TypoScript Syntax and In-depth Study

Extension Key: doc_core_ts

Language: en

Version: 4.4.0

Keywords: typoscript, syntax, forAdmins, 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.

Core Manual

This document is a Core Manual. Core Manuals address the built in functionality of TYPO3 and are designed to provide the reader with in-depth information. Each Core Manual addresses a particular process or function and how it is implemented within the TYPO3 source code. These may include information on available APIs, specific configuration options, etc.

Core Manuals are written as reference manuals. The reader should rely on the Table of Contents to identify what particular section will best address the task at hand.

Table of Contents

Introduction

About this document

This document describes the syntax of TypoScript. It also covers the nature of TypoScript and what the differences are between the various contexts in which it can be used (i.e. templates and TSconfig).

If the concept of TypoScript itself is not clear, please read the appendix "What is TypoScript?". Otherwise feel free to ignore it.

What's new

This version of the manual was updated for TYPO3 4.4. The changes include a note on the new possibility to use conditions also in page and user TSconfig. Additionally all texts were checked and gently reworked.

The manual has been moved to the new documentation template.

Credits

This document was formerly maintained by Michael Stucki and François Suter.

Additions have been made by Sebastian Michaelsen.

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_core_ts/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).

Syntax

Introduction

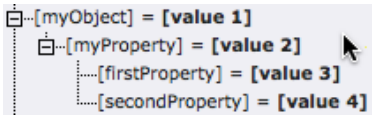
TypoScript is like a (large) multidimensional PHP array (see "Appendix A – What is TypoScript?"). Values are arranged in a tree-like hierarchy. The "branches" are indicated with periods (".") - a syntax borrowed from for example JavaScript and which conveys the idea of defining objects/properties.

Example:

```
myObject = [value 1]
myObject.myProperty = [value 2]
myObject.myProperty.firstProperty = [value 3]
myObject.myProperty.secondProperty = [value 4]
```

Referring to "myObject" we might call it an "*object with the value [value 1] and the property, 'myProperty' with the value [value 2]. Furthermore 'myProperty' has its own two properties, 'firstProperty' and 'secondProperty' with a value each ([value 3] and [value 4]).*"

The TYPO3 backend contains tools that can be used to visualize the tree structure of TypoScript. They are described in the relevant section further in this document (see "TypoScript Templates" and "TSconfig"). The above piece of TypoScript would look like this:



Contexts

There are two contexts where TypoScript is used: templates, where TypoScript is used to actually define what will appear in the TYPO3 frontend, and TSconfig, where it is used to configure settings of the TYPO3 backend. TSconfig is further subdivided into User TSconfig (defined for backend users or user groups) and Page TSconfig (defined for pages in the page tree).

Some parts of TypoScript are available in both contexts, some only in one or the other. Any difference is always mentioned in this manual.

Each context has its own chapter in this manual. It also has its own reference in a separate manual (see "Next steps" at the end of this manual).

TypoScript Syntax

TypoScript is parsed in a very simple way; line by line. This means that abstractly said each line normally contains three parts based on this formula:

```
[Object Path] [Operator] [Value]
```

Example:

```
myObject.myProperty = value 2
```

The **object path** (in this case "myObject.myProperty") is like the variable name in a programming language. The object path is the first block of non-whitespace characters on a line until one of the characters "=<>{(" (space included) is found. **Use only A-Z, a-z, 0-9, "-", "_" and periods (.) for Object Paths!**

The **operator** (in this case it is "=") can be one of the characters "=<>{(". The various operators are described below.

The **value** (in this case "value 2") is whatever characters follow the operator until the end of the line, but trimmed for whitespace at each end. Notice that values are *not* encapsulated in quotes! The value starts after the operator and ends with the line break.

Comments

When a line starts with "/" or "#" it is considered to be a comment and will be ignored.

Example:

```
// This is a comment
/ This also is a comment (only ONE slash is needed)
myObject = HTML
myObject.value = <strong> HTML - code </strong>
# This line also is a comment.
```

Comment blocks

When a line starts with "/*" or "*/" it defines the beginning or the end of a comment section respectively. Anything inside a comment section is ignored.

Rules:

/* and */ MUST be the very first characters of a trimmed line in order to be detected.

Comment blocks are not detected inside a multi-line value block (see parenthesis operator below).

Example:

```
/* This is a comment
 .. and this line is within that comment which...
  ends here:
*/  ... this is not parsed either though - the whole line is still within the comment
myObject = HTML
myObject.value (
  Here's a multiline value which
  /*
    This is not a comment because it is inside a multi-line value block
  */
)
```

Value assignment: The "=" operator

This simply assigns a value to an object path.

Rules:

Everything after the "=" sign and *up to the end of the line* is considered to be the value. In other words: You don't need to quote anything!

Be aware that the value will be trimmed, which means stripped of whitespace at both ends.

Value modifications: The ":=" operator

This operator assigns a value to an object path by calling a predefined function which modifies the existing value of the current object path in different ways.

This is very useful when a value should be modified without completely redefining it again.

Rules:

The portion after the ":=" operator and *to the end of the line* is split in two parts: A function and a value. The function is specified right next to the operator (trimmed) and holding the value in brackets (not trimmed).

There are six predefined functions:

- **prependString**: Adds a string to the beginning of the existing value.
- **appendString**: Adds a string to the end of the existing value.
- **removeString**: Removes a string from the existing value.

- **replaceString**: Replaces old with new value. Separate these using "|".
- **addToList**: Adds a comma-separated list of values to the end of a string value. There is no check for duplicate values, and the list is not sorted in any way.
- **removeFromList**: Removes a comma-separated list of values from an existing comma-separated list of values.

There is a hook inside class.t3lib_tsparser.php which can be used to define more such functions.

Example:

```
myObject = TEXT
myObject.value = 1,2,3
myObject.value := addToList(4,5)
myObject.value := removeFromList(2,1)
```

produces the same result as:

```
myObject = TEXT
myObject.value = 3,4,5
```

Code blocks: The { } signs

Opening and closing curly braces are used to assign many object properties in a simple way at once. It's called a block or nesting of properties.

Rules:

- Everything on the same line as the opening brace ("{"), but that comes *after* it is ignored.
- The "}" sign *must* be the first non-space character on a line in order to close the block. Everything on the same line, but after "}" is ignored.
- Blocks can be nested. This is actually recommended for **improved readability**.
- **Note**: You cannot use conditions inside of braces (except the [GLOBAL] condition which will be detected and reset the brace-level to zero)
- **Note**: Excessive end braces are ignored, but generate warnings in the TypoScript parser.

Example:

```
myObject = TEXT
myObject.field = title
myObject.ifEmpty.data = leveltitle:0
```

could also be written as:

```
myObject = TEXT
myObject {
      field = title
      ifEmpty {
            data = leveltitle:0
      }
}
```

Multi-line values: The ( ) signs

Opening and closing parenthesis are used to assign a *multi-line value*. With this method you can define values which span several lines and thus include line breaks.

Rules:

Note: The end-parenthesis is extremely important. If it is not found, the parser considers the following lines to be part of the value and does not return to parsing TypoScript. This includes the [GLOBAL] condition which will not save you in this case! So don't miss it!

Example:

```
myObject = HTML
myObject.value (
  <p class="warning">
    This is HTML code.
  </p>
)
```

Object copying: The "<" sign

The "<" sign is used to copy one object path to another. The whole object is copied - both value and properties - and it overrides any old objects and values at that position.

Example:

```
myObject = HTML
myObject.value = <p class="warning">This is HTML code.</p>

myOtherObject < myObject
```

The result of the above TypoScript is two independent sets of objects/properties which exactly the same (duplicates). They are *not* references to each other but actual copies:

Another example with a copy within a code block:

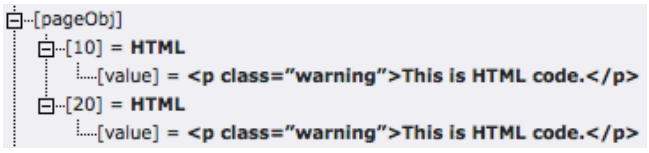
```
pageObj {
      10 = HTML
      10.value = <p class="warning">This is HTML code.</p>
      20 < pageObj.10
}
```

Here also a copy is made, although inside the "pageObj" object. Note that the copied object is referred to with its full path ("pageObj.10"). When **copying on the same level**, you can just refer to the copied object's name, **prepended by a dot**.

The following produces the same result as above:

```
pageObj {
      10 = HTML
      10.value = <p class="warning">This is HTML code.</p>
      20 < .10
}
```

which – in tree view – translates to:



Note: When the original object is changed after copying, the copy does not change! Take a look at the following code:

```
someObject = TEXT
someObject {
      value = Hello world!
      wrap = <p>|<p>
}
anotherObject < someObject
someObject.wrap = <h1>|<h1>
```

The value of the "wrap" property of "anotherObject" is " | ". It it **not** "<h1>|<h1>" because this change happens **after** the copying. This example may seem trivial, but it's easy to loose the oversight in larger pieces of TypoScript.

References: the "=<" sign

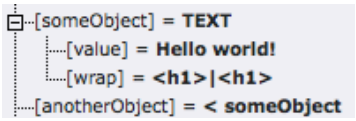
In the context of TypoScript Templates it is possible to create references from one object to another. References mean that multiple positions in an object tree can use the same object at another position without making an actual copy of the object but by simply pointing to the objects full object path.

The obvious advantage is that a **change of code to the original object affects all references**. It avoids the risk mentioned above with the copy operator to forget that a change at a later point does not affect earlier copies. On the other hand there's the reverse risk: it is easy to forget that changing the original object will have an impact on all references. References are very convenient, but should be used with caution.

Example:

```
someObject = TEXT
someObject {
      value = Hello world!
      wrap = <p>|<p>
}
anotherObject =< someObject
someObject.wrap = <h1>|<h1>
```

In this case, the "wrap" property of "anotherObject" will indeed by "<h1>|<h1>". In tree view the properties of the reference are not shown. Only the reference itself is there:



Remember: References are only available in TypoScript templates, not in TSconfig.

Object unsetting: The ">" sign:

This is used to unset an object and all of its properties.

Example:

```
myObject = HTML
myObject.value = <strong> HTML - code </strong>

myObject >
```

In this last line "myObject" is totally wiped out (removed).

Conditions: Lines starting with "["

Conditions break the parsing of TypoScript in order to evaluate the content of the condition line. If the evaluation returns true parsing continues, otherwise the following TypoScript is ignored until the next condition is found, at which point a new evaluation takes place. The next section in this document describes conditions in more details.

Rules:

Conditions apply *only* when outside of any code block (i.e. outside of any curly braces).

Example:

```
[browser = msie]
page.10.value = Internet Explorer
[else]
page.10.value = Not an Internet Explorer browser!
[end]
```

Conditions

There is a *possibility* of using so called *conditions* in TypoScript. Conditions are simple control structures, that evaluate to TRUE or FALSE based on some criteria (externally validated) and thereby determine, whether the TypoScript code following the condition and ending where the next condition is

found, should be parsed or not.

Examples of a condition could be:

- Is the browser "Internet Explorer"?
- Is a usergroup set for the current session?
- Is it Monday?
- Is the GET parameter "&language=uk" set?
- Is it my mother's birthday?
- Do I feel lucky today?

Of these examples admittedly the first few are the most realistic. In fact they are readily available in the context of TypoScript Templates. But a condition can theoretically evaluate any circumstance and return either TRUE or FALSE which subsequently means the parsing of the TypoScript code that follows.

Where conditions can be used

The *detection of conditions* is a part of the TypoScript syntax but the *validation* of the condition content always relies on the context where TypoScript is used. Therefore in plain syntax highlighting (no context) conditions are just highlighted and nothing more. In the context of TypoScript Templates there is a whole section of TSref which defines the syntax of the condition contents for TypoScript Templates. For "Page TSconfig" and "User TSconfig" conditions are implemented since TYPO3 4.3. Basically they work the same way as conditions in TypoScript templates do, but there are some small differences. For details see the according section "Conditions" in TSconfig.

The syntax of conditions

A condition always has its own line and the line is detected by " [" (square bracket) being the first character on that line:

```
(Some TypoScript)
[ condition 1 ][ condition 2]
(Some TypoScript only parsed if condition 1 or condition 2 are met.)
[GLOBAL]
(Some TypoScript)
```

As you can see from this example, the line "**[GLOBAL]**" also is a condition. It is built-in into TypoScript and always returns TRUE. The line "[condition 1][condition 2]" is another condition. If "[condition 1][condition 2]" is TRUE, then the TypoScript in the middle would be parsed until [GLOBAL] (or [END]) resets the conditions. After that point the TypoScript is parsed for any case again.

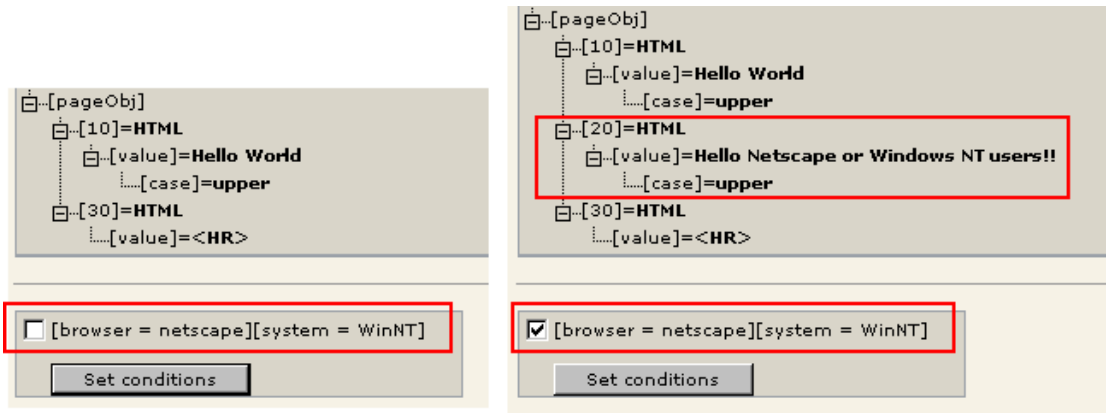
Notice: The condition line "[condition 1][condition 2]" conveys the idea of *two conditions* being set, but from the TypoScript parsers point of view the *whole line* is the condition - it is in the context of TypoScript Templates that the condition line content is broken down into smaller units ("[condition 1]" and "[condition 2]") which are individually evaluated and connected by a logical OR before they return the resulting TRUE or FALSE value. (That is all done with the class t3lib_matchCondition).

Here is an example of some TypoScript (from the context of TypoScript Templates) where another text is outputted if you use the Internet Explorer web browser (instead of for example Internet Explorer) or use Windows NT as operating system:

```
pageObj.10 = HTML
```

```
pageObj.10.value = Hello World
pageObj.10.value.case = upper

[browser = msie][system = WinNT]
pageObj.20 = HTML
pageObj.20 {
  value = Hello Internet Explorer or Windows NT users!
  value.case = upper
}

[GLOBAL]
pageObj.30 = HTML
pageObj.30.value = <hr>
```

You can now use the Object Browser to actually see the difference in the parsed object tree depending on whether the condition evaluates to TRUE or FALSE (which can be simulated with that module as you can see):



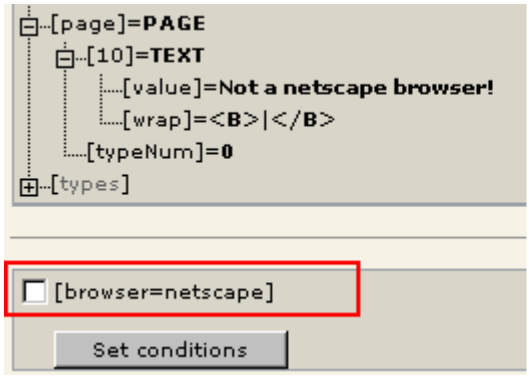
The special [ELSE], [END] and [GLOBAL] conditions

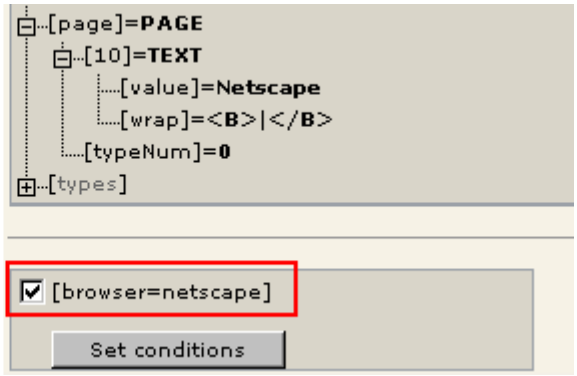
There's a special condition called [ELSE] which will return TRUE if the previous condition returned FALSE. To end an [ELSE] condition you can use either [END] or [GLOBAL]. For all three conditions you can also use them in lower case.

Here's an example of using the [ELSE]-condition (also in the context of TypoScript Templates):

```
page.typeNum = 0
page = PAGE
page.10 = TEXT

[browser=msie]
page.10.value = Internet Explorer

[else]
page.10.value = Not an Internet Explorer browser!

[end]

page.10.wrap = <strong>|</strong>
```

Here we have one output text if the browser is Internet Explorer and another if not. Anyways the text is wrapped by | as we see, because this wrap is added outside of the condition block (here after the [END]-condition).





The fact that you can "enable" the condition in the TypoScript Object Browser is a facility provided to simulate the outcome of any conditions you insert in a TypoScript Template. Whether or not the conditions correctly validate is only verified by actually getting a (in this example) Internet Explorer browser and hitting the site.

Another example could be if you wanted to do something special in case a bunch of conditions is NOT true. There's **no negate-character**, but you could do this:

```
[browser=msie][usergroup=3]
  # Enter nothing here!
[else]
  page.10.value = This text is only displayed if the conditions above are not TRUE!
[end]
```

Where to insert conditions in TypoScript?

Conditions can be used *outside* of confinements (curly braces) only!

So, this is valid:

```
someObject {
  1property = 234
}
[browser=msie]
someObject {
  2property = 567
}
```

But this is ***not valid***:

```
someObject {
  1property = 234
  [browser=msie]
  2property = 567
}
```

When parsed with syntax highlighting you will see this error:

```
someObject {
  1property = 234
  [browser=netscape] - ERROR: Line 2: Object Name String, "[browser" contains invalid character "[".
  2property = 567
}
```

This means that the line was perceived as a regular definition of "[object path] [operator] [value]" and not as a condition.

The [GLOBAL] condition

However for the special condition [GLOBAL] (which resets any previous condition scope), it is a bit

different since that will be detected at *any line* except within multiline value definitions.

```
someObject {
  1property = 234
  [GLOBAL]
  2property = 567
}
```

But you will still get some errors if you syntax highlight it:

```
someObject {
  1property = 234
  [GLOBAL] - ERROR: Line 2: On return to [GLOBAL] scope, the script was short of 1 end brace(s)
  2property = 567
} - ERROR: Line 4: An end brace is in excess.
```

The reason for this is that the [GLOBAL] condition aborts the confinement started in the first line resulting in the first error ("... short of 1 end brace(s)"). The second error appears because the end brace is now in excess since the "brace level" was reset by [GLOBAL].

So, in summary; the special [global] (or [GLOBAL]) condition will break TypoScript parsing within braces at any time and return to the global scope (unless entered in a multiline value). This is true for any TypoScript implementation whether other condition types are possible or not. Therefore you can use [GLOBAL] (put on a single line for itself) to make sure that following TypoScript is correctly parsed from the top level. This is normally done when TypoScript code from various records is combined.

Summary

- Conditions are detected by "[" as the first line character (whitespace ignored).
- Conditions are evaluated in relation to the context where TypoScript is used. They are widely used in TypoScript Templates but not at all used with "Page TSconfig" or "User TSconfig".
- Special conditions [ELSE], [END] and [GLOBAL] exist.
- Conditions can be used outside of confinements (curly braces) only. However the [GLOBAL] condition will always break a confinement if entered inside of one.

Includes

You can also add include-instructions in TypoScript code. Availability depends on the context, but it works with TypoScript templates, Page TSconfig and User TSconfig.

An include-instruction looks like this:

```
<INCLUDE_TYPOSCRIPT: source="FILE: fileadmin/html/mainmenu_typoscript.txt">
```

- It must have its own line in the TypoScript template, otherwise it is not recognized.
- It is processed BEFORE any parsing of TypoScript (contrary to conditions) and therefore does not care about the nesting of confinements within the TypoScript code.

The "source" parameter points to the source of the included content. The string before the first ":" (in the example it is the word "FILE") will determine which source the content is coming from. This is the only option available:

Option	Description
FILE	A reference to a file relative to PATH_site. Must be less than 100 KB. Cannot contain ".." (double periods, back path). If you prefix the relative path with such as "EXT:myext/directory/file.txt" then the file included will be searched for in the extension directory of extension "myext", subdirectory "directory/file.txt".

TypoScript Templates

Constants

What are constants?

Constants are values defined in the "Constants"-field of a template. They follow the syntax of ordinary TypoScript!

Note, reserved name: The object or property "file" is always interpreted as data type "resource".

Note: Toplevel "object" TSConstantEditor cannot be used. It's reserved for configuration of the Constant Editor module.

Example:

Here "bgCol" is set to "red" and "file.toplogo" is set to "logo*.gif" which is found in the resource-field of the template.

```
bgCol = red
topimg.width = 200
topimg.file.pic2 = fileadmin/logo2.gif
file.toplogo = logo*.gif
```

This could also be defined in other ways, e.g. like this:

```
bgCol = red
file {
  toplogo = logo*.gif
}
topimg {
  width = 200
  file.pic2 = fileadmin/logo2.gif
}
```

(The objects in bold are the reserved word "file" and the properties are always of data type "resource".

Using constants

Constants are inserted in the template-setup by performing an ordinary str_replace operation! You insert them like this:

```
{$bgCol}
{$topimg.width}
{$topimg.file.pic2}
{$file.toplogo}
```

Example:

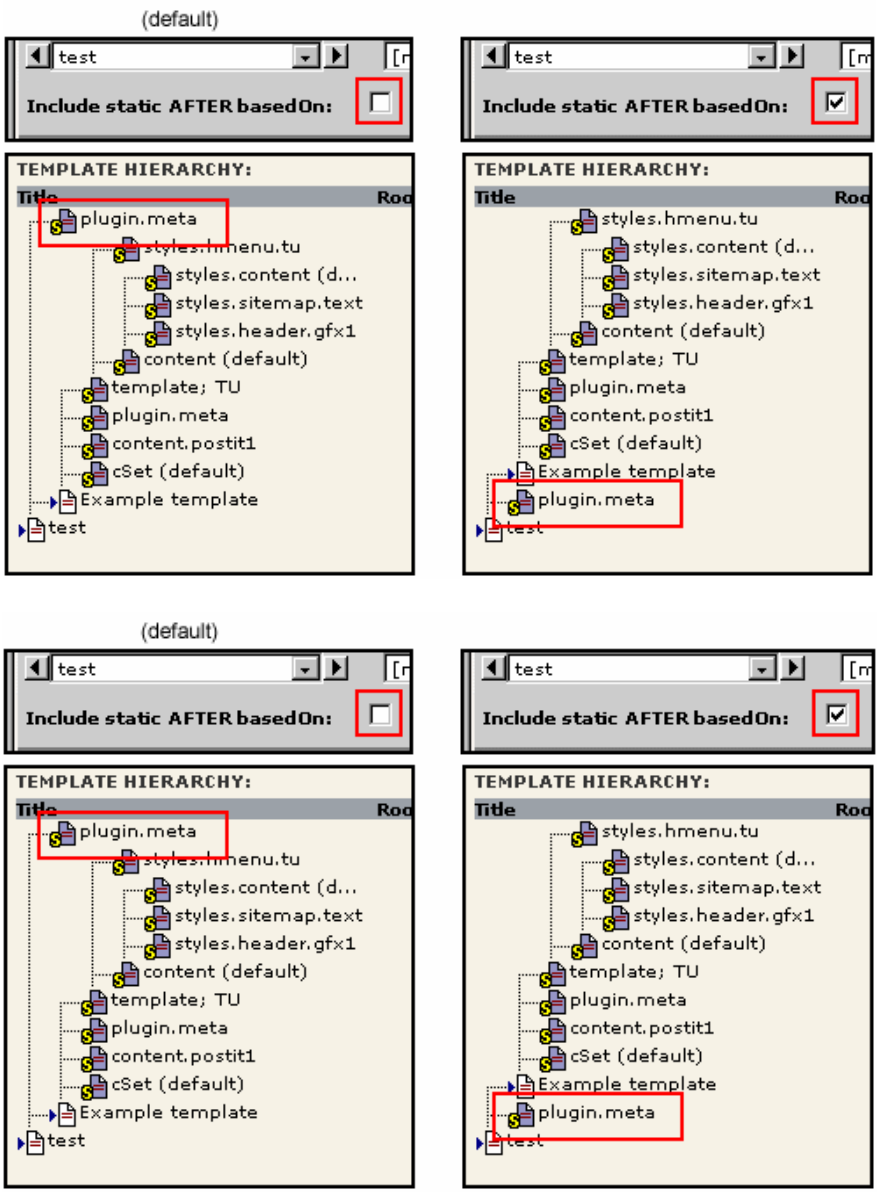
```
page = PAGE
page.typeNum = 0

page.bodyTag = <body bgColor="{$bgCol}">
page.10 = IMAGE
page.10.file = {$file.toplogo}
```

Only defined constants are substituted.

Constants in included templates are also substituted as the whole template is just one large chunk of text.

Constants are case sensitive.

You should use a systematic naming scheme for constants. Seek inspiration in the code-examples around.

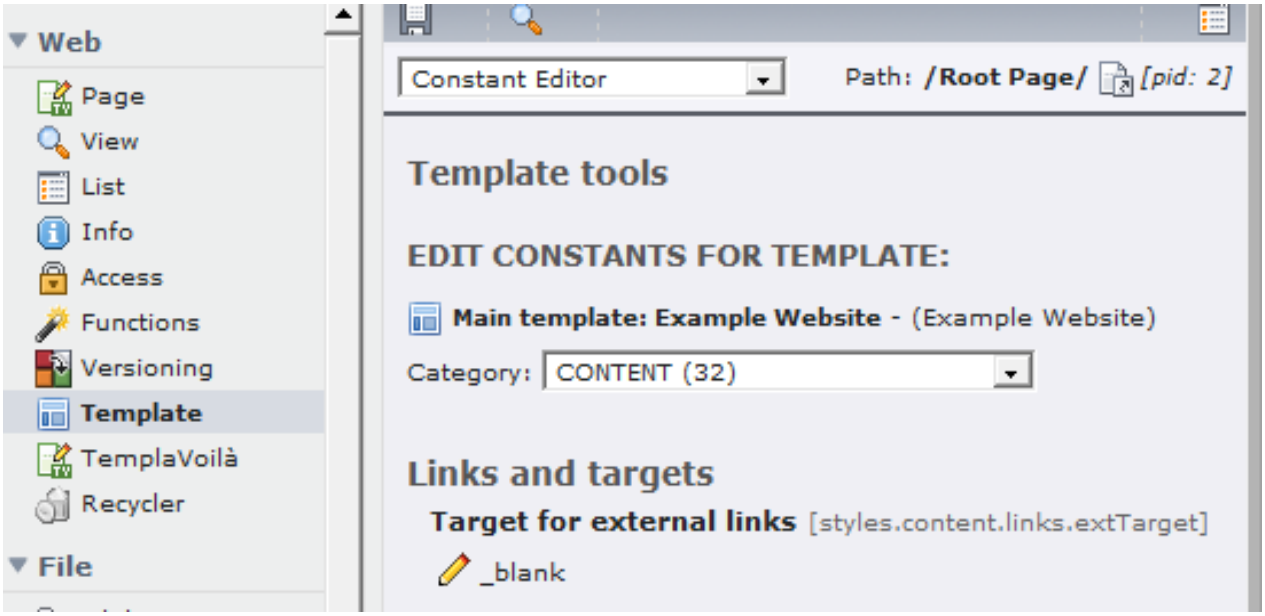


Notice how the constants in the setup code are substituted. In the Object Browser, you can monitor the constants with or without substitution. Also notice that the value "logo*.gif" was resolved to the resource "uploads/tf/logo_01.gif"

(Note: The "Constants display" function is not available if you select "Crop lines".)

Declaring constants for the Constant Editor

You can put comments anywhere in your TypoScript. Comments are always ignored by the parser when the template is processed. But the backend module Web > Template has the ability to utilize comments in the constant editor that makes simple configuration of a template even easier than constants already make it themselves.



When the TypoScript "Constant Editor" parses the template, *all* comments before every constant-definition are registered. You can follow a certain syntax to define what category the constant should be in, which type it has and what explanation there is about the constant. This is an example containing several constant definitions:

```
styles.content.textStyle {
    # cat=content/cText/1; type=; label= Bodytext font: This is the fontface used for text!
  face =
    # cat=content/cText/2; type=int[1-5]; label= Bodytext size
  size =
    # cat=content/cText/3; type=color; label= Bodytext color
  color =
  color1 =
  color2 =
  properties =
}
```

It's totally optional to make the comments before your constants compliant with this system, but it's very useful later on if you want others to make simple corrections to your template or distribute the template in a template-archive or such.

Default values:

The default value of a constant is determined by the value the constant has BEFORE the last template (the one you're manipulating with the module) is parsed (previous templates are typically included static_template-records!), unless the mark ###MOD_TS:EDITABLE_CONSTANTS### is found in the last template, in which case constant-definitions before this mark are also regarded default-values.

This means that all constant values - or values after the mark ###MOD_TS:EDITABLE_CONSTANTS### if present - in the template-record you're manipulating are regarded to be your customized extensions.

Comments:

How the comments are perceived by the module:

- All comments set on lines before the constant wherever it's found in the templates are parsed sequentially.
- Each line is split by the ";" (semicolon) character, that separates the various parameters
- Each parameter is split by the "=" (equal) sign to separate the parameter "key" and the "value".

Keys:

cat=

- Comma-separated list of the categories (case-insensitive) that the constant is a member of. You should *list only one category*, because it usually turns out to be confusing for users, if one and the same constant appears in multiple categories!
- If the chosen category is *not* found among the default categories listed below, it's regarded a new category.
- If the category is empty (""), the constant is excluded from the editor!

Predefined Categories

Category	Description
basic	Constants of superior importance for the template-layout. This is dimensions, imagefiles and enabling of various features. The most basic constants, which you would almost always want to configure.
menu	Menu setup. This includes fontfiles, sizes, background images. Depending on the menutype.
content	All constants related to the display of pagecontent elements.
page	General configuration like metatags, link targets.
advanced	Advanced functions, which are used very seldom.

Subcategories:

There are a number of subcategories to use. Subcategories are entered after the category-name separated by a slash "/". Example: "basic/color/a"

This will make the constant go into the "BASIC"-category, be listed under the "COLOR"-section and probably be one of the top-constants listed, because the "a" is used to sort the constants in a subcategory. If "a" was not entered, the default is "z" and thus it would be one of the last colors to select. As the third parameter here, you can choose whatever you like.

You can use one of the predefined subcategories or define your own. If you use a non-existing subcategory, your constant will just go into the subcategory "Other".

Predefined Subcategories

Standard subcategories (in the order they get listed in the Constant Editor):

Subcategory	Description
enable	Used for options that enable or disable primary functions of a template.
dims	Dimensions of all kinds; pixels, widths, heights of images, frames, cells and so on.
file	Files like background images, fonts and so on. Other options related to the file may also enter.
typo	Typography and related constants.
color	Color setup. Many colors will be found with related options in other categories though.

links	Links: Targets typically.
language	Language specific options.

Subcategories based on the default content elements

cheader,cheader_g,ctext,ctextpic,cimage,cbullets,ctable,cuploads,cmultimedia,cmailform,csearch,clogin,csplash,cmenu,cshortcut,clist,cscript,chtml

These are all categories reserved for options that relate to content rendering for each type of tt_content element. See static_template "content (default)" and "styles.content (default)" for examples.

Custom Subcategories

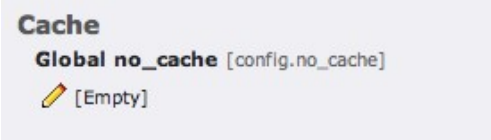
To define your own Subcategory put a comment including the parameter "customsubcategory". Here is an example:

```
# customsubcategory=cache=LLL:EXT:myext/locallang.xml:cache
```

This line defines the new Subcategory "cache" which will be available for your Constants defined AFTER this line. Usage example:

```
#cat=Site conf/cache/a; type=boolean; label=Global no_cache
config.no_cache = 0
```

Will look in the Constant Editor like this:



type=

Type	Description
int [low-high]	Integer, opt. in range "low" to "high"
int+	Positive integer
offset [L1,L2,...L6]	Comma-separated integers. Default is "x,y", but as comma separated parameters in brackets you can specify up to 6 labels being comma separated! If you wish to omit one of the last 4 fields, just don't enter a label for that element.
color	HTML color
wrap	HTML-code that is wrapped around some content.
options [item1,item2,...]	Selectbox with values/labels item1, item2 etc. Comma-separated. Split by "=" also and in that case, first part is label, second is value
boolean [truevalue]	Boolean, opt. you can define the value of "true", def.=1
comment	Boolean, checked= "", not-checked = "#".
file [ext-list/IMAGE_EXT]	Selectorbox with resources. Opt. list allowed extensions (no space in list!), eg. "[ttf]" or "[txt,html,htm]". You can also enter "[IMAGE_EXT]" in which case the default image-extensions are listed. (used for datatype "imgResource")
string (the default)	Just a string value
user	...

label=

Text string, trimmed.

Split by the first ":" to separate a header and body of the comment. The header is displayed on it's own line in bold.

This can be localized by using the traditional "LLL" syntax. Example:

```
#cat=Site conf/cache/a; type=boolean; label=LLL:EXT:examples/locallang.xml:config.no_cache
config.no_cache = 0
```

Note that a single string is referenced (not one for the header and one for the description). This means that the localized string must contain the colon separator (":"). Example:

```
<label index="config.no_cache">Global no_cache:Check to box to turn off all cache</label>
```

TSConstantEditor.[category]

In addition to using constants, you can also configure a category in the constant editor by a special top-level TypoScript "object" in the *constants*-field. The name is "TSConstantEditor" and any properties to this object will NOT be substituted like any other constant normally would.

Property:	Data type:	Description:	Default:
header	string	Header, displayed in upper-case.	
description	string, break by //	Description, enter "//" to create a line break.	
bulletlist	string, break by //	Lines for a bulletlist, enter "//" (double-slash) in order to break to next bullet.	
image	image	This is an optional image you can attach to the category. The image would normally show a given configuration of the template and contain numbered marks, that indicate positions that are referred to by the constants, listed in the number-array. The image must be located in "gfx/" in the module path OR be a file from the resource-list of the template.	
Array, 1-20	list of constant-names	Each number refers to a number-mark on the image and all constants that are listed at each number will get a little number-icon by it's header.	

[TSConstantEditor.[category]]

Example:

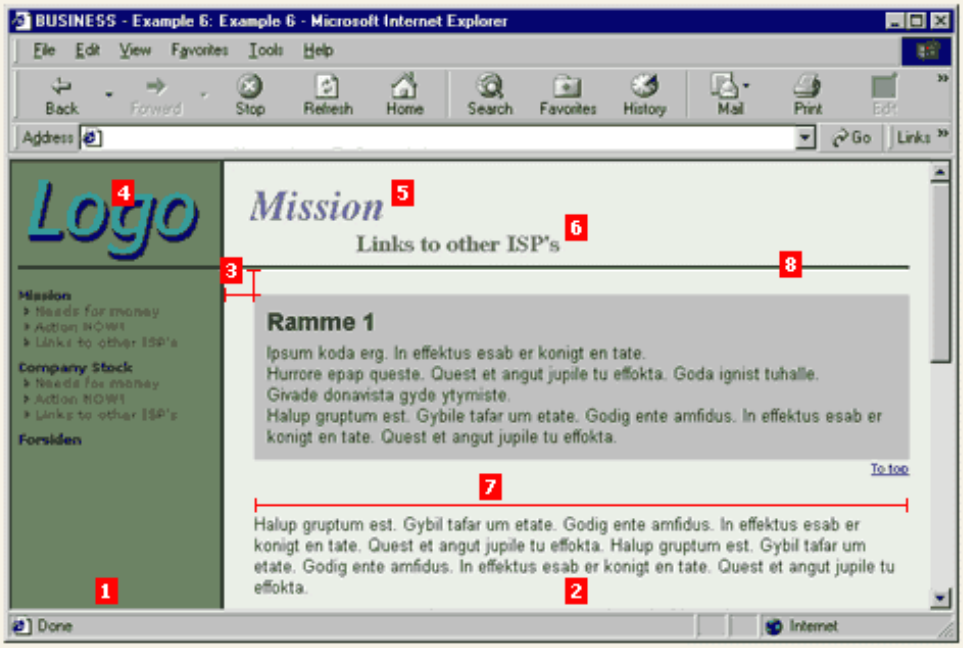
```
## TSConstantEditor Configuration
TSConstantEditor.basic {
  header = Standard Template "BUSINESS"
  description = BUSINESS is a framebased template in a very simple layout, based on ....
  bulletlist = Left-frame image in the top. The dimensions are fixed to ....
  image = gfx/BUSINESS_basic.gif

  1 = leftFrameWidth,menu.file.bgImg,menu.bgCol
  2 = page.file.bgImg,bgCol
  3 = contentOffset
  4 = file.logo
  5 = page.L0.titleWrap
  6 = page.L1.titleWrap
  7 = contentWidth,styles.content.imgtext.maxW
  8 = page.lineCol
}
```

This example shows how the static template "BUSINESS", which you find in the system extension "statictemplates", is configured for the **basic**-module. This is how it looks like in TYPO3:

STANDARD TEMPLATE "BUSINESS"

BUSINESS is a framebased template in a very simple layout, based on text. There are 2 frames; a menuframe (to the left) and a page-frame.

- Left-frame image in the top. The dimensions are fixed to 150 x 80 pixels (normally you are free to choose the dimensions yourself!) (4)
- 2-level textual menu. Select: font-teg-properties like face, size and color for each level. Select: graphical bullet for the 2nd level and optionally a mouseover-bullet.
- The pagetitles of the 1st and 2nd level for a given page is displayed in the top. Select font-tag-properties for each title (like with the menus) (5 and 6)
- Define background-images of each frame, the left and page frame. Also select the width of the left frame. (1)
- Optional ruler across the page. Select the color. (8)



User-defined (custom) constants fields

...

Sorting out details

More about syntax, semantics and TypoScript compared to XML and XSLT

If you think you perfectly understand TypoScript now (you might already...), then don't bother with this section. I even risk confusing you again. But anyways, here it is - more theoretical information on TypoScript including references to the relationship between XML and XSLT:

XML and TypoScript - all about syntax:

A chunk of "TypoScript code" is like an XML document - it only contains information in a structured way, nothing else. But in order to store information in TypoScript or XML you need to follow the **syntax** - rules about *how* the information values can be inserted in the structure. The syntax is like the grammar for a human language defines in which order words can be combined.

For XML such rules include that "all tags must be ended, e.g. ... or
", correct nesting, using lowercase for element and attribute names etc. If you follow these rules, an XML document is called 'well formed'". For TypoScript similar rules exist like "The = operator assigns the rest of the text string as the value to the preceding object path" or "A line starting with # or / is a comment and therefore ignored".

XSLT and "TypoScript Templates" - all about semantics (meaning, function):

This is syntactically valid XML:

```
<asdf>qwerty</asdf>
```

And this is syntactically valid TypoScript:

```
asdf = qwerty
```

And this is syntactically valid English:

```
footballs sing red
```

But none of these examples make sense without some reference which defines how elements, values and words can be combined in order to form *meaning* - they need a *context*. This is called **semantics**. For human languages we have it internally because we know footballs can't sing and you can't "sing red" - we know it's nonsense while the sentence is correctly formed. For an XML document you have a DTD or schema which defines if the element "<asdf>" exists on that level in the document (and if not, then it's nonsense) and for TypoScript you have a *reference document* for the context where the TypoScript syntax is used to define an information hierarchy - for instance the "TSref" for TypoScript Templates or the "TSconfig" document for "Page TSconfig" or "User TSconfig".

So an XML document makes sense only if you know the relationship of the information stored inside of the document and that is required if you want an XSLT stylesheet to transform one XML document to another document. In fact an XSLT stylesheet is like a translator for XML - it translates one "language" of XML into another "language" of XML.

Similarly TypoScript is used as *the syntax* to build "TypoScript Templates" (*containing semantics - meaning*); the information only makes sense if it follows the rules defined in the "TSref" document.

BTW, the comparison of "TypoScript Templates" and "XSLT" is intentional since both can be described as *declarational programming languages* - programming by *configuring values* which *instructs* a *real procedural program* (e.g. the TypoScript Frontend Engine which is written in PHP) how to output data.

XSL was not the way to go as the XSL proposals were public from November 1999 which is a little later than TypoScript was born. Anyways, they were a brand new technology and it did not seem smart using it until it was more stable or had proved to be useful and supported. At that time there certainly would

be no significant XSL(T) processors.

But TypoScript is also a concept that fits the PHP and TYPO3 configuration very well (although it has been taken very far in certain areas). TypoScript is basically a large object-like structure of information which can be set from text-files (DB -records...) and the TYPO3 default PHP frontend code just reacts to the settings in TypoScript.

TypoScript was not destined to be a procedural language and it is not today! It could be compared to the Windows registration database which is a similar bunch of hierarchical configuration.

For more on syntax and semantics, you can read this article that I found on the net.

Where is TypoScript used?

This question cannot be answered completely since this document only describes the syntax of TypoScript and not any of the contexts where TypoScript syntax is applied for configuration; theoretically anyone could use the TypoScript parser class in TYPO3 to generate configuration for their own local extensions using their own local "semantics" (see another chapter in this document).

But at least we can mention the three main applications of the TypoScript syntax as used in the core parts of TYPO3:

- **Page TSconfig**: Customization of branches of the Page tree.
- **User TSconfig**: Customization of users and their groups.
- **TypoScript Templates**: Definition and customization of each website found in the page tree.

Page TSconfig

Each page record in TYPO3 has a field where you can enter "TSconfig code". The main idea with Page TSconfig is that you can configure individual behavior for separate parts of the page tree. This is possible because the TypoScript code entered in the TSconfig fields is accumulated for all pages in the root line of the current position in the page tree starting from the root and going outwards. Thus TypoScript settings in TSconfig fields of outer pages can override those settings of pages closer to the root.

For instance you may have two separate websites located in separate branches of the page tree. The one website might support content from only the "normal" column while the other website supports it for both the "normal" and "border" column. Since the Page module by default shows all four possible columns you may want to instruct the page module to show only the normal column and normal + border column respectively. But this will only be possible if you can somehow tell the system that from this page and outwards, use only "normal" column and from that page and outwards use only "normal" + "border" column. So in the root page of the two-column website you enter this line in the TSconfig field:

And likewise for the one-column website you enter this value in the TSconfig field of the root page:

For any subpage of the root page where the configuration was entered the "Page" module will receive the value of the property "colPos_list". Accordingly only the configured columns will be shown.

The objects and properties you can use here are generally defined in the document "TSconfig" in addition to local extension documents.

User TSconfig

Each frontend and backend user and group has a field for input of TSconfig code. The main idea with User TSconfig is that you can configure individual behavior for groups and even users themselves. This gives you the possibility to set values, which are very detailed, much more detailed than what you want to set as permission settings in the main forms for users/groups. For instance you could configure how many clipboard pads a user should see or whether a "Save document and add new" button should appear for all forms for this user - stuff which is clearly too detailed for a spot in the main form for permissions and settings.

Like with Page TSconfig the content of the TSconfig fields are accumulated in a certain order; the order of member groups and finally the users own fields settings. Thus a setting for a user will override the setting for one of his member groups.

Here is an example of what you can do with User TSconfig for a backend user. This line will enable the "Save document and create new" button in editing forms:

The objects and properties you can use here are generally defined in the document "TSconfig" in addition to local extension documents.

TypoScript Templates

The most (in)famous and extensive use of TypoScript is in TypoScript Templates for the frontend engine. There TypoScript is used to configure how the frontend engine will put together the website output. This is probably also where TypoScript clashes most with traditional ideas of template building in web design and confuses people to think of TypoScript as a programming language - with the result that they find it even more confusing. (If TYPO3 has a scripting language it is *not* TypoScript but PHP!)

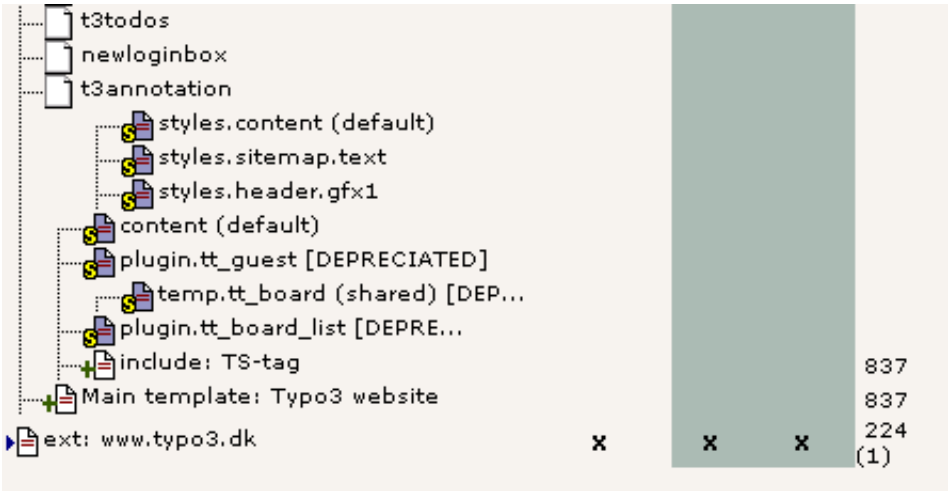
This introduction to TypoScript tries to eliminate this confusion. Therefore let us make two statements about how TYPO3 handles templates:

- **No fixed template method**: TYPO3 does *not* offer *one fixed way* to dealing with templates for websites; rather you are *set free* to choose the way *you* find most appealing. You can use:
  - **HTML templates**: Configure TYPO3 to facilitate external HTML-templates with markers and subparts. Popular and familiar for most people. Please see the tutorial "Modern Template Building, Part 1".
  - **Fluid templates**: Configure TYPO3 to use extbase and fluid (available as system extensions since TYPO3 4.3) for templating. This allows Fluid-style variables with curly braces.
  - **External Templating Engines**: Configure TYPO3 to use XSLT stylesheets with an XSLT processor. This is done either by an extension providing this functionality or by writing your own extension for it.
  - **Custom PHP**: Configure TYPO3 to call your own PHP code which generates content in any way you may prefer. This might include using third party templating engines!
  - **TS content objects**: Build the page by the "content objects" of the Frontend Engine. These cObjects are *accessible/programmable* through the TypoScript syntax.

– **TypoScript Templates *determine* the method**: No matter which template method (see list above) you would like to use TYPO3 needs to be told *which one*! And *this* is what the TypoScript Template does first and foremost; it is used to configure basic and advanced behaviors of the frontend engine so that the site rendering gets done.

A TypoScript Template works a little like the Page TSconfig; it is a database record attaching its TypoScript content to a certain page and from that page and outwards the configuration contained in the TypoScript will affect the pages until a new template record is found which overrides properties from earlier in the tree. Thus TypoScript Template records are effectively defining which page is the root page of a website:

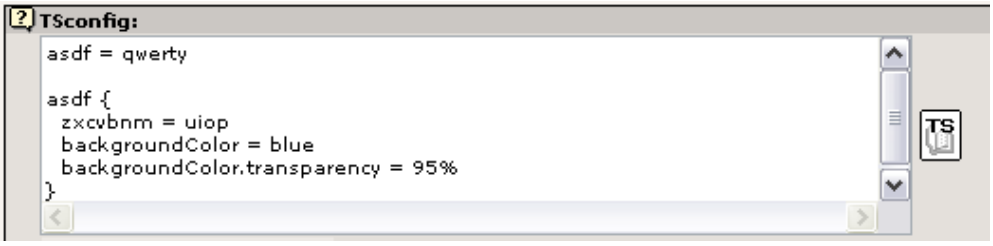
TypoScript Templates contain a field for the TypoScript configuration code ("Setup" field) but a template record like the one on the picture above ("NEW typo3.org") can also contain references to other template records and files which contain predefined generally useful TypoScript code which is included and thus reusable/sharable across templates. The order of included TypoScript template records/files can be seen visually with the Template Analyzer, which you find inside the "Template" module in the backend (if it is not there, install the system extension "tstemplate_analyzer"):



To get more background knowledge about how TypoScript templates work, please read the document "TypoScript Templates". To read about all standard objects and properties you can use in building TypoScript templates you should refer to the TypoScript reference - TSref. For practical examples when you want to learn by doing, look at the Getting Started manual.

Entering TypoScript

Since TypoScript has a line-based syntax which most of all resembles what is found in simple text based configuration files, it is also authored in ordinary textarea input fields inside of TYPO3. An example is the "TSconfig" box of a regular page header:



This is how TypoScript is typically entered - directly in form fields. Since TYPO3 4.2 a JavaScript-based editor is available for TypoScript templates only. It provides line-numbering and syntax highlighting. Since TYPO3 4.3, it also provides auto-completion.

If you don't see the code with syntax highlighting as in the screenshot above, make sure the system extension "t3editor" is loaded. Also the editor is available only when editing TypoScript from the Web > Template module (function: "Info/Modify") and not when editing the whole template record.

Other helpful features:

- There is the "TS wizard" icon which is often found to the right of the textarea - this can help you finding properties for the current TypoScript context.
- There also is the ability to insert an include-tag in any TypoScript field (see later in this document) which refers to an external file which can contain TypoScript - and that file can be edited with an external editor with whatever benefits that has.

Parsing, storing and executing TypoScript

Parsing TypoScript

This means that the TypoScript text content is transformed into a PHP array structure by following the rules of the TypoScript syntax. But still the meaning of the parsed content is not evaluated.

During parsing, syntax errors may occur when the input TypoScript text content does not follow the rules of the TypoScript syntax. The parser is however very forgiving in that case and it only registers an error internally while it will continue to parse the TypoScript code. Syntax errors can therefore be seen only with a tool that analyzes the syntax - like the syntax highlighter does.

The class "t3lib_tsparser" is used to parse TypoScript content. Please see the appendix "The TypoScript parser API" in this document for details.

Storing parsed TypoScript

When TypoScript has been parsed it is stored in a *PHP array* (which is often serialized and cached in the database afterward). If you take the TypoScript from the introduction examples and parse it, you

will get a result like below:

First, the TypoScript:

```
asdf = qwerty
asdf {
  zxcvbnm = uiop
  backgroundColor = blue
  backgroundColor.transparency = 95%
}
```

Then after parsing it with the function "parse()" in the t3lib_tsparser class, the internal variable $this->setup in that class will contain a PHP array which looks like this (with the print_r() PHP function):

```
Array
(
  [asdf] => qwerty
  [asdf.] => Array
  (
    [zxcvbnm] => uiop
    [backgroundColor] => blue
    [backgroundColor.] => Array
    (
      [transparency] => 95%
    )
  )
)
```

You can also print the array by an API function in TYPO3, namely t3lib_div::view_array() or just debug(). Then it looks like this:

asdf	qwerty	
asdf.	zxcvbnm	uiop
	backgroundColor	blue
	backgroundColor.	transparency 95%

As you see the value ("blue") of the property "backgroundColor" can be fetched by this PHP code:

```
$this->setup['asdf.']['backgroundColor']
```

So you can say that TypoScript offers a text-based *interface* for getting values into a multidimensional PHP array from a simple text field or file. This can be very useful if you need to take that kind of input from users without giving them direct access to PHP code - hence the reason why TypoScript came into existence.

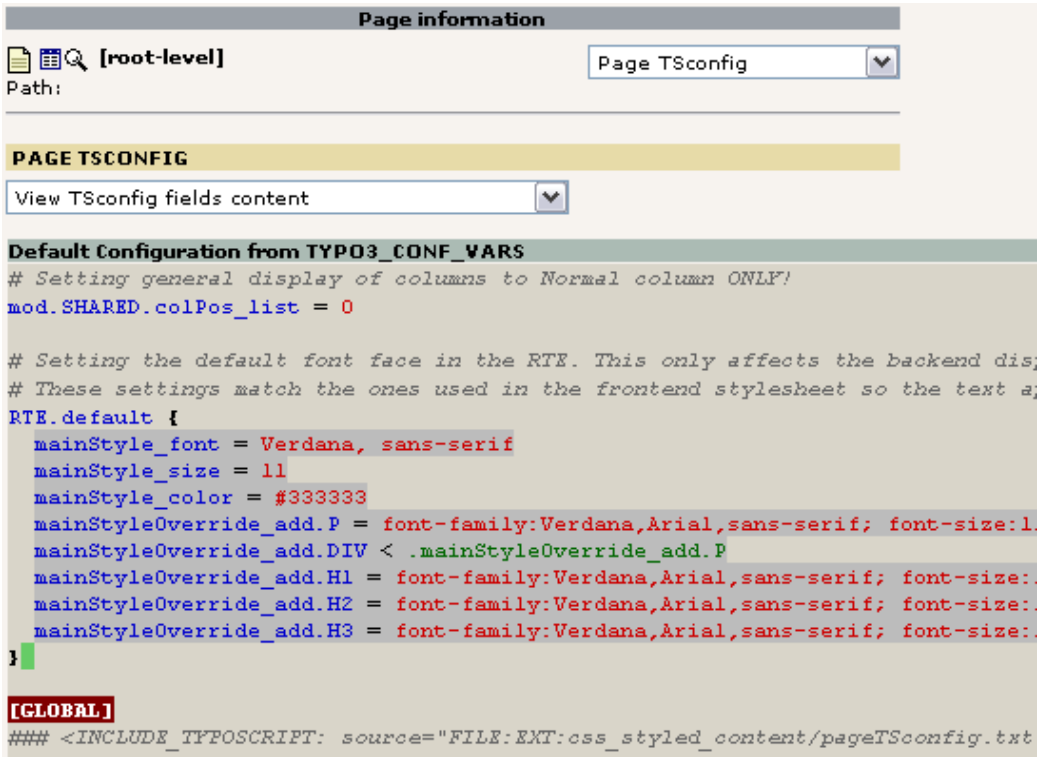
"Executing" TypoScript

Since TypoScript itself contains only information(!) you cannot "execute" it. The closest you come to "executing" TypoScript is when you take the PHP array with the parsed TypoScript structure and pass it to a PHP function which *then* performs whatever actions according to the values found in the array. This is the syntax/semantics debate again.

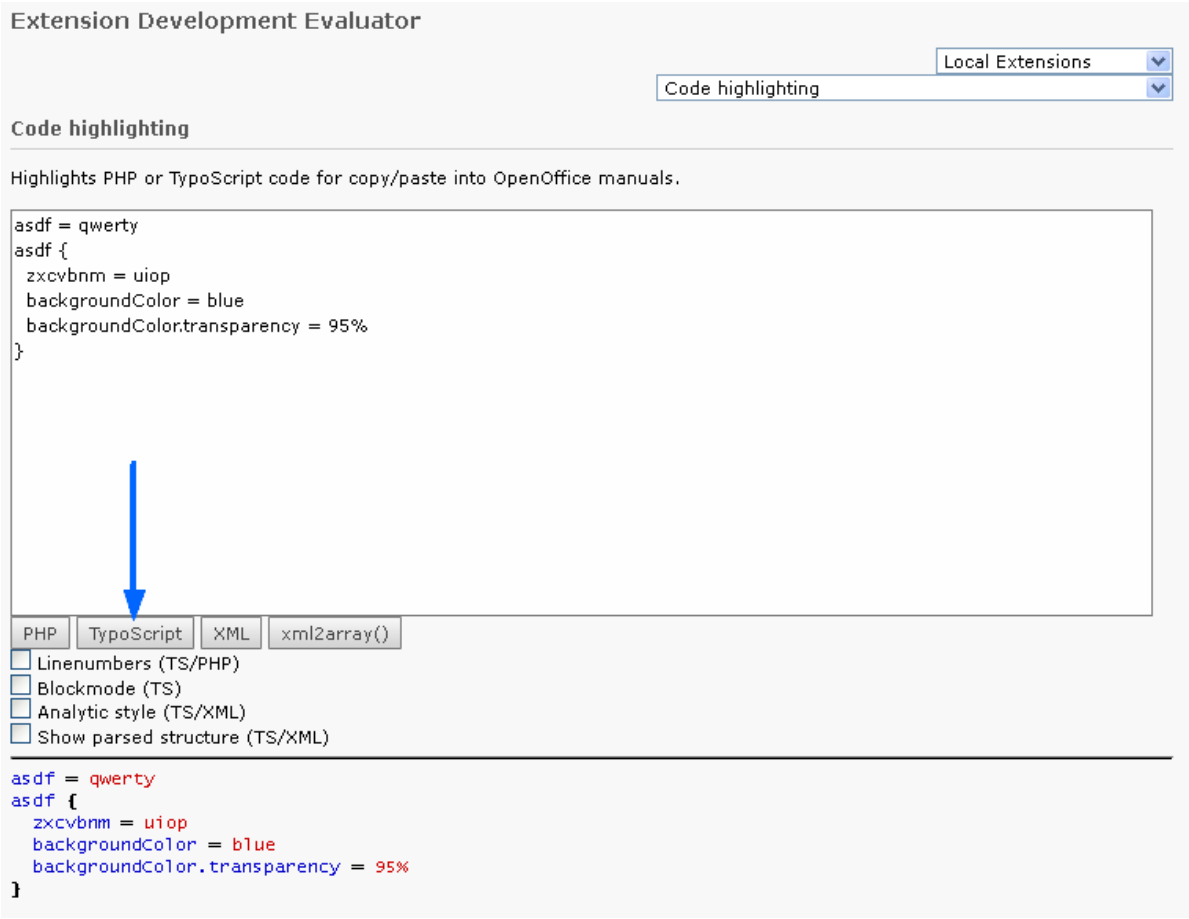
Syntax highlighting and debugging

Syntax highlighting of TypoScript code is done by various analysis applications in TYPO3 like the Template Analyzer for TypoScript Templates or the User Admin module or Page TSconfig function in the Info module. These typically allows you to view the TypoScript in each context highlighted with syntax.

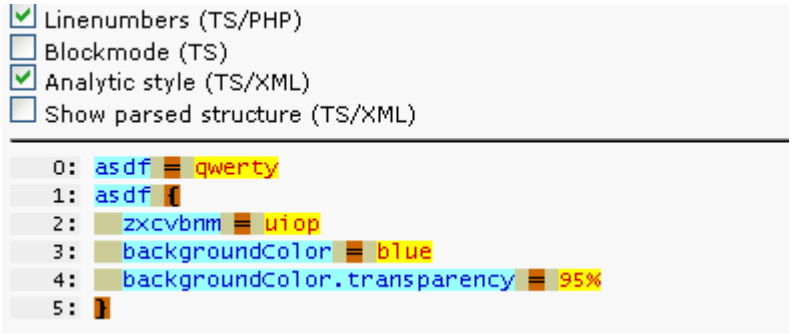
Here is an example from Page TSconfig:



In the extension "extdeveval" you will also find a tool, "Code highlighting", which can analyze TypoScript code ad hoc. This is what you get, when you just press the TypoScript button:



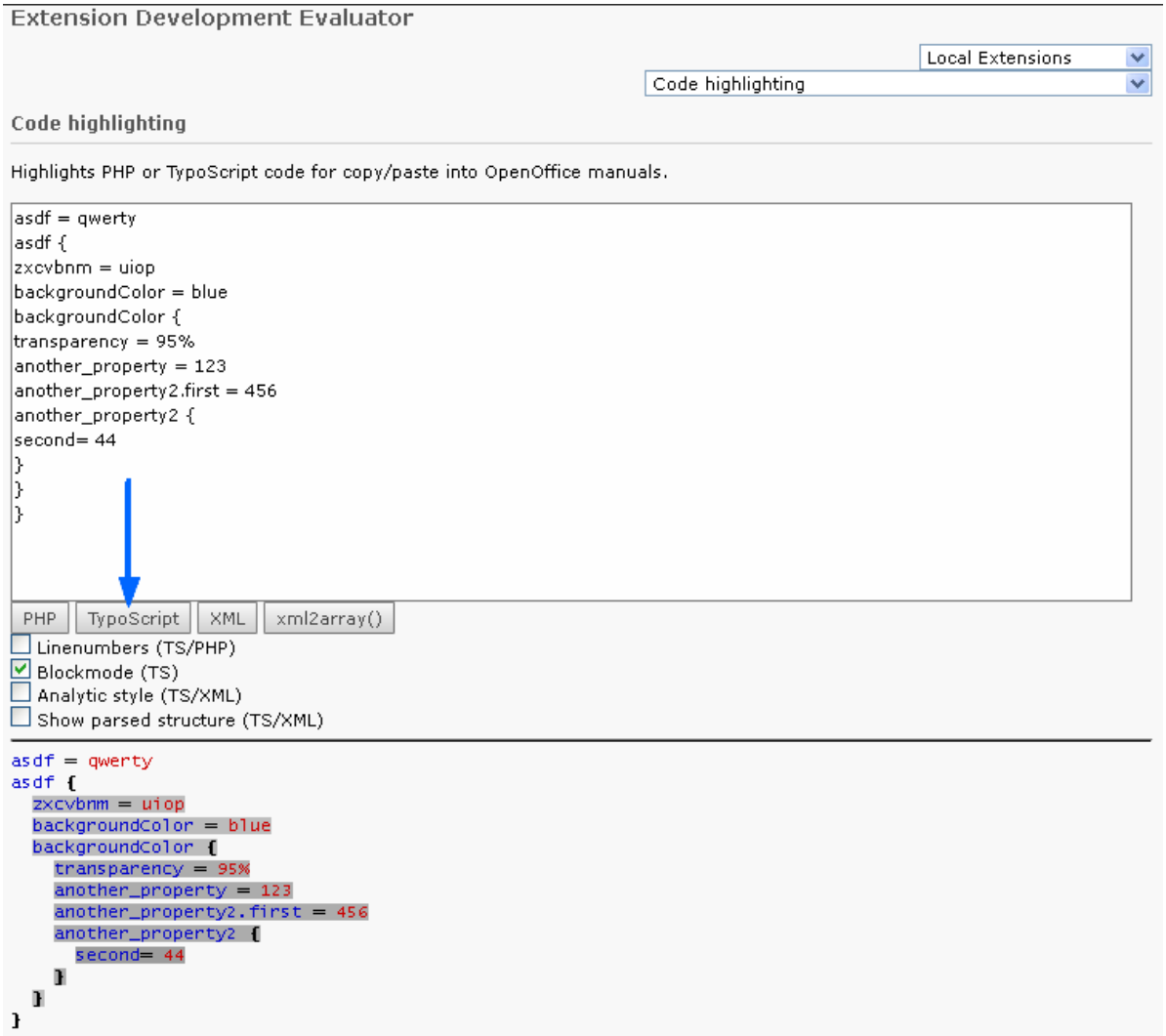
There are various modes of display:



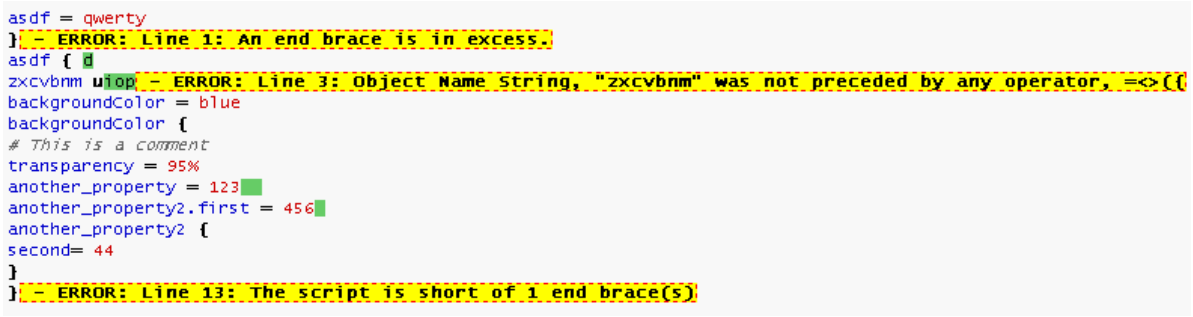
The analytic mode (displayed above) colors all parts of the syntax:

- **Light blue** background: Object and property names
- **Yellow** background: Values
- **Olive green** background: Whitespace
- **Brown** background: Operators

Another mode is the "Block mode", which simply indents the TypoScript code:



Finally you will be warned if syntax errors are found and ignored data will also be highlighted in green:



Debugging

Debugging TypoScript for syntax errors can be done with this tool and any other place where the syntax highlighter is used. But this will only tell you if something is *syntactically* wrong with the code - whether you combine objects and properties *semantically* correctly depends on the context and cannot be told by the TypoScript parser itself.

The TYPO3 system extension "t3editor" offers advanced functions, which can also be useful for debugging. Its auto-completion functionality for example only offers properties to be chosen, which in fact are semantically allowed at a certain place. Currently t3editor is available when editing the Setup field of a TypoScript template – not for Page TSconfig or User TSconfig (writing summer 2011).

Myths, FAQ and Acknowledgments

This section contains a few remarks and answers to questions you may still have. So here it goes:

Myth: "TypoScript is a scripting language"

This is misleading to say since you will think that TypoScript is like PHP or JavaScript while it is not. From the previous pages you have learned that TypoScript strictly speaking is just a syntax. However when the TypoScript syntax is applied to create TypoScript Templates then it begins to look like programming and the parallel to XSLT might also hint at that.

In any case TypoScript is NOT comparable to a scripting language like PHP or JavaScript. In fact, if TYPO3 offers any scripting language it is PHP itself! TypoScript is only an API which is often used to configure underlying PHP code.

Finally the name "TypoScript" is misleading as well. We are sorry about that; too late to change that now.

Myth: "TypoScript has the same syntax as JavaScript"

TypoScript was designed to be simple to use and understand. Therefore the syntax looks like JavaScript objects to some degree. But again; it is very dangerous to say this since it all stops with the syntax - TypoScript is still not a procedural programming language!

Myth: "TypoScript is a proprietary standard"

Since TypoScript is not a scripting language it does not make sense to claim this in comparison to PHP, JavaScript, Java or whatever *scripting language*.

However compared to XML or PHP arrays (which also contain *information*) you can say that TypoScript is a proprietary syntax since a PHP array or XML file could be used to contain the same information as TypoScript does. But this is *not* a drawback: For storage and exchange of *content* TYPO3 uses SQL (or XML if you need to), for storage of *configuration values* XML is not suitable anyways - TypoScript is much better at that job (see below).

To claim that TypoScript is a proprietary standard as an argument against TYPO3 is really unfair since the claim makes it sound like TypoScript is a whole new programming language or likewise. Yes, the TypoScript *syntax* is proprietary but extremely useful and when you get the hang of it, it is very easy to use. In all other cases TYPO3 uses official standards like PHP, SQL, XML, XHTML etc. for all *external* data storage and manipulation.

The most complex use of TypoScript is probably with the TypoScript Template Records. It is understandable that TypoScript has earned a reputation of being complex when you consider how much of the Frontend Engine you can configure through TypoScript Template records. But basically TypoScript is just an API to the PHP functions underneath. And if you think there are a lot of options there it would be no better if you were to use the PHP functions directly! Then there would be maybe even more API documentation to explain the API and you wouldn't have the streamlined abstraction provided by TypoScript Templates. This just served to say: The amount of features and the time it would take to learn about them would not be eliminated, if TypoScript was not invented!

Myth: "TypoScript is very complex"

TypoScript is simple in nature. But certainly it can quickly become complex and get "out of hand" when the amount of code lines grows! This can partly be solved by:

- Disciplined coding: Organize your TypoScript in a way that you can visually comprehend.
- Use the Syntax Highlighter to analyze and clean up your code - this gives you overview as well.

Why not XML instead?

A few times TypoScript has been compared with XML since both "languages" are frameworks for storing information. Apart from XML being a W3C standard (and TypoScript still not... :-)) the main

difference is that XML is great for large amounts of information with a high degree of "precision" while TypoScript is great for small amounts of "ad hoc" information - like configuration values normally are.

Actually a data structure defined in TypoScript could also have been modeled in XML. Currently you *cannot* use XML as an alternative to TypoScript (writing of October 2011), but this may happen at some point. Lets present this fictitious example of how a TypoScript structure could also have been implemented in "TSML" (our fictitious name for the non-existing TypoScript Mark-Up Language):

```
styles.content.bulletlist = TEXT
styles.content.bulletlist {
  current = 1
  trim = 1
  if.isTrue.current = 1
        # Copying the object "styles.content.parseFunc" to this position
  parseFunc < styles.content.parseFunc
  split {
    token.char = 10
    cObjNum = 1
    1.current < .cObjNum
    1.wrap = <li>
  }
    # Setting wrapping value:
  fontTag = <ol type="1"> | </ol>
  textStyle.altWrap = {$styles.content.bulletlist.altWrap}
}
```

That was 17 lines of TypoScript code and converting this information into an XML structure could look like this:

```
<TSML syntax="3">
  <styles>
    <content>
      <bulletlist>
        TEXT
        <current>1</current>
        <trim>1</trim>
        <if>
          <isTrue>
            <current>1</current>
          </isTrue>
        </if>
          <!-- Copying the object "styles.content.parseFunc" to this position -->
        <parseFunc copy="styles.content.parseFunc"/>
        <split>
          <token>
            <char>10</char>
          </token>
          <cObjNum>1</cObjNum>
          <num:1>
            <current>1</current>
            <wrap>&lt;li&gt;</wrap>
          </num:1>
        </split>
          <!-- Setting wrapping value: -->
        <fontTag>&lt;ol type=&quot;1&quot;&gt; | &lt;/ol&gt;</fontTag>
        <textStyle>
          <altWrap>{$styles.content.bulletlist.altWrap}</altWrap>
        </textStyle>
      </bulletlist>
    </content>
  </styles>
</TSML>
```

That was 33 lines of XML - the double amount of lines! And in bytes probably also much bigger. This example clearly demonstrates *why not XML*! XML will just get in the way, it is not handy for what TypoScript normally does. But hopefully you can at least use this example in your understanding of what TypoScript is compared to XML.

The reasonable application for using XML as an alternative solution to TypoScript is if an XML editor existed which in some way made the entering of XML data into a structure like this possible and easy.

The TypoScript parser API

Introduction

If you want to deploy TypoScript in your own TYPO3 applications it is really easy. The TypoScript parser is readily available to you and the only thing that may take a little more effort than the instantiation of PHP is if you want to define conditions for TypoScript.

Basically this chapter will teach you how you can parse your own TypoScript strings into a PHP array structure. The exercise might even help you to further understand the straight forward nature of TypoScript.

Notice that the following pages are for experienced TYPO3 developers and require a good knowledge of PHP.

Parsing custom TypoScript

Lets imagine that you have created an application in TYPO3, for example a plug-in. You have defined certain parameters editable directly in the form fields of the plug-in content element. However you want advanced users to be able to set up more detailed parameters. But instead of adding a host of such detailed options to the interface - which would just clutter it all up - you rather want advanced users to have a text area field into which they can enter configuration codes based on a little reference you make for them.

The reference could look like this:

Root level

Property	Data type	Description	Default
colors	->COLORS	Defining colors for various elements.	
adminInfo	->ADMINFO	Define administrator contact information for cc-emails	
headerImage	file-reference	A reference to an image file relative to the websites path (PATH_site)	

[TLO]

->COLORS

Property	Data type	Description	Default
backgroundColor	HTML-color	The background color of ...	white
fontColor	HTML-color	The font color of text in ...	black
popUpColor	HTML-color	The shadow color of the pop up ...	#333333

[colors]

->ADMINFO

Property	Data type	Description	Default
cc_email	string	The email address that ...	
cc_name	string	The name of ...	
cc_return_adr	string	The return address of ...	[servers]
html_emails	boolean	If set, emails are sent in HTML.	false

[adminInfo]

So these are the "objects" and "properties" you have chosen to offer to your users of the plug-in. This reference defines *what information makes sense* to put into the TypoScript field (semantically), because you will program your application to use this information as needed.

A case story

Now let's imagine that a user inputs this TypoScript configuration in whatever medium you have offered (e.g. a textarea field). (In a syntax highlighted version with line numbers it would look like the listing, which indicates that there are no *syntax errors* and everything is fine in that regard.)

```
 0: colors {
 1:   backgroundColor = red
 2:   fontColor = blue
 3: }
 4: adminInfo {
 5:   cc_email = email@email.com
 6:   cc_name = Copy Name
 7: }
 8: showAll = true
 9:
10: [UserIpRange = 123.456.*.*]
11:
12:   headerImage = fileadmin/img1.jpg
13:
14: [ELSE]
15:
16:   headerImage = fileadmin/img2.jpg
17:
18: [GLOBAL]
19:
20:   // Wonder if this works... :-)
21: wakeMeUp = 7:00
```

(Syntax highlighting of TS (and XML and PHP) can be done with the extension "extdeveval").

In order to parse this TypoScript we can use the following code provided that the variable $tsString contains the above TypoScript as its value:

```
 3: require_once(PATH_t3lib.'class.t3lib_tsparser.php');
 4:
 5: $TSparserObject = t3lib_div::makeInstance('t3lib_tsparser');
 6: $TSparserObject->parse($tsString);
 7:
 8: echo '<pre>';
 9: print_r($TSparserObject->setup);
10: echo '</pre>';
```

- Line 3: The TypoScript parser class is included (most likely already done in both frontend and backend of TYPO3).
- Line 5: Creates an object of the parser class.
- Line 6: Initiates parsing of the TypoScript content of the string $tsString.
- Line 8-10: Outputs the parsed result which is located in $TSparserObject->setup.

The result of this code being run will be this:

```
Array
```

```
(
  [colors.] => Array
  (
    [backgroundColor] => red
    [fontColor] => blue
  )

  [adminInfo.] => Array
  (
    [cc_email] => email@email.com
    [cc_name] => Copy Name
  )

  [showAll] => true
  [headerImage] => fileadmin/img2.jpg
  [wakeMeUp] => 7:00
)
```

Now your application could use this information in a manner like this:

```
echo '<table bgcolor="'.$TSparserObject->setup['colors.']['backgroundColor'].'">
  <tr>
    <td>
      <font color="'.$TSparserObject->setup['colors.']['fontColor'].'">HELLO WORLD!</font>
    </td>
  </tr>
</table>';
```

As you can see some of the TypoScript properties (or *object paths*) which are found in the reference tables above are implemented here. There is not much mystique about this and in fact this is how all TypoScript is used in its respective contexts; **TypoScript contains simply configuration values that make our underlying PHP code act accordingly - parameters, function arguments, as you please; TypoScript is an API to instruct an underlying system.**

This also means that now we can begin to meaningfully talk about invalid information in TypoScript - it is obvious that two properties are entered in TypoScript but do not make any sense: "showAll" and "wakeMeUp". Both properties are not defined in the reference tables and therefore they should neither be implemented in the PHP code of course. However no errors are issued by the parser since the syntax used to define those properties is still right. The only problem is that they are irrelevant; it is like defining a variable in PHP and then never using it! A waste of time - and probably confusing later.

As noted there exists only the input mode of t3editor to do "semantics-checking". However, this only works during input, not at a later time. It might be interesting and very helpful some day if we had that as well so we could also be warned if we use non-existing properties (which could just be spelling errors).

Implementing custom Conditions

Now we know how to parse TypoScript and the only thing we still want to do is to implement support for custom conditions. As stated a few places *the evaluation* of a condition is external to TypoScript and all you need to do in order to have an external process deal with conditions is to pass an object as the second parameter to the parse-function. This is done in the code listing below:

```
 1: require_once(PATH_t3lib.'class.t3lib_tsparser.php');
 2:
 3: class myConditions {
 4:   function match($conditionLine) {
 5:     if ($conditionLine === '[TYPO3 IS GREAT]') {
 6:       return TRUE;
 7:     }
 8:   }
 9: }
10: $matchObj = t3lib_div::makeInstance('myConditions');
11:
```

```
12: $TSparserObject = t3lib_div::makeInstance('t3lib_tsparser');
13: $TSparserObject->parse($tsString, $matchObj);
14:
15: debug($TSparserObject->setup);
```

Here go some notes to this listing:

- Lines 3-10 define a very simple class with a function, match(), inside.
  The function "match()" must exist and take a string as its argument and the match function must also return a boolean value. This function should be programmed to evaluate the condition line according to your specifications.
  Currently, if a condition line contains the value "[TYPO3 IS GREAT]" then the condition will evaluate to true and the subsequent TypoScript will be parsed.
- Line 13: Here the instantiated object, $matchObj, of the "myConditions" class is passed to the parser.
- Line 15: Just a little side note: Instead of using PHPs "print_r()" function we use the classic debug() function in TYPO3 which prints an array in an HTML table - some of us think this is the nicest way to look into the content of an array (make your own opinion from the screenshot below).

Anyways, let's test the custom condition class from the code listing above. This is done by parsing this TypoScript code:

```
 0: someOtherTS = 123
 1:
 2: [TYPO3 IS GREAT]
 3:
 4: message = Yes
 5: someOtherTS = 987
 6:
 7: [ELSE]
 8:
 9: message = No
10:
11: [GLOBAL]
12:
13: someTotallyOtherTS = 456
```

With this listing we would expect to get the object path "message" set to "Yes" since the condition line "[TYPO3 IS GREAT]" matches the criteria for what will return true. Lets try:

someOtherTS	987
message	Yes
someTotallyOtherTS	456

According to this output it worked!

Lets try to alter line 2 to this:

```
1:
2: [TYPO3 IS great]
3:
```

The parsed result is now:

someOtherTS	123
message	No
someTotallyOtherTS	456

As you can see the value of "message" is now "No" since the condition returned FALSE. The string "[TYPO3 IS great]" is obviously *not* the same as "[TYPO3 IS GREAT]"! The value of "someOtherTS" was also changed to "123" which was the value set before the condition and since the condition was not TRUE the overriding of that former value did not happen like in the first case.

A realistic example

Most likely you don't want to evaluate conditions based on their bare string value. More likely you want to set up rules for a syntax and then parse the condition string. One example could be this modified condition class which will implement support for the condition seen in the TypoScript listings in the former section, "[UserIpRange = 123.456.*.*]":

```
 1: class myConditions {
 2:   function match($conditionLine) {
 3:       // Getting the value inside of the square brackets:
 4:     $insideSqrBrackets = trim(ereg_replace('\]$', '', substr($conditionLine, 1)));
 5:
 6:       // Splitting value into a key and value based on the "=" sign
 7:     list($key, $value) = explode('=', $insideSqrBrackets, 2);
 8:
 9:     switch(trim($key)) {
10:       case 'UserIpRange':
11:         return t3lib_div::cmpIP(t3lib_div::getIndpEnv('REMOTE_ADDR'), trim($value)) ?
TRUE : FALSE;
12:       break;
13:       case 'Browser':
14:         return $GLOBALS['CLIENT']['BROWSER'] == trim($value);
15:       break;
16:     }
17:   }
18: }
```

This class works in this way:

- Line 4: The square brackets in the start (and possibly end as well) of the condition line is removed.
- Line 7: The condition line without square brackets is exploded into a key and a value separated by the "=" sign; we are trying to implement the concept of evaluating a data source to a value.
- Line 9-16: This switch construct will allow the "key" to be either "UserIpRange" or "Browser" (the datasource pointer) and the value after the equal sign is of course interpreted accordingly.

Lets try and parse the TypoScript listing from the former section:

```
 0: colors {
 1:   backgroundColor = red
 2:   fontColor = blue
 3: }
 4: adminInfo {
 5:   cc_email = email@email.com
 6:   cc_name = Copy Name
 7: }
 8: showAll = true
 9:
10: [UserIpRange = 123.456.*.*]
11:
```

```
12:   headerImage = fileadmin/img1.jpg
13:
14: [ELSE]
15:
16:   headerImage = fileadmin/img2.jpg
17:
18: [GLOBAL]
19:
20:   // Wonder if this works... :-)
21: wakeMeUp = 7:00
```

The result of parsing this will be an array like this:

colors.	backgroundColor	red
	fontColor	blue
adminInfo.	cc_email	email@email.com
	cc_name	Copy Name
showAll	true	
headerImage	fileadmin/img2.jpg	
wakeMeUp	7:00	

As you can see the "headerImage" property value stems from the [ELSE] condition section and thus the "[UserIpRange = 123.456.*.*]" must still have evaluated to FALSE - which is actually no wonder since nobody can have the IP address range "123.456.*.*"!

Lets change line 10 of the TypoScript to this:

```
 9:
10: [UserIpRange = 192.168.*.*]
11:
```

Since I'm currently on an internal network with an IP number which falls into this space, the condition should now evaluate to TRUE when the TypoScript is parsed:

colors.	backgroundColor	red
	fontColor	blue
adminInfo.	cc_email	email@email.com
	cc_name	Copy Name
showAll	true	
headerImage	fileadmin/img1.jpg	
wakeMeUp	7:00	

... and in fact it does!

Implementing combined Conditions

Conditions can be combined using OR and AND. This feature already is implemented for TypoScript Templates. Here the explanation, how that was done. For an overview of the resulting possibilities see the chapter "Conditions" in TSref. It contains short information about the syntax and an overview of the available conditions.

In the context of TypoScript Templates you can place several "conditions" in the same (real) condition:

```
[browser = msie][browser = opera]
someTypoScript = 123
[GLOBAL]
```

They are evaluated by OR-ing the result of each sub-condition (done in the class t3lib_matchCondition). We could implement something alike and maybe even better. For instance we could implement a syntax like this:

```
[ CON 1 ] && [ CON 2 ] || [ CON 3 ]
```

This will be read like "Returns TRUE if condition 1 and condition 2 are TRUE OR if condition 3 is TRUE". In other words we implement the ability to AND and OR conditions together.

The implementation goes as follows:

```
 1: class myConditions {
 2:
 3:   /**
 4:    * Splits the input condition line into AND and OR parts
 5:    * which are separately evaluated and logically combined to the final output.
 6:    */
 7:   function match($conditionLine) {
 8:       // Getting the value from inside of the wrapping
 9:       // square brackets of the condition line:
10:     $insideSqrBrackets = trim(ereg_replace('\]$', '', substr($conditionLine, 1)));
11:
12:       // The "weak" operator, OR, takes precedence:
13:     $ORparts = split('\][[:space:]]*\|\|[[:space:]]*\[', $insideSqrBrackets);
14:     foreach($ORparts as $andString) {
15:       $resBool = FALSE;
16:
17:         // Splits by the "&&" and operator:
18:       $ANDparts = split('\][[:space:]]*\&\&[[:space:]]*\[', $andString);
19:       foreach($ANDparts as $condStr) {
20:         $resBool = $this->evalConditionStr($condStr) ? TRUE : FALSE;
21:         if (!$resBool) break;
22:       }
23:
24:       if ($resBool) break;
25:     }
26:     return $resBool;
27:   }
28:
29:   /**
30:    * Evaluates the inner part of the conditions.
31:    */
32:   function evalConditionStr($condStr) {
33:       // Splitting value into a key and value based on the "=" sign
34:     list($key, $value) = explode('=', $condStr, 2);
35:
36:     switch(trim($key))    {
37:       case 'UserIpRange':
38:         return t3lib_div::cmpIP(t3lib_div::getIndpEnv('REMOTE_ADDR'), trim($value)) ?
TRUE : FALSE;
39:       break;
40:       case 'Browser':
41:         return $GLOBALS['CLIENT']['BROWSER']==trim($value);
42:       break;
43:     }
44:   }
45: }
```

With this implementation I can make a condition line like this:

```
 9:
10: [UserIpRange = 192.168.*.*] && [Browser = msie]
11:
12:   headerImage = fileadmin/img1.jpg
13:
```

So if I'm in the right IP range AND have the right browser the value of "headerImage" will be "fileadmin/img1.jpg"

If we modify the TypoScript as follows, the same condition applies but if the browser is Firefox then the condition will evaluate to TRUE regardless of the IP range:

```
 9:
10: [UserIpRange = 192.168.*.*] && [Browser = msie] || [Browser = firefox]
11:
12:   headerImage = fileadmin/img1.jpg
```

This is because the conditions are read like the parenthesis levels show:

("UserIpRange = 192.168.*.*" **AND** "Browser = msie") **OR** "Browser = firefox"

The order of the "||" and "&&" operators may be a problem now. For instance:

```
 9:
10: [UserIpRange = 192.168.*.*] || [UserIpRange = 212.237.*.*] && [Browser = msie]
11:
12:   headerImage = fileadmin/img1.jpg
```

I would like it to read as "If User IP Range is either #1 or #2 provided that the browser is MSIE in any case!". But right now it will be TRUE if the User IP range is 192.168.... OR if either the range is 212.... and the browser is MSIE.

Formally, this is what I want:

("UserIpRange = 192.168.*.*" **OR** "UserIpRange = 212.237.*.*") **AND** "Browser = msie"

My solution is to implement a second way of OR'ing conditions together - by simply implying an OR between two "condition sections" if no operator is there. Thus the line above could be implemented as follows:

```
 9:
10: [UserIpRange = 192.168.*.*][UserIpRange = 212.237.*.*] && [Browser = msie]
11:
12:   headerImage = fileadmin/img1.jpg
```

Line 10 will be understood in this way:

```
[UserIpRange = 192.168.*.*](implied OR here!)[UserIpRange = 212.237.*.*] && [Browser = msie]
```

The function match() of the condition class will have to be modified as follows:

```
1:   /**
2:    * Splits the input condition line into AND and OR parts
3:    * which are separately evaluated and logically combined to the final output.
4:    */
5:   function match($conditionLine) {
6:       // Getting the value from inside of the wrapping
```

```
 7:       // square brackets of the condition line:
 8:     $insideSqrBrackets = trim(ereg_replace('\]$', '', substr($conditionLine, 1)));
 9:
10:       // The "weak" operator, OR, takes precedence:
11:     $ORparts = split('\][[:space:]]*\|\|[[:space:]]*\[', $insideSqrBrackets);
12:     foreach($ORparts as $andString) {
13:       $resBool = FALSE;
14:
15:         // Splits by the "&&" and operator:
16:       $ANDparts = split('\][[:space:]]*\&\&[[:space:]]*\[', $andString);
17:       foreach($ANDparts as $subOrStr) {
18:
19:           // Split by no operator between ] and [ (sub-OR)
20:         $subORparts = split('\][[:space:]]*\[', $subOrStr);
21:         $resBool = FALSE;
22:         foreach($subORparts as $condStr) {
23:           if ($this->evalConditionStr($condStr)) {
24:             $resBool = TRUE;
25:             break;
26:           }
27:         }
28:
29:         if (!$resBool) break;
30:       }
31:
32:       if ($resBool) break;
33:     }
34:     return $resBool;
35:   }
```

That's it.

Addendum to the reference for our application

Remember in the previous sections? We defined three tables with properties that could be used in TypoScript in the context of our case-story application. To that reference we should now add a section with conditions which defines the following:

#1: Line syntax:

A condition is split into smaller parts which are connected using a logical AND or a logical OR. Each sub-part of the condition line is separated by "] (Operator) [" where operator can be "&&" (AND) , "||" (OR) or nothing at all (also meaning OR "below" AND in order).

The format of the condition line therefore is:

[COND1] || [COND2] && [COND3] [COND4] ....etc

where the operators have precedence as indicated by these illustrative parenthesis:

[COND1] || ([COND2] && ([COND3] [COND4]))

(Notice: Between COND3 and COND4 the blank space is implicitly an OR.)

#2: Subpart syntax:

For each subpart (for example "[COND 1]") the content is evaluated as follows:

[KEY = VALUE]

where the key denotes a type of condition from the table below:

Key	Description	Example
UserIpRange	Returns TRUE if the client's remote IP address matches the pattern given as value. The value is matched against REMOTE_ADDR by the function t3lib_div::cmpIP(), which you can consult for details on the syntax.	[UserIpRange = 192.168.*.*]
Browser	Returns TRUE, if the client's browser matches one of the keywords below. Values you can use: **konqu** = Konqueror **opera** = Opera **msie** = Microsoft Internet Explorer **net** = Netscape (or any other) Values are evaluated against the output of the function t3lib_div::clientInfo() which can be consulted for details on the values for browsers. **Note**: These values are **examples**, which fit to the code we have built above. In current TYPO3 versions the available values have changed! **For an overview over the values currently possible, always consult TSref!**	[Browser = msie]

Appendix A – What is TypoScript?

People are often confused about what TypoScript (TS) is, where it can be used and have a tendency to think of it as something complex. This chapter has been written in the hope of clarifying these issues.

First let's start with a basic truth:

- TypoScript is a *syntax* for defining information in a hierarchical structure using simple ASCII text content.

This means that:

- TypoScript itself does not "do" anything - it just contains information.
- TypoScript is *only* transformed into function when it is passed to a program which is designed to act according to the information in a TypoScript information structure.

So strictly speaking TypoScript has no function in itself, only when used in a certain context. Since the context is almost always to *configure* something you can often understand TypoScript as *parameters* (or function arguments) passed to a function which acts accordingly (e.g. "background_color = red"). And on the contrary you will probably never see TypoScript used to store information like a database of addresses - you would use XML or SQL for that.

PHP arrays

In the scope of its use you can also understand TypoScript as a non-strict way to enter information into a *multidimensional array*. In fact when TypoScript is parsed, it is *transformed into a PHP array*! So when would you define static information in PHP arrays? You would do that in configuration files - but probably not to build your address database!

This can be summarized as follows:

- When TypoScript is *parsed* it means that the information is transformed into a *PHP array* from where TYPO3 applications can access it.
- So the *same* information could in fact be defined in TypoScript *or directly* in PHP; but the syntax would be different for the two of course.
- TypoScript offers convenient features which is the reason why we don't just define the information directly with PHP syntax into arrays. These features include a relaxed handling of syntax errors, definition of values with less language symbols needed and the ability of using an object/property metaphor, etc.

TypoScript syntax, object paths, objects and properties

See, that is what this document is about - the *syntax* of TypoScript; the rules you must obey in order to store information in this structure. Obviously I'll not explain the full syntax here again but just give an example to convey the idea.

Remember it is about storing information, so think about TypoScript as *assigning values to variables*: The "variables" are called "object paths" because TypoScript easily lends itself to the metaphor of "objects" and "properties". This has some advantages as we shall see but at the same time TypoScript is designed to allow a very simple and straight forward assignment of values; simply by using the equal sign as an operator:

```
asdf = qwerty
```

Now the object path "asdf" contains the value "qwerty".

Another example:

```
asdf.zxcvbnm = uiop
asdf.backgroundColor = blue
```

Now the object path "asdf.zxcvbnm" contains the value "uiop" and "asdf.backgroundColor" contains the value "blue". According to *the syntax* of TypoScript this could also have been written more comfortably as:

```
asdf {
  zxcvbnm = uiop
  backgroundColor = blue
}
```

What happened here is that we broke down the full *object path*, "asdf.zxcvbnm" into its components "asdf" and "zxcvbnm" which are separated by a period, ".", and then we used the curly brace operators, { and } , to bind them together again. To describe this relationship of the components of an *object path* we normally call "asdf " *the object* and " zxcvbnm " *the property* of that object.

So although the terms *objects* and *properties* normally hint at some context (semantics) we may also use them purely to describe the various parts of an object path without considering the context and meaning. Consider this:

```
asdf {
  zxcvbnm = uiop
  backgroundColor = blue
  backgroundColor.transparency = 95%
}
```

Here we can say that "zxcvbnm" and "backgroundColor" are *properties* of (the object) "asdf". Further, "transparency" is a property of (the object / the property) "backgroundColor" (or "asdf.backgroundColor").

Note about perceived semantics

You may now think that "backgroundColor = blue" makes more sense than "zxcvbnm = uiop" but having a look at the **syntax** only it doesn't! The only reason that "backgroundColor = blue" seems to make sense is that in the *English language* we understand the words "background color" and "blue" and automatically imply some meaning. We understand the **semantics** of it. But to a machine like a computer the word "backgroundColor" makes just as little sense as "zxcvbnm" unless it has been programmed to understand either one, e.g. to take its value as the background color for something. In fact "uiop" could be an alias for blue color values and "zxcvbnm" could be programmed as the property setting the background color of something.

This just serves to point one thing out: Although most programming languages and also TypoScript use function, method, keyword and property names which humans can often deduct some meaning from, it ultimately is the programming reference, DTD or XML-Schema which defines the meaning.

Note about the internal structure when parsed into a PHP array

As stated in the previous chapter TypoScript can be understood as a lightweight way to enter information into a multidimensional PHP array. Let's take the TypoScript from above as an example:

```
asdf {
  zxcvbnm = uiop
  backgroundColor = blue
  backgroundColor.transparency = 95%
}
```

When parsed, this information will be stored in a PHP array which could be defined as follows:

```
$TS['asdf.']['zxcvbnm'] = 'uiop';
$TS['asdf.']['backgroundColor'] = 'blue';
$TS['asdf.']['backgroundColor.']['transparency'] = '95%';
```

Or alternatively you could define the information in that PHP array like this:

```
$TS = array(
  'asdf.' => array(
    'zxcvbnm' => 'uiop',
    'backgroundColor' => 'blue',
    'backgroundColor.' => array (
      'transparency' => '95%'
    )
  )
)
```

The information inside a PHP array like that one is used by TYPO3 to apply the configurations, which you have set.

Next steps

If you are looking for an overview of the available objects in TypoScript templates, have a look at TSref, the TypoScript Reference.

All properties and values for TSconfig fields are listed in the document "TSconfig".