TYPO3 Documentation

Release 1.0.0

Elmar Hinz

June 07, 2016

CHAPTER 1

Introduction

This extensions ships a TypoScript parser, that is suited to replace the original TypoScript parser for frontend rendering. In fact a family of parsers has been introduced, specialized on different tasks.

- FE: TypoScriptConditionsProcessor
- FE: TypoScriptProductionParser
- BE: TypoScriptSyntaxParser

1.1 What it is not

1.1.1 No Boost in Performance

The parsing of TypoScript just takes a few milliseconds. Hence, it's not the primary goal to speed up the performance but to improve the architecture. The algorithm is twice as fast as the original algorithm, but with the split into conditions proprocessor and processor the time is about the same again.

1.2 What it is

1.2.1 Standalone Usage

It's possible to use the TypoScript parser standalone outside of the TYPO3 CMS if you like the TypoScript syntax and want to use it for configuration in other fields. This is possible with or without the conditions preprocessor.

1.2.2 Improving the error detection

The error detection covers the error detection of the origional parser and tries be be a little better already. Also the displaying of the line numbers has been worked upon. See Screenshots!

Having done this prove of concept that the replacement of the original syntax highlighter can be done further debugging features are planned:

- Do syntax highlighting of conditions, instead of printing them in one color.
- Detect the difference of objects and properties, because only objects are allowed ot be copied by reference.
- (Related) Throw verbose errors from TS objects, catch them and and display them into the backend.

1.2.3 New Architecture

The reason to write a new TypoScript parser is, to get a modern architecture for it:

- easy to understand
- easy to debug
- easy to extend

A modern parser makes it more easy to get rid of flaws in TypoScript and to add new features like if-else conditions, that work the way you are used to from other languages or enhance error debugging.

1.2.4 Condition Preprocessor

Condition evaluation is done by a preprocessor. By separtion of the condition preprocessing it becomes possible to use the TypoScript parser without bothering with conditions and focus on it's task.

On the other hand by isolating the conditions it becomes possible to enhance the condition preprocessing easily. For example it becomes easy to introduce an [ELSEIF] element.

As with the old parser the condition matching is handled by a third object. Exchanging this object enables the development of conditions, that address a completly different field than the TYPO3 CMS.

1.2.5 Public Presentation

This is a public presentation of the parser. Should it replace the old parser of the core? If yes, it needs to be tested in the wild before until it is really stable. This is the extension to do so.

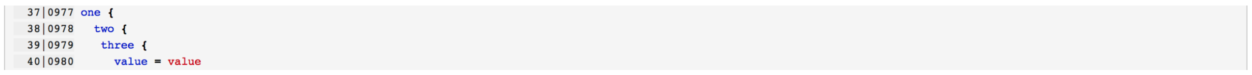
1.3 Differences

- Backslash doesn't escape anything.
- Escaping of dots in object keys is not supported.
- Backslash is an allowed character in the keys (for PHP namespaces).

CHAPTER 2

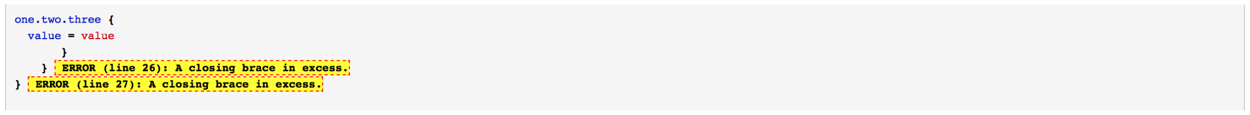
Screenshots

2.1 Line numbering



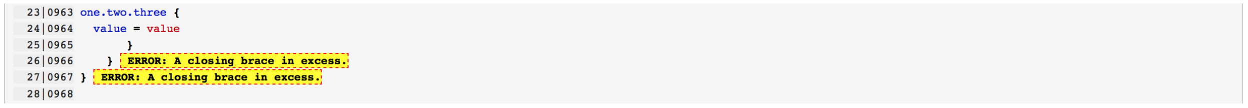
```
37|0977 one {
38|0978   two {
39|0979    three {
40|0980      value = value
```

Fig. 2.1: The line numbers show the numbering of the template and the overall numbering within the template tree.



```
one.two.three {
  value = value
       }
    }  ERROR (line 26): A closing brace in excess.
}  ERROR (line 27): A closing brace in excess.
```

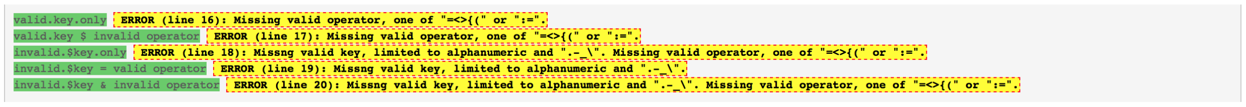
Fig. 2.2: When line numbering is turned off the error messages contain the line number instead.



```
23|0963 one.two.three {
24|0964   value = value
25|0965        }
26|0966     }  ERROR: A closing brace in excess.
27|0967 }  ERROR: A closing brace in excess.
28|0968
```

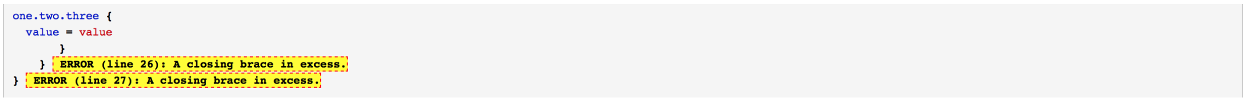
Fig. 2.3: When line numbering is turned on the error messages don't duplicate the information.

2.2 Types of errors



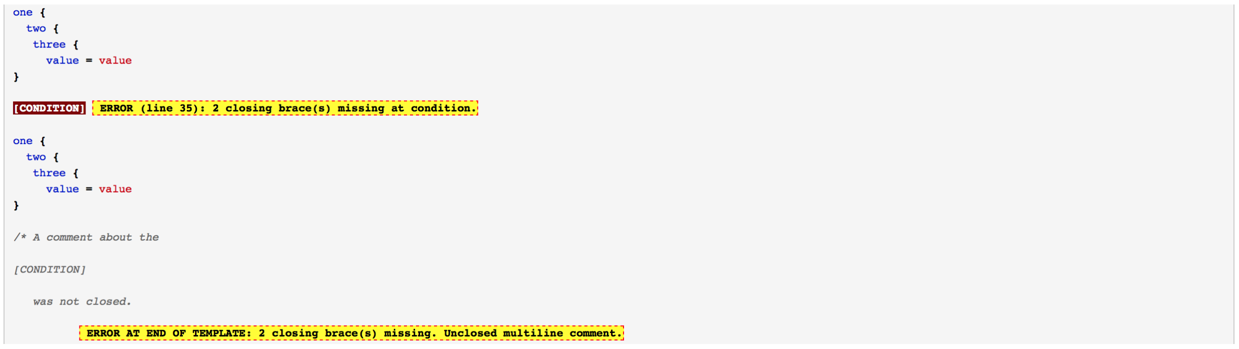
```
valid.key.only  ERROR (line 16): Missing valid operator, one of "=<>{(" or ":=".
valid.key $ invalid operator  ERROR (line 17): Missing valid operator, one of "=<>{(" or ":=".
invalid.$key.only  ERROR (line 18): Missng valid key, limited to alphanumeric and ".-_\". Missing valid operator, one of "=<>{(" or ":=".
invalid.$key = valid operator  ERROR (line 19): Missng valid key, limited to alphanumeric and ".-_\".
invalid.$key & invalid operator  ERROR (line 20): Missng valid key, limited to alphanumeric and ".-_\". Missing valid operator, one of "=<>{(" or ":=".
```

Fig. 2.4: For invalid lines it is assumed that the user want's to enter an operator line. It is checked for invalid key and operator.



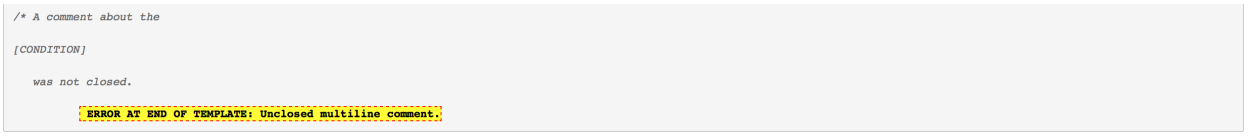
```
one.two.three {
  value = value
       }
    }  ERROR (line 26): A closing brace in excess.
}  ERROR (line 27): A closing brace in excess.
```

Fig. 2.5: Braces in access are shown in the line where they occur.



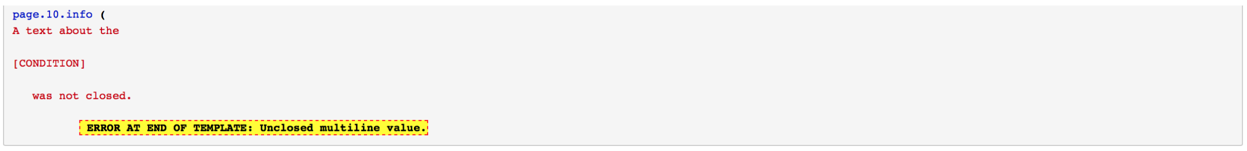
```
one {
  two {
   three {
     value = value
}

[CONDITION]  ERROR (line 35): 2 closing brace(s) missing at condition.

one {
  two {
   three {
     value = value
}

/* A comment about the

[CONDITION]

   was not closed.

           ERROR AT END OF TEMPLATE: 2 closing brace(s) missing. Unclosed multiline comment.
```

Fig. 2.6: Missing closing braces are detected at conditions and at the end of the template.



```
/* A comment about the

[CONDITION]

   was not closed.

           ERROR AT END OF TEMPLATE: Unclosed multiline comment.
```

Fig. 2.7: An unclosed multiline comment is detected at the end of the template. Multiline comments can be used to comment out parts of the script. Included elements like conditions don't result in an error.



```
page.10.info (
A text about the

[CONDITION]

   was not closed.

           ERROR AT END OF TEMPLATE: Unclosed multiline value.
```

Fig. 2.8: An unclosed multiline value is detected at the end of the template.

CHAPTER 3

Administration

Install the extension, clear caches and check of your frontend is rendered as expected and if you get the advanced error feedback in the backend.

If anything goes wrong, uninstall and report the issue.

https://github.com/elmar-hinz/TX.tsp/issues

3.1 Technical Implementation

The origional parser is not fully replaced but extended by XCLASS registration. The extended class serves as adapter to the standalone classes.

CHAPTER 4

Research

4.1 CoreTypoScriptParserTyposcriptParser

4.1.1 Overview

The method `parse()` is a preprocessor that handels including and excluding of template parts by condtions.

It doesn't parse the incoming lines to end first, but delegates the parts immediately to `parseSub()` (a kind of depth-first parsing of the template tree).

The method `doSyntaxHighlight()` is responsible to generate a syntax highlighted `HTML` string. It also calls the preprocessor `parse()` but sets a flag, that disables the coditions, so that all parts are evaluated.

The latter is strange in two aspects. It doesn't make sense to send syntax highlighting through a conditioning preprocessor. It doesn't make sense to parse into an array tree, when one actually want's a `HTML` string as result.

4.1.2 Conditions

Inn the method `parse()` the template is branched into rendered and non-rendered parts based on conditions. The condition evalutation is delegated to a `$matchObj` that is injected by parameter.

For each condition the method creates a hash and stores it into `$this->sections` array. This are used by the `TemplateService` to cache the rendered templates matching combinations of conditions that evaluate to true.

4.1.3 Line numbering

There is a line number offset that sums up the line numbers of previously rendered templates. It is advanced at end of `parse()`.

The line numbers of the current template are tracked by `$this->rawP` in the main loop of `parseSub()` and also for the condition sections that evaluate to false in the method `nextDivider()`. `$this->rawP` is reset to zero at the beginning of the rendering of the current template in the method `parse()`.

4.1.4 Error handling

method `error($errorString, $severity = 2)`.

This method collects into $this->errors[] = [a, b, c, d] with:

- a = error message

- b = severity
- c = line number
- d = template line number offset

Collected messages:

- 'Script is short of XXX braces.'
- 'An end brace is in excess.'
- 'On return to [GLOBAL] scope, the script was short of XXX braces.'
- 'A multiline value section is not ended with a parenthesis!'
- 'Object Name String, contains invalid character XXX. Must be alphanumeric or one of: "_:-."'
- 'Object Name String XXX was not followed by any operator, =<>({'
- '### ERROR: XXX' (Error to be extract form an error comment created in previous parsing steps like during template includes.)

4.1.5 Syntax highlighting

Highlighted parsing is controlled by the method `doSyntaxHighlight()`.

It sets the flag `$this->syntaxHighLight` to true and the template string is parsed. The flag activates the additional highlighting functionality during the process of parsing. Finally the method `syntaxHighlight_print()` is called to format the collected results including the error messages.

Registration of highlighted parts of lines is done during parsing by the method `regHighLight()` if the above flag is set. The parts are collected into

- `$this->highLightData`
- `$this->highLightData_bracelevel`

Both arrays count per line, the first one the higlighted sections of the line, the second one the depth of brace nesting.

4.1.6 Breakpoints

A breakpoint is a line number in `$this->breakPointLN` to break the execution of the rendering. The method `parseSub()` returns with a marker `[_BREAK]`. This marker stops the further execution of the main loop in `parse()`.

4.2 TemplateService

`TemplateService` is a service that makes use of the parser. A main task of TemplateService is, to cache the rendered template for different combinations of conditions of a page.

4.3 ExtendedTemplateService

The class `ExtendedTemplateService` contains method for the TS module in TYPO3 backend. It extends TemplateService.

Lessons Learned

The overall time to parse the TypoScript of a website takes just a few milliseconds. It is not a critical part of the overall page rendering time. Yet the development of this extension was also focused on performance.

5.1 Time to parse the templates vs. time to parse TypoScript

When measured with the TYPO3 core time tracker (admin panel) the template parsing takes a few hundred milliseconds. When measuring and summing up all calls to the TypoScript parse function (TypoScriptParser::parse()) it takes just a few milliseconds. The difference is most likley to be explained by I/O calls to read the templates.

5.2 Non-Recursive Parser

The `Non-Recursive Parser` is the approach taken by this parser. The whole rendering happens within one function by using simple loop structures. Calls to itself or other methods are avoided as far as reasonable. This turns out to be twice as fast as the recursive `Original TypoScript Parser`.

5.3 Original TypoScript Parser

The original parser of the TYPO3 core uses recursive calls to handle the nesting of the braces of the object name pathes.

5.4 JSON Parser

The idea of the `JSON Parser` was, to use the PHP function `json_decode` to create the large `TypoScript` tree consisting of hundreds of PHP arrays on the binary level. `TypoScript` was rewritten to a valid `JSON` string as input.

Unfortunately `json_decode` does merging but not recursive merging. As overwriting is a feature of `TypoScript` this requires to prepare the `JSON` rendering by any approach to do the overwriting in advance. An array was created, containing the full object path as key and the value as value to solve this. Although this creates no nested tree, it takes time.

Together with the conversion to a `JSON` string in the second step, there is no advantage in speed. Taking the non-recursive approach to handle the two steps, it ends up in a similar speed as the `Original TypoScript Parser`.

Known Issues

6.1 No Exceptions are Thrown

The TypoScript production parser currently doesn't throw execptions. It expects valid TS as input. The syntax higlighting parser is designed to inspect TS for mistakes.

The original parser doesn't throw exceptions either. Modules of the backend are not prepared to catch exeptions from the parser and break if execeptions would be thrown from insane TS.

6.2 Intolerant for Insane TS

The TypoScript production parser will silently break, if feed with insane TS. It is optimized for speed and is less tolerant for insane TS than the origional parser.

This means in rare cases code that works for the original parser may break with the TypoScript production parser. Use the syntax highlighting parser to fix the TS code.