TypoScript in 45 minutes

Extension Key: doc_tut_ts45

Language: en

Version: 1.1.0

Keywords: forAdmins, forBeginners, forIntermediates

Official documentation

This document is included as part of the official TYPO3 documentation. It has been approved by the TYPO3 Documentation Team following a peer-review process. The reader should expect the information in this document to be accurate - please report discrepancies to the Documentation Team (documentation@typo3.org). Official documents are kept up-to-date to the best of the Documentation Team's abilities.

Tutorial

This document is a Tutorial. Tutorials are designed to be step-by-step instructions specifically created to walk a beginner through a particular task from beginning to end. To facilitate effective learning, Tutorials provide examples to illustrate the subjects they cover. In addition, Tutorials provide guidance on how to avoid common pitfalls and highlight key concepts that should be remembered for future reference.

Table of Contents

Introduction

About this document

This document is meant to give a short introduction to how TypoScript works and what TypoScript really is. It helps to really understand the code instead of just copying and pasting.

What's new

This document is the “published” version of the wiki page http://wiki.typo3.org/Ts45min. All examples should work with the latest stable release branch, TYPO3 version 4.4.

Credits

This manual was originally created in German by members of the TYPO3 community in the wiki on wiki.typo3.org and then translated and corrected by more community members, so thank you TYPO3 community members ;-) for all your efforts.

Feedback

For general questions about the documentation get in touch by writing to documentation@typo3.org.

If you find a bug in this manual, please file an issue in this manual's bug tracker: http://forge.typo3.org/projects/typo3v4-doc_tut_ts45/issues

Maintaining quality documentation is hard work and the Documentation Team is always looking for volunteers. If you feel like helping please join the documentation mailing list (typo3.projects.documentation on lists.typo3.org).

TypoScript - a quick overview

Introduction

The goal of this introduction is not the thought, "Finally it's working!", but, "Finally I grasped it!". In other words, this introduction is to give you a comprehension of how TypoScript works. Normally, one adds some arbitrary properties to objects; but, for someone who knows TypoScript, it is clear that such a practice won´t work like that. It saves time to know what goes on. That way, troubleshooting becomes easier and time is economized as one learns TypoScript. Everything else is just luck.

The goal of this introduction is not to have a running TYPO3 installation at the end, but to offer an explanation why it works.

Backend Configuration

TypoScript has influence in many different places. If TypoScript is being used in the User/Usergroup TypoScript field or in the page TypoScript field, it will change the looks and behavior of the forms in the backend.

The frontend rendering is being defined by the TypoScript in the TypoScript template. This document only handles the frontend rendering part and only with TypoScript in general.

Prerequisites

We assume that the reader has a TYPO3 system up and running and that the basic operations are known. The difference between pages and content elements will not be elaborated here. We also assume that you know that the content of a page is put together by a combination of various content elements. Just to make sure, we point out the fact that these content elements are being stored in the table tt_content. The database field "CType" defines which content element type we have. Depending on CType a certain mask is loaded.

For a better understanding of TYPO3 and TypoScript, it is helpful to have a look at the database. The extension phpmyadmin allows an easy and comfortable way to access the database from the backend, thus allowing an overview on the relationships between the pages, tt_content and the backend. It should be known that the PID (Page ID) stands for the ID of a page and the UID (Unique ID) stands for a unique record.

Why TypoScript?

Strictly speaking, TypoScript is a configuration language. We cannot program with it, but we can configure many things very comprehensively with it. With TypoScript, we define the rendering of the website. We define our navigation, some fixed content, but also how individual content elements will be rendered on a page.

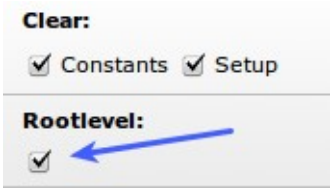
TYPO3 is a content management system with the goal to separate content and design. TypoScript is the glue which joins those together again. The content which is located in the database is read and prepared by TypoScript, and rendered on the frontend.

To render a website, we only have to define what and how content will be rendered.

- The "what" is controlled by the backend - there the pages and content are generated.
- The "how" is controlled by TypoScript.

With TypoScript we define how the individual content elements will be rendered in the frontend, if, for example, a DIV container wraps the element, or if the header <h1> will be integrated.

The TypoScript which defines how the pages is being rendered is located in the "main" template. In this the root level flag is set.



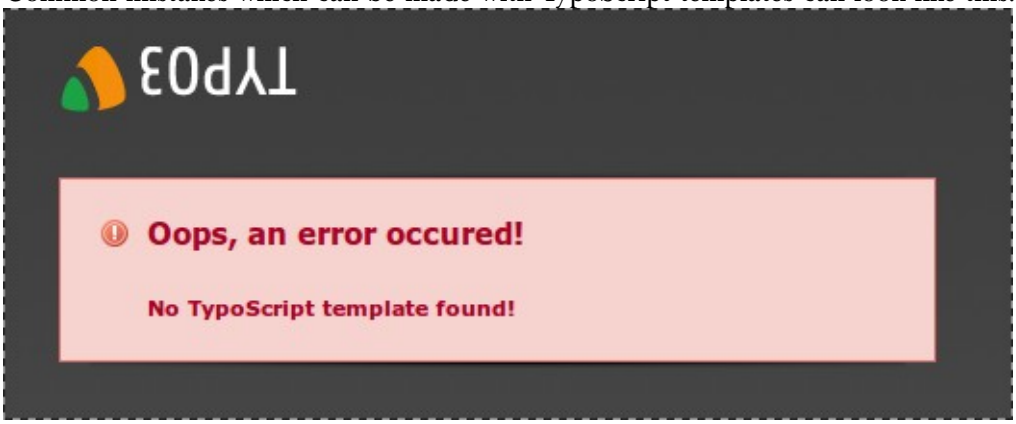
If the frontend has to render a page, TYPO3 searches along the page tree to find a "main" template. Normally, there are some more templates aside the "main" template. How these play together can be seen nicely in the template analyzer. For now, we assume that we only have one template.

The TypoScript syntax is very easy. On the left side, objects and their properties are defined which will get certain values assigned to them. Objects and properties (which in turn can hold objects as well) are separated by a dot ".".

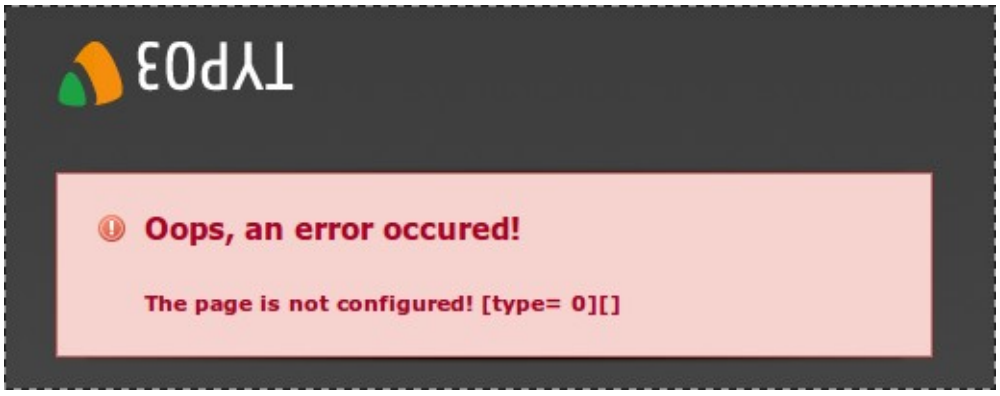
The term template

The term template has a duplicate meaning in TYPO3. On the one hand, there is the HTML template which serves as the skeletal structure in which the content will be rendered. On the other hand, there is the TypoScript template which can be created in the pages.

Common mistakes which can be made with TypoScript templates can look like this:



"No TypoScript template found": This warning appears if no template can be found which has the root level flag enabled in the page tree.



"The page is not configured": This warning appears if the root level flag is enabled, but no PAGE Object can be found.

The following code suffices to circumvent this warning:

```
page = PAGE
page.10 = TEXT
```

```
page.10.value = Hello World
```

TypoScript is just an array

TypoScript is just stored in a PHP array, internally. It is used and evaluated by various classes according to object types.

```
page = PAGE
page.10 = TEXT
page.10.value = Hello World
page.10.wrap = <h2>|</h2>
```

will be converted to the following PHP array

```
$data['page'] = 'PAGE';
$data['page.'][10] = 'TEXT';
$data['page.']['10.']['value'] = 'Hello World';
$data['page.']['10.']['wrap'] = '<h2>|</h2>';
```

On evaluation, the object "PAGE" will be created first, and the parameter $data['page.'] will be assigned. The object "PAGE" will then search for all properties which define it. In this case, it will just find a numeric entry "10" which has to be evaluated. A new object "TEXT" with the parameter $data['page.'] ['10.'] will be created. The object "TEXT" only knows the parameter "value", so it will set its content accordingly. All further parameters will be passed to the function stdWrap. (That's how "TEXT" is implemented. We will elaborate on stdWrap later). There the property 'wrap' is known, and the text "Hello world" will be inserted at the pipe (|) position and returned.

This relationship is important to understand the behavior of TypoScript in many cases. If, for example, the TypoScript is extended with the following line:

```
page.10.myFunction = Magic!
```

the entry will be taken into the PHP array:

```
$data['page.']['10.']['myFunction'] = 'Magic!';
```

However, neither the object TEXT nor the function stdWrap knows the property "myFunction". Consequently, the entry will have no effect.

No semantic error checking is done. This should especially be considered whilst troubleshootin

First steps

In the setup of the main template, the basic rendering is defined.

TypoScript essentially consists of objects which have certain properties. Some of these properties can accept other objects, others stand for certain functions or define the behavior of the object.

For rendering, the object PAGE is responsible.

```
# the object mypage is defined as PAGE object
mypage = PAGE

# it has the property typeNum
mypage.typeNum = 0

# and an object "10" of type TEXT
mypage.10 = TEXT

# the object "10" has in turn a property called "value"
mypage.10.value = Hello World
```

The PAGE object offers, in addition to numerous properties, an endless number of objects which can only be identified by their numbers (a so-called content array). This means that they only consist of numbers, and will get sorted accordingly when they are rendered. First, the object with the smallest number will be rendered; at the end, the object with the biggest number. The order of the TypoScript is

irrelevant.

```
mypage.30 = TEXT
mypage.30.value = This is last

# Rendering would first output object number 10, then 20 and 30. Object number 25 would
logically be
# outputted between 20 and 30
mypage.20 = TEXT
mypage.20.value = I'm the middle

# This is the first outputted object
mypage.10 = TEXT
mypage.10.value = Hello World!

# here we create a second object for the print view
print = PAGE
print.typeNum = 98
print.10 = TEXT
print.10.value = This is what the printer will see.
```

Every entry is stored in a multidimensional PHP array. Every object and every property, therefore, is unique. We could define an arbitrary number of PAGE objects; however, the typeNum has to be unique. For every typeNum, there can be only one PAGE object.

In the example, for the parameter typeNum = 98, a different output mode is created. By using typeNum, various output types can be defined. Typically, typeNum = 0 is used for the HTML output. The request for HTML would be index.php?id=1, respectively, and index.php?id=1&type=98 for the print output. The value of &type defines which PAGE object is displayed. That is why it is possible to have print output, HTML output and even PDF output in one and the same configuration. In doing so, configurations which are used in all of the views can be copied and changed just a little bit in the new object. (For example, normal page content can be copied into the print view but not the menu.)

Note

The output of these examples where both normal text. Especially with output formats like WML the HTTP header should be changed etc. This is not covered here.'

The previous example would look like this PHP array:

```
$TypoScript['mypage'] = 'PAGE';
$TypoScript['mypage .']['typeNum'] = 0;
$TypoScript['mypage .']['10'] = 'TEXT';
$TypoScript['mypage .']['10.']['value'] = 'Hello World!';
$TypoScript['mypage .']['20'] = 'TEXT';
$TypoScript['mypage .']['20.']['value'] = 'I'm the middle';
$TypoScript['mypage .']['30'] = 'TEXT';
$TypoScript['mypage .']['30.']['value'] = 'This is last';
```

Empty spaces at the start and end will be removed by TYPO3 (trim()).

With the "=" sign, we saw the basic assignment: a value is assigned.

```
# = Value is set
test = TEXT
test.value = Holla

# < Object will be copied
# mypage.10 returns "Holla"
mypage.10 < test

# The copied object will be changed
# The change has no effect on mypage.10
test.value = Hello world

# <= means that the object is referenced (the object is linked)
test.value = Holla
mypage.10 <= test
.
# - Object which is referenced changes
```

```
# - changes HAVE an effect on mypage.10
# - mypage.10 will return Hello world
test.value = Hello world
```

Objects are always written with capital letters; parameter and functions typically in camel case (first word lower case, next word starts with a capital letter, no space between words). There are some exceptions to this.

With the "." as a separator parameter, functions and child objects are referenced and can be set accordingly with values.

```
mypage.10.wrap = <h1>|</h1>
```

Which objects, parameters, and functions exist can be referenced in the TypoScript Reference (TSRef).

If some objects are wrapped in each other, and many parameters are assigned, it can get complicated.

```
mypage = PAGE
mypage.typeNum = 0
mypage.10 = TEXT
mypage.10.value = Hello world
mypage.10.typolink.parameter = http://www.typo3.org/
mypage.10.typolink.additionalParams = &nothing=nothing

# ATagParams unfortunately does not use the standardized "camelCase"
mypage.10.typolink.ATagParams = class="externalwebsite"
mypage.10.typolink.extTarget = _blank
mypage.10.typolink.title = The website of TYPO3
mypage.10.postCObject = HTML
mypage.10.postCObject.value = This Text also appears in the link text
mypage.10.postCObject.value.wrap = |, because the postCobject is executed before the
typolink function
```

To keep it simple, the curly brackets {} are allowed to define object levels. Parenthesis () are for writing text on more than one line. The above example can be rewritten like the following example:

```
mypage = PAGE
mypage {

  typeNum = 0

  10 = TEXT
  10 {

     value = Hello world
     typolink {

        parameter = http://www.typo3.org/
        additionalParams = &nothing=nothing

        # ATagParams unfortunately does not use the standardized "camelCase"
        ATagParams = class="externalwebsite"

        extTarget = _blank
        title = The website of TYPO3
     }

     postCObject = HTML
     postCObject {

        value = This Text also appears in the link text
        value {
           wrap (
            |, because the postCObject is executed before the typolink function
            )
         }
      }
   }
}
```

The danger of typographic errors is reduced, and the script is easier to read. In addition, if we would like to rename mypage, we would only have to change the first two lines, instead of the entire script.

Reading content records

Note

The following paragraphs serve as an example and for a better understanding of the background and relationships. The following scripts are from css_styled_content, and it's not necessary to write them by hand. If a content element has to be rendered totally different, or you programmed an extension with new content elements, it will be necessary to understand the relationships.

We do not want to enter all content with TypoScript - that would be tiresome, and we can't expect an editor to do that.

So, we create a content element with the type "TEXT", and create a TypoScript which will gather the content automatically. This example will create a page with a headline and the text from all content elements which are on the current page.

First, we create the PAGE object so there will be some rendering at all. In this object PAGE we will create the object CONTENT, which can be controlled with various TypoScript parameters.

```
page = PAGE
page.typeNum = 0

# The content-object executes a database query and loads the content
page.10 = CONTENT
page.10.table = tt_content
page.10.select {

     # "sorting" is a column from the tt_content table and
     # keeps track of the sorting order which is given in the backend
     orderBy = sorting

     # normal column
     where = colPos = 0
}

# For every result-line from the database query the renderObj is executed
# and the internal data array is filled with the content. This ensures that we
# can call the .field property and we get the according value
page.10.renderObj = COA
page.10.renderObj {

  10 = TEXT

  # The field tt_content.header normally holds the headline.
  10.field = header

  10.wrap = <h1>|</h1>

  20 = TEXT

  # The field tt_content.bodytext holds the content text
  20.field = bodytext

  20.wrap = <p>|</p>
}
```

The object CONTENT executes a SQL query on the database. The query is controlled by "select". "Select" defines that we want all records from the column 0 (which is the column "NORMAL" in the backend), and that the result will be sorted according to the field "sorting". If the property pidInList is not set or has been removed, the query will be limited to the current page only. For example, if the page with ID 100 is referenced, the CONTENT object will only return records from the page with pid=100.

The property renderObj defines how the records get rendered. Therefore, it will be defined as COA (Content Object Array) which can hold a arbitrary number of TypoScript objects. In this case, two TEXT objects are used, which will be rendered one after the other. The order of the rendering is not

controlled by the order in TypoScript, but by the number with which they are defined. The TEXT object "10" will be created before the TEXT object "20".

The challenge is to render all content elements of type text like the web designer predetermined. Therefore, we have to create a definition for every field (e.g., for images, image size, image position, on top, index, etc.)

The various content elements

If we want to render an image instead of a text, we have to choose different fields from tt_content, and also render them differently than standard text. The same applies to "text with image", "headline", etc.

The type of a content element is stored in the column tt_content.CType. In the following example, we show that with the CASE object, we can differentiate how the individual content elements will get rendered.

```
10.renderObj = CASE
10.renderObj {

  # the field CType will be used to differentiate
  key.field = CType

  # The content type "headline" is stored internally as "header"
  header = TEXT
  header.field = header
  header.wrap = <h1>|</h1>

  # Text is used for the text content element
  text = COA
  text {

    10 = TEXT
    # the field tt_content.header normally holds the headline.
    10.field = header
    10.wrap = <h1>|</h1>

    20 = TEXT
    # the field tt_content.bodytext holds the content text
    20.field = bodytext
    20.wrap = <p>|</p>

  }

  # ... other definitions follow here
}
```

css_styled_content

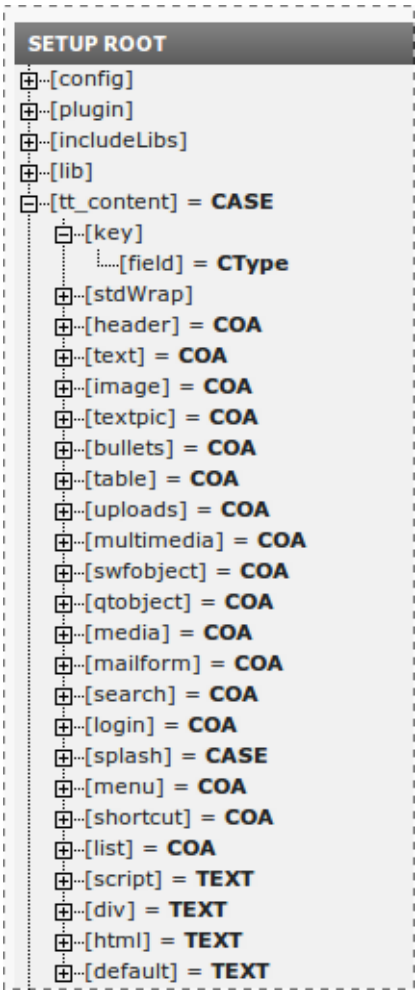
It would be tiresome to program this for every TYPO3 installation, because the elements are the same, or have very similar functionality. For this reason, TYPO3 offers "static templates". The current version comes with "css_styled_content". It has a meaningful definition for every existing content element.

The usage is comparably easy. The definitions are available as tt_content objects.

```
10.renderObj < tt_content
```

This assignment is also the default configuration of the CONTENT element. If the static template "css_styled_content" is available, there is no need to use the parameter "renderObj".

So for every content element in TYPO3, there is a corresponding definition in css_styles_content. In the object browser, it would look like this:



So it is comprehensible which content element is configured in what way. If a content element has to be configured completely different, then it should be clear that this can be done with tt_content.internal identifier of the content element. Here follows an example of how the standard properties of the header can be overwritten:

```
# Because TYPO3 saves everything in one big array the properties that are not overwritten
# are preserved and could result in strange behavior. That's why the old properties should
be deleted.
tt_content.header >

# Every header will be rendered with H1, independent of the properties in the content
element.
tt_content.header = TEXT
tt_content.header.wrap = <h1>|</h1>
tt_content.header.field = header
```

But, not only doesn't "renderObj" have to be recreated; the CONTENT object is also already defined in css_styled_content.

styles.content.get

```
# our code so far
page.10 = CONTENT
page.10.table = tt_content
page.10.select {

    # Use the sorting of the backend we could as well use the date field or the header
    orderBy = sorting

    # normal column
    where = colPos = 0
}
```

Thanks to css_styled_content, it suffices to write the following to achieve the same:

```
# Returns content from the "normal" column (colPos = 0)
page.10 < styles.content.get
```

For the other columns there are default definitions, as well:

```
# Returns content from the "left" column (colPos = 1)
page.10 < styles.content.getLeft

# Returns content from the "right" column (colPos = 2)
page.10 < styles.content.getRight

# Returns content from the "border" column (colPos = 3)
page.10 < styles.content.getBorder
```

In css_styled_content the border is defined as follows:

```
# The normal column is copied
styles.content.getBorder < styles.content.get

# after that colPos is altered
styles.content.getBorder.select.where = colPos=3
```

Create a menu

Until now, we learned how the page content is rendered; however, the page navigation is missing.

TYPO3 offers a special menu object HMENU (H stands for hierarchical) to easily build different kinds of menus.

The menu should be built like a nested list:

```
<ul>
   <li>first level</li>
   <li>first level
       <ul>
           <li>second level</li>
       </ul>
   </li>
   <li>first level</li>
</ul>
```

In order to keep oversight, we create a new folder and a new extension template. In here, we define a new object which we can add to the main template, later. In this way, we can define a diversity of objects apart and safe keep them for future projects. The extension template can be added with the use of "include basis template".

Normally, these objects are defined as sub-objects of "lib". We could use any term that hasn't been assigned, yet.

```
lib.textmenu = HMENU
lib.textmenu {

  # we define the first level as text menu
  1 = TMENU

  # We define the ''NO''rmal state
  1.NO.allWrap = <li>|</li>

  # We define the ''ACT''ive state
  1.ACT = 1
  1.ACT.wrapItemAndSub = <li>|</li>

  # Wrap the whole menu
  1.wrap = <ul class="level1">|</ul>

  # The second level should be configured exactly the same.
  # In-between the curly brackets objects can be copied.
  # With the dot "." we define that the object can be found in the brackets
  2 < .1
  2.wrap = <ul class="level2">|</ul>
  3 < .1
  3.wrap = <ul class="level3">|</ul>
}
```

The object HMENU allows us to create a diversity of menus. For every menu level, an arbitrary menu object can be created which does the rendering. Thus, it is possible to create a GMENU on the first level, and to use TMENU for the 2nd and 3rd level.

The first menu level will be defined with the number one, the second 2, etc. Naturally, it is not allowed to have missing numbers. (For example, if the third menu level is not defined, the fourth will not be rendered.)

On every menu level, we can configure various states of menu items – see menu items (NO = "normal"; ACT = "pages in the root line" (means current page, the parents, grandparents, etc.); CUR = "the current page"). In doing so, pay special attention to the fact that aside the normal state ("NO"), every state has to be activated first (i.e. ACT = 1).

Henceforth, we can use the menu and implement it at our page.

```
page.5 < lib.textmenu
```

Insert content in a template

We now know how to render content, and how to build a menu; but we still do not have a real website, yet.

We could build a website with COAs, and create the HTML skeleton with TypoScript. Although, this would be very complex, and prone to errors. If the HTML template has been built by a template designer and is delivered completely done, it gets even more complicated, especially with slight changes in the layout, afterward.

Therefore, we have the element TEMPLATE, with which we can parse an HTML template, and insert the menu, content, and so on, at the right place.

```
page.10 = TEMPLATE
page.10 {
 template = FILE

 # We load the HTML template
 template.file = fileadmin/test.tmpl

 # Text-areas
 # <!-- ###MENU### begin -->
 # Here is a example content as placeholder, everything which is in-between the markers
will
 # be replaced by the content of the sub-parts, in this case the menu
 # <!-- ###MENU### end -->

 subparts {
   MENU < lib.textmenu
   INHALT < styles.content.get
   SPALTERECHTS < styles.content.getRight
 }

 # Marks are single marker. i.e. there is no start and end marker,
 # instead the marker is replaced directly. ###LOGO### will
 # be replaced by the logo
 marks {
   LOGO = IMAGE

   # The Graphic logo*.gif will be added in the resource-field of the TypoScript template
   LOGO.file = logo*.gif

   # The logo links to the page with ID 1
   LOGO.stdWrap.typolink.parameter = 1
 }
 workOnSubpart = DOCUMENT
}
```

An alternative to this solution could be the extension automaketemplate, with which it's possible to abandon markers, completely. Instead, it uses IDs as references, and thus allows better cooperation with the template designer.

Another alternative would be the extension templavoila. This extension provides a visual user interface. This is not recommended for beginners, though.

Using css_styled_content

We already saw that we can define the different content elements of TYPO3, ourselves. But css_styled_content reduces the amount of work, with about 2000 lines of TypoScript.

It is rewarding - even if it doesn't make sense in the beginning - to have a good look at TypoScript. In TYPO3, we have to be at the page which has the setup template. Then, in the module "Template", we choose "Template Analyzer" from the selector box on top of the window.

A list with active and integrated TypoScript templates appears. These are evaluated from top to bottom by TYPO3, and joined together into one configuration array.

With a click on "EXT:css_styled_content/static/", the contents of that template will be displayed. First, the constants will appear, and then the setup TypoScript.

The extension css_styled_content will add many classes in HTML elements. This has the advantage that it's not necessary to enter all classes by hand. It suffices to find out which HTML element has which class, and to add CSS styles to it.

```
Example:
<div class="csc-textpic-imagewrap">...
```

The descriptions of the classes are simple and - if the TYPO3 internals are familiar - intuitive. All classes start with "csc"; this stands for "css_styled_content". In the example, this is followed by "textpic", which stands for the TypoScript element "textpic" (text with image). "imagewrap" suggests that the DIV container wraps around an image.

What's happening in detail can be understood by making an empty page with only one element, and then checking out the generated source code of that page.

For example, headlines are normally enumerated so that the first headline can be handled specifically. For HTML tables, the classes "odd" and "even" are inserted so that it's easy to color table rows differently. In the same manner, the table columns can be handled individually.

For the HTML purists, it means that many css classes will be inserted which are obsolete. In order to get rid of those, one needs to edit a lot of the css_styled_content extension.

COA TypoScript Objects

The TypoScript objects are implemented in TYPO3 by corresponding classes. For the various requirements while rendering a web page, there are various objects. These objects have a number of properties. For example, the object IMAGE has a method wrap and a method titleText. In the TypoScript reference, we can look up what kind of value this object expects. For wrap, a data type wrap is being expected - meaning a text which is separated by a pipe (|). To add several functions (e.g., "wrap.crop = 100") is therefore useless.

The object receives the parameter (as described above) in a PHP array (e.g., $conf['wrap.']['crop']='100';). This array can contain an arbitrary number of entries. Only those entries which are referenced by the object will be used, though (e.g., $conf['wrap'] or $conf['titleText']).

In the case "titleText", the data type is "string / stdWrap". This means that both text (string) and a method of type stdWrap are allowed. Which property stdWrap evaluates can be looked up in the stdWrap. Hence, we are allowed to augment the method "titleText" with various properties from stdWrap (e.g., titleText.field = header). In doing so, the value of titleText will be filled with standard text, and afterward the stdWrap functions are executed.

So, it is not necessary to guess which object will get manipulated in that way; but, it suffices to look up this information in the reference.

For the rendering of a web page, many more objects are being used. The challenge is to combine all of them artfully.

In the section "Reading content records", we show how we can use the CONTENT object to execute a query on the database and return the content of a page. The object receives a list of content elements which are created one after the other (normally in sorting order). Therefore, we use the object CASE to differentiate the type of the elements (CType) and render them differently.

It is absolutely necessary to know the various TypoScript objects and functions.

Object, executing the database queries

- CONTENT offers the functionality to access arbitrary tables of TYPO3 internals. This doesn't just include tt_content, but also tables of extensions, and so on, can be referenced. The function select allows us to generate complex SQL queries.
- RECORDS offers the functionality to reference specific data records. This is very helpful, if the same text has to be on all pages. By using RECORDS, a single content element can be defined which will be shown. Thus, an editor can edit the content without the need to copy the element numerous times. The object is also being used if the content element "insert record" is used.

  In the following example, the email address of the address record is rendered and linked as e-mail at the same time.

```
page.80 = RECORDS
page.80 {
	source = 1
	tables = tt_address
	conf.tt_address = COA
	conf.tt_address {
		20 = TEXT
		20.field = email
		20.typolink.parameter.field = email
	}
}
```

- HMENU imports the page tree and offers comfortable ways to generate a page menu. Aside the menu which renders the page tree, there are some special menus which allow various ways of usage. This object imports the internal structure for the menu. How this menu will be rendered depends on menu objects like TMENU (text-menu) or GMENU (graphical menu). For every menu level, the object can be changed. Within a menu level, there various menu items. For

every item, we can define the differing states (NO = normal, ACT = active, etc.).

Objects to render content

- IMAGE the rendering of an image

```
lib.logo = IMAGE
lib.logo {
  file = fileadmin/logo.gif
  file.width = 200
  stdWrap.typolink.parameter = 1
}
```

lib.logo holds the logo with a width of 200 pixel and is being linked with the page with PID 1.

- HTML / TEXT for the rendering of standard text or the content of fields. Fundamental difference: the HTML-object implements the stdWrap functionality on .value

```
lib.test1 = TEXT
lib.test1.field = uid

lib.test2 = HTML
lib.test2.value.field = uid
```

- FILE imports the content of a file directly
- TEMPLATE replaces a marker with content in a template

```
page.10 = TEMPLATE
page.10 {
  template = FILE
  template.file = fileadmin/test.tmpl
  subparts {
    HELLO = TEXT
    HELLO.value = replaces the content in between the markers ###HELLO### and ###HELLO###
  }
  marks {
    Test = TEXT
    Test.value = the marker Test will be replaced by this text
  }
  workOnSubpart = DOCUMENT
}
```

- MULTIMEDIA renders multimedia objects
- IMGTEXT allows to generate images inline with text. Is used for the content element "text with image"
- FORM generates a HTML-form

further objects

- CASE object allows case differentiation. In css_styled_content, this object is used for rendering different objects according to their type.
- COA - Content Object Array - allows us to combine an arbitrary number of objects.
- COA_INT - non cached. This element will be generated at each call. This is useful with time and date or user dependent data, for example.
- LOAD_REGISTER / RESTORE_REGISTER objects allow us to fill the global array $GLOBALS["TSFE"]->register[] with content. This object returns nothing. Single values and complete TypoScript objects can be used. In doing so, the register works as a stack: with every call, a further element is stacked. With RESTORE_REGISTER, the element on top can be removed.
- USER and USER_INT are for user defined functions. Every plugin is such an object. USER_INT is the noncached variant.

- IMG_RESOURCE is being used by IMAGE. The resource is returned (the content), which normally is the SRC attribute of the IMG tag. If images are scaled, this object serves as a calculation basis for the new files, which are stored in the /typo3temp folder.
- EDITPANEL - This object is inserted if a backend user is logged in, and the option "Display Edit Icons" is set in the frontend admin panel. If the admin panel is inserted, the pages will not be cached, anymore. Icons for sorting order, editing, removing, and so on, will be shown.
- GIFBUILDER is used for generating GIF files dynamically. Various texts and images can be combined, and much more. The GIFBUILDER itself offers some objects, like TEXT or IMAGE, which are not related to the standard TEXT or IMAGE objects, respectively. While working with the GIFBUILDER, we have to watch out that we do not confuse the objects, even though they have the same name.

We did not introduce all objects which exist in TypoScript, but we have named the most important one

TypoScript functions

TypoScript functions are used to change and adjust the output of content elements. The most popular function is the standard wrap, better known as stdWrap. Whether an object implements a certain function, or not, is shown in TSRef, column data type..

Property	Data type	Description	Default
file	imgResource		
imageLinkWrap	->imageLinkWrap	[...]	
if	->if	[...]	
altText **titleText**	String/stdWrap	[...]	

[Example:(cObject).IMAGE]

The first line in this example (property = file) tells us that file is of the data type imgResource. This means that we can use the imgResource functions on the file property.

Sometimes functions are - for better recognition - marked with an arrow (like -> if).

If there are multiple entries separated by a slash, it means that you have various possibilities to use that element. In the example above, you can see this with titleText and altText. Both can be either plain string or stdWrap. So, you can enter a plain string and do nothing more; or you can adjust and change your string by using stdWrap features on it; or you can leave the string empty all together and generate the content with stdWrap, only.

Some important and frequently used functions are presented in the following subsections. This chapter is about introducing those functions, and where they can be used. All details to them, however, can be found at TSRef, and not here.

imgResource

The functions of the data type "imgResource" relate to the modifications of pictures, as the name suggests. The object IMAGE has the property "file", which is inherited from the data type "imgResource".

This, for example, allows an image to be resized:

```
temp.myImage = IMAGE
temp.myImage {

        file = toplogo.gif
        file.width = 200
        file.height = 300

}
```

Enter maximum size (or minimum size):

```
temp.myImage = IMAGE
temp.myImage {

        file = toplogo.gif

        # maximum size
        file.maxW = 200
        file.maxH = 300

        # minimum size
        file.minW = 100
        file.minH = 120

}
```

and also the direct access to an ImageMagick function:

```
temp.myImage = IMAGE
temp.myImage {

        file = toplogo.gif
        file.params = -rotate 90
```

One of the most common and best examples for the use of imgResource is the implementation of pictures dynamically from the Media field in the page properties. This has the advantage that editors are able to change the pictures without using TypoScript. This allows us to have changing header images for different areas of a website with a few lines of TypoScript.

```
temp.dynamicHeader = IMAGE
temp.dynamicHeader {
        file {

                # Define path to the images
                import = uploads/media/

                import {

                        # If there are no images on this page, search recursive down the
page tree
                        data = level:-1, slide

                        # Enter the field in which the image is defined
                        field = media

                        # define which of the images will be displayed
                        # (in this case the first it encounters)
                        listNum = 0

                }
}
```

The path "uploads/media/" is the location of the files which are inserted in the pages properties "files" section. (For TYPO3 version 4.2.x, this is in the tab "Resources".) The TypoScript in the brackets of import completely consists of stdWrap functions, which are used to define from where and which image will be imported. Finally, stdWrap returns the file name of the image, which will then be imported from the import path (uploads/media).

imageLinkWrap

By using "imageLinkWrap", we can wrap the image with a link to the PHP script "showpic.php". The script will open the image in a new window with predefined parameters like window background, image size, etc. This function can be used to create "click-to-enlarge" functionality. (A small image (thumbnail) is clicked to open a new window and show the image in its original size.)

```
temp.meinBild = IMAGE
temp.meinBild {

        file = toplogo.gif

        imageLinkWrap = 1
        imageLinkWrap {

                # activate ImageLinkWrap
                enable = 1

                # define the body tag for the new window
                bodyTag = <body class="BildOriginal">

                # wrap the image (Close the window if it is clicked)
                wrap = <a href="javascript:close();"> | </a>

                # Width of the image(m allows proportional scaling)
                width = 800m

                # height of the image
                height = 600

                # create a new window for the image
```

```
                JSwindow = 1

                # open a new window for every image (instead of using the same window)
                JSwindow.newWindow = 1

                # Padding for the new window
                JSwindow.expand = 17,20
        }

}
```

numRows

In TypoScript, there are not only big mighty functions, but also small mighty ones. An example is the function numRows, which sole purpose is to return the number of lines from a select query. Just like the object CONTENT, numRows uses the select function. The query is generated similarly in both cases. The difference is only whether the number of lines is returned, or the actual content of those lines.

In cooperation with the "if" function, it is possible to generate some nice stuff. An example is a stylesheet for the content of the right column in the backend, which is only used if there actually is any content in the right column.

```
temp.headerdata = TEXT
temp.headerdata {
        value = <link rel="stylesheet" type="text/css"
href="fileadmin/templates/rechteSpalte.css">

        # if the select returns at least 1 line insert the stylesheet
        if.isTrue.numRows {

                # check if this page
                pidInList = this

                # has content in table tt_content
                table = tt_content

                # SQL: WHERE colPos = 2
                select.where = colPos=2
      }
}

# copy temp.headerdata in page.headerData.66 (and overwrite page.headerData.66)
page.headerData.66 < temp.headerdata
```

Or, use another template if there is content in the right column:

```
#a COA (Content Object Array) allows to merge many objects
temp.maintemplate= COA
temp.maintemplate {

        # 10 will only be embedded, if the if-Statement returns „true“
        10 = COA
        10 {
                # we use a copy of the select from css_styled_content
                if.isTrue.numRows < styles.content.getRight

                10 = TEMPLATE
                10 {
                        template = FILE
                        template.file = fileadmin/templates/template-2column.html
                }
        }

        # 20 will only be embedded, if the if-Statement returns „true“
        20 = COA
        20 {
                if.isFalse.numRows < styles.content.getRight
                10 = TEMPLATE
                10 {
                        template = FILE
                        template.file = fileadmin/templates/template.html
                }
```

```
        }
}
```

select

The function "select" generates a SQL SELECT query, which is used to read records from the database. The select function automatically checks whether the records might be "hidden", or "deleted", or if they have a "start and end date". If pidInList is used (meaning a list of pages is rendered), the function also checks if the current user is allowed to see all records.

With the help of the select function, it is possible to show the content of a page on all pages, for example.

```
temp.leftContent = CONTENT
temp.leftContent {

    table = tt_content
    select {

        # the page with ID = 123 is the source
        pidInList = 123

        # sorting is like in the backend
        orderBy = sorting

        # content of the left column
        where = colPos=1

        # defines the field with the lanuguage-ID in tt_content
        languageField = sys_language_uid
    }
}

# defines the field with the lanuguage-ID in tt_content
marks.LEFT < temp.leftContent
```

split

The split function can be used to split given data at a predefined character, and process the single pieces afterward. At every iteration, the current index key "SPLIT-COUNT" is stored (starting with 0).

By using "split", we could, for example, read a table field and wrap every single line with a certain code (e.g., generate a HTML table which can be used to show the same content on more than one page):

```
 # Example
 20 = TEXT

 # The content of the field "bodytext" is imported (from $cObj->data-array)
 20.field = bodytext
 20.split {

   # The separation character (char = 10 is newline) is defined.
   token.char = 10

   # We define which element will be used.
   # By using optionSplit we can distinguish between elements.
   # A corresponding element with the number must be defined!
   # Here the option split property is used.
   # Alternating the number 1 and 2 are being used for rendering.
   # In this example the classes "odd" and "even" are used so we can style a table in zebra
style.
    cObjNum = 1 || 2

    # The first element is being defined (which is referenced by cObjNum)
    # The content is imported using stdWrap->current
    1.current = 1

    # The element is wrapped
    1.wrap = <TR class="odd"><TD valign="top"> | </TD></TR>

    # The 2nd element is determined and wrapped
```

```
    2.current = 1
    2.wrap = <TR class="even"><TD valign="top"> | </TD></TR>
  }

  # A general wrap to create valid table markup
  20.wrap = <TABLE border="0" cellpadding="0" cellspacing="3" width="368"> | </TABLE>
```

if

The perhaps most difficult TYPO3 function is the "if" function, because every programmer who is familiar with it instinctively misuses it. Therefore, we have some examples to show how it works.

Generally, the if function returns "true" if all conditions are fulfilled. This resembles a Boolean AND combination. If "false" should be returned, we can use the "negate" option to negate the result (!(true)).

```
10 = TEXT
10 {

    # Content of the text-element
    value = The L parameter is passed as GET var

    # Results in "true" and leads to rendering the upper value if the
    # GET/POST parameter is passed with a value which is not 0
    if.isTrue.data = GP:L
}
```

With the use of "if" it is also possible to compare values. Therefore we use the parameter if.value.

```
10 = TEXT
10 {

    # WARNING: this value resembles the value of the text element not that of the "if"
    value = 3 is bigger than 2

    # compare parameter of the "if"-funtion
    if.value = 2

    # please note: the sorting order is backwards, this example
    # returns the sentence "3 isGreaterThan 2"
    if.isGreaterThan = 3
}
```

Because the properties of the "if" function implement stdWrap functions, all kinds of variables can be compared.

```
10 = TEXT
10 {
    # value of the text element
    value = The record can be shown because the starting date has passed.

    # condition of the if-clause
    if.value.data = date:U

    # condition backwards again: start time isLessThan date:U
    if.isLessThan.field = starttime
}
```

typolink

Typolink is the TYPO3 function which allows us to generate all kinds of links. If possible, one should use this function to generate links, because these will be "registered" in TYPO3. This is a prerequisite, for example, for realURL, which will generate search engine friendly paths; or for the anti-spam protection on email addresses. So, if you feel the urge to use <a href="..."> - don't.

The functionality of typolink is basically very easy. Typolink links the specified text according to the defined parameters. One example:

```
temp.link = TEXT
temp.link {

    # this is the defined text
    value = Examplelink
```

```
    # here comes the typolink function
    typolink {

        # whats the goal of the link?
        parameter = http://www.example.com/

        # with which target(_blank opens a new window)
        extTarget = _blank

        # and add a class to the link so we can style it.
        ATagParams = class="linkclass"
    }
}
```

The example above will generate this HTML code: Examplelink

Typolink works almost like a wrap - the content which is defined by value, for example, will be wrapped by the HTML anchor tag. If no content is defined, it will be generated automatically. With a link to a page, the page title will be used. With an external URL, the URL will be shown.

One can shorten this example, because the "parameter" tag from the typolink function does some of the thinking for you. Here, the short example will generate the exact same HTML code.

```
temp.link2 = TEXT
temp.link2 {

    # again the defined text
    value = Examplelink

    # the parameter with the summary of the parameters of the first example (explanation
follows below)
    typolink.parameter = www.example.com _blank linkclass
}
```

The "parameter"-part from the typolink function analyzes the entry on specific characters and converts The "parameter" part from the typolink function analyzes the entry on specific characters, and converts the respective sections. Initially, the parameter entry will be separated at the blank spaces. If then a dot "." is found (if the case may be in before a slash "/"), an external link will be generated. If the dot "." is found after a slash "/", a file link is generated. If an @ is found, an e-mail link would be generated. If a integer is found, like "51", an internal link to the page with the id "51" will be generated. If a leading hash "#" is found, a certain content element will be linked (for example, for a link to the content element with the id #234 on the current page. In order to link to the page with ID 51 and content element #234, one would use 51#234).

The second part of the parameter describes the goal of the link. Normally, this would be - like in the first example - defined by extTarget (for external links) or target (for internal links); but, it can be overwritten by using a second parameter.

The third part will be converted to a class attribute for the link.

If only a class attribute is needed, and no target, one has to fill the target part anyway, because the function would not recognize the class being in the third place. So, without a target, the line would be the following (with a minus sign "-" as divider):

```
typolink.parameter = www.example.com - linkclass
```

With the usage of the typolink function and the target attribute, it's also possible to open links in JavaScript-popups.

```
temp.link = TEXT
temp.link {

    # the link text
    value = Open a popup window

    typolink {
        # 1. Parameter = PageID of the target page, 1 2. parameter = size of the popup
        parameter = 10 500x400
```

```
            # The title tag of the link
            title = Click here to open a popup window

            # The parameters of the popup window
            JSwindow_params = menubar=0, scrollbars=0, toolbar=0, resizable=1

        }
}
```

It is important to note that many of the properties of typolink are of the type stdWrap. This means that values can be calculated, or read out of the database.

```
lib.stdheader >
lib.stdheader = TEXT
lib.stdheader {
   field = header
   typolink.parameter.field = header_link
   wrap = <h2>|</h2>
}
```

The headline will be displayed, and a link will be placed with a goal which is defined in the field header_link. The first line removes the default settings from css_styled_content.

encapsLines

EncapsLines is short for "encapsulate lines". This TypoScript function allows us to define how single lines in the content are wrapped. For example, if nothing is defined, a <p> or a will wrap the element. Another example would be to automatically replace all <b> tags with a tag.

A simple example:

In the RTE we have the following text:

```
A simple text without anything special

<div class="myclass">Some text with a wrapping div tag.</div>
```

In TypoScript we have the following:

```
encapsLines {
       # define which tags will be seen as wrappers
       encapsTagList = div,p

       # Lines which are not already encapsulated with tags from the
       # encapsTagList will be wrapped with <p>-tags
       wrapNonWrappedLines = <p>|</p>

       # replace all DIV tags with P tags
       remapTag.DIV = P

       # if a line is empty insert a empty space
       innerStdWrap_all.ifEmpty =  
}
```

The result will look like this in HTML code:

```
<p>A simple text without anything special.</p>
<p> </p>
<p class=""myclass">Some text with a wrapping div tag.</p>;
```

With most TYPO3 projects, the following code will not be necessary. But in the extension "css_styled_content", some settings are defined with this function, which can be changed, if needed. Therefore follows an example from the standard configuration from "css_styled_content", to clarify its functionality.

```
lib.parseFunc_RTE {

       nonTypoTagStdWrap.encapsLines {

              # wrapping tags
              encapsTagList = div,p,pre,h1,h2,h3,h4,h5,h6
```

```
                # convert all DIV tags to <p> tags
                remapTag.DIV = P

                # wrap all lines which are not wrapped at all with the <p> tag
                nonWrappedTag = P

                # replace all empty lines with the empty space code
                 innerStdWrap_all.ifBlank =  

                # here the - infamous - bodytext is placed
                addAttributes.P.class = bodytext

                # use addAttributes if no other attribute is set
                addAttributes.P.class.setOnly=blank
        }
}
```

Comparing the first example with the second, you might notice that apparently there are 2 parameters which do the same thing: firstly, "wrapNonWrappedLines"; and secondly, "nonWrappedTag". The difference is that "nonWrappedTag" can be extended, whereas "wrapNonWrappedLines" needs a comprehensive wrapping tag. If already wrapped lines like | are wrapped, and "wrapNonWrappedLines" is defined as | , the result would be a mixture of P tags with and without classes, instead of a consistent wrap.

To demonstrate it clearly, to get rid of the mostly annoying class="bodytext", you don't need to do more than insert the following line:

```
lib.parseFunc_RTE.nonTypoTagStdWrap.encapsLines.addAttributes.P.class >
```

file link

With the function "file link" we can generate - as the name suggests - a file link. While doing so, not just a link to the file is being generated, but "filelink" also allows us to add an icon and display the file size.

```
temp.example = TEXT
temp.example {

      # link description and file name at the same time
      value = my_image.png

      filelink {

            # Path to the file
            path = fileadmin/images/

            # The file should have an icon
            icon = 1

            # The icon will be wrapped
            icon.wrap = <span class="icon">|</span>

            # The icon has to be linked to the file as well
            icon_link = 1

            # Instead of the symbol for the filetype the file
            # will be displayed as an icon if it is of type png or gif
            icon_image_ext_list = png,gif

            # The size will be displayed as well
            size = 1

            # Wraps the filesize (with regard to the empty spaces)
            size.noTrimWrap = | (| Bytes) |

            # Rendering of the filesize will be done in bytes
            size.bytes = 1

            # Abbreviations for the various filesize units
            size.bytes.labels =  | K| M| G
```

```
                # Wrap for the whole element
                stdWrap.wrap = <div class="filelink">|</div>
        }
}
```

parseFunc

This function parses the main part of the content, i.e., the content which has been entered in the Rich Text Editor. The function is responsible for the fact that the content is not rendered exactly as it was entered in the RTE. Some default parsing rules are implemented in "css_styled_content", and some of those we explained in the encapsLines chapter. If we would like to change how TYPO3 wraps something, most of the time this can be done with a parseFunc instruction. We could also use parseFunc to search and replace a certain string.

In the following example, every occurrence of "COMP" is replaced by "My company name".

```
page.stdWrap.parseFunc.short {
 COMP = My company name
 }
```

The various possibilities to change the default behavior are easily found, by using the TypoScript object browser. All possibilities for how parseFunc can alter the rendering can be found here: parseFunc.

tags

The function "tags" is used in combination with parseFunc to get custom tags. For example, in the extension "css_styled_content", a custom tag <LINK> is defined to create simple links.

```
tags {
        # Here the name of the new tag is defined
        link = TEXT

        # here how the tag is processed/parsed
        link {
                current = 1
                typolink {

                        parameter.data = parameters:allParams

                        extTarget = {$styles.content.links.extTarget}

                        target = {$styles.content.links.target}

                }

                parseFunc.constants=1
        }
}
```

This function is especially useful, if a certain type of element is being used very often by the editors, and we would like to make things easier for them. We are able to provide a way that the editors do not have to format this manually every time, they just have to enter the tag, and the formatting is done automatically.

HTMLparser

The HTML parser defines how content is processed. Normally, it's used as a subfunction of parseFunc. For example, we could define that all links will be set with an absolute value (for example, for a newsletter):

```
page.stdWrap.HTMLparser = 1
page.stdWrap.HTMLparser {
      keepNonMatchedTags=1

      # Here we define the domain which will be placed in front of the relative path
      tags.a.fixAttrib.href.prefixRelPathWith=http://www.example.com/

      # All links without a target will receive a target = _blank
      tags.a.fixAttrib.target.default=_blank
```

```
}
```

The function HTMLparser is extremely mighty, because every content can be altered before it's rendered. We could define custom tags - e.g., internal links are stored as follows: <link http://www.typo3.org/>Linktext</link> thus a custom tag is being used. This custom tag can be defined in all fields - also headlines - on which a parser has been defined.

The following example allows the <u> tag in headlines. The default definition from "css_styled_content" will be altered. The function htmlSpecialChars will be deactivated, so the <u> remains untouched. Thereafter, the parseFunc function is used, and defined that aside the tag "u", no other tags will be allowed. Thus, all tags apart from the <u> will be removed.

```
# In the headline the <u> tag shall be allowed
# Apart from that all elements have to be parsed as usual
lib.stdheader.10.setCurrent.htmlSpecialChars = 0
lib.stdheader.10.setCurrent.parseFunc {
  allowTags = u
  denyTags = *
  constants=1
  nonTypoTagStdWrap.HTMLparser = 1
  nonTypoTagStdWrap.HTMLparser {
    keepNonMatchedTags=1
    htmlSpecialChars = 2
    allowTags = u
    removeTags = *
  }
}
```

This example once again shows how important the stdWrap function actually is. The function setCurrent is of Type string/stdWrap, and thus allows the usage of parseFunc.

using stdWrap correctly

The function stdWrap includes a wide variety of functions and parameter. Some are trivial, the use of some are hard to find. Here we will commit ourselves to the basic principle and highlight a few special functions/properties.

The stdWrap-property can only be used if its defined explicitly. If we have a property of type "wrap" then there are no stdWrap-properties. By default either a property of type stdWrap is presented or a property offers for example "string/stdWrap".

```
10 = IMAGE
10.stdWrap.typolink...
```

The object has a property stdWrap of type stdWrap.

```
10 = HTML
10.value = Hello World
10.value.typolink …
```

The object HTML in contrast has a property of type string/stdWrap. We can add a string and in addition we can use stdWrap properties.

Heed the order

An important limitation should be highlighted:

The single functions are executed in the order specified by the reference.

If we don't pay attention to this fact the results might look different from what we expected.

```
10 = TEXT
10.value = Hello World
10.case = upper
10.field = header # assuming the header contains “typo3” (small case characters)
10.stdWrap.wrap = <strong>|</strong>

# results in the following:
<STRONG>TYPO3</STRONG>
```

The following happens in this example: First, the value of the TEXT object is set to "Hello world". We know that the TypoScript configuration is stored in an array. The sorting in this array is not like the sorting in TypoScript. The sorting in the array is constrained by definitions of the ordering of stdWrap. This order is mirrored by the reference. After a short look into the TSRef it should be clear that, first, "field" is processed, thereafter stdWrap (and with it "stdWrap.wrap"), and in the end, "case".

use stdWrap recursively

Because the stdWrap function can be called recursively, it is possible to change the execution order.

The function "prioriCalc" permits easy mathematical expressions. If set to 1, the content is calculated; however, the calculations are done from left to right (no mathematical precedence like "*" before "+", etc.). The following example looks as if the content of field "width" gets 20 added to it.

```
10 = TEXT
10.field = width   # Assumption: "width" is 100
10.wrap = |+20
10.prioriCalc = 1
```

This is not the case. The result which will be rendered is: "100+20". The function "prioriCalc" is executed before the function wrap is executed, and thus only calculates the result of "field" - the expression "100". In order to get the result we anticipated, we have to make sure that "field" and "wrap" are executed before "prioriCalc" is executed. This can be achieved by using the following expression:

```
10.stdWrap.wrap = |+20
```

The stdWrap function will be executed after "field", but before "prioriCalc", thus "100+20" is wrapped,

and after that the function "prioriCalc" is executed, resulting in the value "120".

The data type

While TypoScripting, it's crucial to know what kind of datatype we are handling. Especially with stdWrap, we have noticed it's becoming common practice to combine functions arbitrarily, until the anticipated result is achieved by accident.

Only if the stdWrap functionality is mentioned explicitly, the stdWrap functions like field, data, or typolink can be used.

lang: multilanguage functionality

stdWrap offers a property "lang" with which it is possible to translate simple texts which are implemented via TypoScript on a page.

```
10 = TEXT
10.value = Imprint
10.lang.de = Impressum
10.typolink.parameter = 10
```

However, text like these are hard to translate by external editors. Especially with unknown languages, this can become a challenge.

For this case, it is best to handle the translations with constants. These can be placed together at a specific place, and implemented into TypoScript.

```
# Constants
text.imprint = Imprint
text.de.imprint = Impressum

# Setup
10 = TEXT
10.value = {$text.imprint}
10.lang.en = {$text.en.imprint}
10.typolink.parameter = 10
```

This way the translation is not depending on the TS configuration of the item.

cObject

The parameter cObject can be used to replace the content with a TypoScript object. This can be a COA, a plugin or a text like in this example:

```
10.typolink.title.cObject = TEXT
10.typolink.title.cObject.value = Impressum
10.typolink.title.cObject.lang.en = Imprint
```

Outlook

The manual is still maintained in the wiki, changes made there will regularly be merged into this manual. If you want to change something use the wiki page at http://wiki.typo3.org/Ts45min, if you think you found a bug and want to report it use the issue tracker at http://forge.typo3.org/projects/typo3v4-doc_tut_ts45/issues