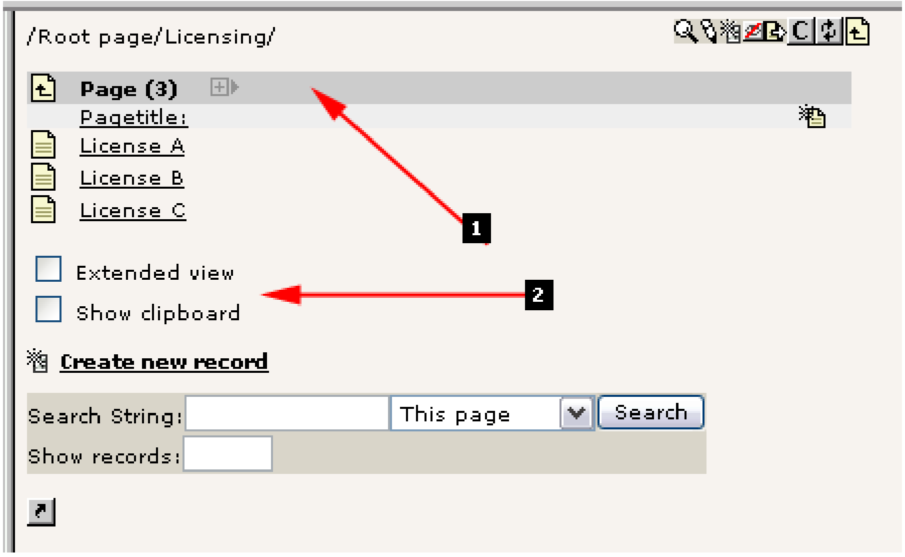
In the process of skinning TYPO3 with CSS styles you should proceed from general to specific. This means

- First, create styles for main elements like BODY, H2, H3, P, PRE, INPUT etc. making the interface look as you want in *general*.
- Secondly, create specific style rules for specific scripts as needed.

If you look inside "typo3/stylesheet.css" you will see that this is the way that stylesheet proceeds. In fact it might not be so bad an idea to take this stylesheet as an example for your own! In that case you can either choose to totally substitute the default stylesheet, "typo3/stylesheet.css", with a new one (by $TBE_STYLES['stylesheet']) or simply create an additional stylesheet (set up by $TBE_STYLES['styleSheetFile_post']) which will be included as the last one - and in this stylesheet you override any of the previous rules you want to change (recommended method).

Addressing specific elements in the backend

Lets say you want to specifically style the two elements shown in this image:

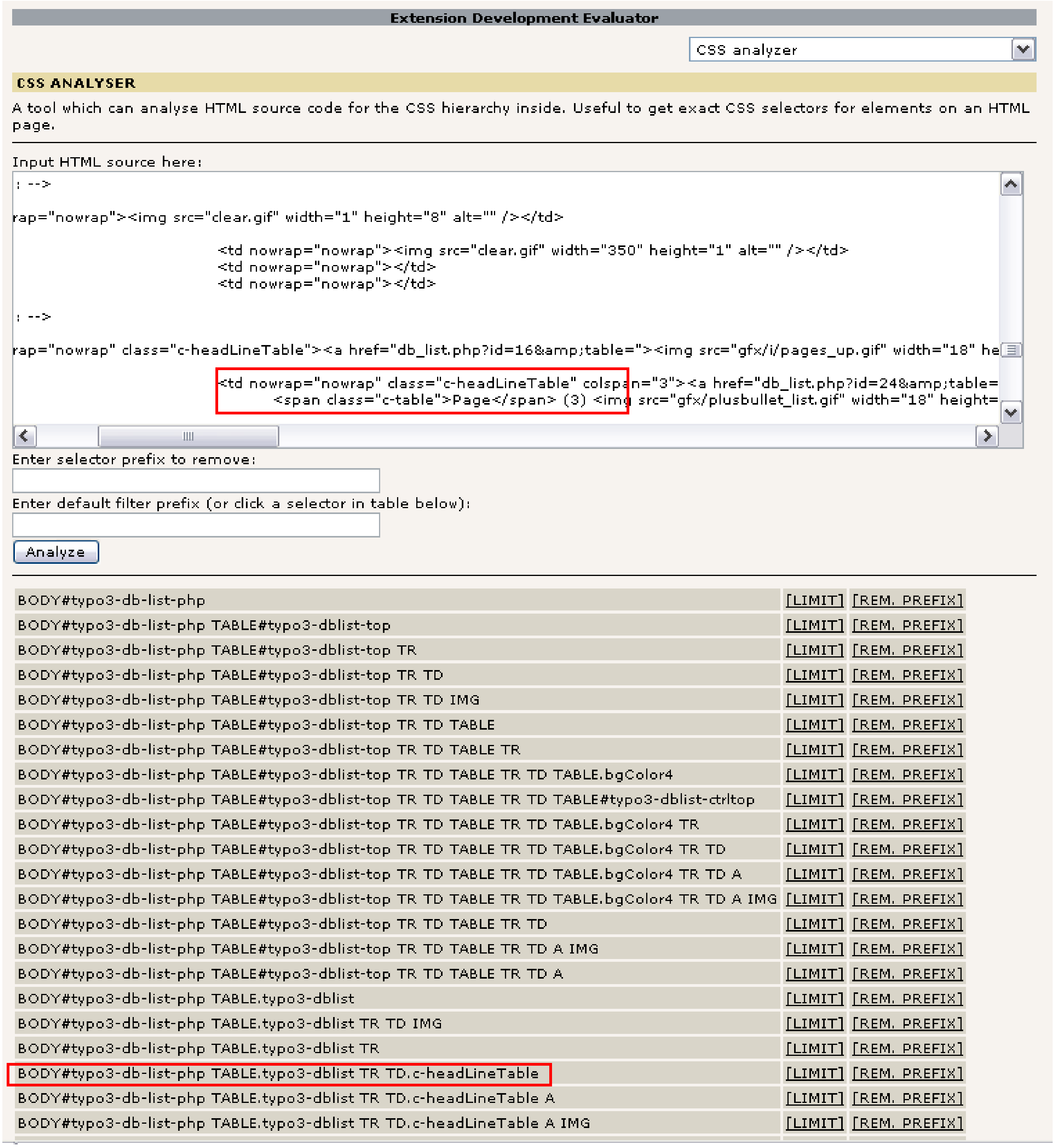


- #1 should be blueish in the background
- #2 should have a dotted border around

What you do is this:

- Right-click the frame, select “Show HTML source” or whatever your browser allows you.
- Paste the HTML source of the script into the tool “CSS analyzer” found in the extension “extdeveval” - this will analyse the hierarchy of CSS selectors.
- Find your selector, write CSS rules!

In this screenshot you can see how I have pasted the HTML source of the script into the tool mentioned and in return I get a nice overview of the CSS selectors inside:



CSS ANALYSER

A tool which can analyse HTML source code for the CSS hierarchy inside. Useful to get exact CSS selectors for elements on an HTML page.

Input HTML source here:

```
; -->

rap="nowrap"><img src="clear.gif" width="1" height="8" alt="" /></td>

                        <td nowrap="nowrap"><img src="clear.gif" width="350" height="1" alt="" /></td>
                        <td nowrap="nowrap"></td>
                        <td nowrap="nowrap"></td>

; -->

rap="nowrap" class="c-headLineTable"><a href="db_list.php?id=16&amp;table="><img src="gfx/i/pages_up.gif" width="18" he

                        <td nowrap="nowrap" class="c-headLineTable" colspan="3"><a href="db_list.php?id=24&amp;table=
                               <span class="c-table">Page</span> (3) <img src="gfx/plusbullet_list.gif" width="18" height=
```

Enter selector prefix to remove:

Enter default filter prefix (or click a selector in table below):

BODY#typo3-db-list-php	[LIMIT]	[REM. PREFIX]
BODY#typo3-db-list-php TABLE#typo3-dblist-top	[LIMIT]	[REM. PREFIX]
BODY#typo3-db-list-php TABLE#typo3-dblist-top TR	[LIMIT]	[REM. PREFIX]
BODY#typo3-db-list-php TABLE#typo3-dblist-top TR TD	[LIMIT]	[REM. PREFIX]
BODY#typo3-db-list-php TABLE#typo3-dblist-top TR TD IMG	[LIMIT]	[REM. PREFIX]
BODY#typo3-db-list-php TABLE#typo3-dblist-top TR TD TABLE	[LIMIT]	[REM. PREFIX]
BODY#typo3-db-list-php TABLE#typo3-dblist-top TR TD TABLE TR	[LIMIT]	[REM. PREFIX]
BODY#typo3-db-list-php TABLE#typo3-dblist-top TR TD TABLE TR TD TABLE.bgColor4	[LIMIT]	[REM. PREFIX]
BODY#typo3-db-list-php TABLE#typo3-dblist-top TR TD TABLE TR TD TABLE#typo3-dblist-ctrltop	[LIMIT]	[REM. PREFIX]
BODY#typo3-db-list-php TABLE#typo3-dblist-top TR TD TABLE TR TD TABLE.bgColor4 TR	[LIMIT]	[REM. PREFIX]
BODY#typo3-db-list-php TABLE#typo3-dblist-top TR TD TABLE TR TD TABLE.bgColor4 TR TD	[LIMIT]	[REM. PREFIX]
BODY#typo3-db-list-php TABLE#typo3-dblist-top TR TD TABLE TR TD TABLE.bgColor4 TR TD A	[LIMIT]	[REM. PREFIX]
BODY#typo3-db-list-php TABLE#typo3-dblist-top TR TD TABLE TR TD TABLE.bgColor4 TR TD A IMG	[LIMIT]	[REM. PREFIX]
BODY#typo3-db-list-php TABLE#typo3-dblist-top TR TD TABLE TR TD	[LIMIT]	[REM. PREFIX]
BODY#typo3-db-list-php TABLE#typo3-dblist-top TR TD TABLE TR TD A IMG	[LIMIT]	[REM. PREFIX]
BODY#typo3-db-list-php TABLE#typo3-dblist-top TR TD TABLE TR TD A	[LIMIT]	[REM. PREFIX]
BODY#typo3-db-list-php TABLE.typo3-dblist	[LIMIT]	[REM. PREFIX]
BODY#typo3-db-list-php TABLE.typo3-dblist TR TD IMG	[LIMIT]	[REM. PREFIX]
BODY#typo3-db-list-php TABLE.typo3-dblist TR	[LIMIT]	[REM. PREFIX]
BODY#typo3-db-list-php TABLE.typo3-dblist TR TD.c-headLineTable	[LIMIT]	[REM. PREFIX]
BODY#typo3-db-list-php TABLE.typo3-dblist TR TD.c-headLineTable A	[LIMIT]	[REM. PREFIX]
BODY#typo3-db-list-php TABLE.typo3-dblist TR TD.c-headLineTable A IMG	[LIMIT]	[REM. PREFIX]

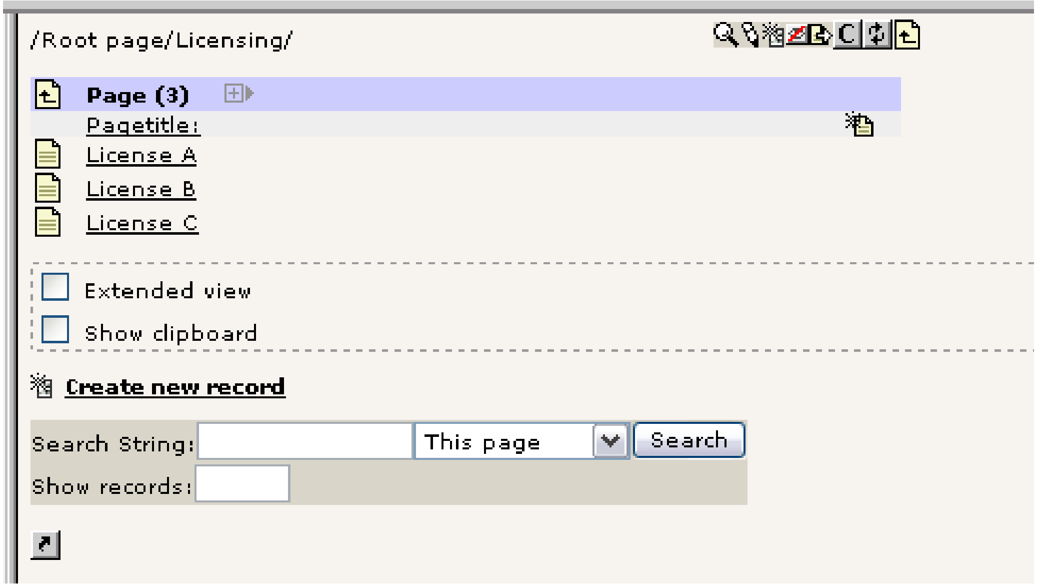
In less than 10 seconds this has allowed me to spot that the exact address of the header cell is “BODY#typo3-db-list-php TABLE.typo3-dblist TR TD.c-headLineTable” and I can now add to my stylesheet:

```
BODY#typo3-db-list-php TABLE.typo3-dblist TR TD.c-headLineTable {
      background-color: #ccccff;
}
```

Likewise I could easily find that the two selector boxes were encapsulated in a DIV section which I could address like this:

```
BODY#typo3-db-list-php DIV#typo3-listOptions {
      border: dotted 1px #999999;
}
```

The result was:

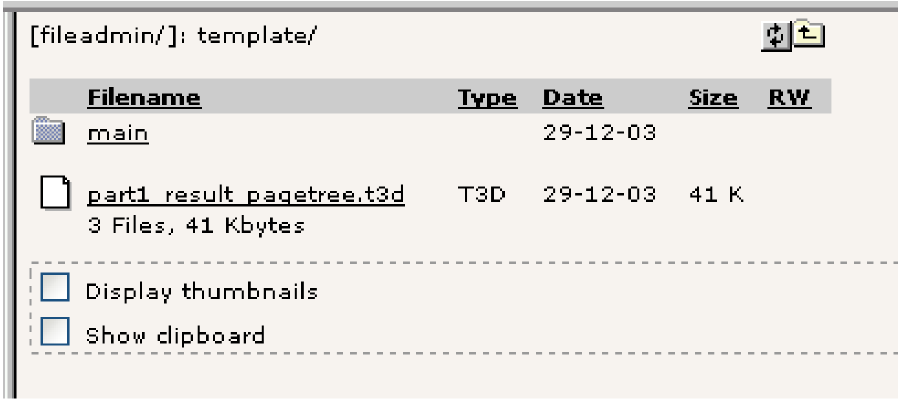

/Root page/Licensing/

Page (3)
Pagetitle:
License A
License B
License C

☐ Extended view
☐ Show clipboard

Now, as you can see the selector contained “BODY#typo3-db-list-php” which is a specific address to the Web > List module (using its script ID!). If I wanted my styles to be more general so also the File > Filelist module would affected, then I could (in this case) remove the BODY#... part:

```
DIV#typo3-listOptions {
      border: dotted 1px #999999;
}
```


[fileadmin/]: template/

Filename	Type	Date	Size	RW
main		29-12-03		
part1_result_pagetree.t3d	T3D	29-12-03	41 K	
3 Files, 41 Kbytes				

☐ Display thumbnails
☐ Show clipboard

Skinning database record icons with variations

Introduction

Database records in TYPO3 has and icon associated which can be shown in the interface. But the icon might change according to internal settings in the record; other icons might be used as alternatives to the default and for each possible icon certain “states” might reflect on how the icon look. For instance, if a record has the “hidden” flag set, the icon should be gray with a red cross over in order to reflect this state visually.

Until version 3.6.0 TYPO3 has automatically calculated new versions of database icons when needed by the system. Thus you needed to supply only one icon - all variations would be automatically generated and stored in typo3temp/. However this auto generation depended on GDlib with gif-support and that has been a well known problem for many years since not everyone has access to these features.

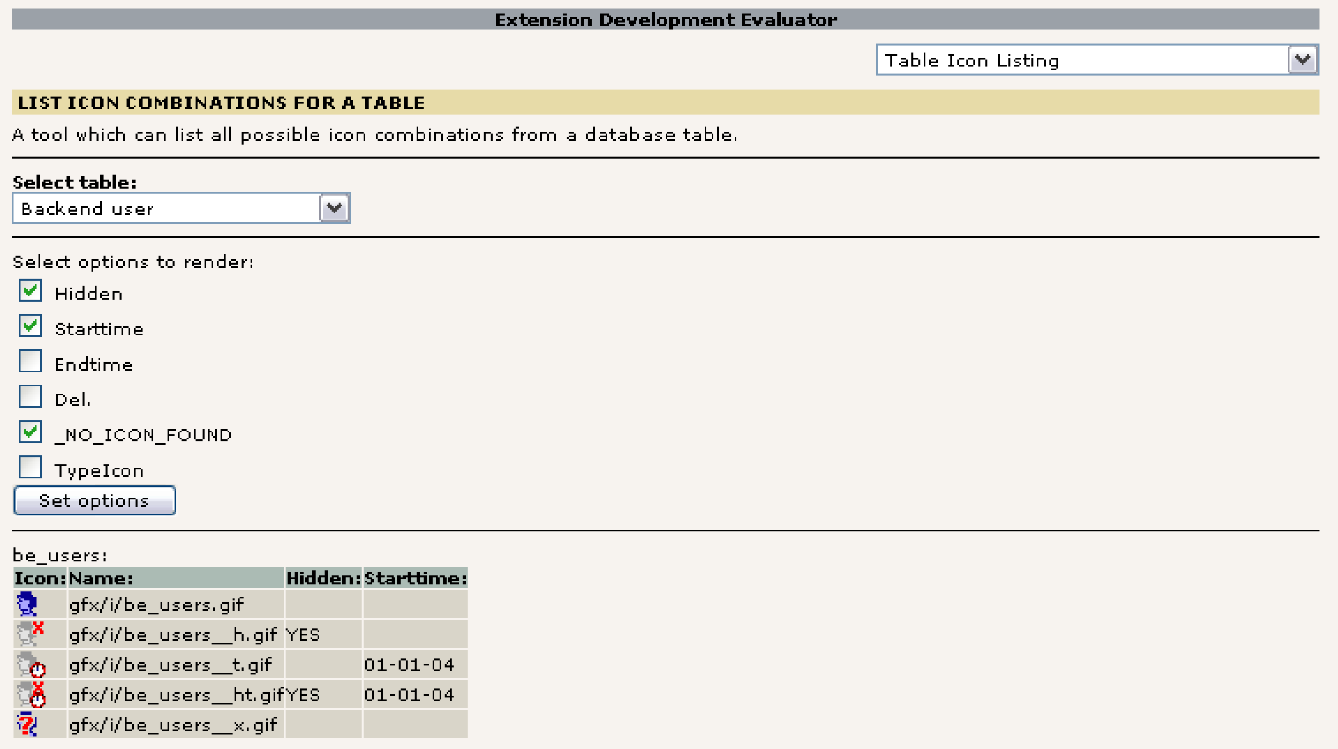
In TYPO3 3.6.0 the automatic generation is disabled (can be enabled by setting $TYPO3_CONF_VARS['GFX']['noIconProc']=0) and instead most icons have their most used states shipped along pre-rendered instead.

This solution not only solves the last mandatory dependency on GDlib for TYPO3 but also provides a way for skinning of various icon states - since skinned icons would be too hard to do processing for!

Pre-rendered icon states

The number of variations for an icon of a database record depends on configuration in $TCA. The most easy way to get an overview of the icons you would need to produce as variations is to use the tool "Table Icon Listing" in the "extdeveval" extension.

This is an example of how that tool shows the icons for "Backend Users" and their variations:



Extension Development Evaluator

Table Icon Listing

LIST ICON COMBINATIONS FOR A TABLE

A tool which can list all possible icon combinations from a database table.

Select table: Backend user

Select options to render:

- ☑ Hidden
- ☑ Starttime
- ☐ Endtime
- ☐ Del.
- ☑ _NO_ICON_FOUND
- ☐ TypeIcon

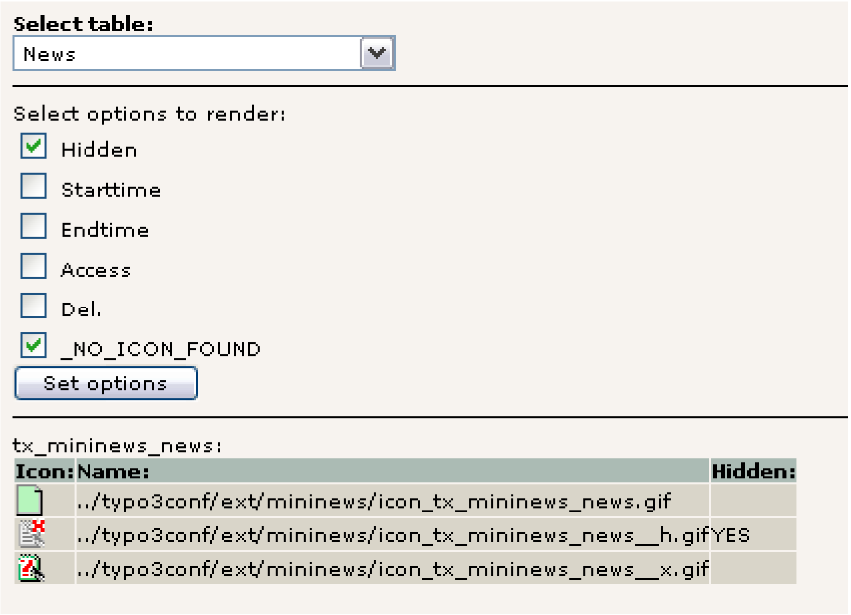
be_users:

Icon:	Name:	Hidden:	Starttime:
	gfx/i/be_users.gif		
	gfx/i/be_users__h.gif	YES	
	gfx/i/be_users__t.gif		01-01-04
	gfx/i/be_users__ht.gif	YES	01-01-04
	gfx/i/be_users__x.gif		

Notice, the default icon is "gfx/i/be_users.gif"

- If the hidden flag is set, the icon name is "be_users__h.gif"
- If the starttime is set, the icon name is "be_users__t.gif"
- If both starttime and hidden flag is set, the icon name is "be_users__ht.gif"
- If an icon carries a state that is not found, then show "be_users__x.gif" (default icon for a state that does not have an icon. If this icon is not set a generalized default icon is shown; thus a record with a special state will never be shown just plain!)

For an extension like "mininews" we can perform the same analysis:



Select table: News

Select options to render:

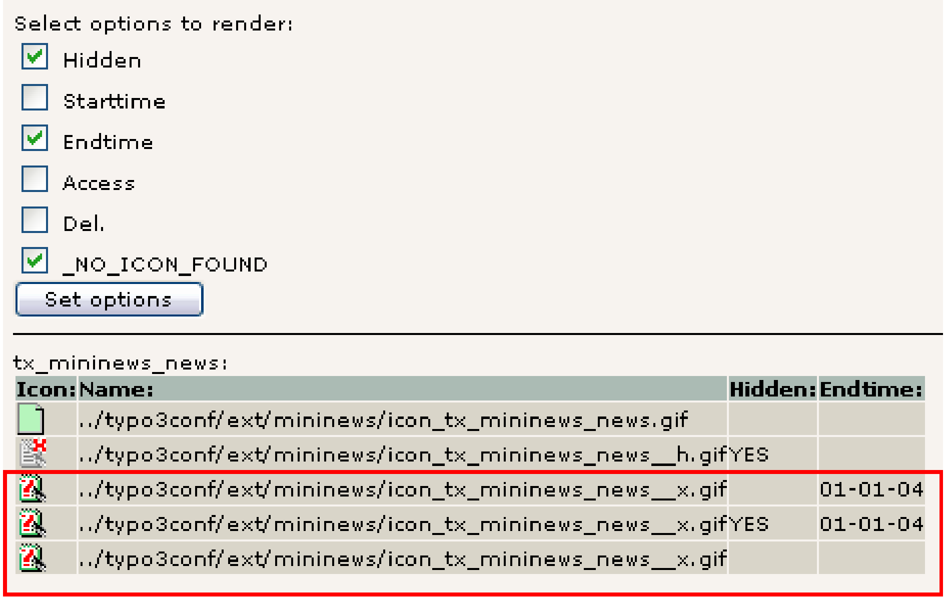
☑ Hidden
☐ Starttime
☐ Endtime
☐ Access
☐ Del.
☑ _NO_ICON_FOUND

tx_mininews_news:

Icon:	Name:	Hidden:
	../typo3conf/ext/mininews/icon_tx_mininews_news.gif	
	../typo3conf/ext/mininews/icon_tx_mininews_news__h.gif	YES
	../typo3conf/ext/mininews/icon_tx_mininews_news__x.gif	

Again, notice how the variations over "icon_tx_mininews_news.gif" is prefixed with "flags" like "__h" and "__x"

If we enable more of the render options we might eventually hit a combination of options which is *not* found pre-rendered though:



Select options to render:

☑ Hidden
☐ Starttime
☑ Endtime
☐ Access
☐ Del.
☑ _NO_ICON_FOUND

tx_mininews_news:

Icon:	Name:	Hidden:	Endtime:
	../typo3conf/ext/mininews/icon_tx_mininews_news.gif		
	../typo3conf/ext/mininews/icon_tx_mininews_news__h.gif	YES	
	../typo3conf/ext/mininews/icon_tx_mininews_news__x.gif		01-01-04
	../typo3conf/ext/mininews/icon_tx_mininews_news__x.gif	YES	01-01-04
	../typo3conf/ext/mininews/icon_tx_mininews_news__x.gif		

As you can see the "endtime" flag has no icon associated with it.

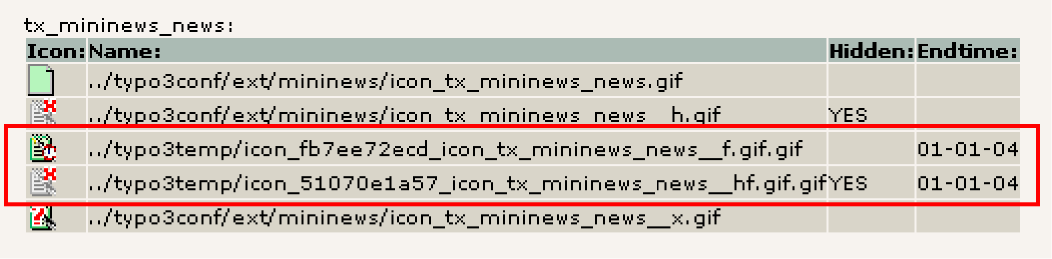
Automatic creation of pre-made variations

In order to create variations for inclusion in your extensions (for the default icon) you can enable the rendering of icons if you like (in localconf.php):

```
$TYPO3_CONF_VARS['GFX']['noIconProc']=0;
```

Then you reload the "Table Icon Listing" and the icons are generated in typo3temp/:

If you want the new icons to be included in the extension you simply

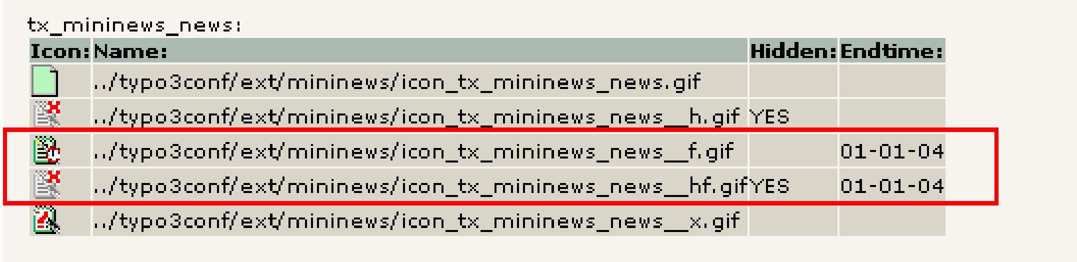


tx_mininews_news:

Icon:	Name:	Hidden:	Endtime:
	../typo3conf/ext/mininews/icon_tx_mininews_news.gif		
	../typo3conf/ext/mininews/icon_tx_mininews_news__h.gif	YES	
	../typo3temp/icon_fb7ee72ecd_icon_tx_mininews_news__f.gif.gif		01-01-04
	../typo3temp/icon_51070e1a57_icon_tx_mininews_news__hf.gif.gif	YES	01-01-04
	../typo3conf/ext/mininews/icon_tx_mininews_news__x.gif		

- Move them from typo3temp/ into the extension folder (here "typo3conf/ext/mininews/")
- Rename them to the expected names, e.g. "icon_fb7ee72ecd_icon_tx_mininews_news__f.gif.gif" to "icon_tx_mininews_news__f.gif" (remember to also remove the "double-gif" in the extension!)

And after another reload you will be assured that the icon is found correctly:



tx_mininews_news:

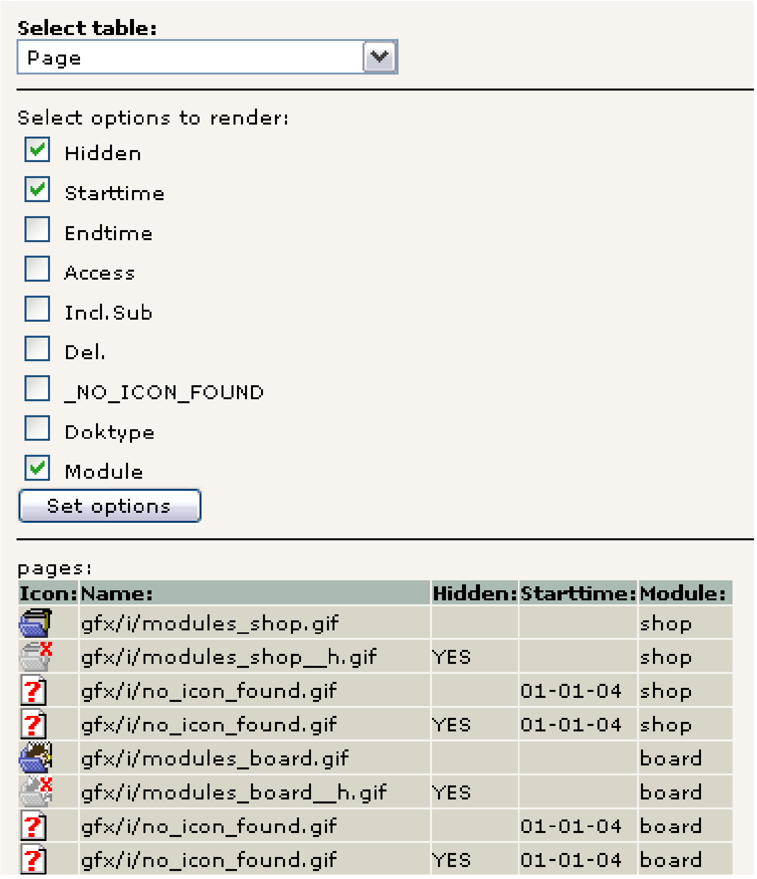
Icon:	Name:	Hidden:	Endtime:
	../typo3conf/ext/mininews/icon_tx_mininews_news.gif		
	../typo3conf/ext/mininews/icon_tx_mininews_news__h.gif	YES	
	../typo3conf/ext/mininews/icon_tx_mininews_news__f.gif		01-01-04
	../typo3conf/ext/mininews/icon_tx_mininews_news__hf.gif	YES	01-01-04
	../typo3conf/ext/mininews/icon_tx_mininews_news__x.gif		

(**Tip for code hackers:** Inside "ext/extdeveval/mod1/class.tx_extdeveval_iconlister.php there is a line with a function call, "$this->renameIconsInTypo3Temp();" which is commented out - if you uncomment this function call it will rename icons made in typo3temp/ to filenames that can be copied directly into the extension you are making. Basically this removes "icon_fb7ee...." from the temporary file!)

Limits to number of pre-made icons

Since the number of combinations can be staggering you might often have to settle for a compromise where you define which states are the most likely to occur and then give those priority when you create variations - otherwise you might have to make hundreds of icons!

Thus you can find that the pages table does not have pre-made icons for all "Module" icons. Only the "hidden" state has been considered general enough to allow for a pre-made icon - enabling starttime results in a "no_icon_found.gif" version as you can see below:



Select table:
Page

Select options to render:

- ☑ Hidden
- ☑ Starttime
- ☐ Endtime
- ☐ Access
- ☐ Incl.Sub
- ☐ Del.
- ☐ _NO_ICON_FOUND
- ☐ Doktype
- ☑ Module

pages:

Icon:	Name:	Hidden:	Starttime:	Module:
	gfx/i/modules_shop.gif			shop
	gfx/i/modules_shop__h.gif	YES		shop
?	gfx/i/no_icon_found.gif		01-01-04	shop
?	gfx/i/no_icon_found.gif	YES	01-01-04	shop
	gfx/i/modules_board.gif			board
	gfx/i/modules_board__h.gif	YES		board
?	gfx/i/no_icon_found.gif		01-01-04	board
?	gfx/i/no_icon_found.gif	YES	01-01-04	board

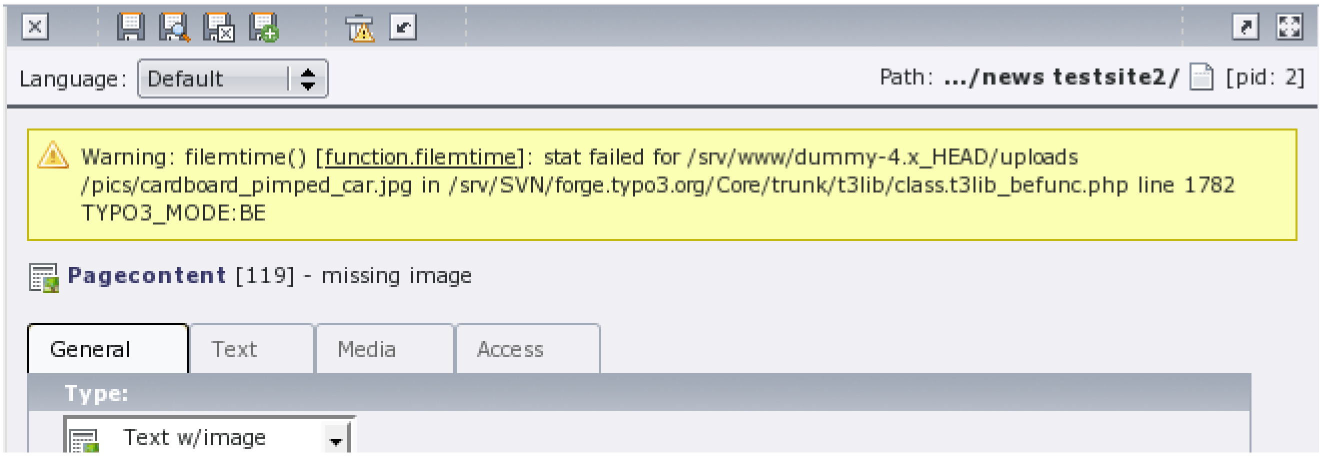
Error and Exception Handling

Introduction

Since version 4.3.0 TYPO3 comes with an build-in error and exception handling system. Admins can configure how errors and exceptions should be displayed in Backend and Frontend. Errors and exception can be logged to all available logging systems in TYPO3 including t3lib_div::syslog() which is – among other features - able to send error messages by mail (see example setups below).

Screenshots

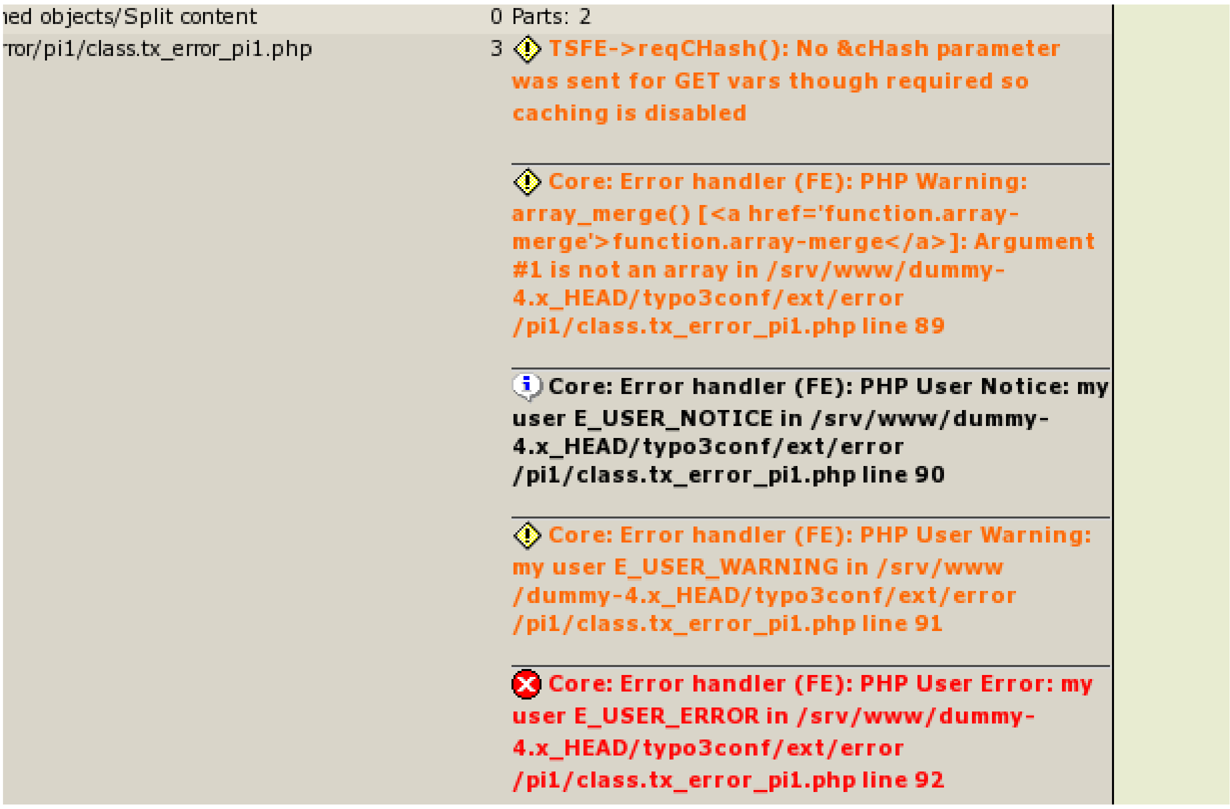
A PHP warning in the Backend:



Language: Default
Path: .../news testsite2/ [pid: 2]
Pagecontent [119] - missing image
General
Text
Media
Access
Type:
Text w/image

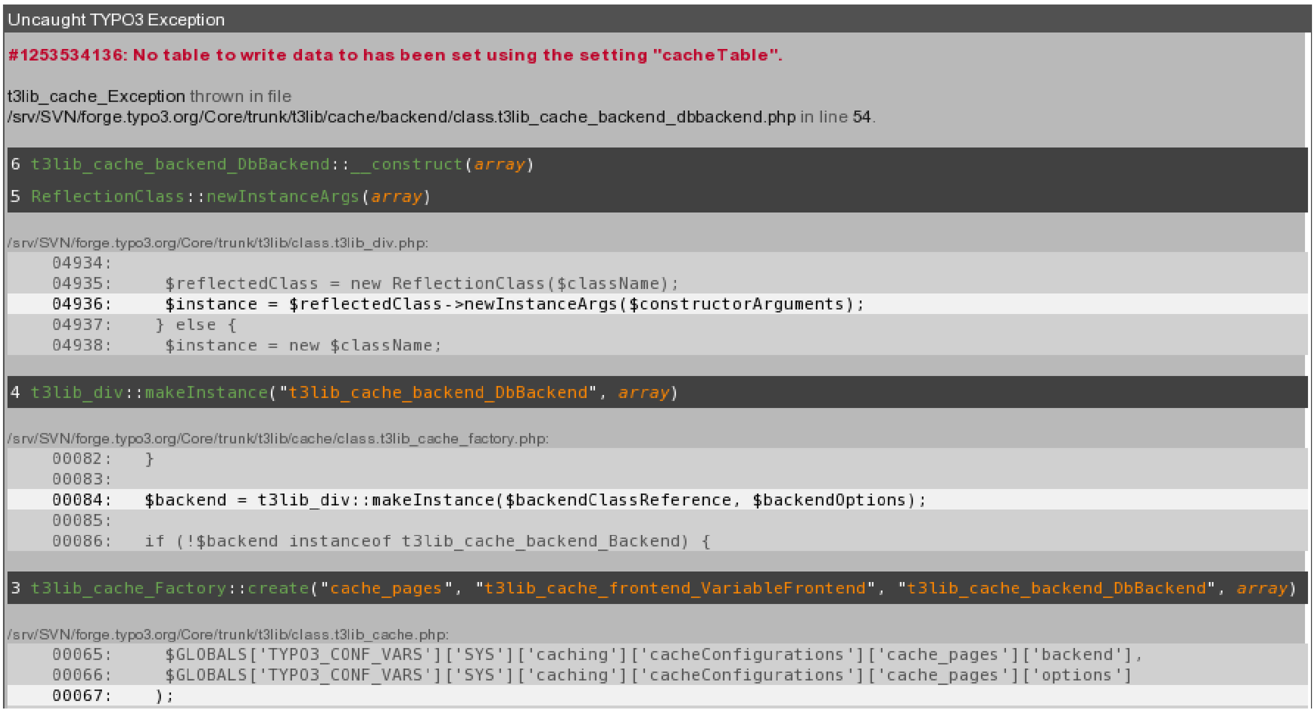
Warning: filemtime() [function.filemtime]: stat failed for /srv/www/dummy-4.x_HEAD/uploads/pics/cardboard_pimped_car.jpg in /srv/SVN/forge.typo3.org/Core/trunk/t3lib/class.t3lib_befunc.php line 1782 TYPO3_MODE:BE

Some PHP errors in the adminpanel:



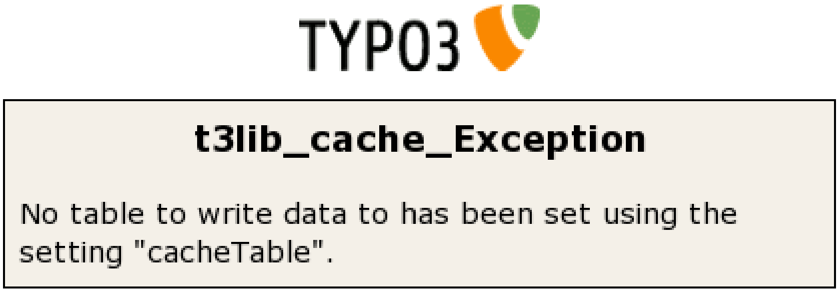
```
ıed objects/Split content                0 Parts: 2
rror/pi1/class.tx_error_pi1.php          3 TSFE->reqCHash(): No &cHash parameter
                                           was sent for GET vars though required so
                                           caching is disabled

                                           Core: Error handler (FE): PHP Warning:
                                           array_merge() [<a href='function.array-
                                           merge'>function.array-merge</a>]: Argument
                                           #1 is not an array in /srv/www/dummy-
                                           4.x_HEAD/typo3conf/ext/error
                                           /pi1/class.tx_error_pi1.php line 89

                                           Core: Error handler (FE): PHP User Notice: my
                                           user E_USER_NOTICE in /srv/www/dummy-
                                           4.x_HEAD/typo3conf/ext/error
                                           /pi1/class.tx_error_pi1.php line 90

                                           Core: Error handler (FE): PHP User Warning:
                                           my user E_USER_WARNING in /srv/www
                                           /dummy-4.x_HEAD/typo3conf/ext/error
                                           /pi1/class.tx_error_pi1.php line 91

                                           Core: Error handler (FE): PHP User Error: my
                                           user E_USER_ERROR in /srv/www/dummy-
                                           4.x_HEAD/typo3conf/ext/error
                                           /pi1/class.tx_error_pi1.php line 92
```

An exception shown by the debug exception handler:



```
Uncaught TYPO3 Exception
#1253534136: No table to write data to has been set using the setting "cacheTable".

t3lib_cache_Exception thrown in file
/srv/SVN/forge.typo3.org/Core/trunk/t3lib/cache/backend/class.t3lib_cache_backend_dbbackend.php in line 54.

6 t3lib_cache_backend_DbBackend::__construct(array)
5 ReflectionClass::newInstanceArgs(array)

/srv/SVN/forge.typo3.org/Core/trunk/t3lib/class.t3lib_div.php:
    04934:
    04935:      $reflectedClass = new ReflectionClass($className);
    04936:      $instance = $reflectedClass->newInstanceArgs($constructorArguments);
    04937:     } else {
    04938:      $instance = new $className;

4 t3lib_div::makeInstance("t3lib_cache_backend_DbBackend", array)

/srv/SVN/forge.typo3.org/Core/trunk/t3lib/cache/class.t3lib_cache_factory.php:
    00082:    }
    00083:
    00084:    $backend = t3lib_div::makeInstance($backendClassReference, $backendOptions);
    00085:
    00086:    if (!$backend instanceof t3lib_cache_backend_Backend) {

3 t3lib_cache_Factory::create("cache_pages", "t3lib_cache_frontend_VariableFrontend", "t3lib_cache_backend_DbBackend", array)

/srv/SVN/forge.typo3.org/Core/trunk/t3lib/class.t3lib_cache.php:
    00065:      $GLOBALS['TYPO3_CONF_VARS']['SYS']['caching']['cacheConfigurations']['cache_pages']['backend'],
    00066:      $GLOBALS['TYPO3_CONF_VARS']['SYS']['caching']['cacheConfigurations']['cache_pages']['options']
    00067:     );
```

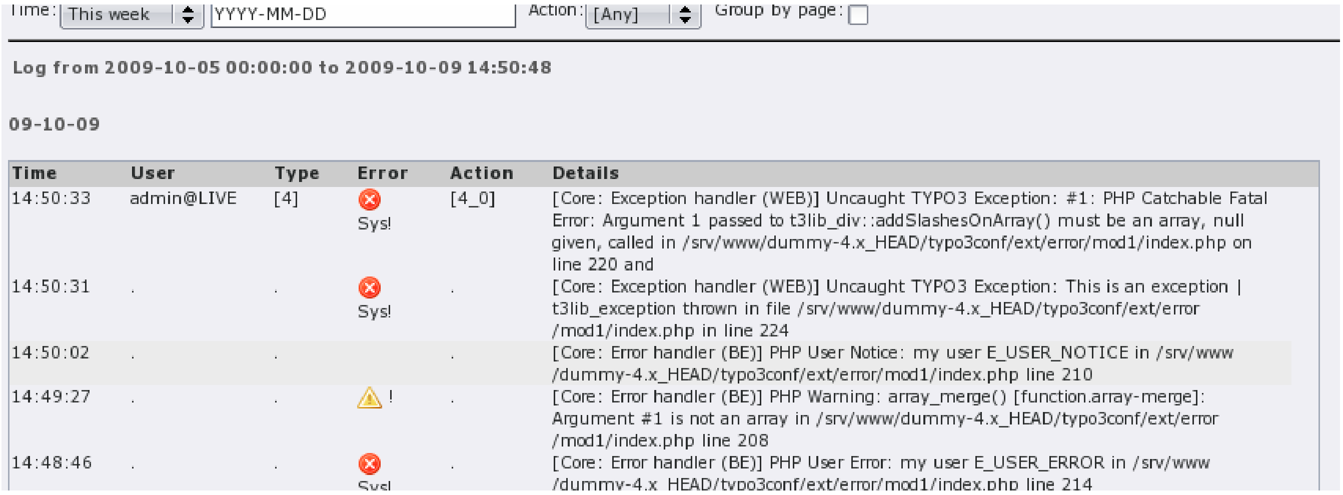
Same exception shown by the production exception handler:



t3lib_cache_Exception

No table to write data to has been set using the setting "cacheTable".

Errors and exceptions shown in belog (Tools->Log):

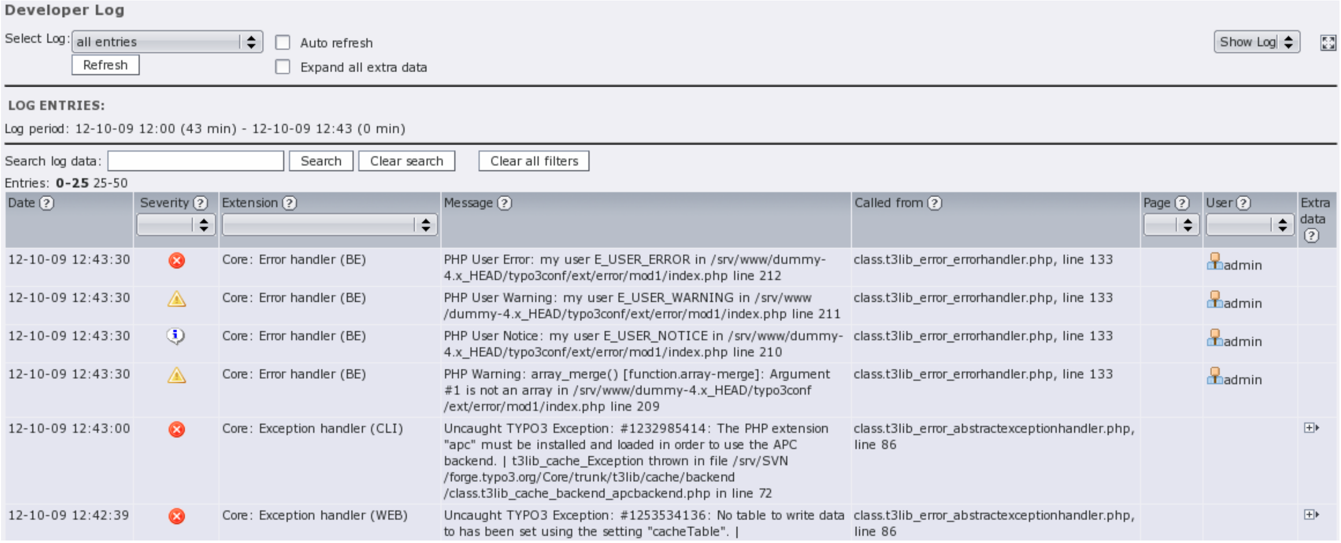


Log from 2009-10-05 00:00:00 to 2009-10-09 14:50:48

09-10-09

Time	User	Type	Error	Action	Details
14:50:33	admin@LIVE	[4]	✖ Sys!	[4_0]	[Core: Exception handler (WEB)] Uncaught TYPO3 Exception: #1: PHP Catchable Fatal Error: Argument 1 passed to t3lib_div::addSlashesOnArray() must be an array, null given, called in /srv/www/dummy-4.x_HEAD/typo3conf/ext/error/mod1/index.php on line 220 and
14:50:31	.	.	✖ Sys!	.	[Core: Exception handler (WEB)] Uncaught TYPO3 Exception: This is an exception | t3lib_exception thrown in file /srv/www/dummy-4.x_HEAD/typo3conf/ext/error/mod1/index.php in line 224
14:50:02	.	.		.	[Core: Error handler (BE)] PHP User Notice: my user E_USER_NOTICE in /srv/www/dummy-4.x_HEAD/typo3conf/ext/error/mod1/index.php line 210
14:49:27	.	.	⚠ !	.	[Core: Error handler (BE)] PHP Warning: array_merge() [function.array-merge]: Argument #1 is not an array in /srv/www/dummy-4.x_HEAD/typo3conf/ext/error/mod1/index.php line 208
14:48:46	.	.	✖ Sys!	.	[Core: Error handler (BE)] PHP User Error: my user E_USER_ERROR in /srv/www/dummy-4.x_HEAD/typo3conf/ext/error/mod1/index.php line 214

Errors and exceptions shown in the devLog (using extension “devlog”):



Developer Log

Select Log: all entries ☐ Auto refresh
Refresh ☐ Expand all extra data
Show Log

LOG ENTRIES:

Log period: 12-10-09 12:00 (43 min) - 12-10-09 12:43 (0 min)

Search log data: Search Clear search Clear all filters

Entries: **0-25** 25-50

Date	Severity	Extension	Message	Called from	Page	User	Extra data
12-10-09 12:43:30	✖	Core: Error handler (BE)	PHP User Error: my user E_USER_ERROR in /srv/www/dummy-4.x_HEAD/typo3conf/ext/error/mod1/index.php line 212	class.t3lib_error_errorhandler.php, line 133		admin	
12-10-09 12:43:30	⚠	Core: Error handler (BE)	PHP User Warning: my user E_USER_WARNING in /srv/www/dummy-4.x_HEAD/typo3conf/ext/error/mod1/index.php line 211	class.t3lib_error_errorhandler.php, line 133		admin	
12-10-09 12:43:30	ℹ	Core: Error handler (BE)	PHP User Notice: my user E_USER_NOTICE in /srv/www/dummy-4.x_HEAD/typo3conf/ext/error/mod1/index.php line 210	class.t3lib_error_errorhandler.php, line 133		admin	
12-10-09 12:43:30	⚠	Core: Error handler (BE)	PHP Warning: array_merge() [function.array-merge]: Argument #1 is not an array in /srv/www/dummy-4.x_HEAD/typo3conf/ext/error/mod1/index.php line 209	class.t3lib_error_errorhandler.php, line 133		admin	
12-10-09 12:43:00	✖	Core: Exception handler (CLI)	Uncaught TYPO3 Exception: #1232985414: The PHP extension "apc" must be installed and loaded in order to use the APC backend. | t3lib_cache_Exception thrown in file /srv/SVN/forge.typo3.org/Core/trunk/t3lib/cache/backend/class.t3lib_cache_backend_apcbackend.php in line 72	class.t3lib_error_abstractexceptionhandler.php, line 86			⊞
12-10-09 12:42:39	✖	Core: Exception handler (WEB)	Uncaught TYPO3 Exception: #1253534136: No table to write data to has been set using the setting "cacheTable". |	class.t3lib_error_abstractexceptionhandler.php, line 86			⊞

Configuration

All configuration options related to error and exception handling are found in $TYPO3_CONF_VARS[SYS]:

Key	Datatype	Description
displayErrors	integer	Configures whether PHP errors should be displayed. • 0 = Do not display any PHP error messages. Overrides the value of “exceptionalErrors” and sets it to 0 (= no errors are turned into exceptions), the configured “productionExceptionHandler” is used as exception handler • 1 = Display error messages with the registered error handler, the configured “debugExceptionHandler” is used as exception handler • 2 = Display errors only if client matches TYPO3_CONF_VARS[SYS][devIPmask]. If devIPmask matches the users IP address the configured “debugExceptionHandler” is used for exceptions, if not “productionExceptionHandler” will be used. • -1 = Default setting. With this option, you can override the PHP setting "display_errors". If devIPmask matches the users IP address the configured “debugExceptionHandler” is used for exceptions, if not “productionExceptionHandler” will be used.
errorHandler	string	Classname to handle PHP errors. Leave empty to disable error handling. Default: “t3lib_error_ErrorHandler”. This class will register itself as error handler. It is able to write error messages to all available logging systems in TYPO3 (t3lib_div::syslog, t3lib_div::devlog() and to the sys_log table). Additionally the errors can be displayed as flash messages in the Backend or in the adminpanel in Frontend. The flash messages in Backend are only displayed if the error and exception handling is in “debug-mode”, which is the case when the configured “debugExceptionHandler” is registered as exception handler (see: TYPO3_CONF_VARS[SYS][displayErrors]). Errors which are registered as “exceptionalErrors” will be turned into exceptions (to be handled by the configured exceptionHandler).
errorHandlerErrors	integer	The E_* constant that will be handled by the error handler Default: E_ALL ^ E_NOTICE
exceptionalErrors	integer	The E_* constant that will be handled as an exception by the error handler. Default: "E_ALL ^ E_NOTICE ^ E_WARNING ^ E_USER_ERROR ^ E_USER_NOTICE ^ E_USER_WARNING" (4341) and "0" if displayError=0. Refer to the PHP documentation for more details on this value.
productionExceptionHandler	string	Classname to handle exceptions that might happen in the TYPO3-code. Leave empty to disable exception handling. **Default**: “t3lib_error_ProductionExceptionHandler”. This exception handler displays a nice error message when something went wrong. The error message is logged to the configured logs. **Note**: The configured productionExceptionHandler is used if displayErrors is set to “0” or to “-1” and devIPmask doesn't match.

Key	Datatype	Description
debugExceptionHandler	string	Classname to handle exceptions that might happen in the TYPO3-code. Leave empty to disable exception handling. **Default**: "t3lib_error_DebugExceptionHandler". This exception handler displays the complete stack trace of any encountered exception. The error message and the stack trace is logged to the configured logs. **Note**: The configured debugExceptionHandler is used if displayErrors is set to "1" and if displayErrors is "-1" or "2" and the devIPmask matches.
enable_errorDLOG	boolean	Whether errors should be written to the developer log (requires an installed *devlog extension).
enable_exceptionDLOG	boolean	Whether exceptions should be written to the developer log (requires an installed *devlog extension).
syslogErrorReporting	integer	Configures which PHP errors should be logged to the configured syslogs (see: [SYS][systemLog]). If set to "0" no PHP errors are logged to the syslog. Default is "E_ALL ^ E_NOTICE" (6135).
belogErrorReporting	integer	Configures which PHP errors should be logged to the "syslog" table (extension: belog). If set to "0" no PHP errors are logged to the sys_log table. Default is "E_ALL ^ E_NOTICE" (6135).

The table below shows which values can be set by the user and which are set by TYPO3.

Values in black can be changed in localconf.php.

Values in red are set by TYPO3.

displayErrors	errorHandlerErrors	exceptionalErrors	errorHandler	devIPmask	exceptionHandler	display_errors (php_ini)
-1	E_ALL ^ E_NOTICE	E_ALL ^ E_NOTICE ^ E_WARNING ^ E_USER_ERROR ^ E_USER_NOTICE ^ E_USER_WARNING	t3lib_error_ErrorHandler	match	$TYPO3_CONF_VARS['SYS']['debugExceptionHandler']	Not changed
				no match	$TYPO3_CONF_VARS['SYS']['productionExceptionHandler']	
0	E_ALL ^ E_NOTICE	0 (no errors are turned into exceptions)	t3lib_error_ErrorHandler	Doesn't matter	$TYPO3_CONF_VARS['SYS']['productionExceptionHandler']	0 (Off)
1	E_ALL ^ E_NOTICE	E_ALL ^ E_NOTICE ^ E_WARNING ^ E_USER_ERROR ^ E_USER_NOTICE ^ E_USER_WARNING	t3lib_error_ErrorHandler	Doesn't matter	$TYPO3_CONF_VARS['SYS']['debugExceptionHandler']	1 (On)
2	E_ALL ^ E_NOTICE	E_ALL ^ E_NOTICE ^ E_WARNING ^ E_USER_ERROR ^ E_USER_NOTICE ^ E_USER_WARNING	t3lib_error_ErrorHandler	match	$TYPO3_CONF_VARS['SYS']['debugExceptionHandler']	1 (On)
				no match	$TYPO3_CONF_VARS['SYS']['productionExceptionHandler']	0 (Off)

t3lib_error_ErrorHandler

The class t3lib_error_ErrorHandler is the default error handler in TYPO3.

Functions:

- Can be registered for all, or for only a subset of the PHP errors which can be handled by an error handler
- Displays error messages as flash messages in the Backend (if exceptionHandler is set to “t3lib_error_DebugExceptionHandler”). Since flash messages are integrated in the Backend template, PHP messages will not destroy the Backend layout.
- Displays errors as TsLog messages in the adminpanel.
- Logs error messages to t3lib_div::syslog() which is able to write error messages to a file, to the web server's error_log, the system's log and it can send you errors and exceptions in an email. t3lib_div::syslog() offers a hook and can be extended by user-defined logging methods.
- Logs error messages to t3lib_div::devLog() if “enable_errorDLOG” is enabled (depending on the devlog extension used, this might require an existing DB connection).
- Logs error messages to the sys_log table. Logged errors are displayed in the belog extension (Tools->Log) (works only if there is an existing DB connection).

t3lib_error_ProductionExceptionHandler

Functions of t3lib_error_ProductionExceptionHandler

- Shows brief exception message using t3lib_timeTrack::debug_typo3PrintError() which can be manipulated by a hook
- Logs exception messages to t3lib_div::syslog() which is able to write exception messages to a file, to the web server's error_log, the system's log and it can send you errors and exceptions in an email. t3lib_div::syslog() offers a hook an can be extended by user-defined logging methods.
- Logs exception messages to t3lib_div::devlog() if “enable_exceptionDLOG” is enabled (depending on the devlog extension used, this might require an existing DB connection).
- Logs exception messages to the sys_log table. Logged errors are displayed in the belog extension (Tools->Log) (works only if there is an existing DB connection).

t3lib_error_DebugExceptionHandler

Functions of t3lib_error_DebugExceptionHandler

- Shows detailed exception messages and full trace of an exception
- Logs exception messages to t3lib_div::syslog() which is able to write exception messages to a file, to the web server's error_log, the system's log and it can send you errors and exceptions in an email. t3lib_div::syslog() offers a hook an can be extended by user-defined logging methods.
- Logs exception messages to t3lib_div::devlog() if “enable_exceptionDLOG” is enabled (depending on the devlog extension used, this might require an existing DB connection).
- Logs exception messages to the sys_log table. Logged errors are displayed in the belog extension (Tools->Log) (works only if there is an existing DB connection).

Examples

Debugging and development setup

Very verbose configuration which logs and displays all errors and exceptions.

[File: localconf.php]

```
$TYPO3_CONF_VARS['SYS']['displayErrors'] = '1';
$TYPO3_CONF_VARS['SYS']['devIPmask'] = '*';
$TYPO3_CONF_VARS['SYS']['errorHandler'] = 't3lib_error_ErrorHandler';
$TYPO3_CONF_VARS['SYS']['errorHandlerErrors'] = E_ALL ^ E_NOTICE;
$TYPO3_CONF_VARS['SYS']['exceptionalErrors'] = E_ALL ^ E_NOTICE ^ E_WARNING ^ E_USER_ERROR ^
E_USER_NOTICE ^ E_USER_WARNING;
$TYPO3_CONF_VARS['SYS']['debugExceptionHandler'] = 't3lib_error_DebugExceptionHandler';
$TYPO3_CONF_VARS['SYS']['productionExceptionHandler'] = 't3lib_error_DebugExceptionHandler';
$TYPO3_CONF_VARS['SYS']['systemLogLevel'] = '0';
$TYPO3_CONF_VARS['SYS']['systemLog'] =
'mail,test@localhost.local,4;error_log,,2;syslog,LOCAL0,,3;file,/abs/path/to/logfile.log';
$TYPO3_CONF_VARS['SYS']['enable_errorDLOG'] = '1';
$TYPO3_CONF_VARS['SYS']['enable_exceptionDLOG'] = '1';
```

[File: .htaccess]

```
php_flag display_errors on
php_flag log_errors on
php_value error_log /path/to/php_error.log
```

Production setup

Example for a production configuration which displays only errors and exceptions if the devIPmask matches. Errors and exceptions are only logged if their level is at least 2 (=Warning).

[File: localconf.php]

```
$TYPO3_CONF_VARS['SYS']['displayErrors'] = '2';
$TYPO3_CONF_VARS['SYS']['devIPmask'] = '[your.IP.address]';
$TYPO3_CONF_VARS['SYS']['errorHandler'] = 't3lib_error_ErrorHandler';
$TYPO3_CONF_VARS['SYS']['systemLogLevel'] = '2';
$TYPO3_CONF_VARS['SYS']['systemLog'] = 'mail,test@localhost.local,4;error_log,,2;syslog,LOCAL0,,3';
$TYPO3_CONF_VARS['SYS']['enable_errorDLOG'] = '0';
$TYPO3_CONF_VARS['SYS']['enable_exceptionDLOG'] = '0';
$TYPO3_CONF_VARS['SYS']['syslogErrorReporting'] = E_ALL ^ E_NOTICE ^ E_WARNING;
$TYPO3_CONF_VARS['SYS']['belogErrorReporting'] = '0';
```

[File: .htaccess]

```
php_flag display_errors off
php_flag log_errors on
php_value error_log /path/to/php_error.log
```

Performance setup

Since the error and exception handling and also the logging need some performance, here's an example how to disable error and exception handling completely.

[File: localconf.php]

```
$TYPO3_CONF_VARS['SYS']['displayErrors'] = '0';
$TYPO3_CONF_VARS['SYS']['devIPmask'] = '';
$TYPO3_CONF_VARS['SYS']['errorHandler'] = '';
$TYPO3_CONF_VARS['SYS']['debugExceptionHandler'] = '';
$TYPO3_CONF_VARS['SYS']['productionExceptionHandler'] = '';
$TYPO3_CONF_VARS['SYS']['systemLog'] = '';
$TYPO3_CONF_VARS['SYS']['enable_errorDLOG'] = '0';
$TYPO3_CONF_VARS['SYS']['enable_exceptionDLOG'] = '0';
$TYPO3_CONF_VARS['SYS']['syslogErrorReporting'] = '0';
$TYPO3_CONF_VARS['SYS']['belogErrorReporting'] = '0';
```

[File: .htaccess]

```
php_flag display_errors off
php_flag log_errors off
```

Extending the error and exception handling

If you want to register your own error or exception handler, simply include the class and insert its name to “productionExceptionHandler”, “debugExceptionHandler” or “errorHandler”.

Example:

```
$TYPO3_CONF_VARS['SYS']['errorHandler'] = 'myOwnErrorHandler';
$TYPO3_CONF_VARS['SYS']['debugExceptionHandler'] = 'myOwnDebugExceptionHandler';
$TYPO3_CONF_VARS['SYS']['productionExceptionHandler'] = 'myOwnProductionExceptionHandler';
```

An error or exception handler class must register an error (exception) handler in its constructor. Have a look at the files in t3lib/error/ to see how this should be done.

If you want to use the built-in error and exception handling but extend it by your own functionality, simply derive your class from the error and exception handling classes shipped with TYPO3 and register this class as error (exception) handler.

Example:

```
class tx_postExceptionsOnTwitter extends t3lib_error_DebugExceptionHandler {
    function echoExceptionWeb(Exception $exception) {
        $this->postExceptionsOnTwitter($exception);
    }
    function postExceptionsOnTwitter($exception) {
        // do it ;-)
    }
}

$TYPO3_CONF_VARS['SYS']['debugExceptionHandler'] = 'tx_postExceptionsOnTwitter';
$TYPO3_CONF_VARS['SYS']['productionExceptionHandler'] = 'tx_postExceptionsOnTwitter';
```

Data Formats

<T3DataStructure>

Introduction

TYPO3 offers an XML format, T3DataStructure, which defines a hierarchical data structure. In itself the data structure definition doesn't do much - it is only a back bone for higher level applications which can add their own configuration inside.

Such applications can be:

- "FlexForms" - a TCEform type which will allow users to build information hierarchies (in XML) according to the Data Structure. In this sense the Data Structure is like a DTD for the backend which can render a dynamic form based on the Data Structure
- "TemplaVoila" - an extension which uses the Data Structure as backbone for mapping template HTML to data.

This documentation of a data structure will document the general aspects of the XML format and leave the details about FlexForms and TemplaVoila to be documented elsewhere.

Some other facts about Data Structures (DS):

- A Data Structure is defined in XML with the document tag named "<T3DataStructure>"
- The XML format generally complies with what can be converted into a PHP array by t3lib_div::xml2array() - thus it directly reflects how a multidimensional PHP array is constructed.
- A Data Structure can be arranged in a set of "sheets". The purpose of sheets will depend on the application. Basically sheets are like a one-dimensional internal categorization of Data Structures.
- Parsing a Data Structure into a PHP array is incredibly easy - just pass it to t3lib_div::xml2array() (see example below)
- "DS" is sometimes used as short for Data Structure

Elements

This is the list of elements and their nesting in the Data Structure. This could probably be expressed by a DTD or XML schema (anyone?). Words will have to do for now.

Elements nesting other elements ("Array" elements):

All elements defined here cannot contain any string value but *must* contain another set of elements.

(In a PHP array this corresponds to saying that all these elements must be arrays.)

Element	Description	Child elements
<T3DataStructure>	Document tag	<meta> <ROOT> *or* <sheets>
<meta>	Can contain application specific meta settings	
<ROOT> <[field name]>	Defines an "object" in the Data Structure • <ROOT> is reserved as tag for the first element in the Data Structure. The <ROOT> element must have a <type> tag with the value "array" and then define other objects nested in <el> tags. • [field name] defines the objects name	<type> <section> <el> <[application tag]>
<sheets>	Defines a collection of "sheets" which is like a one-dimensional list of independent Data Structures	<[sheet name]>
<TCEforms>	Contains details about visual representation of sheets. If there is only a single sheet, applies to implicit single sheet.	<sheetTitle> <cshFile>
<sheetTitle>	Title of the sheet. Mandatory for any sheet except the first (which gets "General" in this case). Can be a plain string or a reference to language file using standard LLL syntax. Ignored if sheets are not defined for the flexform.	
<cshFile>	CSH language file for fields inside the flexform. Refer to section on T3locallang of this document on the format of language files and to section Content Sensitive Help of "Inside TYPO3" document for information about CSH.	

Element	Description	Child elements
<[sheet ident]>	Defines an independent data structure starting with a <ROOT> tag. **Notice:** Alternatively it can be a plain value referring to another XML file which contains the <ROOT> structure. See example below.	<ROOT>
<el>	Contains a collection of Data Structure "objects"	<[field name]>

Elements containing values ("Value" elements):

All elements defined here must contain a string value and no other XML tags whatsoever!

(In a PHP array this corresponds to saying that all these elements must be strings or integers.)

Element	Format	Description
<type>	Keyword string: "array", [blank] (=default)	Defines the type of object. • "array" means that the object simply contains a collection of other objects defined inside the <el> tag on the same level If the value is "array" you can use the boolean "<section>". See below. • Default value means that the object does not contain sub objects. The meaning of such an object is determined by the application using the data structure. For FlexForms this object would draw a form element. **Notice:** If the object was <ROOT> this tag must have the value "array"
<section>	Boolean, 0/1	Defines for an object of the type <array> that it must contain other "array" type objects. The meaning of this is application specific. For FlexForms it will allow the user to select between possible arrays of objects to create in the form. For TemplaVoila it will select a "container" element for another set of elements inside. This is quite fuzzy unless you understand the contexts.

Example: FlexForm configuration in "mininews" extension

Simple example of a data structure used to define a FlexForm element in TCEforms. Notice the application specific section <TCEforms> (see documentation for FlexForms).

```
<T3DataStructure>
        <meta>
                <langDisable>1</langDisable>
        </meta>
  <ROOT>
    <type>array</type>
    <el>
      <field_templateObject>
                <TCEforms>
                        <label>LLL:EXT:mininews/locallang_db.php:tt_content.pi_flexform.select_template<
/label>

                        <config>
                                <type>select</type>
                                <items>
                                        <numIndex index=”0”>
                                                <numIndex index=”0”></numIndex>
                                                <numIndex index=”1”>0</numIndex>
                                        </numIndex>
                                </items>
                                <foreign_table>tx_templavoila_tmplobj</foreign_table>
                                <foreign_table_where>
                                        AND tx_templavoila_tmplobj.pid=###STORAGE_PID###
                                        AND
tx_templavoila_tmplobj.datastructure="EXT:mininews/template_datastructure.xml"
                                        AND tx_templavoila_tmplobj.parent=0
                                        ORDER BY tx_templavoila_tmplobj.title
                                </foreign_table_where>
                                <size>1</size>
                                <minitems>0</minitems>
                                <maxitems>1</maxitems>
                        </config>
                </TCEforms>
      </field_templateObject>
    </el>
  </ROOT>
</T3DataStructure>
```

Example #2

More complex example of a FlexForms structure, using two sheets, "sDEF" and "s_welcome" (snippet from "newloginbox" extension).

```
<T3DataStructure>
  <sheets>
        <sDEF>
          <ROOT>
                <TCEforms>
                        <sheetTitle>LLL:EXT:newloginbox/locallang_db.php:tt_content.pi_flexform.sheet_general</sheetTitle>
                </TCEforms>
            <type>array</type>
            <el>
              <show_forgot_password>
                        <TCEforms>
                                <label>LLL:EXT:newloginbox/locallang_db.php:tt_content.pi_flexform.show_forgot_password</label>
                                <config>
                                        <type>check</type>
                                </config>
                        </TCEforms>
              </show_forgot_password>
            </el>
          </ROOT>
    </sDEF>
    <s_welcome>
          <ROOT>
                <TCEforms>
                        <sheetTitle>LLL:EXT:newloginbox/locallang_db.php:tt_content.pi_flexform.sheet_welcome</sheetTitle>
                </TCEforms>
            <type>array</type>
            <el>
              <header>
                        <TCEforms>
                                <label>LLL:EXT:newloginbox/locallang_db.php:tt_content.pi_flexform.header</label>
                                <config>
                                        <type>input</type>
                                        <size>30</size>
                                </config>
                        </TCEforms>
              </header>
              <message>
                        <TCEforms>
                                <label>LLL:EXT:newloginbox/locallang_db.php:tt_content.pi_flexform.message</label>
                                <config>
                                        <type>text</type>
                                        <cols>30</cols>
                                        <rows>5</rows>
                                </config>
                        </TCEforms>
              </message>
            </el>
          </ROOT>
    </s_welcome>
  </sheets>
</T3DataStructure>
```

Sheet references

If Data Structures are arranged in a collection of sheets you can choose to store one or more sheets externally in separate files. This is done by setting the value of the <[sheet ident]> tag to a relative file reference instead of being a definition of the <ROOT> element.

Example

Taking the Data Structure from Example #2 above we can now rearrange it in three files:

Main Data Structure:

```
<T3DataStructure>
  <sheets>
        <sDEF>fileadmin/sheets/default_sheet.xml</sDEF>
    <s_welcome>fileadmin/sheets/welcome_sheet.xml</s_welcome>
  </sheets>
</T3DataStructure>
```

fileadmin/sheets/default_sheet.xml

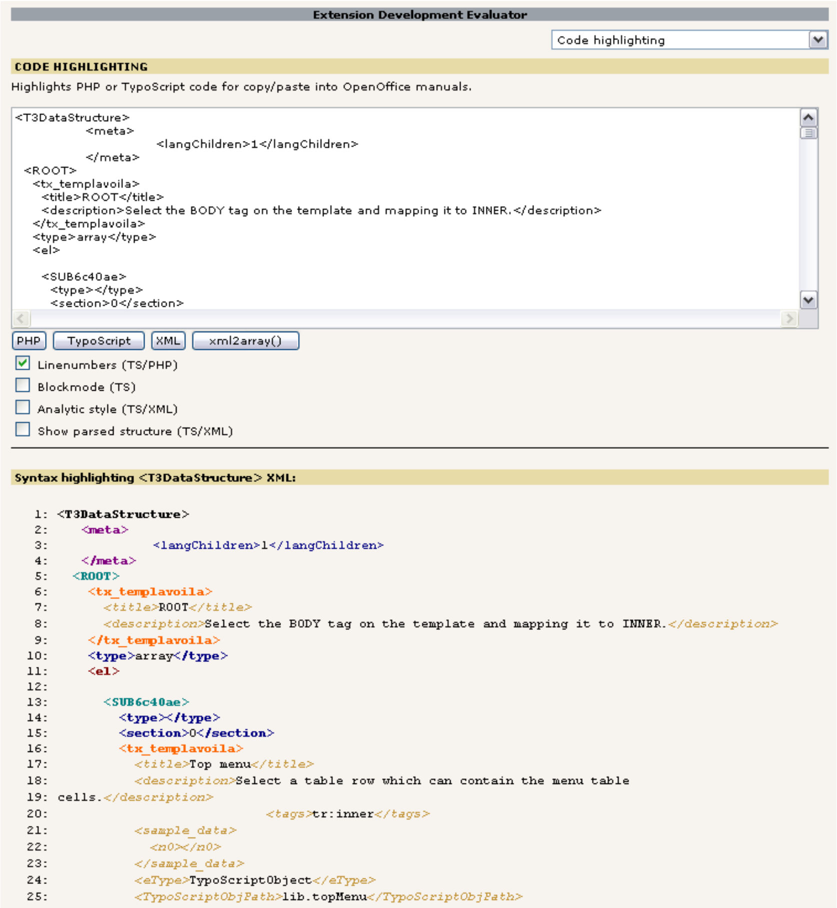
```
<T3DataStructure>
  <ROOT>
        <TCEforms>
                <sheetTitle>LLL:EXT:newloginbox/locallang_db.php:tt_content.pi_flexform.sheet_general</sheetTitle>
        </TCEforms>
    <type>array</type>
    <el>
      <show_forgot_password>
                <TCEforms>
                        <label>LLL:EXT:newloginbox/locallang_db.php:tt_content.pi_flexform.show_forgot_password</label>
                        <config>
                                <type>check</type>
                        </config>
                </TCEforms>
      </show_forgot_password>
    </el>
  </ROOT>
</T3DataStructure>
```

fileadmin/sheets/welcome_sheet.xml

```
<T3DataStructure>
  <ROOT>
        <TCEforms>
                <sheetTitle>LLL:EXT:newloginbox/locallang_db.php:tt_content.pi_flexform.sheet_welcome</sheetTitle>
        </TCEforms>
    <type>array</type>
    <el>
      <header>
                <TCEforms>
                        <label>LLL:EXT:newloginbox/locallang_db.php:tt_content.pi_flexform.header</label>
                        <config>
                                <type>input</type>
                                <size>30</size>
                        </config>
                </TCEforms>
      </header>
      <message>
                <TCEforms>
                        <label>LLL:EXT:newloginbox/locallang_db.php:tt_content.pi_flexform.message</label>
                        <config>
                                <type>text</type>
                                <cols>30</cols>
                                <rows>5</rows>
                        </config>
                </TCEforms>
      </message>
    </el>
  </ROOT>
</T3DataStructure>
```

Syntax highlighting of a Data Structure

You can syntax highlight a data structure using the extension “extdeveval” and the code highlighter. Just copy the DS XML content into the form:



Highlights PHP or TypoScript code for copy/paste into OpenOffice manuals.

```
<T3DataStructure>
        <meta>
                <langChildren>1</langChildren>
        </meta>
  <ROOT>
    <tx_templavoila>
      <title>ROOT</title>
      <description>Select the BODY tag on the template and mapping it to INNER.</description>
    </tx_templavoila>
    <type>array</type>
    <el>

      <SUB6c40ae>
        <type></type>
        <section>0</section>
```

☑ Linenumbers (TS/PHP)

☐ Blockmode (TS)

☐ Analytic style (TS/XML)

☐ Show parsed structure (TS/XML)

Syntax highlighting <T3DataStructure> XML:

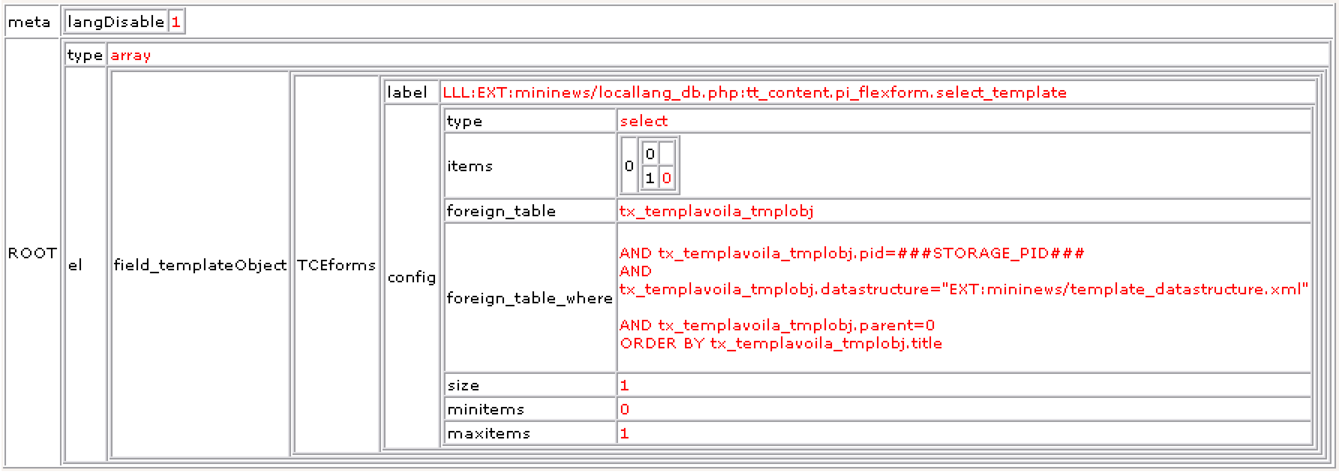
```
<T3DataStructure>
    <meta>
              <langChildren>1</langChildren>
    </meta>
   <ROOT>
     <tx_templavoila>
       <title>ROOT</title>
       <description>Select the BODY tag on the template and mapping it to INNER.</description>
     </tx_templavoila>
     <type>array</type>
     <el>

       <SUB6c40ae>
         <type></type>
         <section>0</section>
         <tx_templavoila>
           <title>Top menu</title>
           <description>Select a table row which can contain the menu table
cells.</description>
                        <tags>tr:inner</tags>
           <sample_data>
             <n0></n0>
           </sample_data>
           <eType>TypoScriptObject</eType>
           <TypoScriptObjPath>lib.topMenu</TypoScriptObjPath>
```

Parsing a Data Structure

You can convert a Data Structure XML document into a PHP array by the function t3lib_div::xml2array(). Taking the simple DS above:

```
<T3DataStructure>
        <meta>
                <langDisable>1</langDisable>
        </meta>
  <ROOT>
    <type>array</type>
    <el>
      <field_templateObject>
                <TCEforms>
                        <label>LLL:EXT:mininews/locallang_db.php:tt_content.pi_flexform.select_template<
/label>

                        <config>
                                <type>select</type>
                                <items>
                                        <numIndex index=”0”>
                                                <numIndex index=”0”></numIndex>
                                                <numIndex index=”1”>0</numIndex>
                                        </numIndex>
                                </items>
                                <foreign_table>tx_templavoila_tmplobj</foreign_table>
                                <foreign_table_where>
                                        AND tx_templavoila_tmplobj.pid=###STORAGE_PID###
                                        AND
tx_templavoila_tmplobj.datastructure="EXT:mininews/template_datastructure.xml"
                                        AND tx_templavoila_tmplobj.parent=0
                                        ORDER BY tx_templavoila_tmplobj.title
                                </foreign_table_where>
                                <size>1</size>
                                <minitems>0</minitems>
                                <maxitems>1</maxitems>
                        </config>
                </TCEforms>
      </field_templateObject>
    </el>
  </ROOT>
</T3DataStructure>
```

Passing this to the xml2array function and you will get an array like this (screen shot from "extdeveval"):



```
meta   langDisable 1
ROOT   type array
       el   field_templateObject   TCEforms   label    LLL:EXT:mininews/locallang_db.php:tt_content.pi_flexform.select_template
                                              config   type                select
                                                       items               0   0
                                                                               1 0
                                                       foreign_table       tx_templavoila_tmplobj
                                                       foreign_table_where AND tx_templavoila_tmplobj.pid=###STORAGE_PID###
                                                                           AND
                                                                           tx_templavoila_tmplobj.datastructure="EXT:mininews/template_datastructure.xml"

                                                                           AND tx_templavoila_tmplobj.parent=0
                                                                           ORDER BY tx_templavoila_tmplobj.title
                                                       size                1
                                                       minitems            0
                                                       maxitems            1
```

As you can see the format of the XML generated by t3lib_div::array2xml() is designed to reflect the array structures PHP can contain and thus the transformation to and from XML with the functions t3lib_div::array2xml() and t3lib_div::xml2array() is very easy and quick.

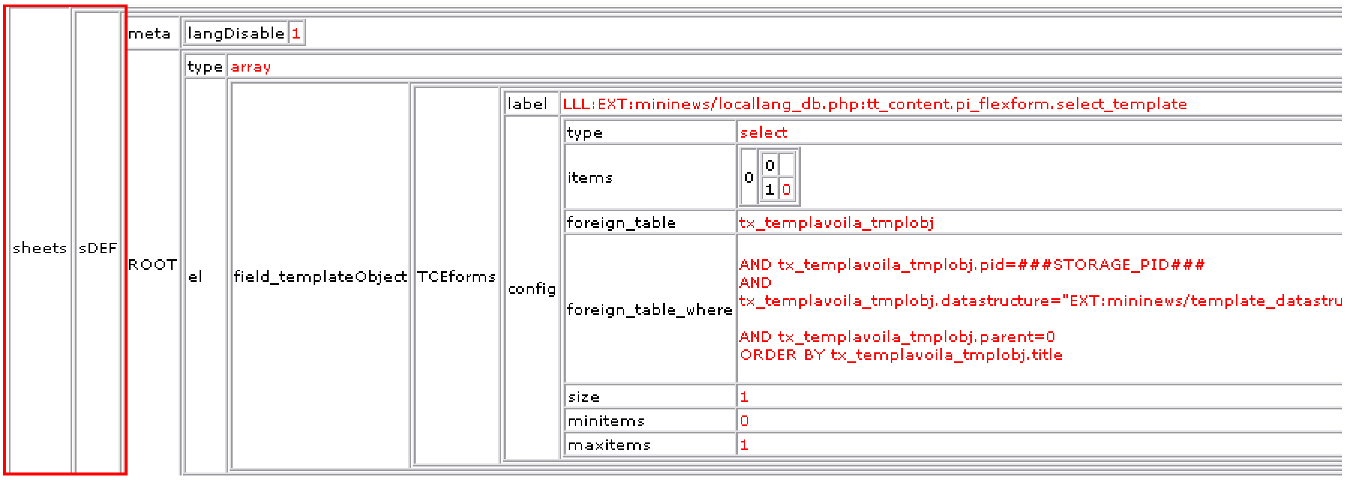
API functions for sheets

If you have a DS with sheets inside you might need to resolve the references:

```
<T3DataStructure>
  <sheets>
        <sDEF>fileadmin/sheets/default_sheet.xml</sDEF>
    <s_welcome>fileadmin/sheets/welcome_sheet.xml</s_welcome>
  </sheets>
</T3DataStructure>
```

This is done by t3lib_div::resolveSheetDefInDS() or t3lib_div::resolveAllSheetsInDS(). In fact, even if you don't have sheets in your file but just want to stay compatible with DS XML *with* sheets you should use this function. For instance these function calls will parse the DS into an array (screen shot above) and resolve the sheet definition, in this case creating a default sheet "sDEF" (screen shot below):

```
$treeDat = t3lib_div::xml2array($inputCode);
$treeDat = t3lib_div::resolveAllSheetsInDS($treeDat);
```



```
sheets  sDEF  meta   langDisable 1
              ROOT   type array
                     el   field_templateObject   TCEforms   label    LLL:EXT:mininews/locallang_db.php:tt_content.pi_flexform.select_template
                                                            config   type                select
                                                                     items               0   0
                                                                                             1 0
                                                                     foreign_table       tx_templavoila_tmplobj
                                                                     foreign_table_where AND tx_templavoila_tmplobj.pid=###STORAGE_PID###
                                                                                         AND
                                                                                         tx_templavoila_tmplobj.datastructure="EXT:mininews/template_datastru

                                                                                         AND tx_templavoila_tmplobj.parent=0
                                                                                         ORDER BY tx_templavoila_tmplobj.title
                                                                     size                1
                                                                     minitems            0
                                                                     maxitems            1
```

Applications

For a more practical understanding of Data Structures you should study some of the applications of Data Structures:

- FlexForms - using Data Structures as a “DTD” for rendering a hierarchical editing form which saves the content back into XML
- TemplaVoila - using Data Structures for mapping content to HTML template files.

<T3locallang>

Introduction

This XML format is used for "locallang-XML" (llXML) files, a format TYPO3 uses for storage of interface labels and translations of them. The format is parsed by t3lib_div::xml2array() which means that tag-names and "index" attribute values are inter-related in significance. The content is always in utf-8.

llXML files must be used from inside extension directories (system/global/local).

See "Inside TYPO3" for more details about locallang-files and the application of this format.

llXML files from installed extensions are translated by a backend tool (extension "llxmltranslate").

A "locallang-XML" file contains a set of labels in the default language (must always English!). The translated labels are located in systematically named external files in typo3conf/l10n/[language key]/. Optionally the main files can contain the translations directly but that should only be used in special cases since it constrains the translation process too much. It is also possible with a specific file reference to use other external files than the automated ones.

Elements

This is the elements and their nesting in the locallang-XML format.

Elements nesting other elements (“Array” elements):

All elements defined here cannot contain any string value but *must* contain another set of elements.

(In a PHP array this corresponds to saying that all these elements must be arrays.)

Element	Description	Child elements
<T3locallang>	Document tag	<meta> <data> <orig_hash> <orig_text>
<meta>	Contains meta data about the locallang-XML file. Used in translation, but not inside TYPO3 directly.	<labelContext> <description> <type> <csh_table>
<data>	Contains the data for translations **Notice:** The contents in the <data> tag is *all that is needed for labels inside TYPO3*. Everything else is meta information for the translation tool!	<languageKey>
<orig_hash>	Contains hash-integers for each translated label of the default label at the point of translation. This is used to determine if the default label has changed since the translation was made.	<languageKey>
<orig_text>	Contains the text of the default label that was the basis of the translated version! The original text is used to show a diff between the original base of the translation and the new default text so a translator can quickly see what has changed.	<languageKey>
<languageKey>	Array of labels for a language. The "index" attribute contains language key. There are two cases in the context “<data>” to note: • index = “default”: Array of default labels. • index = [language key] : • If string: Pointer to external file containing translation, e.g. “EXT:csh_dk/lang/dk.locallang_csh_web_info.xml” • If array: Translations inline in main file (deprecated) • [If not existing, *recommended*]: Translations in external default file typo3conf/l10n/	<label> *Alternatively, when used under <data> it can be a string pointing to an external "include file"!*
<labelContext>	Array of context descriptions of the default labels. The "index" attribute contains label key	<label>

Elements containing values ("Value" elements):

All elements defined here must contain a string value and no other XML tags whatsoever!

All values are in utf-8.

(In a PHP array this corresponds to saying that all these elements must be strings or integers.)

Element	Format	Description
<label> (under <data>)	string	Value of a original/translated label. The "index" attribute contains label key.
<label> (under <orig_hash>)	integer	Hash of a translated label. The "index" attribute contains label key.
<label> (under <orig_text>)	string	Original default value of a translated label used for making a diff if the original has changed. The "index" attribute contains label key.
<label> (child of <labelContext>)	string	Description of a default labels context. This should be used where it cannot be clear for the translation where the default labels occur. Sometimes the context is important for the translator in order to translate correctly. The "index" attribute contains label key.
<description>	string	Description of the file contents.
<type>	string	Type of content. Possible values are: ● "module" : Used for labels in the backend modules. ● "database" : Used for labels of database tables and fields. ● "CSH" : Used for Context Sensitive Help (both database tables, fields, backend modules etc.)

<csh_table>	string	(Only when the type is "CSH"!)
		For CSH it is important to know what "table" the labels belong to. A "table" in the context of CSH is an identification of a group of labels. This can be an actual table name (containing all CSH for a single table) or it can be module names etc. with a prefix to determine type. See CSH section in "Inside TYPO3" for more details.
		Examples:
		`<csh_table>xMOD_csh_corebe</csh_table> (General Core CSH)`
		`<csh_table>_MOD_tools_em</csh_table> (For Extension Mgm. module)`
		`<csh_table>pages</csh_table> (For "pages" table)`

<T3locallangExt>

External include files contains a sub-set of the tags of the <T3locallang> format. Basically they contain the <data>, <orig_hash> and <orig_text> tags but with "<languageKey>" tags inside only for the specific language they used.

When the include file is read the information for the selected language key is read from each of the three tags and merged into the internal array.

Element	Description	Child elements
<T3locallangExt>	Document tag for the external include files of "<T3locallang>"	<data> <orig_hash> <orig_text>
<data>	*See <data> element of <T3locallang> above.*	
<orig_hash>	*See <data> element of <T3locallang> above.*	
<orig_text>	*See <data> element of <T3locallang> above.*	

Example: locallang-XML file for a backend module

This example shows a standard locallang-XML file for a backend module. Notice how the <orig_hash> section is included which means that translators can spot if an original label changes. However the "<orig_text>" section would have been needed if translators were supposed to also see the difference. But typically that is not enabled since it takes a lot of space up.

```
<T3locallang>
    <meta type="array">
        <description>Standard Module labels for Extension Development Evaluator</description>
        <type>module</type>
        <csh_table/>
        <labelContext type="array"/>
    </meta>
    <data type="array">
        <languageKey index="default" type="array">
            <label index="mlang_tabs_tab">ExtDevEval</label>
            <label index="mlang_labels_tabdescr">The Extension Development Evaluator tool.</label>
        </languageKey>
        <languageKey index="dk" type="array">
            <label index="mlang_tabs_tab">ExtDevEval</label>
            <label index="mlang_labels_tabdescr">Evalueringsværktøj til udvikling af extensions.</label>
        </languageKey>
....
    </data>
    <orig_hash type="array">
        <languageKey index="dk" type="array">
            <label index="mlang_tabs_tab" type="integer">114927868</label>
            <label index="mlang_labels_tabdescr" type="integer">187879914</label>
        </languageKey>
    </orig_hash>
</T3locallang>
```

Example: locallang-XML file (CSH) with reference to external include file

The main XML file looks like this. Notice the tag "csh_table" has a value which is important for CSH content so it can be positioned in the right category.

In the <data> section you can see all default labels. But notice how the value for the "dk" translation is a reference to an external file! The contents of that file is shown below this listing.

```
<T3locallang>
    <meta type="array">
        <description>CSH for Web&gt;Info module(s) (General Framework)</description>
        <type>CSH</type>
        <csh_table>_MOD_web_info</csh_table>
        <labelContext type="array"/>
    </meta>
    <data type="array">
        <languageKey index="default" type="array">
            <label index=".alttitle">Web &gt; Info module</label>
            <label index=".description">The idea of the Web&gt;Info ...</label>
            <label index=".details">Conceptually the Web&gt;Info mod...functionality.</label>
            <label index="_.seeAlso">_MOD_web_func,</label>
            <label index="_.image">EXT:lang/cshimages/pagetree_overview_10.png</label>
            <label index=".image_descr">The Web&gt;Info module a....
&quot;info_pagetsconfig&quot;.</label>
        </languageKey>
        <languageKey index="dk">EXT:csh_dk/lang/dk.locallang_csh_web_info.xml</languageKey>
    </data>
</T3locallang>
```

The include file (for "dk") looks like below.

```
<T3locallangExt>
    <data type="array">
        <languageKey index="dk" type="array">
            <label index="pagetree_overview.alttitle">Sidetræ overblik</label>
        </languageKey>
    </data>
    <orig_hash type="array">
        <languageKey index="dk" type="array">
            <label index="pagetree_overview.alttitle" type="integer">92312309</label>
        </languageKey>
    </orig_hash>
    <orig_text type="array">
        <languageKey index="dk" type="array">
            <label index="pagetree_overview.alttitle">Pagetree Overview</label>
        </languageKey>
    </orig_text>
</T3locallangExt>
```