Inside TYPOlight

Overview

- Part 1: TYPOlight folders
- Part 2: TYPOlight framework
- Part 3: Libraries
- Part 4: Life cycle of a front end request
- Part 5: Data container arrays
- Part 6: Customizing TYPOlight

Part 1: TYPOlight folders

Part 1: TYPOlight folders

- plugins
- system
- templates
- tl_files
- typolight

- system
  - config
  - drivers
  - html
  - libraries
  - logs
  - modules
  - themes
  - tmp

Part 1: TYPOlight folders

- plugins
  - External scripts that are used system-wide (TinyMCE, MooTools, SWFobject, phpmailer etc.)
- templates
  - Custom templates, include files, SQL dumps
  - The folder is not touched by the live update script
- tl_files
  - Central file management
- typolight
  - Administration area in a subfolder allows for additional protection via .htaccess file

Part 1: TYPOlight folders

- system/config
  - Central storage location for configuration files
- system/drivers
  - Mixture between controller, model and view (e.g. DC_Table.php)
  - Database adapters (e.g. DB_Mysql.php, DB_Oracle.php)
- system/html
  - Cache directory that is accessible via HTTP (e.g. thumbnails)
- system/libraries
  - Libraries abstract various tasks like database communication, file operations (SMH), safely retrieving user input, sending e-mails, calculating dates etc.

Part 1: TYPOlight folders

- system/logs
  - Storage location for log files (not accessible via HTTP)
- system/modules
  - Central storage location for modules
  - Even the back end itself is “just” a module
  - The core can be extended by any functionality
- system/themes
  - Storage location for back end themes
- system/tmp
  - Cache directory that is not accessible via HTTP

Part 2: TYPOlight framework

Part 2: TYPOlight framework

- TYPOlight and MVC
  - MVC = Model-View-Controller
  - MVC elements exist in TYPOlight
  - Still it is not a classic MVC framework
- Deviation list
  - Models are only used for users
  - Drivers are a combination of controller, model and view with extended CRUD functionality (create, read, update, delete)
  - Back end views (e.g. forms) are being rendered automatically
  - No typical URI routing in favor of search-engine-friendly URLs in the front end

Part 2: TYPOlight framework – Models

- Models
  - Only implemented for users in TYPOlight
  - ```
    $this->import('BackendUser', 'User');
    echo $this->User->isAdmin; // False

    $this->User->admin = 1;
    $this->User->save();

    echo $this->User->isAdmin; // True
    ```
  - Models only play a minor part, because the goal was to program comprehensive drivers that can create different views and process forms on the basis of meta informations
  - Instead of creating a model, a controller and several views for every table, the driver is supposed to cover it all automatically
    ```

Part 2: TYPOlight framework – Views

- Views
  - **Layouts** (e.g. fe_page)
  - **Views** (e.g. mod_newslist)
  - **Partials** (e.g. layout_short)
  - The “TYPOlight vocabulary” does not draw this distinction; the term “template” is used for all kinds of views
- Loading views
  - TYPOlight searches the “templates” folder first
  - Then all active modules
  - First hit wins (if the template e.g. exists in the “backend” module, another template with the same name in the “news” module will never be loaded)

Part 2: TYPOlight framework – Views

- Parsing views
  - **Template::parse()** loads a view
  - replaces the wildcards within it
  - returns the result as string
- Outputting views
  - **Template::output()** loads a view
  - replaces the wildcards within it
  - executes additional actions
  - prints the result to the screen

Part 2: TYPOlight framework – Views

- BackendTemplate::output()
  - Loads the rich text editor configuration
  - Inserts the dynamic JavaScript and CSS files
  - Executes the “outputBackendTemplate”-Hook
  - Adds the copyright notice
  - Checks and enables the GZip compression
  - Sends the HTTP headers
  - Outputs the XHTML code
  - Prints the debug information (if active)

Part 2: TYPOlight framework – Views

- FrontendTemplate::output()
  - Generates the search index URL
  - Reads the article keywords
  - Executes the “outputFrontendTemplate”-Hook
  - Stores the cache file and sends the cache header
  - Replaces the insert tags (if active)
  - Adds the file to the search index (if active)
  - Inserts the copyright notice
  - Checks and enables the GZip compression
  - Sends the HTTP header
  - Outputs the XHTML code
  - Prints the debug information (if active)

Part 2: TYPOlight framework – Views

- Dynamic scripts
  - **TL_CSS**: Allows you to add CSS files
  - **TL_JAVASCRIPT**: Allows you to add JavaScript files
  - **TL_HEAD**: Allows you to add individual code
  - `$GLOBALS['TL_CSS'][] = 'system/modules/news/style.css';`
- Automatic back end views
  - **List view**: Lists the records of a table
  - **Parent view**: Lists the child records of a parent record
  - **Tree view**: Lists hierarchical records as a tree
  - Automatic form rendering saves us from having to create a separate view for every table and every action

Part 2: TYPOlight framework – Controller

- Controller functionality (CRUD)
  - **list()**: Lists all records
  - **show()**: Lists a single record
  - **create()**: Renders a form to create a new record
  - **save()**: Saves a new record
  - **edit()**: Renders a form to edit an existing record
  - **update()**: Updates an existing record
  - **delete()**: Deletes a record

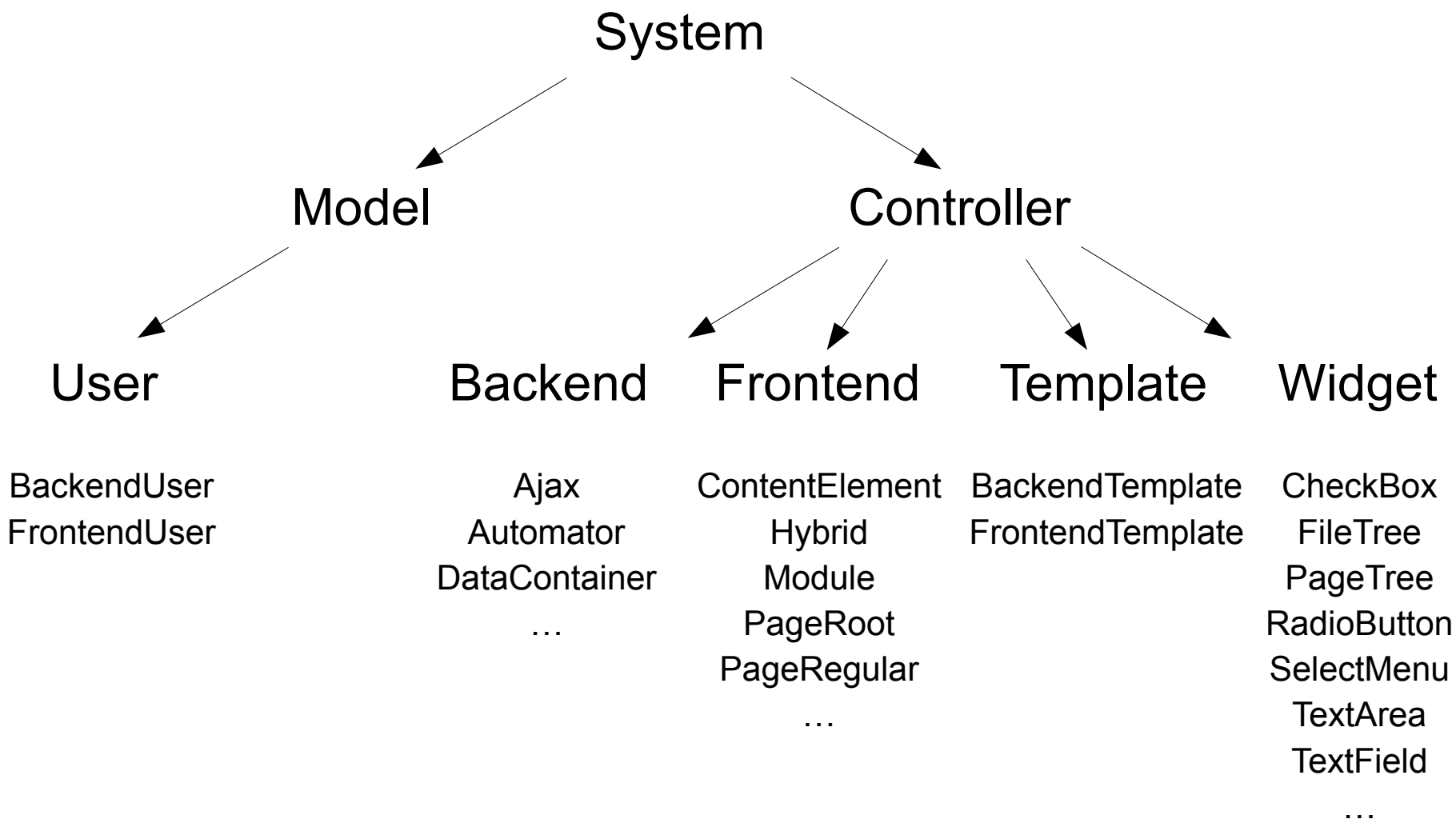
Part 2: TYPOlight framework – Controller

- Additional TYPOlight driver functions
  - **cut()**: Moves a record
  - **copy()**: Duplicates a record
  - **deleteAll()**: Deletes multiple records at once
  - **editAll()**: Edits multiple records at once
  - **undo()**: Restores a deleted record
  - Restoration of former versions of a record
- “Virtual controller”
  - At run time, a virtual controller is created on the basis of the DCA configuration, which takes care of rendering forms, validating user input and saving data to the database
  - Offers more functionality than a CRUD controller

Part 3: Libraries

Part 3: Libraries

- System architecture



Part 3: Libraries – System

- Base class „System“
  - Contains system-wide methods
  - **import()** instantiates other objects
  - **log()** adds an entry to the log table
  - **reload()** reloads the current page
  - **redirect()** redirects to another page
  - **parseDate()** returns a formatted date
  - **setCookie()** writes a cookie
- Advantages of System::import()
  - Detects Singletons automatically
  - Checks whether an object already exists

Part 3: Libraries – Controller

- class Controller extends System
  - **getFrontendModule()** returns a front end module
  - **getArticle()** returns an article
  - **getContentElement()** returns a content element
  - **resizeImage()** generates a thumbnail in system/html
  - **printArticleAsPdf()** exports an article as PDF file
  - **replaceInsertTags()** replaces insert tags
  - **sendFileToBrowser()** triggers the “save as …” dialogue
  - **getFrontendUrl()** generates a front end URL
  - **removeOldFeeds()** removes deprecated XML files
  - …

Part 3: Libraries – Controller

- class Backend extends Controller
  - **getBackendModule()** returns a back end module
  - **getSearchablePages()** returns all searchable pages
  - **createPageList()** returns the pages as drop-down menu
  - **createFileList()** returns the files as drop-down menu
- class Frontend extends Controller
  - **getPageIdFromUrl()** returns the ID of the current page
  - **getRootIdFromUrl()** returns the ID of the current root page
  - **jumpToOrReload()** reloads the page or redirects to another one
  - **getLoginStatus()** checks whether a user is logged in
  - **parseMetaFile()** parses a “meta.txt” file

Part 3: Libraries – Database abstraction

- Database abstraction
  - SQL92 standard as common denominator
  - Only specific functions like LIMIT are encapsulated in extra methods which are defined per adapter
  - Thus, the interface remains uncomplicated and flexible
  -
```
$db = $this->Database;

$stmt = $db->prepare('SELECT * FROM tl_user WHERE name=?');
$stmt->limit(1); // Inconsistent, therefore encapsulated
$user = $stmt->execute('Theo Test');

while ($user->next())
{
    echo $user->name;
}
```
  - Easy access to the fields of the result set

Part 3: Libraries – Database abstraction

- Advantages of the DB abstraction library
  - Supports complex queries like joins or subqueries
  - Automatic escaping prevents SQL injections
  - Lazy initialization of result sets
  - Consistent and database-independent interface
- Restrictions of the DB abstraction library
  - No abstraction layer to create and modify tables
  - The abstraction library does not provide for the special requirements of BLOB/CLOB fields in Oracle
  - Only MySQL is in fact completely supported
  - No control whether a programmer abides by the SQL92 standard (it is possible to write specific queries)

Part 3: Libraries – File operations

- File permissions and the Safe Mode Hack
  - PHP as an Apache module typically runs under the user “wwwrun”, “nobody” or “www-data”
  - However, files that have been uploaded via FTP typically belong to the FTP user (e.g. “web5” or “xa2387”)
  - The server denies the PHP process (and thus TYPOlight) access to the supposedly alien files
- Solutions
  - Run PHP as CGI with suPHP
  - Run the PHP process under the same user who owns the files that have been uploaded via FTP
  - Execute file operations via FTP (Safe Mode Hack)

Part 3: Libraries – File operations

- Abstraction layer „Files“
  - Depending on the configuration settings, the Files library loads a PHP or FTP adapter to modify files
  - **mkdir()** creates a new directory
  - **rmdir()** removes a directory
  - **fopen()** opens a file
  - **fputs()** writes to a file
  - **fclose()** closes a file
  - **rename()** renames a file or folder
  - **copy()** duplicates a file or folder
  - **delete()** deletes a file
  - **chmod()** changes the access rights of a file or folder

Part 3: Libraries – File operations

- File modification via “Files”
  - Works similar to the native PHP functions
  - ```
    $this->import('Files');
    ```

    ```
    $fh = $this->Files->fopen('system/tmp/test.txt', 'wb');
    $this->Files->fputs($fh, 'This is a test.');
    $this->Files->fclose($fh);
    ```
  - File paths have to be relative!
- Easy operation via “File” and “Folder”
  - Utility classes to modify files or folders
  - Supports creating folders recursively
  - Provides file information like path, extension, access time, width and height or MIME type
    ```

Part 3: Libraries – Security in TYPOlight

- Security in TYPOlight
  - Input library encapsulates reading user input
  - **Step 1**: HTML entities are being decoded
  - **Step 2**: Unicode entities are being decoded (XSS prevention)
  - **Step 3**: JavaScript snippets are being removed (in „strict mode“ all event attributes are being removed as well)
  - **Step 4**: Disallowed HTML tags are being removed
  - **Step 5**: Potentially dangerous characters are being encoded
- Additional XSS protection
  - Do **not** add the <script> tag to the list of allowed tags
  - Otherwise it is possible to embed JavaScript in all HTML fields

Part 3: Libraries – Security in TYPOlight

- Reading the server environment
  - The Environment library allows you to read the server environment independently from the operating system
  - Potentially dangerous code is being removed (e.g. `$_SERVER['HTTP_USER_AGENT']` can contain JavaScript code)
- Securing forms
  - If a form is being submitted, TYPOlight checks whether it actually comes from the same site (referer check)
  - Some anonymizers and security tools hide the referer address which leads to an error message in TYPOlight
  - If the referer check is being disabled (never recommended), all forms should at least contain a security question (Captcha)
  - A captcha additionally protects you against spam

Part 3: Libraries – Security in TYPOlight

- Login and authentication
  - A TYPOlight session is bound to the PHP session and the IP address of the user
  - IP binding can be disabled in version 2.7 (not recommended)
  - Active sessions are stored in the database
  - The cookie only contains a checksum and no relevant data like expiration time, ID or other user information
  - Recall extension allows for persistent logins in the front end
- Switching accounts and previewing the front end
  - The implementation supports session switching
  - Administrators can switch to other users (both in the back end as well as in the front end preview)

Part 3: Libraries – Security in TYPOlight

- Storing encrypted data
  - Every field can be stored encrypted
  - Configurable in the data container array
  - `$GLOBALS['TL_DCA']…['eval']['encrypt'] = true;`
  - Encryption requires an encryption key that is set up during the installation process (once data is encrypted, it can only be decrypted with this key!)
  - Requires the PHP module “mcrypt”
- Encryption in the TYPOlight core
  - Encryption is not being used in the core so far
  - Custom modules that store sensitive data (e.g. credit card information) should use this feature

Part 3: Libraries – Widgets

- Widgets
  - Widgets = form fields
  - Standard fields like TextField, CheckBox, SelectMenu
  - TYPOlight-specific fields like PageTree, FileTree or wizards
- Base class „Widget“
  - Provides common functionality
  - **generate()** returns a form field
  - **validate()** validates the user input
  - **hasErrors()** checks whether there have been errors
  - **getErrors()** returns the error messages as array

Part 3: Libraries – Widgets

- Outputting widgets
  - **generateLabel()** returns the label
  - `<label for="ctrl_name">Name</label>`
  - **generate()** returns the field
  - `<input type="text" id="ctrl_name" name="name" />`
  - **generateWithError()** returns the field with error message
  - `<p class="error">Please fill in the field.</p>`
    `<input type="text" id="ctrl_name" name="name" />`
  - **generateWithError(**true**)** reverses the order
  - `<input type="text" id="ctrl_name" name="name" />`
    `<p class="error">Please fill in the field.</p>`

Part 3: Libraries – Widgets

- Outputting error messages
  - **getErrors()** returns the error messages as array
  - **getErrorAsString()** returns the first error message
  - **getErrorAsString(2)** returns the third error message
  - **getErrorsAsString()** returns all error messages as string, separated by a line break (`<br />`)
  - **getErrorsAsString(', ')** returns all error messages as string, separated by a comma
  - **getErrorAsHTML()** returns the first error message as HTML string (`<p class="error">…</p>`)
  - **getErrorAsHTML(2)** returns the third error message

Part 3: Libraries – Widgets

- Default view

```
<!-- View (actually partial) -->
<?php echo $this->generateLabel(); ?>
<?php echo $this->generateWithError(); ?>

<!-- Output -->
<label for="ctrl_name">Your name</label>
<input type="text" id="ctrl_name" name="name" />

<!-- Output with error message -->
<label for="ctrl_name">Your name</label>
<p class="error">Please fill in the field.</p>
<input type="text" id="ctrl_name" name="name" />

<!-- Reverse order → generateWithError(true) -->
<label for="ctrl_name">Your name</label>
<input type="text" id="ctrl_name" name="name" />
<p class="error">Please fill in the field.</p>
```

Part 3: Libraries – Widgets

- Complex example

```
<!-- View (actually partial) -->
<fieldset>
<?php if ($this->hasErrors()): ?>
<p class="flash"><?php echo $this->getErrorAsString(); ?></p>
<?php endif; ?>
<div>
  <?php echo $this->generateLabel(); ?><br />
  <?php echo $this->generateWithError(); ?>
</div>
</fieldset>

<!-- Output -->
<fieldset>
<p class="flash">Please fill in the field.</p>
<div>
  <label for="ctrl_name">Your name</label><br />
  <input type="text" id="ctrl_name" name="name" />
</div>
</fieldset>
```

Part 3: Libraries – Widgets

- Input validation
  - **Mandatory field**: the field must not be empty
  - **Minimum length**: must not contain less than n characters
  - **Maximum length**: must not contain more than n characters
  - **Digits & letters**: only digits and letters are allowed
  - **Date & time**: only date and time formats are allowed
  - **E-mail address**: input must be a valid e-mail address
  - **Phone number**: input must be a valid phone number
  - **URL**: input must be a valid URL or domain
  - **Percent**: input must be a number between 0 and 100
  - Individual regular expressions can be added using the "addCustomRegexp" hook

Part 3: Libraries – Creating feeds

- class Feed extends System
  - Getter and setter methods for properties
  - **addItem()** adds a FeedItem
  - **generateRss()** returns the feed in RSS format
  - **generateAtom()** returns the feed in Atom format
- class FeedItem extends System
  - Getter and setter methods for properties
  - **addEnclosure()** adds an enclosure to the item

Part 3: Libraries – Creating feeds

- Simplified example

```php
<?php

$feed = new Feed();

$feed->title = 'TYPOlight user meeting 2009';
$feed->description = 'Information about the user meeting';

$item = new Item();

$item->title = 'Record participation';
$item->description = 'More than 70 participants!';

$feed->addItem($item);

echo $feed->generateRss();

?>
```

Part 3: Libraries – Creating feeds

- RSS output

```
<?xml version="1.0" encoding="UTF-8"?>
<rss version="2.0">
  <channel>
    <title>TYPOlight user meeting 2009</title>
    <description>Information about the user …</description>
    <link>…</link>
    <language>…</language>
    <pubDate>…</pubDate>
    <item>
      <title>Record participation</title>
      <description><![CDATA[More than 70 …]]></description>
      <link>…</link>
      <pubDate>…</pubDate>
      <guid>…</guid>
    </item>
  </channel>
</rss>
```

Part 3: Libraries – Periodic command scheduler

- Periodic command scheduler
  - Automatic script execution in certain intervals
  - Supports hourly, daily and weekly execution
  - Does not support exact scheduling like cron jobs
  - Can be used in custom extensions
  - `$GLOBALS['TL_CRON']['hourly'][] = array('Rates', 'update');`
- Daily execution
  - Recreation of the feed files
  - Purging of the temporary directory
- Weekly execution
  - Recreation of the style sheets and XML sitemaps

Part 3: Libraries – Periodic command scheduler

- Usage with a real cron job
  - The PCM can be triggered by a real cron job
  - Hourly execution of the cron.php file in the TYPOlight folder
  - `0 * * * * php /home/www/typolight/cron.php`
- Removing the triggers
  - Layouts be_login.tpl and fe_page.tpl
  - ```
    <!-- indexer::stop -->
    <img src="<?php echo $this->base; ?>cron.php" … />
    <!-- indexer::continue -->
    ```
  - The three lines need to be removed completely
    ```

Part 4: Life cycle of a front end request

Part 4: Life cycle of a FE request – Initialization

- __autoload()
  - Classes are loaded automatically in TYPOlight
  - The “libraries” folder is being searched first
  - Then all modules folders
  - Strict alphabetical order and no distinction between active and inactive modules, because the Config object does not even exist at the time the autoloader is defined
  - DOMPDF_autoload (if installed)
  - An exception is thrown if the class cannot be found
- Controller::classFileExists()
  - Checks whether a class or class file exists
  - Considers active and inactive modules

Part 4: Life cycle of a FE request – Initialization

- scan()
  - Scans a folder for subfolders and files
  - Like scandir(), but does not return '.' and '..'
  - Built-in cache and open_basedir compatibility
- specialchars()
  - Converts special characters into HTML entities
  - Like htmlspecialchars(), but does not modify ampersands to prevent double conversions (`&`)
- deserialize()
  - Reconverts a serialized array into an array
  - Like unserialize(), but returns the argument in case of an error

Part 4: Life cycle of a FE request – Initialization

- trimsplit()
  - Splits a string by a regular expression
  - Like preg_split(), but additionally executes trim()
- ampersand()
  - Converts all ampersands in a string into entities (argument true) or single ampersand characters (argument false)
- natcaseksort()
  - Extends the PHP function natcasesort()
  - Allows you to sort an array by keys, using a case insensitive "natural order" algorithm

Part 4: Life cycle of a FE request – Initialization

- array_insert()
  - Inserts a value at a certain position within an array
  - The value can be another array
- array_duplicate()
  - Duplicates a certain array member
  - The copy is added right after the original
- array_move_up()
  - Moves a certain array member one position up
  - Equates to exchanging two members

Part 4: Life cycle of a FE request – Initialization

- array_move_down()
  - Moves a certain array member one position down
  - Equates to exchanging two members
- array_delete()
  - Removes a certain array member
  - Recalculates the array keys
- array_is_assoc()
  - Checks whether an array is associative
  - If the keys are numeric and in a continuous ascending order, the array is considered not to be associative

Part 4: Life cycle of a FE request – Initialization

- mbstring.php
  - Substitute library for the PHP “mbstring” library
  - E.g. required on Strato shared hosting accounts
  - Provides functions to binary-safely modify international strings and non ASCII characters
  - Most important: utf8_strtolower() and utf8_strtoupper()
- php.ini customization
  - Suppress session IDs in URLs (PHPSESSID)
  - Define an error and exception handler
  - Set the path to the error.log file
- Starting the PHP session

Part 4: Life cycle of a FE request – Configuration

- Loading the Config object
  - The localconfig.php file is being loaded first to check whether there are any inactive extensions
  - system/modules/backend/config/config.php
  - system/modules/frontend/config/config.php
  - Then the configuration files (config.php) of the other active extensions are being loaded in alphabetical order
  - At last, the localconfig.php file is being loaded again to override the default configuration with the local settings
- Inactive extensions
  - Are neither searched nor initialized
  - The more inactive extensions, the better the performance

Part 4: Life cycle of a FE request – Configuration

- Configuration arrays
  - Back end modules
    `$GLOBALS['TL_CONFIG']['BE_MOD']`
  - Back end form fields
    `$GLOBALS['TL_CONFIG']['BE_FFL']`
  - Back end page types
    `$GLOBALS['TL_CONFIG']['TL_PTY']`
  - Front end modules
    `$GLOBALS['TL_CONFIG']['FE_MOD']`
  - Content elements
    `$GLOBALS['TL_CONFIG']['TL_CTE']`
  - Front end form fields
    `$GLOBALS['TL_CONFIG']['TL_FFL']`

Part 4: Life cycle of a FE request – Configuration

- Loading the default objects
  - Environment object to read the server environment
  - Input object to process user input
- Further configuration
  - Error_reporting is set according to the localconfig.php file
  - The time zone is set according to the localconfig.php file
  - The relative path to TYPOlight is calculated (if not set yet)
  - The mbstring encoding is set according to the localconfig.php file
  - The browser language is determined and stored
- Referer check
  - Only if there is form data

Part 4: Life cycle of a FE request – Finding a page

- Loading pages from the cache
  - TYPOlight looks for a cached version
  - Checks the expiration time and outputs it if it is valid
  - If the page is loaded from the cache, we are done
  - All following steps can be saved by using the cache!
- Loading the FrontendUser object
  - A database connection is being established
  - The User object is only initialized at this point
  - Neither authenticate() nor login() are executed
- Determining the login status
  - Checks whether a back end or front end user is logged in

Part 4: Life cycle of a FE request – Finding a page

- Finding the page by the URL
  - The ID or alias of the page is being extracted from the URL
  - The corresponding page is loaded from the database
  - And mapped to a website root page if the alias is not unique
  - At last, the settings from the parent pages are inherited
- Authenticating the user
  - The user session is validated on the basis of the cookie
  - On protected pages, the user's permissions are validated as well

Part 4: Life cycle of a FE request – Loading a page

- Loading the page object
  - Defaults to PageRegular (regular page)
- Loading the page layout
  - Doctype Definition and meta robots tags
  - TYPOlight CSS framework
  - Dynamic scripts (CSS-/JavaScript, <head> tags)
  - Google Analytics ID
- Loading the modules
  - Order: header, left, main, right, footer, custom sections
  - The article module loads articles and content elements
  - The page title and description are being added at the end

Part 4: Life cycle of a FE request – Printing a page

- Outputting the page
  - Template::output() prints the page to the screen
- The template object takes care of
  - Adding the page to the search index
  - Creating or updating the cache version
  - Sending the HTTP headers
  - Enabling the GZip compression
  - Outputting the view
  - (cp. Part 2: TYPOlight framework)

Part 5: Data container arrays

Part 5: Data container arrays – Function

- Table meta data
  - A data container array describes a table
  - Table configuration, table relations, field configuration
  - By this meta data, TYPOlight determines how to list/save records
  - Back end forms are also rendered on the basis of this meta data
- Loading DCA files
  - The DCA files of the active modules are loaded one after the other (backend, frontend and then in alphabetical order)
  - Every module can override the existing configuration
  - The dcaconfig.php file is included at the end, loading local modifications that are not touched by the live update

Part 5: Data container arrays – Structure

- Configuration
  - Configuration of the table itself
  - Relations to other tables
  - Versioning
  - Behaviour when data is edited or deleted
- Listing
  - Defines how records are listed
  - „List view“, „parent view“ or „tree view“
  - Defines the default sorting order
  - Filter configuration (search, filter, sort, limit)

Part 5: Data container arrays – Structure

- Operations
  - Operations (e.g. edit or delete)
  - Global operations (e.g. edit multiple)
  - Access control via button callbacks
- Palettes
  - A palette is a set of form fields
  - Form fields can be grouped and aligned
  - Only allowed fields are shown, so palettes can look differently depending on the user's permissions
  - Palettes can change dynamically e.g. depending on the type of module or content element
  - Subparts of the form can be loaded interactively via Ajax

Part 5: Data container arrays – Structure

- Fields
  - Defines the specific table fields
  - The input type determines the type of form field
- Evaluation
  - Detailed field configuration
  - Input validation (e.g. mandatory field or date field)
  - Field size (e.g. rows and columns of a textarea)
  - Field appearance (e.g. style)
  - Rich text editor configuration
  - Data encryption
  - …

Part 5: Data container arrays – Callbacks

- onload_callback
  - Part of the „configuration“ section
  - Executed when the DataContainer object is initialized
  - Allows you to e.g. check permissions or to modify the data container array dynamically at runtime
- onsubmit_callback
  - Part of the „configuration“ section
  - Executed when a back end form is submitted
  - Allows you to e.g. modify the form data before it is written to the database (used to calculate intervals in the calendar extension)

Part 5: Data container arrays – Callbacks

- ondelete_callback
  - Part of the „configuration“ section
  - Executed when a record is being deleted
  - Runs before the records are actually removed from the database
- paste_button_callback
  - Part of the „listing“ section
  - Allows for individual paste buttons
  - E.g. used in the site structure to enable or disable buttons depending on the access permissions
  - Additional check via load_callback required, because the command can still be entered directly in the URL!

Part 5: Data container arrays – Callbacks

- child_record_callback
  - Part of the „listing“ section
  - Defines how child elements are rendered in “parent view”
  - From version 2.7, child elements can be moved via Drag & Drop (e.g. content elements, format definitions, FAQs etc.)
- label_callback
  - Part of the „listing“ section
  - Allows for individual labels in the list
  - E.g. used in the user module to add status icons to the user list (administrator/user, active/inactive)

Part 5: Data container arrays – Callbacks

- button_callback
  - Part of the „operations“ section
  - Allows for individual navigation icons
  - E.g. used in the site structure to enable or disable buttons depending on the user permissions
  - Additional check via load_callback required, because the command can still be entered directly in the URL!
- options_callback
  - Part of the „fields“ section
  - Allows you to define an individual function to load data into a drop-down menu or checkbox list
  - Useful for e.g. conditional foreinKey-relations

Part 5: Data container arrays – Callbacks

- input_field_callback
  - Part of the „fields“ section
  - Allows for the creation of individual form fields
  - E.g. used in the back end module “personal data” to generate the “purge data” widget
  - Attention: the field is not saved automatically!
- load_callback
  - Part of the „fields“ section
  - Executed when a form field is initialized
  - Can be used to e.g. load a default value

Part 5: Data container arrays – Callbacks

- save_callback
  - Part of the „fields“ section
  - Executed when a field is submitted
  - Can be used to e.g. add an individual validation routine
  - The return value of the callback function is being saved, so it should always be set!

Part 6: Customizing TYPOlight

Part 6: Customizing TYPOlight

- Covered in the TYPOlight book
  - How to create a custom TinyMCE configuration file and integrate it into the data container array
  - How to customize labels and store the changes update-safe in the system/config/langconfig.php file
  - How to customize data container arrays and store the changes update-safe in the system/config/dcaconfig.php file
  - How to create a custom extension that defines an additional field and adds it to an existing table
- In this workshop
  - Purpose of the different hooks
  - How to extend classes and override methods

Part 6: Customizing TYPOlight – Hooks

- User registration
  - **createNewUser**: executed when a new user registers at the front end (the account can still be inactive)
  - **activateAccount**: executed when a newly registered front end account is activated
  - **setNewPassword**: executed when a password is changed
- Login and logout
  - **checkCredentials**: executed if the login fails due to a wrong password (allows you to e.g. check against another database)
  - **importUser**: executed if a user account cannot be found (allows you to e.g. import users from an LDAP server)
  - **postLogin/postLogout**: executed when a user logs into or off the front end

Part 6: Customizing TYPOlight – Hooks

- Forms
  - **loadFormField**: executed when a form field is loaded
  - **validateFormField**: allows you to add an individual validation routine to a form field
  - **addCustomRegexp**: allows you to add an individual regular expression to the widget validator
  - **postUpload**: executed after a file has been uploaded in a form
  - **processFormData**: executed after a form has been submitted

Part 6: Customizing TYPOlight – Hooks

- URL generation
  - **getPageIdFromUrl**: allows you to add a custom routine to extract the page ID from the URL
  - **generateFrontendUrl**: allows you to add a custom routine to generate front end URLs
- Templates
  - **parseBackendTemplate**: parses a back end template
  - **outputBackendTemplate**: outputs a back end template
  - **parseFrontendTemplate**: parses a front end template
  - **outputFrontendTemplate**: outputs a front end template

Part 6: Customizing TYPOlight – Hooks

- Miscellaneous
  - **getAllEvents**: allows you to add a custom routine to query events in a front end module
  - **getSearchablePages**: allows you to add custom URLs to the search index (URLs should point to valid pages)
  - **postDownload**: executed after a file has been downloaded (e.g. used in the download statistics extension)
  - **replaceInsertTags**: allows you to add custom insert tags

Part 6: Customizing TYPOlight – Extending classes

- Customizing the navigation module
  - The navigation module shall be modified to always display even if there are no subpages, in which case a note shall be printed
  - The functionality of the original class shall be preserved, so future updates do not require maintenance
- Creating a custom extension
  - Module folder “xcustom” (will be loaded last)
  - Holds a file named ModuleMyNavigation.php
  - Which defines the class ModuleMyNavigation
  - Class ModuleMyNavigation extends class ModuleNavigation
  - Only the generate() method will be overridden

Part 6: Customizing TYPOlight – Extending classes

- class ModuleMyNavigation

```php
<?php

class ModuleMyNavigation extends ModuleNavigation
{
    public function generate()
    {
        // Execute the original method
        $buffer = parent::generate();

        if ($buffer == '')
        {
            $buffer = 'There are no subpages';
        }

        return $buffer;
    }
}

?>
```

Part 6: Customizing TYPOlight – Extending classes

- Registering the new class
  - TYPOlight needs to know about the new class
  - Therefore we override the global configuration array FE_MOD in the system/modules/xcustom/config/config.php file
  - `$GLOBALS['TL_CONFIG']['FE_MOD']['navigationMenu']['navigation'] = 'ModuleMyNavigation';`
- Dynamic configuration
  - Thanks to the dynamic configuration, TYPOlight automatically loads the new class upon the next request
  - The navigation module now prints the notice “There are no subpages” instead of not showing at all
  - The modification is update-safe and does not require maintenance