

TypoScript in 45 Minuten

Extension Key: doc_tut_ts45_de

Language: de

Version: 2.0.1

Keywords: forAdmins, forDevelopers, forBeginners, forIntermediates

Copyright 2000-2010, Documentation Team, <documentation@typo3.org>

This document is published under the Open Content License
available from <http://www.opencontent.org/opl.shtml>

The content of this document is related to TYPO3
- a GNU/GPL CMS/Framework available from www.typo3.org

Official documentation translation

This document is a translation of a TYPO3 official manual. It may not be as up to date as the original and is not subject to a peer-review process. Official documentation is only provided in English.

Tutorial

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

Table of Contents

TypoScript in 45 Minuten.....	1	Objekte zur Ausgabe von Inhalten	19
Introduction.....	3	weitere Objekte	20
About this document.....	3	TypoScript Funktionen:	21
What's new.....	3	imgResource	21
Credits.....	3	imageLinkWrap	22
Feedback.....	3	numRows	23
TypoScript: ein kleiner Rundblick	4	select	24
Einleitung	4	split	24
Warum TypoScript	5	if	25
Der Begriff Template	5	typolink	26
auch TypoScript ist nur ein Array	6	encapsLines	28
Erste Schritte	7	filelink	29
Inhalte einlesen	10	parseFunc	30
Die unterschiedlichen Inhaltselemente	11	tags	30
css_styled_content	11	HTMLparser	31
styles.content.get	12	stdWrap richtig nutzen	32
Ein Menü bauen	14	Reihenfolge beachten!	32
Inhalte in ein Template einfügen	16	stdWrap rekursiv nutzen	32
css_styled_content nutzen	17	Der Datentyp	33
COA TypoScript Objekte	18	lang: Mehrsprachigkeit	33
Objekte, die Abfragen der Datenbank durchführen .	18	cObject	33
		Ausblick	34

Introduction

About this document

Dieses Tutorial stellt eine kurze Einführung in die Funktionsweise von TypoScript dar und soll illustrieren, was TypoScript eigentlich ist. Es soll helfen, TypoScript wirklich zu verstehen anstatt immer nur per Copy- und Paste zu arbeiten.

What's new

Dieses Tutorial entspricht nach wie vor der Wiki-Seite <http://wiki.typo3.org/De:ts45min>, alle Änderungen der Wiki-Seite werden – sofern sinnvoll – hier mit aufgenommen.

Credits

Dieses Tutorial ist als Community-Tutorial entstanden. Im Laufe der Zeit haben viele Community-Mitglieder durch ihre Mithile, ihre Anregungen und ihren Support bei der Entstehung dieses Tutorials geholfen.

Maßgeblich beteiligt an dieser Einführung sind:

1. Martin Holtz
2. Susanne Moog

weiter haben durch unzählige Hinweise, Ergänzungen, Korrekturen etc. geholfen:

1. Daniel Brüßler
2. M4rtijn
3. Kees van der Hall
4. und viele weitere TYPO3-Community-Mitglieder

Danke euch allen.

Feedback

Bei generellen Fragen zur Dokumentation kontaktiere bitte documentation@typo3.org (auf Englisch).

Falls ein Fehler in diesem Tutorial auffällt, erstelle bitte einen neuen Bugtracker-Eintrag unter http://forge.typo3.org/projects/typo3v4-doc_tut_ts45/issues mit der Kategorie „doc_tut_ts45_de“.

Gute Dokumentation zu produzieren und auf einem aktuellen Stand zu halten ist harte Arbeit. Das Documentation Team ist daher im auf der Suche nach freiwilligen Helfern. Wenn du Lust hast uns zu helfen, dann komm doch auf unsere Mailingliste (typo3.projects.documentation@lists.typo3.org).

TypoScript: ein kleiner Rundblick

Diese Einführung in TypoScript ist ein erster Schritt. Da wir diese Einführung im Wiki pflegen, liegt es jedem frei, dort Anmerkungen, Hinweise oder Fragen hinzuzufügen. Diese werden dann in die kommenden Versionen einfließen.

Maßgeblich beteiligt an dieser Einführung sind:

- Susanne Moog
- Martin Holtz

weiter haben durch Hinweise, Ergänzungen, Korrekturen etc. geholfen:

- Daniel Brüßler
- M4rtijn
- Kees van der Hall
- und viele weitere Nachtwerker

Einleitung

Das Ziel dieser Einführung ist nicht "ah, jetzt läuft's", sondern "ah, jetzt hab ich es verstanden". Oftmals wird versucht, willkürlich irgendwelche Eigenschaften an diverse Objekte anzuhängen, obwohl für alle, die TypoScript kennen, auf den ersten Blick klar ist, dass das so nicht funktionieren kann. Es spart viel Zeit, zu verstehen was passiert - denn dann ist die Fehlersuche ungleich einfacher. Und die Zeit, die dafür benötigt wird, wird am Ende eingespart. Andernfalls ist es Glücksache!

Das Ziel dieser Einführung ist nicht, am Ende eine laufende TYPO3-Installation stehen zu haben, sondern für alle, die das bereits geschafft haben, eine Erklärung zu bieten, warum es funktioniert.

Backend-Konfiguration

TypoScript hat an unterschiedlichen Stellen einen Einfluss. Wenn TypoScript im Benutzer/Gruppen-TypoScript-Feld oder im Seiten-TypoScript-Feld verwendet wird, dann wird damit das Aussehen und Verhalten der Formulare im Backend beeinflusst.

Die Ausgabe der Webseite (Frontend-Rendering) dagegen wird durch das TypoScript in den TypoScript-Templates festgelegt. Dieser Rundblick beschäftigt sich ausschließlich mit dem Frontend-Rendering und dort nur mit TypoScript im Allgemeinen.

Voraussetzungen

Wir gehen davon aus, dass beim Leser **TYPO3** **installiert** und lauffähig ist, und dass das grundsätzliche Arbeiten mit TYPO3 bekannt ist. Der Unterschied zwischen Seiten und Inhaltselementen wird hier nicht mehr weiter erläutert. Auch wird angenommen, dass bekannt ist, dass der Inhalt einer Seite durch die Kombination von unterschiedlichen Inhaltselementen bestimmt wird. Sicherheitshalber weisen wir darauf hin, dass diese Inhaltselemente alle in der Tabelle `tt_content` gespeichert werden. Mit dem Datenbank-Feld "CType" wird definiert, welches Inhaltselement vorliegt. Je nach Typ werden unterschiedliche Masken geladen.

Für das Verständnis von TYPO3 und TypoScript ist es hilfreich, sich in der Datenbank umzuschauen. Die Extension **Tools>phpMyAdmin** (**phpmyadmin**) ermöglicht einen einfachen und komfortablen Zugriff aus dem Backend auf die Zusammenhänge zwischen den Tabellen `pages`, `tt_content` und dem Backend von TYPO3. Bekannt sollte auch sein, dass die PID (Page ID) für die ID einer Seite steht und die UID (Unique ID) für einen eindeutigen Datensatz.

Warum TypoScript

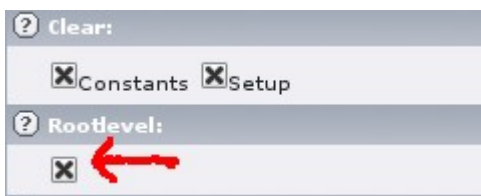
Genaugenommen ist TypoScript eine Konfigurationssprache. Wir können damit nicht programmieren, aber doch sehr viel und umfangreich konfigurieren. Mit TypoScript definieren wir die Ausgabe der Webseite. Wir definieren unser Menü, bestimmte fixe Inhalte, aber auch wie jedes einzelne Inhaltselement, das auf einer Seite angelegt ist, ausgegeben werden soll.

TYPO3 ist ein Content-Management-System mit dem Ziel Inhalt und Gestaltung zu trennen. TypoScript ist der Kleber, der das ganze wieder zusammenfügt. Die Inhalte aus der Datenbank werden durch TypoScript ausgelesen und aufbereitet, anschließend auf der Website ausgegeben.

Um eine Website auszugeben, muss also lediglich definiert werden, was wie ausgegeben werden soll. Das "was" wird über das Backend gesteuert - dort werden Seiten mit Inhalten angelegt - diese sollen ausgegeben werden. Das "wie" wird über TypoScript gesteuert.

Mit TypoScript wird z.B. definiert, wie die einzelnen Inhaltselemente bei der Ausgabe dargestellt werden sollen, ob zum Beispiel ein div-Container das Element umschließt oder die Überschrift in <h1> eingeschlossen werden soll.

Das TypoScript, das definiert, wie die Seite ausgegeben wird, liegt in einem "main"-Template. In diesem ist das Rootlevel-Flag gesetzt.



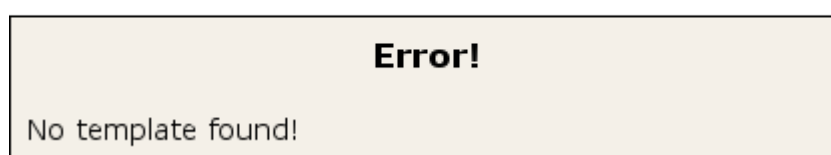
Wenn die Ausgabe einer Seite erzeugt werden soll, sucht TYPO3 entlang des Seitenbaums nach oben die Seiten ab, ob dort ein "main" Template liegt. In der Regel existieren neben dem "main" Template noch weitere Templates. Wie diese zusammen spielen kann im Template-Analyzer nachverfolgt werden. Für den Anfang gehen wir nur von einem Template aus.

Die Syntax von TypoScript ist simpel. Auf der linken Seite werden Objekte mit Ihren Eigenschaften definiert, denen dann bestimmte Werte zugewiesen werden. Objekte und Eigenschaften (die wiederum Objekte enthalten können) werden mit einem Punkt "." getrennt.

Der Begriff Template

Der Begriff Template hat in TYPO3 eine doppelte Bedeutung. Zum Einen gibt es das HTML-Template, die Vorlage oder das Grundgerüst in das später die Inhalte geschrieben werden sollen, zum Anderen gibt es noch die TypoScript-Templates die in den Seiten angelegt werden können.

Übliche Fehler im Zusammenhang mit den TypoScript-Templates können folgendermaßen aussehen:



"No template found": Diese Warnung erscheint, wenn kein Template mit einem gesetzten Rootlevel-Flag im Seitenbaum auf der aktuellen Seite oder auf dem Weg bis zur Root-Seite (Globus) existiert.



Error!

The page is not configured! [type= 0][]

"The page is not configured": Diese Warnung erscheint, wenn das Rootlevel-Flag gesetzt ist, aber kein Objekt PAGE erzeugt wurde. Der folgende Code - in das SETUP-Feld kopiert - reicht bereits um keine Fehlermeldung zu bekommen.

```
page = PAGE
page.10 = TEXT
page.10.value = Hallo Welt
```

auch TypoScript ist nur ein Array

TypoScript wird intern als PHP-Array gespeichert und dann entsprechend des Objekt-Typs von unterschiedlichen Klassen verwendet und ausgewertet.

```
page = PAGE
page.10 = TEXT
page.10.value = Hallo Welt
page.10.wrap = <h2>|</h2>
```

Wird in das folgende PHP-Array umgewandelt:

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

Bei der Auswertung wird somit zuerst das Objekt "PAGE" erzeugt und erhält den Parameter \$data['page.']. Das Objekt "PAGE" sucht dann nach allen Eigenschaften die es definiert hat. In diesem Fall findet es nur einen numerischen Eintrag "10" der auszuwerten ist. Ein neues Objekt "TEXT" mit dem Parameter \$data['page.']['10.'] wird erzeugt. Das Objekt TEXT selbst kennt nur den Parameter "value", setzt also den Inhalt entsprechend. Alle weiteren Parameter werden an die Funktion stdWrap weitergegeben (so ist TEXT implementiert, mehr zu stdWrap etc. später). Dort ist die Eigenschaft 'wrap' bekannt und der Text "Hallo Welt" wird an die Position der Pipe (|) geschrieben und zurückgegeben.

Dieser Zusammenhang ist wichtig für das Verständnis des Verhaltens von TypoScript an vielen Stellen. Wenn zum Beispiel das TypoScript um folgende Zeile erweitert wird:

```
page.10.meineEigeneFunktion = Magie!
```

Dann wird der Eintrag entsprechend ins PHP-Array übernommen:

```
$data['page.']['10.']['meineEigeneFunktion'] = 'Magie!';
```

Allerdings kennt weder das Objekt TEXT noch die von TEXT verwendete Funktion stdWrap eine Eigenschaft "meineEigeneFunktion". Somit bleibt der Eintrag ohne Wirkung.

Auch wenn die Syntax korrekt ist, hat die Eigenschaft keinen Einfluss - eine Plausibilitätsprüfung gibt es in diesem Fall nicht. Insbesondere im Zusammenhang mit dem TypoScript Editor sollte das beachtet werden. Nur weil der freundlich ein Syntax-Highlighting macht, bedeutet es nicht, dass das TypoScript so ausgewertet wird wie gewünscht.

Dieser Zusammenhang sollte beim Arbeiten - insbesondere bei der Fehlersuche - berücksichtigt werden.

Erste Schritte

Im Setup des Main-Templates wird die grundsätzliche Seitenausgabe definiert.

TypoScript besteht im Wesentlichen aus Objekten, die bestimmte Eigenschaften haben. Einigen Eigenschaften können neue Objekte hinzugefügt werden, andere stehen für bestimmte Funktionen oder definieren das Verhalten des Objektes.

Für die Ausgabe ist das Objekt `PAGE` zuständig:

```
# das objekt "meineseite" wird als PAGE Objekt definiert
meineseite = PAGE

# es besitzt die Eigenschaft typeNum
meineseite.typeNum = 0

# und ein Objekt "10" das vom Typ [[De:TSref/TEXT|TEXT]] ist.
meineseite.10 = TEXT

# Das Objekt 10 besitzt wiederum eine Eigenschaft value
meineseite.10.value = Hallo Welt
```

Das `PAGE`-Objekt bietet neben zahlreichen Eigenschaften eine unendliche Anzahl an Objekten, die nur anhand Ihrer Nummer identifiziert werden (ein sogenanntes Content Array). D.h. sie bestehen nur aus Zahlen und sie werden bei der Ausgabe entsprechend sortiert. Zuerst wird das Objekt mit der kleinsten Zahl ausgegeben, am Ende das Objekt mit der größten Zahl. Die Reihenfolge im TypoScript ist dabei unerheblich.

```
meineseite.30 = TEXT
meineseite.30.value = Das ist der Schluss

# Bei der Ausgabe wird zuerst das Objekt 10, dann 20 und danach 30 ausgegeben. Ein Objekt
25 würde entsprechend
# zwischen 20 und 30 ausgegeben werden.
meineseite.20 = TEXT
meineseite.20.value = Ich stehe in der Mitte

# Dies ist das erste Objekt
meineseite.10 = TEXT
meineseite.10.value = Hallo Welt!

# hier wird ein 2tes Page-Objekt für die Druckausgabe erzeugt
druckausgabe = PAGE
druckausgabe.typeNum = 98
druckausgabe.10 = TEXT
druckausgabe.10.value = Dies hier wird per Drucker ausgegeben.
```

Jeder Eintrag wird intern in einem multidimensionalen PHP-Array gespeichert. Jedes Objekt und jede Eigenschaft ist daher eindeutig. Wir können beliebig viele `PAGE`-Objekte definieren, allerdings muss der `typeNum` eindeutig sein - pro `typeNum` darf nur ein `PAGE`-Objekt existieren. Im Beispiel wird mit dem Parameter `typeNum = 98` ein anderer Ausgabemodus erzeugt. Über den `typeNum` können unterschiedliche Ausgabetypen definiert werden. Klassischerweise steht `typeNum=0` für die normale HTML-Ausgabe. Der Aufruf lautet dann für HTML `index.php?id=1` bzw. `index.php?id=1&type=98` für die Druckausgabe. Der Wert von `&type` legt also fest, welches `PAGE`-Objekt ausgegeben wird. So kann in ein und der selben Konfiguration zwischen Druck-Ansicht, HTML-Ansicht oder auch PDF-Ansicht gewechselt werden. Dabei können Konfigurationen, die in allen drei Ansichten benötigt werden, entsprechend kopiert und kleine Änderungen dann im neuen Objekt vorgenommen werden. (zum Beispiel könnte man so den normalen Seiteninhalt in die Druckansicht kopieren, das Menü aber nicht)



Hinweis

Die Ausgabe dieser Beispiele würden beide als normaler Text ausgegeben. Insbesondere bei Ausgabeformaten wie WML müsste der HTTP-Header geändert werden usw. Das soll hier aber nicht behandelt werden.

Intern wird TypoScript als ein einziges PHP-Array verwaltet. Das obige Beispiel würde in PHP z.B. so geschrieben werden:

```
$typoscript['meineseite'] = 'PAGE';
$typoscript['meineseite.']['typenum'] = 0;
$typoscript['meineseite.']['10'] = 'TEXT';
$typoscript['meineseite.']['10.']['value'] = 'Hello World!';
$typoscript['meineseite.']['20'] = 'TEXT';
$typoscript['meineseite.']['20.']['value'] = 'Ich stehe in der Mitte';
$typoscript['meineseite.']['30'] = 'TEXT';
$typoscript['meineseite.']['30.']['value'] = 'Das ist der Schluss';
```

Leerzeichen am Anfang und am Ende werden durch TYPO3 entfernt (trim()).

Mit dem = haben wir die erste einfache Zuweisung kennen gelernt: ein Wert wird gesetzt.

```
# = Wert wird gesetzt
test = TEXT
test.value = Holla

# < Objekt wird kopiert
# meineseite.10 gibt "Holla" aus
meineseite.10 < test

# Objekt, das kopiert wurde, wird geändert
# Änderung hat keine Auswirkungen auf meineseite.10
test.value = Hallo Welt

# => Objekt wird referenziert (auf das Objekt wird nur verwiesen)
test.value = Holla
meineseite.10 => test

# Objekt, das referenziert wird, wird geändert
# Änderung HAT Auswirkungen auf meineseite.10
# meineseite.10 gibt "Hallo Welt" aus
test.value = Hallo Welt
```

Objekte werden immer in Großbuchstaben geschrieben, Parameter und Funktionen in der Regel in camelCase (erstes Wort klein, dann alle folgenden Worte mit einem großen Anfangsbuchstaben, Ausnahmen gibt es allerdings noch einige).

Mit dem . als Trenner werden Parameter, Funktionen oder Kind-Objekte angesprochen und können entsprechend auch mit Werten gesetzt werden.

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

Welche Objekte, Parameter und Funktionen existieren, können wir in der [TypoScript-Referenz](#) nachlesen.

Wenn einige Objekte ineinander verschachtelt und viele Parameter gesetzt werden, dann entsteht viel Tipparbeit.

```
meineseite = PAGE
meineseite.typenum = 0
meineseite.10 = TEXT
meineseite.10.value = Hallo Welt
meineseite.10.typolink.parameter = http://www.example.org/
meineseite.10.typolink.additionalParams = &nix=nix

# ATagParams hält sich leider nicht an die Vorgabe "camelCase"
meineseite.10.typolink.ATagParams = class="externewebseite"
meineseite.10.typolink.extTarget = _blank
```



```
meineseite.10.typolink.title = Die Standard-Beispiel-Webseite
meineseite.10.postCObject = HTML
meineseite.10.postCObject.value = Dieser Text steht auch im Linktext
meineseite.10.postCObject.value.wrap = |, da das postCObject vor der typolink Funktion
ausgeführt wird.
```

Der Einfachheit halber sind geschweifte Klammern `{}` erlaubt um Objektebenen zu definieren, runde Klammern `()` um Texte auch über mehrere Zeilen zu schreiben. Das obige Beispiel könnte man mit Klammern auch so schreiben:

```
meineseite = PAGE
meineseite {

    typeNum = 0

    10 = TEXT
    10 {

        value = Hallo Welt
        typolink {

            parameter = http://www.martinholtz.de/
            additionalParams = &nix=nix

            # ATagParams hält sich leider nicht an die Vorgabe "CamelCase"
            ATagParams = class="externewebseite"

            extTarget = _blank
            title = Die Webseite von Martin Holtz, dem Autor dieser Zeilen.
        }

        postCObject = HTML
        postCObject {

            value = Dieser Text steht auch im Linktext
            value {
                wrap (
                    |, da das postCObject vor der typolink Funktion ausgeführt wird.
                )
            }
        }
    }
}
```

Die Gefahr von Tippfehlern sinkt und die ganze Sache ist übersichtlicher. Zudem würde man, wenn man aus dem Objekt "meineseite" das Objekt "page" machen wollte, nun nur noch die ersten zwei Zeilen statt jeder Zeile ändern müssen. Die Verwendung von „page“ als Bezeichnung für das PAGE-Objekt ist häufig zu finden, vor allem auch in Code-Beispielen auf vielen Tutorial-Seiten.

Inhalte einlesen



Hinweis

Die folgenden Absätze dienen als Beispiel zum Verständnis des Hintergrundes und der Zusammenhänge. Der Code wird von `css_styled_content` bereits geliefert - er muss daher nicht jedesmal von Hand eingegeben werden. Wenn aber ein bestimmtes Inhaltselement ganz anders erstellt werden soll, oder eine eigene Extension ein eigenes Inhaltselement erzeugt, dann ist es notwendig die Zusammenhänge zu verstehen.

Wir wollen nicht alle Inhalte per TypoScript eingeben - das wäre zu mühsam und einem Redakteur können wir das auch nicht zumuten.

Also legen wir einige Inhaltselemente vom Typ `"TEXT"` an und erzeugen ein TypoScript, das uns die Inhalte automatisch holt. Dieses Beispiel erzeugt eine Seite mit den Überschriften und dem Text von allen Seitenelementen auf der aktuellen Seite.

Zuerst wird das `PAGE`-Objekt angelegt, damit überhaupt eine Ausgabe stattfinden kann. Innerhalb des Objektes `PAGE` wird dann an der Stelle "10" das Objekt `CONTENT` angelegt, das über verschiedene TypoScript-Parameter gesteuert wird.

```

page = PAGE
page.typenum = 0

# Das Content-Objekt führt eine Datenbank Abfrage durch und
# lädt den Inhalt
page.10 = CONTENT
page.10.table = tt_content
page.10.select {

    # "sorting" ist ein Tabellenfeld aus
    # der Tabelle tt_content und enthält
    # die Sortierreihenfolge wie im
    # Backend angezeigt
    orderBy = sorting

    # Normale Spalte
    where = colPos = 0
}

# Für jede Ergebnis-Zeile aus der Datenbankabfrage
# wird das renderObj ausgeführt und das interne Daten-Array
# mit den Inhalten gefüllt, so dass z.B. über die .field Eigenschaft
# der Wert des entsprechenden Feldes geholt werden kann.
page.10.renderObj = COA
page.10.renderObj {

    10 = TEXT

    # Im Feld tt_content.header steht üblicherweise die Überschrift.
    10.field = header

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

    20 = TEXT

    # Im Feld tt_content.bodytext steht der Text.
    20.field = bodytext

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

```

Das Objekt `CONTENT` erzeugt eine SQL-Abfrage auf die Datenbank. Die Abfrage wird von "select" gesteuert. Dort wird definiert, dass aus der Tabelle `tt_content`, sortiert nach dem Feld "sorting", alle Datensätze aus der Spalte 0 (das ist üblicherweise im Backend die Spalte "normal") ausgelesen werden sollen. Wenn die Eigenschaft `pidInList` nicht gesetzt ist oder gelöscht wird, so wird die Abfrage auf die

aktuelle Seite beschränkt. D.h. wenn die Seite mit der ID 100 aufgerufen wird, gibt das **CONTENT**-Objekt nur Datensätze zurück, die auf der Seite 100 angelegt sind (pid = 100). Die Eigenschaft `renderObj` definiert dann, wie die Datensätze ausgegeben werden. Dafür wird es als COA (Content Object Array) definiert, das eine beliebige Anzahl unterschiedlicher TypoScript-Objekte aufnehmen kann. In diesem Fall werden zwei **TEXT**-Objekte definiert, diese werden nacheinander ausgegeben. Die Reihenfolge der Ausgabe wird nicht über die Reihenfolge im TypoScript definiert, sondern über die Zahlen mit denen sie definiert sind. Das **TEXT**-Objekt "10" wird daher vor dem **TEXT**-Objekt "20" erzeugt. Möchte man jetzt ein Objekt dazwischen einfügen, muss eine Zahl zwischen 10 und 20 gewählt werden - die Position im TypoScript ist dagegen unwichtig.

Die Herausforderung besteht darin, alle Felder die wir in dem Inhaltselement "Text" haben so auszugeben, wie uns das der Webdesigner vorgegeben hat, dafür müssen wir für jedes Feld (z.B. für Bilder, Bild-Größen, Bild-Positionen, nach oben, index etc.) in der `tt_content` eine Definition anlegen.

Die unterschiedlichen Inhaltselemente

Wenn wir jetzt anstelle eines Textes aber ein Bild ausgeben wollen, dann müssen wir andere Felder aus der `tt_content` verwenden und diese auch anders anzeigen als reinen Text. Das gleiche gilt dann für Text mit Bild, Überschrift, Liste, Tabelle usw. Der Typ des Content-Elements wird im Tabellenfeld `tt_content.CType` gespeichert. Im folgenden Beispiel wird gezeigt, dass mit dem Objekt **CASE** unterschieden werden kann, wie die einzelnen Inhaltselemente dargestellt werden.

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

    # das Feld CType wird für die Fallunterscheidung verwendet.
    key.field = CType

    # Der Inhaltstyp "Überschrift" wird intern als "header" abgelegt.
    header = TEXT
    header.field = header
    header.wrap = <h1>|</h1>

    # Text wird normal als Text verwendet.
    text = COA
    text {

        10 = TEXT
        # Im Feld tt_content.header steht üblicherweise die Überschrift.
        10.field = header
        10.wrap = <h1>|</h1>

        20 = TEXT
        # Im Feld tt_content.bodytext steht der Text.
        20.field = bodytext
        20.wrap = <p>|</p>

    }

    # ... und hier folgen noch einige.
}
```

css_styled_content

Da es mühsam ist, diese Sachen immer wieder neu zu programmieren und in der Regel die Elemente immer gleich oder zumindest ähnlich funktionieren, liefert uns TYPO3 einige "statische" Templates mit. Das derzeit aktuelle ist "`css_styled_content`". Es enthält für alle möglichen Inhaltselemente sinnvolle Definitionen.

Die Anwendung ist vergleichbar einfach. Die Definitionen stehen als `tt_content` Objekt zur Verfügung.

```
10.renderObj < tt_content
```

Diese Zuordnung ist auch die Default-Konfiguration des `CONTENT`-Elements, wenn also das static-Template "css_styled_content" zum Setup hinzugefügt wird, dann ist es nicht mehr nötig, den Parameter "renderObj" zu setzen.

Für jedes einzelne Inhaltselement in TYPO3 gibt es also eine entsprechende Definition in `css_styled_content`. Im Objektbrowser sieht das dann wie folgt aus:

Es kann also einfach nachvollzogen werden, welches Inhaltselement wie konfiguriert wird. Und wenn ein Inhaltselement gänzlich anders konfiguriert werden soll, dann sollte jetzt klar sein, dass das via `tt_content.interne Bezeichnung des Inhaltselements` gemacht werden kann. Hier ein Beispiel, wie man die Standardeinstellungen für Überschriften überschreibt:

```
# Da TYPO3 intern alles in einem großen Array speichert, würden Eigenschaften, die nicht
# überschrieben
# werden, erhalten bleiben und könnten für merkwürdige Nebeneffekte sorgen. Daher löscht
# man zunächst
# die alten Einstellungen
tt_content.header >

# Jede Überschrift wird immer als h1 angezeigt, unabhängig von den Einstellungen
# im Inhaltselement
tt_content.header = TEXT
tt_content.header.wrap = <h1>|</h1>
tt_content.header.field = header
```

Aber nicht nur das *renderObj* wird von `css_styled_content` bereits zur Verfügung gestellt, sondern auch das Content-Objekt.

styles.content.get

```
# unser bisheriger Code
page.10 = CONTENT
page.10.table = tt_content
page.10.select {

    # Sortierung aus dem Backend übernehmen.
    # Wir könnten auch ein Datumsfeld die Überschrift oder
    # sonst etwas nehmen.
    orderBy = sorting

    # Normale Spalte
    where = colPos = 0
}
```

Dank `css_styled_content` reicht die folgende Eingabe, um den gleichen Effekt zu erzielen:

```
# Gibt die Spalte "normal" (colPos = 0) aus
page.10 < styles.content.get
```

Alternativ gibt es für die anderen Spalten auch Default-Definitionen:

```
# Inhalt der linken Spalte ausgeben
page.10 < styles.content.getLeft

# Inhalt der rechten Spalte ausgeben
page.10 < styles.content.getRight

# Inhalt der Rand-Spalte ausgeben
page.10 < styles.content.getBorder
```

In der `css_styled_content` wird z.B. die Rand-Spalte wie folgt definiert:

```
# die normale Spalte wird kopiert
styles.content.getBorder < styles.content.get

# und anschließend wird nur die colPos angepasst.
```

```
styles.content.getBorder.select.where = colPos=3
```

Ein Menü bauen

Wir haben bisher gelernt, wie der Inhalt auf der Seite ausgegeben wird. Allerdings fehlt die für eine Website übliche Navigation.

Dafür bietet TYPO3 ein spezielles Menü-Objekt **HMENU** (H für hierarchisch). Aufbauend auf diesem Menü können unterschiedliche Arten von Menüs erzeugt werden.

Das Menü soll als eine geschachtelte Liste aufgebaut werden:

```
<ul>
  <li>auf der 1. Ebene</li>
  <li>auf der 1. Ebene
    <ul>
      <li>auf der 2.Ebene</li>
    </ul>
  </li>
  <li>auf der 1. Ebene</li>
</ul>
```

Um den Überblick nicht zu verlieren, legen wir in einem neuen Sysfolder ein Extension-Template an. Innerhalb dieses Templates definieren wir ein neues Objekt, das wir dann später dem Main-Template hinzufügen können. So können unterschiedliche Objekte einfacher auseinander gehalten werden und einfach für weitere Projekte verwendet werden. Das Extension-Template wird dann im Main-Template unter "Include basis template:" hinzugefügt.

Üblicherweise werden diese Objekte als Unterobjekte von "lib" definiert. Es könnte aber auch jede beliebige und noch nicht verwendete Bezeichnung genommen werden.

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

    # wir definieren die erste Menüebene als Textmenü
    1 = TMENU

    # Wir definieren den Normalzustand
    1.NO.allWrap = <li>|</li>

    # Wir definieren den ACTiven Zustand
    1.ACT = 1
    1.ACT.wrapItemAndSub = <li>|</li>

    # Die ganze Menü-Ebene schachteln wir in ein UL
    1.wrap = <ul class="ebene1">|</ul>

    # Die zweite Ebene soll genau so angelegt werden.
    # innerhalb der geschweiften Klammer können wir
    # Objekte auch kopieren, indem wir mit dem
    # "." anzeigen, dass das Objekt innerhalb der
    # Klammer existiert.
    2 < .1
    2.wrap = <ul class="ebene2">|</ul>
    3 < .1
    3.wrap = <ul class="ebene3">|</ul>
}
```

Das Objekt **HMENU** ist ein Objekt, das es ermöglicht die unterschiedlichsten Menüs zu erzeugen. Für jede Ebene kann ein beliebiges Menü-Objekt verwendet werden, dass das Rendering der Ebene übernimmt. So ist es möglich in der ersten Ebene ein grafisches Menü (**GMENU**) zu erzeugen und in der zweiten und dritten Ebene ein Textmenü (**TMENU**) zu erzeugen. Die erste Menü-Ebene wird also über die 1 definiert, die zweite über die 2 usw. Natürlich dürfen keine Lücken existieren - wenn die dritte Ebene nicht definiert ist, wird auch die vierte Ebene nicht erzeugt.

Auf jeder Menü-Ebene können für die verschiedene **Menü-Zustände** (NO="normal", ACT="Seiten in der Rootline, d.h. aktuelle Seite und deren Eltern-, Großeltern, Ur... usw.-Seiten", CUR="aktuelle Seite" etc.) unterschiedliche Definitionen konfiguriert werden. Dabei ist darauf zu achten, dass außer dem normalen Zustand ("NO") alle anderen Zustände explizit aktiviert werden müssen (z.B ACT=1).

Nun können wir dieses Menü verwenden und es in unsere neue Seite einfügen:

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

Inhalte in ein Template einfügen

Wir haben gesehen, wie wir Inhalte ausgeben und Menüs aufbauen können, aber eine richtige Website haben wir damit immer noch nicht.

Wir könnten die Website mit COAs erstellen und das ganze HTML-Gerüst mit TypoScript nachbilden. Allerdings ist das eine aufwändige und fehlerträchtige Arbeit. Wenn das HTML-Template auch noch von einem Template-Designer fix und fertig geliefert wird, dann wird es - vorallem auch bei jeder Änderung - richtig anstrengend.

Daher gibt es das Element **TEMPLATE**, mit dem ein HTML-Template eingelesen wird, in das dann an die richtige Stelle das Menü, der Inhalt und noch weitere Informationen (wie das Logo) eingefügt werden können.

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

    # Wir laden unsere HTML-Vorlage
    template.file = fileadmin/test.tpl

    # Textbereiche:
    # <!-- ###MENU### begin -->
    # Hier steht Beispielinhalt als Platzhalter. Alles zwischen den Markern wird
    # durch den Inhalt des subparts, in diesem Fall durch den Inhalt des
    # Menüs, ersetzt.
    # <!-- ###MENU### end -->
    subparts {
        MENU < lib.textmenu
        INHALT < styles.content.get
        SPALTERECHTS < styles.content.getRight
    }

    # Marks sind einzelne Marker. D.h. es gibt kein begin oder end, sondern
    # der Marker wird direkt ersetzt.
    # ###LOGO###
    # Wird durch das Logo ersetzt.
    marks {
        LOGO = IMAGE

        # Die Grafik logo*.gif wird über das Ressourcenfeld des
        # TypoScript-Templates hinzugefügt.
        LOGO.file = logo*.gif

        # Das Logo linkt auf die Seite mit der ID 1
        LOGO.stdWrap.typolink.parameter = 1
    }
    workOnSubpart = DOCUMENT
}
```

Alternativ zu diesem Weg existiert eine Extension **Template Auto-parser** (**automaketemplate**) (contact: kasper) mit deren Hilfe es möglich ist, auf die Marker zu verzichten und stattdessen auf gesetzte IDs zuzugreifen. Das ermöglicht eine reibungslosere Zusammenarbeit mit den Template-Designern.

Eine weitere Alternative ist **Templavoila** (**templavoila**) (contact: dmitry). Sie ermöglicht eine sehr visuelle Vorgehensweise.

css_styled_content nutzen

Dass wir die Definitionen für die unterschiedlichen Content-Elemente von TYPO3 selber programmieren können, haben wir bereits gesehen. `Css_styled_content` nimmt uns diese Arbeit mit seinen rund 2000 Zeilen TypoScript allerdings ab.

Es lohnt sich - auch wenn es anfangs nicht so klar ist was da passiert - das TypoScript genauer anzusehen: Wir müssen in TYPO3 auf der Seite sein, auf der das Setup-Template eingerichtet ist. Dann wählen wir im Modul "Template" den Eintrag "Template Analyzer" aus der Auswahlbox.

Es erscheint eine Liste mit den aktiven und eingebundenen TypoScript-Templates. Diese werden der Reihe nach (von oben nach unten) von TYPO3 ausgewertet und intern zu einem großen Konfigurationsarray zusammengefügt.

Mit einem Klick auf "EXT:css_styled_content/static/" wird der Inhalt des Templates dargestellt. Zuerst erscheinen die Konstanten danach das Setup-TypoScript.

Die Extension `css_styled_content` setzt an allen möglichen Stellen Klassen in die HTML-Elemente. Dieses hat zum Vorteil, dass es nicht mehr nötig ist, das selber zu machen, sondern in der Regel genügt, herauszusuchen welche Klasse welchen Effekt hat, und diese per CSS anzupassen.

Beispiel:

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

Die Bezeichnungen der Klassen sind einfach und - wenn die TYPO3-Interna ein wenig bekannt sind - intuitiv. Alle Klassen fangen mit "csc" an; das steht für "css_styled_content". Gefolgt wird dieses im Beispiel von "textpic", das wiederum für das TypoScript Element "textpic" (Text mit Bild) steht. "imagewrap" legt nahe, dass der Div-Container ein Bild umschließt (wrapt).

Was genau alles gemacht wird, kann aber auch einfach nachvollzogen werden, wenn man in eine leere Seite ein einzelnes Element einfügt und dann den Quelltext betrachtet.

In der Regel werden z.B. Überschriften durchnummeriert, so dass die erste Überschrift besonders behandelt werden kann. Bei Tabellen werden die Klassen "odd" (dt. "ungerade") und "even" (dt. "gerade") hinzugefügt, so dass es einfach ist, die Tabelle mit Zebrastreifen zu versehen. Genauso können die unterschiedlichen Spalten direkt angesprochen werden.

Für HTML Puristen bedeutet das allerdings, dass an sehr vielen Stellen CSS-Klassen hinzugefügt werden, die in dem aktuellen Projekt nicht verwendet werden. Um diese überflüssigen Klassen loszuwerden, muss im Zweifel fast die komplette `css_styled_content`-Extension geändert werden.

COA TypoScript Objekte

Die TypoScript-Objekte werden durch entsprechende Klassen in TYPO3 implementiert. Für die unterschiedlichen Anforderungen bei der Ausgabe einer Website gibt es unterschiedliche Objekte. Diese Objekte haben dann unterschiedliche Eigenschaften. So besitzt das Objekt `IMAGE` unter anderem eine Methode `wrap` und eine Methode `titleText`. In der TypoScript-Referenz kann dann der Datentyp nachgeschlagen werden, was dieses Objekt also für einen Wert erwartet. Bei `wrap` wird auch ein Datentyp `wrap` erwartet, also ein Text der durch eine Pipe (`|`) getrennt wird. An das `wrap` weitere Funktionen anzuhängen (z.B. `wrap.crop = 100`) ist daher sinnlos. In der Praxis werden solche Versuche jedoch immer wieder gesehen - und das obwohl mit einem Blick in die Referenz klar sein sollte, welche Methoden/Eigenschaften erwartet werden.

Das Objekt bekommt die Parameter wie bereits oben erläutert als PHP-Array übergeben, (z.B. `$conf['wrap.']['crop']='100'`;) dieses Array kann beliebig viele unterschiedliche Einträge enthalten, verwendet werden aber nur die, die das Objekt auch abrufen (z.B. `$conf['wrap']` oder `$conf['titleText']`).

In dem Fall `titleText` ist der Datentyp `string / stdWrap`, das bedeutet dass sowohl ein Text (`string`) als auch eine Methode vom Typ `stdWrap` erlaubt ist. Welche Eigenschaften `stdWrap` auswertet, können wir wieder in der [Referenz nachlesen](#). Somit dürfen wir an dieser Stelle die Methode `titleText` um beliebige Eigenschaften aus `stdWrap` erweitern (z.B.: `titleText.field = header`). Dabei wird der Wert für `titleText` zuerst mit dem normalen Text gefüllt und danach wird die `stdWrap` Funktion ausgeführt.

Es ist also nicht nötig, zu raten welches Objekt wie manipuliert werden kann, sondern es reicht diese Information in der Referenz nachzulesen.

Für die Ausgabe einer Website werden aber mehrere Objekte benötigt, die Herausforderung besteht darin diese geschickt zu kombinieren.

Im Abschnitt [Inhalte einlesen](#) wird gezeigt, wie mit dem TypoScript Objekt `CONTENT` eine Abfrage auf die Datenbank erzeugt wird und der Inhalt einer Seite ausgelesen wird. Das Objekt erhält dabei eine Liste von allen Inhaltselementen einer Seite die nacheinander - üblicherweise in der Sortierreihenfolge - erstellt werden. Dafür wurde das Objekt `CASE` verwendet, damit in Abhängigkeit vom Typ des Inhaltselements (CType) die Ausgabe unterschiedlich gerendert werden kann.

Es ist also unbedingt nötig, die verschiedenen TypoScript-Objekte und -Funktionen zu kennen.

Objekte, die Abfragen der Datenbank durchführen

- `CONTENT` bietet die Möglichkeit auf beliebige Tabellen innerhalb von TYPO3 zuzugreifen. Das heißt nicht nur `tt_content` sondern auch Tabellen von Extensions etc. können ausgelesen werden. Die Funktion `select` ermöglicht es komplexe SQL-Abfragen zu erstellen.
- `RECORDS` bietet die Möglichkeit bestimmte Datensätze zu holen. Sehr hilfreich, wenn auf allen Seiten der gleiche Text stehen soll. Via `RECORDS` kann dann ein bestimmtes Inhaltselement definiert werden, dass dann angezeigt wird. Somit kann der Inhalt von Redakteuren geändert werden, ohne dass das Element mehrfach kopiert werden muss. Das Objekt wird auch verwendet, wenn das Inhaltselement "Datensätze einfügen" verwendet wird.

Im folgenden Beispiel wird die E-Mail Adresse des Adress-Datensatzes ausgegeben und direkt als E-Mail Link verlinkt.

```
page.80 = RECORDS
page.80 {
    source = 1
    tables = tt_address
    conf.tt_address = COA
    conf.tt_address {
```

```

20 = TEXT
20.field = email
20.typoLink.parameter.field = email
}
}

```

- **HMENU** liest den Seitenbaum ein und bietet viele komfortable Ansätze zum Erzeugen von Menüs. Neben Menüs, die den Seitenbaum abbilden, gibt es noch die Special-Menüs mit denen viele andere Dinge umgesetzt werden können. Dieses Objekt liest intern die Struktur für das Menü ein. Wie dann das Menü dargestellt wird, wird durch Menü-Objekte wie **TMENU** (Text-Menü) oder **GMENU** (Grafisches-Menü) definiert. Für jede Menü-Ebene kann das Objekt gewechselt werden. Innerhalb einer Menü-Ebene gibt es unterschiedliche Menü-Items. Für jedes Item wiederum können unterschiedliche Status (NO=normal, ACT=Aktiv eine Seite in der Rootline, CUR=aktuelle Seite) definiert werden.

Objekte zur Ausgabe von Inhalten

- **IMAGE** die Ausgabe eines Bildes.

```

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

```

lib.logo enthält nun das Logo mit einer Breite von 200 Pixeln und wird verlinkt auf die Seite mit der PID 1.

- **HTML / TEXT** für die Ausgabe von einfachem Text oder dem Inhalt von Feldern. Wesentlicher Unterschied: das HTML-Objekt implementiert die stdWrap-Funktionalität auf .value.

```

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

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

```

- **FILE** importiert direkt den Inhalt einer bestimmten Datei.
- **TEMPLATE** ersetzt in einem Template die Marker durch Inhalte.

```

page.10 = TEMPLATE
page.10 {
    template = FILE
    template.file = fileadmin/test.tpl
    subparts {
        HELLO = TEXT
        HELLO.value = Ersetzt den Inhalt zwischen den beiden Markern ###HELLO### und
        ###HELLO###
    }
    marks {
        Test = TEXT
        Test.value = Der Marker "Test" wird durch diesen Text ersetzt.
    }
    workOnSubpart = DOCUMENT
}

```

- **MULTIMEDIA** rendert Multimedia Objekte.
- **IMGTEXT** ermöglicht es Bilder innerhalb von Text zu erzeugen. Wird verwendet für das Inhalts-Element "Bild mit Text"
- **FORM** erzeugt ein HTML-Formular.

weitere Objekte

- **CASE** das Objekt ermöglicht Fall-Unterscheidungen. In `css_styled_content` wird dieses Objekt dafür verwendet in Abhängigkeit des Felds `CType` unterschiedliche Objekte zu rendern.
- **COA**- Content Object Array - ermöglicht es beliebig viele Objekte zusammenzuführen.
- **COA_INT** - nicht gecached. Diese Elemente werden bei jedem Aufruf neu erstellt und berechnet. Sinnvoll z.B. für Uhrzeiten oder benutzerabhängige Daten
- **LOAD_REGISTER** / **RESTORE_REGISTER** Dieses Objekt ermöglicht es das globale Array `$GLOBALS["TSFE"]->register[]` mit Inhalt zu füllen. Dieses Objekt selber gibt nichts zurück. Es können einzelne Werte aber auch ganze TypoScript-Objekte verwendet werden. Dabei arbeitet das Register als Stack (Stapel), mit jedem Aufruf wird ein weiterer Inhalt oben auf den Stapel gepackt. Mit **RESTORE_REGISTER** können Sie das jeweils oberste Element auch wieder entfernen
- **USER** und **USER_INT** - Benutzerdefinierte Funktionen, jedes Plugin ist ein solches Objekt. **USER_INT** ist dabei die nicht gecachte Variante.
- **IMG_RESOURCE** wird z.B. von **IMAGE** verwendet. Es wird die Ressource zurückgegeben, also der Inhalt, der normalerweise in das `SRC`-Attribut des **IMG**-Tags eingetragen wird. Wenn Bilder skaliert werden, werden durch dieses Objekt die Dateien berechnet und in `typo3temp/` abgelegt.
- **EDITPANEL** Dieses Objekt wird nur eingefügt, wenn ein Backend-User eingeloggt ist und für diesen die Einstellung "Display Edit Icons" im Frontend Admin Panel gesetzt ist. Wenn das Admin-Panel eingefügt ist, dann werden die Seiten nicht mehr gecached. Es werden Icons für das Verschieben, Editieren, Löschen, Verstecken und Erstellen von Datensätzen angezeigt.
- **GIFBUILDER** Der **GIFBUILDER** wird dafür verwendet, GIF-Dateien dynamisch zu erzeugen. Dabei können unterschiedliche Texte kombiniert werden, Bilder übereinander gelegt, Texte erzeugt und vieles mehr. Der **GIFBUILDER** selber bietet weitere Objekte wie **TEXT** oder **IMAGE** an, die allerdings nicht den normalen **TEXT** bzw. **IMAGE**-Objekten entsprechen. Bei Arbeiten mit dem **GIFBUILDER** muss also aufgepasst werden, dass die Objekte nicht verwechselt werden, auch wenn der Name derselbe ist - die Eigenschaften sind unterschiedlich implementiert.

Wir haben hier noch nicht alle Objekte vorgestellt, die in TypoScript existieren. Allerdings sind wir der Meinung, dass dieses hier die wichtigsten Objekte waren.

TypoScript Funktionen:

TypoScript-Funktionen werden genutzt um die Ausgabe bestimmter Elemente zu verändern und anzupassen. Die bekannteste Funktion ist der „stdWrap“. Ob ein Element eine bestimmte Funktion implementiert oder nicht, kann man in der TSref in der Spalte „Data type“ (Datentyp) ablesen.

Beispiel cObj (Content Object / Inhaltselement) IMAGE:

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

Im oberen Beispiel steht z.B. in der ersten Zeile (Property = file) der Datentyp imgResource angegeben. Dies bedeutet, dass wir auf die file-Eigenschaft die Funktionen von imgResource anwenden können.

Manchmal werden Funktionen auch zum besseren Verständnis mit einem kleinen Pfeil vor dem Namen gekennzeichnet (siehe -> if).

Stehen in der Zeile „Data type“ mehrere durch einen Schrägstrich getrennte Werte, so bedeutet dies, dass mehrere Möglichkeiten vorhanden sind, dieses Element zu nutzen. Im obigen Beispiel ist dies bei altText und titleText der Fall, diese haben die Datentypen „string“ und „stdWrap“ was bedeutet, dass ich eine einfache Zeichenkette (String) angeben kann, und Inhalt mittels der stdWrap-Funktionen bearbeiten (der String wird mit stdWrap bearbeitet) oder generieren kann (mit Hilfe von stdWrap wird anderer Inhalt geholt).

Hier werden einige wichtige und häufig verwendete Funktionen vorgestellt. Dabei geht es darum, diese Funktionen vorzustellen, deren Sinn zu erklären. Details zu diesen Funktionen und alle implementieren Eigenschaften findet Ihr dann allerdings in der TSref.

imgResource

Die Funktionen für den Datentyp „imgResource“ beziehen sich auf die Modifikation von Bildern, wie ihr Name schon vermuten lässt. Das Objekt **IMAGE** besitzt die Eigenschaft "file" die vom Datentyp „imgResource“ ist.

Sie ermöglichen zum Beispiel ein Bild in der Größe zu verändern,

```
temp.meinBild = IMAGE
temp.meinBild {
    file = toplogo.gif
    file.width = 200
    file.height = 300
}
```

Maximalgrößen (oder Mindestgrößen) anzugeben,

```
temp.meinBild = IMAGE
temp.meinBild {
    file = toplogo.gif
    # für Maximalgrößen
```

```

    file.maxW = 200
    file.maxH = 300

    # für Mindestgrößen
    file.minW = 100
    file.minH = 120
}

```

und sogar die direkte Angabe eines ImageMagick-Befehls:

```

temp.meinBild = IMAGE
temp.meinBild {

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

}

```

Eines der bekanntesten und schönsten Beispiele für die Benutzung von `imgResource` ist das Einfügen von dynamischen Bildern aus dem Media-Feld in den Seiteneigenschaften. Dies hat den Vorteil, dass Redakteure die Bilder ändern können ohne TypeScript zu nutzen und gleichzeitig auch zum Beispiel Header-Bilder für verschiedene Bereiche über ein wenig TypeScript realisiert werden können:

```

temp.dynamischerHeader = IMAGE
temp.dynamischerHeader {
    file {

        # Pfad zu Importdateien definieren
        import = uploads/media/

        import {

            # wenn kein Bild auf Seite, dann suche rekursiv bis Bild gefunden
            data = level:-1, slide

            # Feld angeben, in welchem das Bild definiert ist
            field = media

            # angeben, die wievielte Datei in dem Feld abgerufen wird
            listNum = 0

        }

    }
}

```

Der Pfad „uploads/media/“ ist der Pfad, in dem die Dateien landen, die man in den Seiteneigenschaften unter „Dateien“ hochladen kann. Der Teil innerhalb der geschweiften Klammern von „import“ besteht komplett aus `stdWrap`-Funktionen, die hier genutzt werden, um anzugeben, von wo und welches Bild genau importiert werden soll. Letztlich liefert `stdWrap` hier den Dateinamen des Bildes der dann aus dem Import-Pfad (uploads/media) importiert werden soll.

imageLinkWrap

Mit Hilfe der Funktion „`imageLinkWrap`“ erzeugt man einen Link um ein Bild auf das PHP-Skript „showpic.php“. Der Link wird im HTML um das `IMG`-Tag gelegt. Das Skript öffnet das Bild in einem neuen Fenster mit festlegbaren Parametern, wie Fensterhintergrund, Bildgröße etc. Diese Funktion kann genutzt werden um „Klick-Vergrößern“ für Bilder zu erzeugen (d.h. Ich habe ein kleines Bild (Thumbnail) und nach einem Klick auf dieses öffnet sich ein neues Fenster mit dem Bild in Originalgröße.).

```

temp.meinBild = IMAGE
temp.meinBild {

    file = toplogo.gif
    imageLinkWrap = 1
}

```

```

imageLinkWrap {

    # ImageLinkWrap aktivieren
    enable = 1

    # Body-Tag für neues Fenster definieren
    bodyTag = <body class="BildOriginal">

    # Das Bild umschließen (hier schließt ein Bild-Klick das geöffnete Fenster)
    wrap = <a href="javascript:close();"> | </a>

    # Breite des Bildes (m ermöglicht proportionales Skalieren)
    width = 800m

    # Höhe des Bildes
    height = 600

    # Ein neues Fenster für das Bild erstellen
    JWindow = 1

    # Für jedes weitere Bild neue Fenster öffnen (statt im gleichen Fenster)
    JWindow.newWindow = 1

    # Padding (Rand) des neuen Fensters
    JWindow.expand = 17,20

}

```

numRows

In TypoScript gibt es nicht nur große, mächtige Funktionen sondern auch kleine, mächtige Funktionen. So zum Beispiel die Funktion `numRows`, die eigentlich nichts anderes macht, als die Anzahl der Zeilen einer `select`-Abfrage zurückzugeben. Genau so wie das Objekt `CONTENT` verwendet `numRows` dafür die Funktion `select`. Die Abfrage wird somit in beiden Fällen gleich erzeugt - nur wird unterschieden ob die Anzahl der Ergebnisdatensätze zurückgegeben wird, oder ob die Ergebnisse selber zurückgegeben werden.

In Zusammenarbeit mit der „if“-Funktion lassen sich damit ganz nette Sachen realisieren, wie zum Beispiel ein Stylesheet für den Inhalt der rechten Spalte, das nur geladen wird, wenn in der rechten Spalte auch Inhalt steht:

```

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

    # wenn das select in Klammern mindestens 1 Zeile liefert, dann wird das Stylesheet
eingebunden
    if.isTrue.numRows {

        # diese Seite überprüfen
        pidInList = this

        # in der Tabelle tt_content
        table = tt_content

        # SQL: WHERE colPos = 2, deutsch: wo Spalte = rechts
        select.where = colPos=2

    }

}

# kopieren von temp.headerdata in page.headerData.66 (und damit überschreiben von
page.headerData.66)
page.headerData.66 < temp.headerdata

```

oder gleich ein anderes Template, wenn Inhalt in der rechten Spalte steht:

```

#ein COA (Content Object Array) ermöglicht es, beliebig viele Objekte zusammenzuführen
temp.maintemplate= COA

```

```
temp.maintemplate {
    # 10 wird nur eingebunden, wenn das if-Statement „wahr“ zurückgibt
    10 = COA
    10 {
        # das select von oben kommt hier als Kopie aus css_styled_content
        if.isTrue.numRows < styles.content.getRight

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

    # 20 wird nur eingebunden, wenn das if-Statement „wahr“ zurückgibt
    20 = COA
    20 {
        if.isFalse.numRows < styles.content.getRight
        10 = TEMPLATE
        10 {
            template = FILE
            template.file = fileadmin/templates/template.html
        }
    }
}
```

select

Die Funktion „select“ erstellt eine SQL SELECT-Query, die man verwendet, um Datensätze aus der Datenbank zu lesen. Die select-Funktion achtet dabei automatisch darauf, ob die Datensätze versteckt, gelöscht oder zeitlich beschränkt sind. Wenn pidInList verwendet wird (also eine Liste von Seiten angegeben wird), überprüft die Funktion auch, ob der aktuelle Benutzer den Datensatz sehen darf.

Mit Hilfe der select-Funktion kann man zum Beispiel den Inhalt einer Spalte einer bestimmten Seite auf allen Seiten anzeigen lassen:

```
temp.linkerInhalt = CONTENT
temp.linkerInhalt {
    table = tt_content
    select {
        # Seite mit ID 123 ist Quelle
        pidInList = 123

        # Reihenfolge wie im Backend angegeben
        orderBy = sorting

        # Inhalt der linken Spalte
        where = colPos=1

        # Definiert das Feld mit der Sprach-ID in tt_content.
        languageField = sys_language_uid
    }
}

# den Marker im Template mit dem temporären Objekt ersetzen
marks.LINKS < temp.linkerInhalt
```

split

Die Split-Funktion wird genutzt, um die Eingabe bei Vorkommen eines bestimmten Zeichens aufzutrennen und die jeweiligen Teile dann einzeln zu verarbeiten.

Bei jeder Iteration wird der aktuelle Index im Schlüssel „SPLIT_COUNT“ gespeichert (beginnend mit 0).

Mit Hilfe von „split“ kann zum Beispiel ein Tabellenfeld ausgelesen und jede einzelne Zeile mit bestimmtem Code gewrappt werden (um ggf. eine HTML-Tabelle mit Zeilen zu generieren, wenn an anderer Stelle der gleiche Inhalt nicht als Tabelle gebraucht wird):

```
# Beispiel
20 = TEXT

# Der Inhalt des Feldes "bodytext" wird importiert (aus $cObj->data-array)
20.field = bodytext
20.split {

    # Das Trennzeichen (char = 10 ist der Zeilenumbruch) wird definiert
    token.char = 10

    # Es wird festgelegt, welches Element verwendet werden soll
    # Über optionSplit kann hier zwischen unterschiedlichen Elementen
    # unterschieden werden. Ein Entsprechendes Element mit der Nummer muss definiert sein!
    # An dieser Stelle wird die optionSplit Eigenschaft verwendet,
    # es wird immer so abwechselnd das Element 1 und dann das Element 2 zum Rendern
    # verwendet - in diesem Beispiel werden abwechselnd den Zeilen die Klassen "odd" oder
    # "even" gegeben, so dass man ein Zebamuster färben könnte
    cObjNum = 1 || 2

    # Das Element 1 wird definiert (das, auf welches sich cObjNum bezieht!)
    # Und der Inhalt wird mittels stdWrap->current importiert.
    1.current = 1

    # Das Element wird gewrappt
    1.wrap = <TR class="odd"><TD valign="top"> | </TD></TR>

    # Das 2te Element wird definiert und gewrappt
    2.current = 1
    2.wrap = <TR class="even"><TD valign="top"> | </TD></TR>
}

# ein genereller Wrap wird um das ganze gelegt, um eine korrekte Tabelle zu erzeugen
20.wrap = <TABLE border="0" cellpadding="0" cellspacing="3" width="368"> | </TABLE>
```

if

Die wohl schwierigste TYPO3-Funktion ist die „if“-Funktion, da jeder, der ein if-Konstrukt in einer klassischen Programmiersprache kennt, diese Funktion instinktiv falsch benutzen wird. Daher hier ein paar Beispiele und was diese bewirken.

Generell gibt die if-Funktion „wahr“ zurück, wenn ALLE Bedingungen erfüllt sind, es sind also boolsche UND-Verknüpfungen. Will man, dass die Funktion bei Erfüllung aller Bedingungen „falsch“ zurückgibt, kann man die „negate“-Option benutzen, das Ergebnis also negieren (!(true)).

```
10 = TEXT
10 {

    # Inhalt des Textelements
    value = Der L-Parameter wird übergeben.

    # liefert „wahr“ und führt zur Anzeige des obigen values, wenn als GET/POST-Parameter
    das L mit
    # einem Wert ungleich 0 übertragen wird
    if.isTrue.data = GP:L
}
```

Mit Hilfe von if ist es auch möglich Werte zu vergleichen. Dazu wird der Parameter if.value genutzt.

```
10 = TEXT
10 {

    # ACHTUNG: hier value = value des Textelements, nicht von if
```

```

value = 3 ist größer als 2

# Vergleichsparameter der if-Funktion
if.value = 2

# bitte beachten: die Reihenfolge ist sozusagen rückwärts, diese Beispiel ergibt als
Satz „3
isGreaterThan 2“
if.isGreaterThan = 3
}

```

Da die einzelnen Eigenschaften der if-Funktion die stdWrap-Funktionen implementieren, können Variablen von überall her damit verglichen werden.

```

10 = TEXT
10 {

    # Wert des Textelements
    value = Der Datensatz kann angezeigt werden, weil der Startzeitpunkt vorbei ist.

    # Abfragewert der Bedingung
    if.value.data = date:U

    # Bedingung, wieder rückwärts zu lesen: starttime isLessThan date:U
    if.isLessThan.field = starttime
}

```

typolink

Typolink ist die TYPO3-Funktion, mit deren Hilfe man alle möglichen Arten von Links generieren kann. Wann immer möglich, sollten Links mit dieser Funktion generiert werden, da diese dann in TYPO3 „registriert“ sind – dies ist die Voraussetzung dafür, dass z.B. realURL aus den Links suchmaschinenfreundliche Pfade generiert oder dafür, dass bei E-Mail-Adressen der Spamschutz klappt. Wann immer ihr also versucht sein solltet, ein `` zu nutzen, – tut es nicht.

Die Funktionsweise von typolink ist im Wesentlichen sehr einfach. Typolink verlinkt den angegebenen Text je nach definierten Parametern. Ein Beispiel:

```

temp.link = TEXT
temp.link {

    # das ist der verlinkte Text
    value = Beispiellink

    # hier kommt die typolink-Funktion
    typolink {

        # wohin soll verlinkt werden?
        parameter = http://www.example.com/

        # mit welchem Target? (_blank wie hier öffnet neues Fenster)
        extTarget = _blank

        # eine zusätzliche Klasse für den Link, damit man ihn auch stylen kann
        ATagParams = class="linkclass"
    }
}

```

Das obige Beispiel generiert diesen HTML-Code: `Beispiellink`

Typolink arbeitet fast wie ein wrap - der Inhalt der z.B. über value vorgegeben wird, wird durch das öffnende und schließende a-Tag gewrapped. Wenn value leer ist, also kein Inhalt vorhanden ist, wird automatisch ein Text erzeugt. Bei einem Link auf eine Seite wird der Seitentitel verwendet, bei einer externen URL wird die URL ausgegeben.

Das Beispiel kann man jedoch abkürzen, da einem der „parameter“-Tag der typolink-Funktion schon etwas Denkarbeit abnimmt. Hier beispielhaft die kurze Variante, die haargenau den gleichen Link erzeugt, wie die obenstehende:

```
temp.link2 = TEXT
temp.link2 {
    # wieder der verlinkte Text
    value = Beispiellink

    # der Parameter mit der Zusammenfassung der oberen Parameter (Erklärung folgt unten)
    typolink.parameter = www.example.com _blank linkclass
}
```

Der „parameter“-Teil der typolink-Funktion analysiert die Eingabe auf bestimmte Zeichen und wandelt anhand der gefundenen Zeichen die jeweiligen Abschnitte um. Zunächst wird die Parameter-Folge an den Leerzeichen aufgeteilt. Findet er dann - wie im Beispiel - im ersten Abschnitt einen Punkt „.“ (ggf. vor einem Slash), generiert er einen externen Link, findet er den Punkt „.“ nach einem Slash „/“, generiert er einen Dateilink, bei einem „@“ würde er einen E-Mail-Link generieren, bei einem einfachen Integer-Wert „51“ einen Link zur Seite mit der ID 51. Durch Voranstellen des Rautezeichens „#“ erreicht man einen Link auf ein bestimmtes Content-Element (z.B. #234 für einen Link auf das Content-Element mit der ID #234 auf der aktuellen Seite, 51#234 für das Content-Element auf der Seite mit der ID 51).

Der zweite Teil des Parameters beschreibt das Target (Ziel) für den Link. Normalerweise wird dies – wie im oberen längeren Beispiel gezeigt – durch extTarget (für externe Links) oder target (für interne Links) gesetzt, kann aber über diesen zweiten Parameter überschrieben werden.

Der dritte Teil wird automatisch in ein Klassenattribut für den Link umgewandelt.

Soll jetzt aber nur das Klassenattribut aber nicht das target gesetzt werden, muss an Stelle des targets trotzdem etwas eingesetzt werden, da die Funktion sonst nicht erkennt, dass die Klasse an dritter Stelle steht. Möchte man also kein target, weil man das Standardtarget nutzen möchte, sondern nur die Klasse, sieht die Zeile so aus (mit einem Divis „-“ als Trenner):

typolink.parameter = www.example.com – linkclass

Mit Hilfe der typolink-Funktion und des target-Attributs ist es auch möglich, Links in Javascript-Popups zu öffnen:

```
temp.link = TEXT
temp.link {
    # der Linktext
    value = Popup-Fenster öffnen

    typolink {
        # 1. Parameter = PageID der Zielseite, 2. Parameter = Größe des Javascript-Popups
        parameter = 10 500x400

        # Das title-Tag des Links
        title = Hier klicken um Popup zu öffnen

        # die Parameter für das JS-Window
        JSwindow_params = menubar=0, scrollbars=0, toolbar=0, resizable=1
    }
}
```

Zu beachten ist auch, dass viele Eigenschaften von typolink vom Typ value/stdWrap sind. D.h. es können Werte berechnet werden oder aus der Datenbank gelesen werden.

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

Die Überschrift wird ausgegeben, dabei wird ein Link gesetzt auf das Ziel das im Feld header_link angegeben ist. Die erste Zeile löscht dabei die Default-Einstellung der css_styled_content.

encapsLines

EncapsLines als Abkürzung von „encapsulate lines“ oder deutsch „Zeilen einkapseln“ ist eine TypoScript-Funktion mit deren Hilfe definiert werden kann, wie einzelne Zeilen des Inhalts umschlossen werden. Also ob zum Beispiel, wenn nichts definiert wird, ein <p> oder ein <div> um das Element kommen soll, oder ob automatisiert alle Vorkommnisse von mit ersetzt werden sollen.

Ein einfaches Beispiel:

Im RTE haben wir diesen Text angegeben:

```
Ein einfacher Text ohne alles.

<div class="myclass">Ein Text mit einem DIV-Tag drum rum.</div>
```

Im TypoScript haben wir jetzt diese Definition:

```
encapsLines {
    # definiere, welche Tags als umschließende Tags gewertet werden
    encapsTagList = div,p

    # Zeilen, die nicht bereits mit Tags der encapsTagList umschlossen sind, werden mit
    # <p>-Tags umschlossen
    wrapNonWrappedLines = <p>|</p>

    # ersetze alle DIV-Tags mit P-Tags
    remapTag.DIV = P

    # falls eine Zeile leer sein sollte, gib ein kodiertes Leerzeichen aus
    innerStdWrap_all.ifEmpty = &nbsp;
}
```

Das Ergebnis sieht als HTML-Code so aus:

```
<p>Ein einfacher Text ohne alles.</p>
<p>&nbsp;</p>
<p class="myclass">Ein Text mit einem DIV-Tag drum rum.</p>
```

Bei den meisten TYPO3-Projekten wird man diese Funktion selten im eigenen Code benötigen. In der Extension "css_styled_content" werden aber mithilfe dieser Funktion einige Einstellungen gesetzt, die man ggf. an die eigenen Bedürfnisse anpassen möchte. Daher hier ein Beispiel aus der Standardkonfiguration von css_styled_content, um die Funktionsweise zu verdeutlichen:

```
lib.parseFunc_RTE {
    nonTypoTagStdWrap.encapsLines {
        # Umschließende Tags
        encapsTagList = div,p,pre,h1,h2,h3,h4,h5,h6

        # alle DIV-Tags in <p> umwandeln
        remapTag.DIV = P
    }
}
```

```

# alle noch nicht umschlossenen Zeilen mit <p> wrappen
nonWrappedTag = P

# Leerzeilen mit kodiertem Leerzeichen ersetzen
innerStdWrap_all.ifBlank = &nbsp;

# hier wird die – häufig beklagte – Klasse bodytext gesetzt
addAttributes.P.class = bodytext

# addAttributes nur einsetzen, wenn noch kein Attribut vorhanden
addAttributes.P.class.setOnly=blank
    }
}

```

Vergleicht man das untere mit dem oberen Beispiel fällt auf, dass es scheinbar zwei Parameter gibt, die das gleiche tun. Zum Einen „wrapNonWrappedLines“, zum Anderen „nonWrappedTag“. Der Unterschied liegt darin, dass „nonWrappedTag“ mittels addAttributes erweitert werden kann, während bei „wrapNonWrappedLines“ der komplette Wrapping-Tag angegeben werden muss. Wenn schon umschlossene Zeilen zum Beispiel mit <p class=“blubb“>|</p> gewrappt sind, und „wrapNonWrappedLines“ auf <p>|</p> steht, ergibt das im Ergebnis eine Mischung aus P-Tags mit und ohne Klasse, statt einem einheitlichen Bild.

Hier nochmal deutlich dargestellt: Um das häufig lästige class=“bodytext“ zu entfernen, ist daher nichts weiter nötig, als folgende Zeile.

```
lib.parseFunc RTE.nonTypoTagStdWrap.encapsLines.addAttributes.P.class >
```

filelink

Mit der Funktion „filelink“ erzeugt man – wie die deutsche Übersetzung vermuten lässt – einen Dateilink. Dabei wird nicht nur der Link zur Datei selbst erzeugt, sondern filelink bietet auch die Möglichkeit, ein Icon zur Datei und deren Größe darzustellen.

```

temp.example = TEXT
temp.example {

    # Linkbeschriftung und gleichzeitig Dateiname des Bilds
    value = mein_bild.png

    filelink {

        # Pfad zur Datei
        path = fileadmin/bilder/

        # Datei soll ein Icon bekommen
        icon = 1

        # Das Icon wird gewrappt
        icon.wrap = <span class="icon">|</span>

        # Das Icon soll auch auf die Datei verlinkt sein
        icon_link = 1

        # Statt des Symbols für den Dateityp wird die eigentliche Datei in klein
        # angezeigt, wenn sie vom Typ png oder gif ist
        icon_image_ext_list = png,gif

        # Die Größe wird auch angezeigt
        size = 1

        # Wrappt die Dateigröße (unter Beachtung der äußeren Leerzeichen)
        size.noTrimWrap = | (| Bytes) |

        # Ausgabe soll als Bytes formatiert werden
        size.bytes = 1
    }
}

```

```

# Abkürzungen für die verschiedenen Byte-Größen
size.bytes.labels = | K| M| G

# Wrap für das gesamte Element
stdWrap.wrap = <div class="filelink">|</div>
}

```

parseFunc

Die parseFunc in TYPO3 ist nicht leicht zu erklären. Zumindest nicht im Deutschen. Was zum großen Teil wohl daran liegt, dass es im Deutschen kein akkurates Wort für das Englische „parse“ gibt, selbst das sonst so hilfreiche LEO-Wörterbuch hat dazu keine anderen Vorschläge. Für unsere Erklärung hier soll das deutsche Wort „verarbeiten“ genügen. (Man könnte es auch mit "abscannen" und verarbeiten umschreiben.)

Diese Funktion verarbeitet den Großteil der Inhalte, die zum Beispiel über den Rich-Text-Editor eingegeben werden. Sie ist unter anderem dafür verantwortlich, dass so mancher Inhalt, den man in den RTE eingibt, nicht genauso wieder herauskommt. Einige Standard-Verarbeitungsregeln sind in der Extension „css_styled_content“ bereits vorhanden, ein Teil davon ist oben unter „encapsLines“ schon beschrieben. Will man so zum Beispiel ändern, wie TYPO3 bestimmte Elemente wrappt, kann man das meist mit einer parseFunc-Anweisung, will man eine Basisfunktion zum Suchen und Ersetzen kann man das mit einer parseFunc-Anweisung. Hier im Beispiel wird jedes Vorkommen von "COMP" im Frontend umgewandelt in "Mein Firmenname", so kann man zum Beispiel Abkürzungen automatisch ausschreiben oder Kleingeschriebenes automatisch großschreiben:

```

page.stdWrap.parseFunc.short {
    COMP = Mein Firmenname
}

```

Die verschiedenen Möglichkeiten das Standardverhalten anzupassen, findet man sehr leicht über den TypoScript-Object-Browser. Die verschiedenen Möglichkeiten die Verarbeitung von Eingaben weitergehend über parseFunc anzupassen findet man in der TSref unter "parseFunc". So lässt sich schnell und einfach die Ausgabe von TYPO3 an die eigenen Bedürfnisse anpassen.

tags

Die Funktion „tags“ wird in Kombination mit parseFunc genutzt, um benutzerdefinierte Tags zu definieren. In der Extension css_styled_content ist zum Beispiel der Tag <LINK> definiert, um einfach Links erzeugen zu können:

```

tags {
    # hier wird der Name des neuen Tags definiert
    link = TEXT

    # und hier die Verarbeitung des neuen Tags
    link {
        current = 1
        typolink {

            parameter.data = parameters.allParams

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

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

        }
    }

    parseFunc.constants=1
}

```

Diese Funktion ist besonders dann nützlich, wenn man eine bestimmte Art von Elementen immer wieder braucht und seinen Redakteuren den Prozess vereinfachen will (so dass sie zum Beispiel nicht immer „von Hand“ formatieren müssen, sondern nur den Tag angeben und automatisch formatiert und umgewandelt wird).

HTMLparser

Der HTML-Parser legt fest, wie Inhalte verarbeitet werden. Er wird meistens als Unterfunktion von `parseFunc` genutzt. So kann man zum Beispiel festlegen, dass alle Links absolut gesetzt werden (beispielsweise für einen Newsletter):

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

    keepNonMatchedTags=1

    # hier wird die Domain definiert, die vor den relativen Pfad gestellt wird
    tags.a.fixAttrib.href.prefixRelPathWith=http://www.example.com/

    # für alle Links ohne definiertes target wird hier das target auf _blank gesetzt
    tags.a.fixAttrib.target.default=_blank
}
```

Die Funktion `HTMLparser` ist extrem mächtig, da damit jeglicher Inhalt vor der Ausgabe angepasst und verändert wird. So können auch eigene Tags definiert werden - intern werden Links z.B. in der Form `<link http://www.typo3.org/>Linktext</link>` gespeichert. D.h. es wird dafür ein eigenes Tag verwendet. Dieses Tag kann wiederum in allen Feldern - auch Überschriften - definiert werden, bei denen ein entsprechender Parser definiert ist.

Das folgende Beispiel erlaubt das `<u>`-Tag in den Überschriften. Hier wird die Standard Definition aus `css_styled_content` angepasst. Die Funktion `htmlSpecialChars` wird deaktiviert, damit das `<u>` erhalten bleibt. Dann wird die Funktion `parseFunc` verwendet und definiert, dass außer dem Tag "u" keine Tags zugelassen sind. Es werden also alle Tags außer dem `<u>`-Tag entfernt.

```
# in der Überschrift soll ein <u>-Tag zugelassen werden,
# ansonsten aber alle Elemente wie gewohnt geparkt werden.
lib.stdheader.10.setCurrent.htmlSpecialChars = 0
lib.stdheader.10.setCurrent.parseFunc {
    allowTags = u
    denyTags = *
    constants=1
    nonTypoTagStdWrap.HTMLparser = 1
    nonTypoTagStdWrap.HTMLparser {
        keepNonMatchedTags=1
        htmlSpecialChars = 2
        allowTags = u
        removeTags = *
    }
}
```

Bei diesem Beispiel wird wieder deutlich, wie wichtig die Funktion `stdWrap` ist. Die Funktion `setCurrent` ist vom Typ `string/stdWrap` und ermöglicht daher dass die Funktion `parseFunc` überhaupt angewendet werden kann.

stdWrap richtig nutzen

Die Funktion `stdWrap` hält eine große Anzahl unterschiedlicher Funktionen und Parameter bereit. Einige sind trivial, der Nutzen anderer ist dann doch schwieriger zu ergründen. An dieser Stelle soll nochmal auf das Grundprinzip eingegangen werden und ein paar besondere Funktionen/Eigenschaften hervorgehoben werden.

Die `stdWrap`-Eigenschaft kann nur verwendet werden, wenn es explizit definiert ist. Wenn eine Eigenschaft vom Typ "wrap" ist, dann sind keine `stdWrap`-Eigenschaften vorhanden. In der Regel wird entweder eine Eigenschaft `stdWrap` vom Typ `stdWrap` angeboten, oder eine Eigenschaft bietet z.B. "string/stdWrap" an.

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

Das Objekt `Image` hat eine Eigenschaft `stdWrap` vom Typ `stdWrap`.

```
10 = HTML
10.value = Hallo Welt
10.value.typolink ...
```

Das Objekt `HTML` dagegen hat eine Eigenschaft `value` vom Typ `string/stdWrap`. Es kann also ein String zugewiesen werden und zusätzlich können `stdWrap` Eigenschaften verwendet werden.

Reihenfolge beachten!

Eine wichtige Einschränkung sollte hier aber hervorgehoben werden: *Die einzelnen Funktionen werden in der Reihenfolge, in der sie in der Referenz angegeben sind, ausgeführt.* Wenn das nicht beachtet wird, kann es passieren, dass einige Ergebnisse anders aussehen als gedacht.

```
10 = TEXT
10.value = Hallo Welt
10.case = upper
10.field = header # nehmen wir an, header enthält "typo3" (kleingeschrieben!)
10.stdWrap.wrap = <strong>|</strong>

# gibt folgendes aus:
<STRONG>TYP03</STRONG>
```

In diesem Beispiel passiert Folgendes: Zuerst wird der Inhalt des Text-Objekts mit "Hallo Welt" gefüllt. Da die TypoScript-Konfiguration in einem Array gespeichert wird, bei dem die Reihenfolge der Definition nicht erhalten bleiben kann, werden die Funktionen nach einer in `stdWrap` definierten Reihenfolge ausgeführt. Diese Reihenfolge spiegelt sich in der Referenz entsprechend wieder. Nach einem kurzen Blick in die Referenz sollte offensichtlich sein, dass zuerst "field", danach "stdWrap" (und damit "stdWrap.wrap") und erst zum Schluss "case" ausgeführt wird.

stdWrap rekursiv nutzen

Da allerdings die `stdWrap` Funktion wiederum rekursiv aufgerufen werden kann, ist es möglich die Reihenfolge damit zu ändern.

Die Funktion "prioriCalc" ermöglicht es einfache mathematische Ausdrücke zu berechnen. Wenn Sie auf 1 gesetzt ist, wird der Inhalt berechnet, wobei allerdings eine einfache Auswertung von links nach rechts stattfindet. Der folgende Code sieht aus, als würde er den Inhalt des Feldes "width" um 20 erhöhen.

```
10 = TEXT
10.field = width # Annahme: "width" ist 100
10.wrap = |+20
```



```
10.prioriCalc = 1
```

Allerdings ist dieses nicht der Fall das Ergebnis, das ausgegeben wird lautet "100+20". Die Funktion "prioriCalc" wird vor der Funktion "wrap" aufgerufen und berechnet daher nur das Ergebnis von "field", also den Ausdruck "100". Damit das richtige Ergebnis "120" ausgegeben wird, muss sichergestellt werden, dass "field" und "wrap" vor "prioriCalc" ausgeführt werden. Das wird mit dem folgenden Ausdruck erreicht:

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

Die stdWrap-Funktion selber wird nach "field" aber vor "prioriCalc" ausgeführt, daher wird "100+20" gewrapped und erst danach wird die Funktion "prioriCalc" ausgeführt und somit der Wert "120" berechnet.

Der Datentyp

Enorm wichtig bei Arbeiten mit TypoScript ist es, den Datentyp der Eigenschaft, die verwendet werden soll, zu berücksichtigen. Insbesondere bei der stdWrap Eigenschaft fällt in der Praxis immer wieder auf, dass die Funktionen irgendwie kombiniert werden bis schließlich per Zufall das Ziel erreicht wird.

Nur wenn explizit die stdWrap-Funktionalität angegeben ist, können stdWrap-Funktionen wie field, data oder typolink auch verwendet werden.

lang: Mehrsprachigkeit

stdWrap stellt eine Eigenschaft "lang" zur Verfügung, mit der es möglich ist, einfache Texte, die in einer Seite per TypoScript eingebunden sind, mehrsprachig zu übersetzen.

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

Allerdings sind solche Texte durch externe Redakteure nur schwer zu übersetzen - gerade bei unbekannten Sprachen kann das schnell zu einer Herausforderung werden. In einem solchen Fall bietet es sich an, die Übersetzung bei Konstanten vorzunehmen. Diese können zentral an einer Stelle gepflegt werden und werden dann in das TypoScript eingefügt.

```
# Constants
text.impressum = Impressum
text.en.impressum = Imprint

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

Dadurch wird die Übersetzung vom eigentlichen TypoScript getrennt.

cObject

Der Parameter cObject kann dafür verwendet werden, den Inhalt durch ein TypoScript-Objekt zu ersetzen. Das kann ein COA, ein Plugin oder ein Text wie in diesem Beispiel sein.

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

Ausblick

Punkte, die wir in den nächsten Versionen berücksichtigen möchten:

- Der Bereich Menüs könnte noch deutlich ausgebaut werden
- Das Zusammenspiel mit TemplaVoila sollte angerissen/erläutert werden (insbesondere FCEs)
- TypoScript Debuggen erläutern
- weitere stdWrap Funktionen erläutern
- erläutern: Datensatz wird geöffnet, Formulare werden gerendert, das wird via TCA und Seiten-TS gesteuert, dann wird gespeichert, die Daten landen in der DB (Sonderweg RTE auch noch erläutern?). Dann wird die Seite im Frontend aufgerufen und die Daten für die Ausgabe werden über TypoScript geholt und HTML wird erzeugt.
- Abkürzungen, am Besten in einem Glossar zum Ausdrucken erklären.

Für weitere Anregungen, Vorschläge, Kritik oder Fragen sind wir natürlich offen. Einfach auf der Wiki-Seite <http://wiki.typo3.org/index.php/De:TSref/45MinutesTypoScript> oder in forge (<http://forge.typo3.org/projects/show/team-docteam>) melden.