



Entwicklerhandbuch

XML Hands On

XSLT - XQuery - MarkLogic



(c) Alex Düsel 2019

Creative Commons Namensnennung-Keine
Bearbeitungen 4.0 International Public License
www.github.com/alexdd/Buch





Dieses Buch wurde mit Tektur CCMS erstellt. Tektur ist ein einfach zu bedienender kollaborativer Editor um **DITA** ¹⁾ Inhalte erstellen, als PDF ausgeben und pflegen zu können. Die Eingabe erfolgt dabei per **WYSIWYG** ²⁾ mit geführter Benutzerinteraktion. Die Inhalte werden als einzelne Topics verwaltet, die in verschiedenen Maps referenziert werden können; Stichwort: **Topic Based Authoring** ³⁾.

Sonstige Features: Rechte- und Rollensystem, Versionskontrolle, konfigurierbarer Workflow mit Review & Approval Funktionen. Auf dem Entwicklerblog ⁴⁾ kann man sich über den Fortschritt informieren.

► *Dieses Buch ist **WORK IN PROGRESS** und dient in erster Linie als Test für Tektur CCMS. Der Feinschliff kommt noch! Momentan schreibe ich alles was mir irgendwie interessant erscheint - je nach zeitlicher Möglichkeit mehr oder weniger umfangreich - auf und binde die Topics in eine Map ein, um ein PDF erzeugen zu können.*

Wie schliesslich Struktur und Inhalt genau aussehen sollen, werde ich mir zu einem späteren Zeitpunkt noch genau überlegen :-]

- 1) https://de.wikipedia.org/wiki/Darwin_Information_Typing_Architecture
- 2) <https://de.wikipedia.org/wiki/WYSIWYG>
- 3) https://en.wikipedia.org/wiki/Topic-based_authoring
- 4) <http://www.tekturcms.de>

Inhalt

1 Anwendungsgebiete	7
1.1 Führende Technologie XSLT	8
1.2 Einige aktuelle und vergangene Szenarien	9
1.3 Professionelle XML Verarbeitung	14
2 Wichtige Konzepte	17
2.1 Push vs. Pull Stylesheets	17
2.2 Eindeutigkeit der Regelbasis	19
2.3 Namespaces	22
3 Ausgewählte Themen	25
3.1 XSLT Uses Cases und Lösung	25
3.1.1 Vortransformationen	25
3.1.2 Vererbung	27
3.1.3 XSLT Streaming	29
3.1.3.1 XSLT Akkumulator	29
3.1.3.2 XSLT Iterator	31
3.1.4 Identifikation mit generate-id()	34
3.1.5 Reguläre Ausdrücke	36
3.2 XQuery Use Cases und Lösung	37
3.2.1 Abfragen mit XQuery	37
3.2.2 Connector zu Marklogic in Oxygen	40
3.2.3 XQuery als Programmiersprache	43
3.2.3.1 Funktionen und Module	45
3.2.3.2 If..then..else Ausdrücke	46
3.2.4 SQL Views in MarkLogic	47
3.3 Testing	53
3.3.1 Validierung mit Schematron	53
3.3.2 Erste Schritte mit Xspec	57
4 Zusätzliches Know-How	59
4.1 GIT Einmaleins	59
4.2 XML Editoren	60
Glossar	63
Abbildungen	65
Literaturverzeichnis	67
Stichwortregister	71

1 Anwendungsgebiete

XML, XSLT, XPATH, XSL-FO und XQuery sind Techniken um baumstrukturierte Daten - im Vergleich zu relationalen Daten - aus verschiedenen Quellen ineinander zu überführen, abzuspeichern, zu versenden, darzustellen und auszuwerten. Einfach gesprochen geht es um die Weiterverarbeitung solcher Daten - das XML kann dabei sowohl die Nutzdaten als auch die Steuerdaten tragen.

Vom Aussehen her sind XML Daten im Prinzip Textdaten. Sie können sehr einfach mit einem Texteditor erstellt werden. Im Gegensatz zu Multimedia-Daten sind keine komplexen Tools, wie z.B. ein Grafikeditor, erforderlich.

Auch relationale Daten können in Form von Tabellen, als Excel Tabelle oder bspw. als komma-separierte Textdatei, aus einem System ausgespult und weiterverarbeitet werden. XML erlaubt es jedoch die Daten semantisch auszuzeichnen. Das geschieht durch das Klammern semantisch zusammengehöriger Elemente mittels Klammer-Tags und weiterer Kategorisierung dieser Informationseinheiten mittels weiterer Properties (Attribute) an diesen Tags. Durch das Verschachteln dieser geklammerten Komponenten entsteht ein Baum, der die Hierarchische Ordnung der Daten widerspiegeln sollte.

Diese Baumstrukturen sind maschinell lesbar und die Daten können, bevor sie von einem Versender zu einem Empfänger gehen, mittels eines automatischen Prozesses validiert werden. Dabei können sowohl der Inhalt als auch die Syntax anhand von definierten Regeln (Schemas) genau überprüft werden.

Der XML Standard ist mittlerweile 20 Jahre alt. Zuvor gab es SGML, das zum Beispiel auch nicht abgeschlossene Tags erlaubt.

Der Übergang von SGML zu XML hat die Sache ein bisschen vereinfacht, eine weitere Vereinfachung brachte JSON als Standard. JSON wird gerne im Webbereich eingesetzt um baumstrukturierte Daten zu verarbeiten. JSON ist jedoch nicht so gut maschinenlesbar und es gibt noch nicht so viele Werkzeuge wie z.B. Code Editoren dafür.

Folgend eine kurze Erläuterung zu den eingangs erwähnten Schlüsselwörtern, um die es sich im weiteren Text drehen wird:

- **XML** ist das Datenformat. Auf XML arbeiten die anderen Technologien. XML ist immer Input für diese Tools.
- **XSLT** transformiert eine XML Instanz in eine andere. Plain Text ist eine Aneinanderreihung von Text Knoten
- **XPATH** erlaubt es, bestimmte Knoten in einem XML Dokument über bedingte Pfadausdrücke zu selektieren.
- **XSL-FO** ist eine weitere XML basierte Auszeichnungssprache, die ein XSL-FO Prozessor einlesen kann, um daraus z.B. ein PDF zu generieren.
- **XQuery** ist eine Abfragesprache ähnlich zu SQL, jedoch werden damit nicht relationale Daten abgefragt sondern baumstrukturierte.

1.1 Führende Technologie XSLT

Ausserhalb der XML Verarbeitung hat XSLT keinen Stellenwert. Im **TIOBE Index** ⁵⁾ von 2003 rangierte XSLT einmal auf Platz 60 an letzter Stelle der Liste ⁶⁾.

Im Bereich XML würde aber ohne XSLT nicht viel gehen. Es gibt einige exotische Anwendungsgebiete in denen XML effizient mit **LISP** Dialekten verarbeitet wird, bspw. die Verarbeitung von, nach XML konvertierten, **EDI X12** ⁷⁾ Nachrichten.

Auch im Bereich Publishing in der Luftfahrt hat sich SGML als Vorreiter von XML im **S1000D** Standard ⁸⁾ wacker gehalten. Hier wird teilweise noch mit proprietäre Programmiersprachen, wie Metamorphosis gearbeitet.

Alternativen zu XSLT finden sich im entsprechendem Wikipedia-Artikel ⁹⁾.

Wir konzentrieren uns hier also erst einmal auf XSLT und insbesondere XSL Stylesheets.

Beispielsweise werden die Autohandbücher führender Hersteller mittels XSL gesetzt, deren Eingabedaten aufbereitet und zur Weiterverarbeitung transformiert.

- XSLT hat gerade noch den Status "Programmiersprache", weil man damit eine Turing Maschine ¹⁰⁾ programmieren kann.
- Mit HTML oder einer Templater Sprache (z.B. JSP) würde das nicht funktionieren.
- XSLT benötigt normalerweise immer eine XML Eingabe. Zumindest ein XML Knoten muss verarbeitet werden. Es gibt aber auch den Spezialfall, dass bspw. mit dem XSLT Prozessor Saxon keine Eingabe notwendig ist, bzw. diese vom XSLT Stylesheet selbst erzeugt wird.
- XSLT ist keine imperative Sprache, d.h es werden keine Anweisungen der Reihe nach abgearbeitet, sondern eine deklarative Sprache, d.h für jedes Ereignis (besser gesagt: für jeden durchlaufenen DOM Knoten) wird eine gefundene - und vom Programmierer deklarierte - Regel angewendet.
- Ausserdem gibt es funktionale Anteile, um bspw. die deklarierten Regeln rekursiv anwenden zu können.
- XSLT wird oft mit XSL gleichgesetzt. Aber XSL ¹¹⁾ ist mehr:
 - Zum einen kommt noch XPATH hinzu: XPATH erlaubt komplizierte Berechnungen und Selektionen auf den DOM Knoten eines XML Dokuments.
 - Zum anderen ist auch XSL-FO ¹²⁾ Bestandteil der XSL Spezifikation. XSL-FO Tags sind Anweisungen für einen XSL-FO Prozessor, der aus einem XSL-FO Dokument ein PDF Dokument generiert. Es sind auch andere Ausgabe-Formate, wie bspw. RTF möglich.

5) <https://de.wikipedia.org/wiki/TIOBE-Index>

6) <https://bit.ly/2ARgKCJ>

7) https://en.wikipedia.org/wiki/ASC_X12

8) <https://en.wikipedia.org/wiki/S1000D>

9) https://de.wikipedia.org/wiki/XSL_Transformation

10) <http://www.unidex.com/turing/utm.htm>

11) W3C Seiten zu The Extensible Stylesheet Language Family (XSL)

12) <https://www.w3.org/TR/xsl/>

1.2 Einige aktuelle und vergangene Szenarien

Einige Beispiele - aktuell und aus vergangenen Tagen:

XML Webseiten

Einen XSLT Prozessor hat jeder Browser eingebaut. Es gab mal eine Zeit, in der es sehr populär war, Webseiten vom Server als XML auszuliefern. XML erlaubt die semantische Auszeichnung des Inhalts, und die strikte Trennung des Inhalts von Layout und Design. Wesentlich besser als dies mit HTML und CSS jemals möglich wäre. Ich denke, u.a. wegen des exzessiven Einsatzes von Javascript (auch inline), hat sich diese Idee nie vollständig durchgesetzt. Schliesslich wurde XHTML spezifiziert und jetzt gibt es HTML5.

Betrachten wir das folgende einfache XML Beispiel:

```
<?xml version="1.0" encoding="UTF-8"?>
<document>
<title>Das ultimative Zwei-Kapitel Dokument</title>
  <chapter>
    <title>Kapitel 1</title>
    <intro>In Kapitel 1 wird kurz gesagt was Sache ist.</intro>
    <content>Um es kurz zu machen, wie der Hase läuft steht in Kapitel 2.</content>
  </chapter>
  <chapter>
    <title>Kapitel 2</title>
    <intro>Hier wird erklärt, wie der Hase läuft.</intro>
    <content>Im Prinzip ist es ganz einfach.</content>
  </chapter>
</document>
```

Ohne XSLT Stylesheet Zuweisung wird der Browser eine Datei mit diesem Inhalt als eingerücktes XML anzeigen - oder die Tags einfach ignorieren und den Textinhalt in einer Zeile darstellen. Fügt man eine Processing Instruction¹³⁾ am Anfang ein, wird ein XSLT Stylesheet vom Browser herangezogen und vor der Darstellung im Browser wird die so deklarierte XML Transformation ausgeführt:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="formatiermich.xsl" ?>
<document>
  <title>Das ultimative Zwei-Kapitel Dokument</title>
  <chapter>
[...]
```

Jetzt kann man das XML einfach im Browser öffnen und alles wird schön formatiert - je nachdem welche Regeln in *formatiermich.xsl* gesetzt sind - angezeigt:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <html>
      <xsl:apply-templates/>
    </html>
  </xsl:template>

  <xsl:template match="document">
```

13) <https://de.wikipedia.org/wiki/Verarbeitungsanweisung>

```

<body>
  <xsl:apply-templates/>
</body>
</xsl:template>

<xsl:template match="document/title">
  <h1>
    <xsl:apply-templates/>
  </h1>
</xsl:template>

<xsl:template match="chapter">
  <div class="chapter">
    <xsl:apply-templates/>
  </div>
</xsl:template>

<xsl:template match="chapter/title">
  <h2>
    <xsl:apply-templates/>
  </h2>
</xsl:template>

<xsl:template match="chapter/intro">
  <div class="intro">
    <i><xsl:apply-templates/></i>
  </div>
</xsl:template>

<xsl:template match="chapter/content">
  <p><xsl:apply-templates/></p>
</xsl:template>
</xsl:stylesheet>

```

Die Processing Instruction hat keinen Einfluss auf den XML Inhalt und wird in einer anderen Eingabeverarbeitung nicht herangezogen.

Serverseitige Konvertierung

Auch eine serverseitige Konvertierung ist gebräuchlich. Ein Beispiel aus vergangenen Tagen - WAP-Seiten¹⁴⁾ für unterschiedliche Handy-Modelle.

Damals hatten die Handys sehr unterschiedliche Displaygrößen. Handybrowser konnten noch nicht ausreichend Javascript und die Skalierung der WAP-Seite für das jeweilige Handy passierte nicht im Handy, sondern vor der Auslieferung auf der Serverseite. Dazu wurde eine XML Quelle mittels verschiedener XSLT Stylesheets in unterschiedliche WML WAP Repräsentationen transformiert.

So würde das Zwei-Kapitel Beispiel von oben im WML Format aussehen (recht einfach gehalten):

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN" "http://www.wap.org/DTD/wml_1.1.xml">
<wml>
  <head>
    <meta name="title" content="Das ultimative Zwei-Kapitel Dokument"/>
  </head>
  <card id="chapter1" title="Kapitel 1">
    <p><i>In Kapitel 1 wird kurz gesagt was Sache ist.</i></p>
    <p>Um es kurz zu machen, wie der Hase läuft steht in Kapitel 2.</p>
  </card>

```

14) https://de.wikipedia.org/wiki/Wireless_Application_Protocol

```
<card id="chapter2" title="Kapitel 2">
  <p><i>Hier wird erklärt, wie der Hase läuft.</i></p>
  <p>Im Prinzip ist es ganz einfach.</p>
</card>
</wml>
```

Eine XSLT Transformation, die die XML Daten von oben in diese WML Darstellung überführt, könnte z.B. so implementiert werden:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

  <xsl:output
    doctype-public="-//WAPFORUM//DTD WML 1.2//EN"
    doctype-system="http://www.wapforum.org/DTD/wml12.dtd"
    indent="yes"/>

  <xsl:template match="document">
    <wml>
      <xsl:apply-templates/>
    </wml>
  </xsl:template>

  <xsl:template match="document/title">
    <head>
      <meta name="title">
        <xsl:attribute name="content">
          <xsl:value-of select="."/>
        </xsl:attribute>
      </meta>
    </head>
  </xsl:template>

  <xsl:template match="chapter">
    <card id="{concat('chapter',count(preceding-sibling::chapter)+1)}">
      <xsl:attribute name="title">
        <xsl:value-of select="title"/>
      </xsl:attribute>
      <xsl:apply-templates select="*[not(self::title)]"/>
    </card>
  </xsl:template>

  <xsl:template match="node()|@*">
    <xsl:copy>
      <xsl:apply-templates select="node()|@*"/>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="processing-instruction()"/>

  <xsl:template match="intro">
    <p><i><xsl:apply-templates/></i></p>
  </xsl:template>

  <xsl:template match="content">
    <p><xsl:apply-templates/></p>
  </xsl:template>
</xsl:stylesheet>
```

Ausgabeformate

Aus einer XML Quelle können auch leicht weitere Format erzeugt werden, bspw. ePub. EPUB¹⁵⁾ ist das Standardformat für eBooks und neben Tags zur Formatierung für den Content, gibt es bspw. auch Anweisungen zum Erzeugen des Inhaltsverzeichnisses oder anderer Navigationsstrukturen.

Weitere gängige Formate sind neben dem oben gezeigten veralteten WML Format, elektronische Ausgabe-Formate wie: CHM¹⁶⁾, EclipseHelp¹⁷⁾, JavaHelp¹⁸⁾, ..., Print-Ausgabe Formate, wie PDF oder Adobe Framemaker¹⁹⁾, oder XML Standard Austauschformate, wie DITA, S1000D, PI-MOD²⁰⁾, JATS²¹⁾ oder TEI²²⁾.

Menschenlesbar machen

Kryptische XML Log-, Daten- oder Konfigurationsfiles können leicht mit XSLT menschenlesbar formatiert werden.

Ein Arbeitskollege im neuen Job kam kürzlich auf mich zu, ob ich um eine Möglichkeit wüsste, wie man sein kryptisches Datenfile für einen Übersetzungsdienst menschenlesbar formatieren könnte - XSLT to the Rescue:

```
<?xml version="1.0" encoding="UTF-8"?><?xml-stylesheet type="text/xsl" href="de.xsl"?>
<jcr:root xmlns:sling="http://sling.apache.org/jcr/sling/1.0"
  xmlns:jcr="http://www.jcp.org/jcr/1.0"
  xmlns:mix="http://www.jcp.org/jcr/mix/1.0"
  xmlns:nt="http://www.jcp.org/jcr/nt/1.0"
  jcr:language="de"
  jcr:mixinTypes="[mix:language]"
  jcr:primaryType="sling:Folder">
  <b_manual
    jcr:primaryType="sling:MessageEntry"
    sling:message="Bedienungsanleitung"/>
    <b_warning
      jcr:primaryType="sling:MessageEntry"
      sling:message="Warnung"/>
    <b_danger
      jcr:primaryType="sling:MessageEntry"
      sling:message="Vorsicht"/>
    <b_note
      jcr:primaryType="sling:MessageEntry"
      sling:message="Notiz"/>
    <b_notice
      jcr:primaryType="sling:MessageEntry"
      sling:message="Hinweis"/>
  [...]
```

Mit einem eingehängten XSLT Stylesheet de.xsl wird so ein XML Datenfile schön formatiert als Tabelle angezeigt:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
```

- 15) <https://de.wikipedia.org/wiki/EPUB>
- 16) [https://de.wikipedia.org/wiki/CHM_\(Dateiformat\)](https://de.wikipedia.org/wiki/CHM_(Dateiformat))
- 17) <https://www.ibm.com/developerworks/library/os-echelp/index.html>
- 18) <https://en.wikipedia.org/wiki/JavaHelp>
- 19) <https://de.wikipedia.org/wiki/FrameMaker>
- 20) <https://www.i4icm.de/forschungstransfer/pi-mod/>
- 21) https://de.wikipedia.org/wiki/Journal_Article_Tag_Suite
- 22) https://de.wikipedia.org/wiki/Text_Encoding_Initiative

```

xmlns:jcr="http://www.jcp.org/jcr/1.0"
xmlns:sling="http://sling.apache.org/jcr/sling/1.0">

<xsl:template match="jcr:root">
  <html>
    <table border="1" cellpadding="5" cellspacing="5">
      <xsl:apply-templates/>
    </table>
  </html>
</xsl:template>

<xsl:template match="*">
  <tr>
    <td>
      <xsl:value-of select="concat(count(preceding::*[@sling:message]) + 1, '. ')" />
    </td>
    <td>
      <xsl:value-of select="name()" />
    </td>
    <td contenteditable="true">
      <xsl:value-of select="@sling:message" />
    </td>
  </tr>
</xsl:template>
</xsl:stylesheet>

```

Hängt man an dieses Beispiel noch ein bisschen Javascript Logik und macht die Felder für die Übersetzungen mittels des HTML5 contenteditable Attributs editierbar, dann bräuchte man nur noch eine Rücktransformation HTML nach XML und hätte schon einen kleinen webbasierten XML Editor gebaut. Genau nach diesem Prinzip funktionieren einige aktuelle XML Editoren.

Diagramme

Nachdem eine SVG Grafik im XML Format vorliegt kann diese auch direkt aus XML Daten mittels XSLT erzeugt werden.

Über das HTML5 <svg> Element kann so eine erzeugte Grafik inline in das ebenfalls durch das XSLT generierte HTML Dokument eingebunden werden.

Betrachten wir unser Beispiel von oben, erweitert um drei neue <block> Elemente:

```

<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="chart.xsl" ?>
<document>
  <title>Das ultimative Zwei-Kapitel Dokument</title>
  <chapter>
    <title>Kapitel 1</title>
    <intro>In Kapitel 1 wird kurz gesagt was Sache ist.</intro>
    <content>Um es kurz zu machen, wie der Hase läuft steht in Kapitel 2.</content>
  </chapter>
  <chapter>
    <title>Kapitel 2</title>
    <intro>Hier wird erklärt, wie der Hase läuft.</intro>
    <content>Im Prinzip ist es ganz einfach. Betrachten wir doch drei gelbe Blöcke:
    </content>
    <block/>
    <block/>
    <block/>
  </chapter>
</document>

```

Wenn wir das XSLT Stylesheet von oben noch um eine Regel für das neue <block> Element ergänzen, so wie hier:

```
<xsl:template match="block">
  <svg style="background-color:yellow" width="30" height="30"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    xmlns="http://www.w3.org/2000/svg"/>
  <br/>
  <br/>
</xsl:template>
```

Dann erhalten wir drei schön formatierte gelbe SVG Blöcke.

Weiterführende Links:

- Client-side image generation with SVG and XSLT²³⁾
- Knotentyp Visualisierung im Apache Jack Rabbit Projekt²⁴⁾

1.3 Professionelle XML Verarbeitung

Vom Single-Source Publishing bis zur Generierung von Java Code aus Klassendiagrammen.

Single Source Publishing

Gängige Formate in der Technischen Dokumentation sind elektronische Ausgabe-Formate wie: CHM, EclipseHelp, JavaHelp, ePub, ..., Print-Ausgabe Formate, wie PDF oder Adobe Framemaker, oder XML Standard Austauschformate, wie DITA, S1000D, PI-MOD oder TEI. Vorteile:

- Bei einer Änderung in der XML Quelle werden auch automatisch alle anschließenden Formate aktualisiert.
- Strikte Trennung von Content (/ Semantik) und Layout/Design.
- Auf der XML Quelle sind XML Features möglich, wie: Modularisierung: Erlaubt die feingranulare Wiederverwendung von Content-Bausteinen, sowie das Verlinken, Filtern, Suchen und Exportieren derselben. Generalisierung ist ein DITA Konzept, welches die Wiederverwendung von angepassten Topics in anderen DITA Systemen ermöglicht. Gültigkeiten erlauben die bedingte Anwendung von Content-Bestandteilen auf Satz und Wort-Ebene. Versionierung und Diffing - Vergleich von Änderungen zwischen Versionen. Intelligente Querverweise: Ein Link zwischen einzelnen XML Topics bleibt versions-treu. Automatischer Satz, inkl. Zusammenhalte- und Trennregeln für Seiten, Absätze und Blöcke (Listen, Tabellen, etc).
- Veraltete Formate können ausgetauscht werden, ohne dass der Content geändert werden muss oder verlorengeht.
- Die XML Quelle kann ohne Aufbereitung in anderen Systemen wiederverwendet werden.
- Es gibt weit verbreitete Standards zur Struktur der XML Quelle.
- Nur das XML muss in der Datenhaltung persistiert werden

23) <http://surguy.net/articles/client-side-svg.xml>

24) <http://jackrabbit.apache.org/jcr/node-type-visualization.html>

- Es gibt spezialisierte XML Datenbanken, die besonders gut auf Baumstrukturen arbeiten. (Dokumente sind per se baum-strukturiert und sind eigentlich für eine relationale Datenbank ungeeignet)

Code Generierung

Die Redaktionssysteme der Technischen Dokumentation der führenden Hersteller in Deutschland haben XML unter der Haube und setzen auf die Single-Source Strategie.

Nachdem man bei XSLT im Format der Ausgabe frei ist, kann auch direkt Plain-Text mit XSLT Regeln generiert werden. Daher liegt es nahe sich jegliche Form von Quelltext aus einer XML Repräsentation erzeugen zu lassen.

Beispielsweise speichern gängige CASE Tools (Computer Aided Software Engineering) UML Diagramme im XML Format ab, so z.B. **ArgoUML** ²⁵⁾.

Diese Klassendiagramme lassen sich mittels XSLT direkt in Java-Code transformieren, wie z.B. in einem kleinen Open Source Projekt (aus vergangenen Tagen) : **Butterfly Code Generator** ²⁶⁾

Es gibt aber auch einen schönen Artikel dazu im Java World Journal²⁷⁾.

Migrationen und Konvertierungen

Für jede erdenkliche Art der Migration eines XML Datenbestands oder eines Datenbank-Dumps / -Exports im XML Format, zwischen Produktversionen oder zwischen Dienstleister- und Dienstnutzer-Systemen bietet sich XSLT als Mittel der Wahl zur schnellen und komplexen Transformation an.

Dabei ist zu beachten, dass XSLT besonders schnell und gut auf verschachtelten Strukturen arbeitet. Entartet ein Baum zur Liste und/oder sind nur geringe Strukturanpassungen notwendig, wird man sich mit einem schnellen SAX Parser leichter tun. Mittels der XSLT3.0 Streaming Option können auch sehr große XML Quellen (Big Data) verarbeitet werden. Saxon bietet bspw. diese Streaming Option²⁸⁾.

25) <http://argouml.tigris.org>

26) <http://butterflycode.sourceforge.net>

27) <https://www.javaworld.com/article/2073998/java-web-development/generate-javabean-classes-dynamically-with-xslt.html>

28) <http://www.saxonica.com/html/documentation/sourcedocs/streaming/>

2 Wichtige Konzepte

XSLT und XQuery erlauben es Probleme auf viele verschiedene Arten zu lösen. Sicherlich wird jeder Programmierer im Laufe der Zeit seinen eigenen Stil entwickeln. Das kommt nicht zuletzt daher, dass man als XSLT Entwickler in vielen Firmen eine Expertenrolle einnimmt.

Umso wichtiger ist es, sich an allgemeine Konzepte, Muster und Best Practices zu halten, um einen schwer wartbaren Wildwuchs zu vermeiden.

Auf den folgenden Seiten wird versucht einige dieser Konzepte zusammenzutragen und mit eigenen Erfahrungen und Ideen zu kombinieren.

Es wird weder der Anspruch auf Vollständigkeit noch auf Korrektheit dieser Informationen erhoben. Das Kapitel soll vielmehr als Denkanstoß mit hoffentlich einigen verwertbaren Ideen dienen.

2.1 Push vs. Pull Stylesheets

XSLT ist eine ereignisgesteuerte, regelbasierte Umgebung zur Konvertierung von XML Daten. Gerade der Vorteil des regelbasierten Ansatzes ist vielen Entwicklern nicht bewusst, und es entsteht Quellcode der aussieht, wie mit XPath angereicherter PHP Code.

Ich frage mich an dieser Stelle immer, wieso nimmt man dann überhaupt XSLT, wenn man keine Template-Match Regeln verwendet, oder nur spärlich verwendet?

Um diesen Umstand aufzuklären ist ein bisschen Theorie notwendig:

Beim "Pull" werden Elemente in der Quellinstanz selektiert und an einer passenden Stelle in der Zielinstanz eingefügt. Diese Vorgehensweise ist vergleichbar mit derer von Template-Engines, wie JSP oder ASP. Das kann in mehreren Stufen erfolgen, bis schrittweise die Quellinstanz in die finale Zielinstanz überführt wurde.

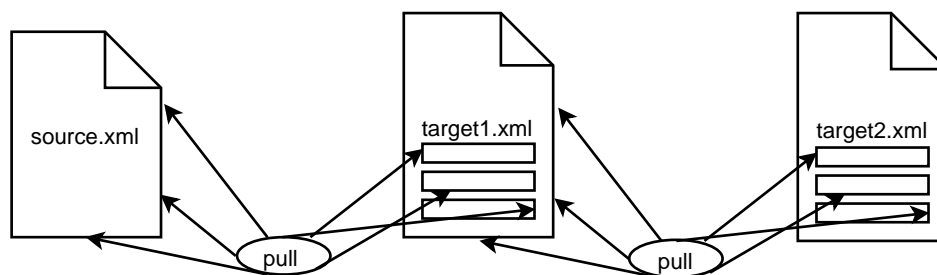


Bild 1: Pull Stylesheet

Beim "Push" werden die Quelldaten schrittweise in die Zieldaten konvertiert. Diese Vorgehensweise kann explorativ erfolgen und beim Transformieren in einen Zwischenschritt entstehen Erkenntnisse, die bei der Weiterverarbeitung nützlich sind. Merke: XSLT steht für eXtensible Stylesheet Transformation.

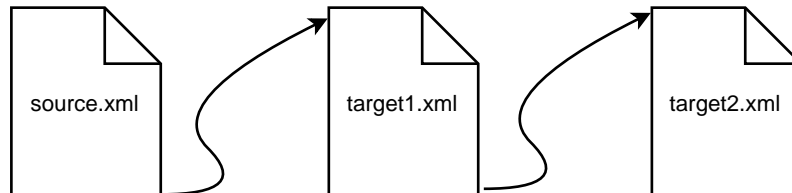


Bild 2: Push Stylesheet

Das bisher Gesagte verdeutlicht zwar den "Pull" Ansatz, was genau aber ge"pusht" wird, ist vermutlich noch unklar. Betrachten wir XML in der Baumdarstellung.

Der XSLT Prozessor unternimmt einen Tiefsuchlauf und überprüft bei jedem Knoten den er betritt, ob in seiner Regelbasis eine Regel existiert, die auf diesen Knoten "matched". Dabei gibt es drei grundsätzliche Möglichkeiten, wie die Knoten des Quellbaums in den Zielbaum kopiert - oder eben nicht kopiert - werden können.

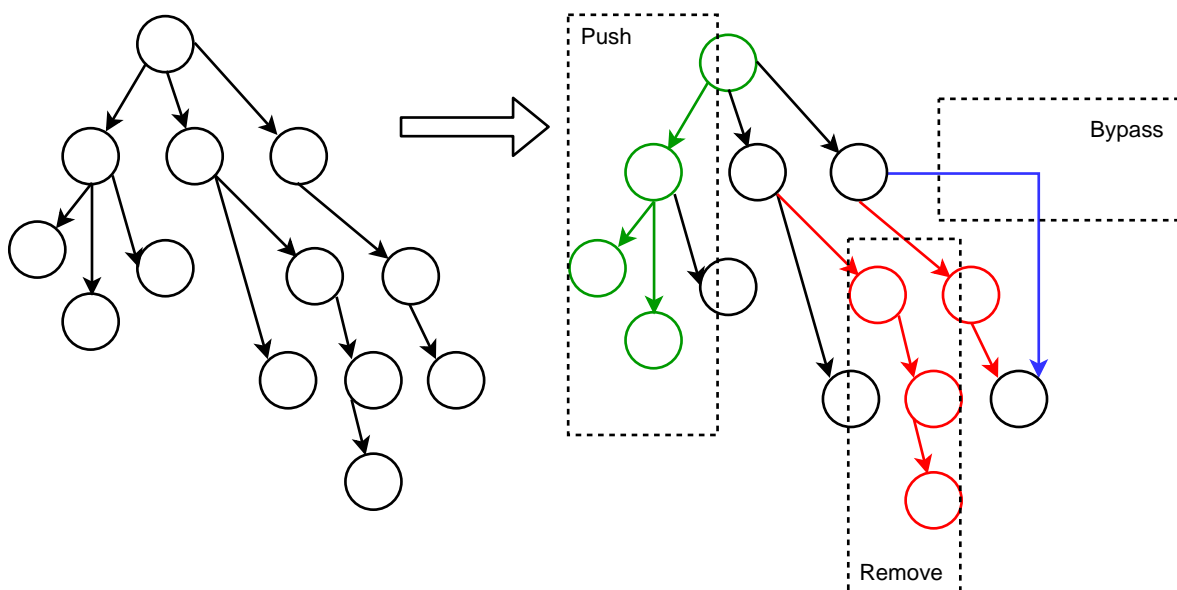


Bild 3: Transformation des Quellbaums in den Zielbaum

Remove ... Beim Betreten einer leeren Match-Regel wird ein Teilbaum nicht kopiert

Bypass ... Beim Betreten einer Match-Regel ohne Kopieranweisung wird der Knoten übersprungen

Push ... Durch gezielte Auswahl einer Knotenmenge auf der Descendant-Achse wird der XSLT Prozessor in eine bestimmte Richtung ge"pusht".

Pull-Stylesheets werden gewöhnlich mit **for-each** Loops programmiert. Dieser Ansatz ist meiner Ansicht nach gebräuchlich, wenn keine großen DTD Änderungen zu erwarten sind, der XML Baum flach strukturiert ist und die Anforderungen an die Konvertierung relativ einfach sind, bspw. beim Auswerten / Konvertieren von Konfigurationsdateien. In allen anderen Fällen sind Push-Stylesheets vorzuziehen, d.h. möglichst wenige **for-each** loops und möglichst viele **Template-Match** Regeln.

2.2 Eindeutigkeit der Regelbasis

Die Regelbasis der XSLT ereignisgesteuerten Laufzeitumgebung (Wir erinnern uns: Jedes Mal wenn der XSLT Prozessor einen Knoten betritt, wird ein Event ausgelöst) kann unendlich viele Regeln aufnehmen. Für die Vollständigkeit, Eindeutigkeit und Konsistenz der Regelbasis ist der Programmierer selbst verantwortlich.

Um die Eindeutigkeit der Regeln zu gewährleisten, gibt es verschiedene Mechanismen.

Reihenfolge der Match-Regeln

Im Normalfall sollte auf einen bestimmten Knoten in einem bestimmten Szenario genau eine Regel matchen. Falls es einen Konflikt gibt, wird zumindest bei Saxon diejenige Regel herangezogen, die im Stylesheet zuletzt deklariert wurde.

Diesen Umstand zu kennen, ist genau dann wichtig, wenn man einen bestehenden Stylesheet-Code übernehmen muss. Getreu dem Motto "Never change a running system" sollte man die Sache diesbzgl. sehr behutsam aufräumen.

Präzedenz der Auswertung

Match-Regeln werden gemäß ihrer Spezifität sortiert und diejenige, die auf einem Knoten in einem bestimmten Szenario am besten zutrifft wird zur Auswertung herangezogen. Grds. werden die Regeln anhand folgender Kriterien sortiert:

1. Importierte Template Regeln haben immer eine niedrigere Priorität als die Regeln des importierenden Stylesheets.
2. Templates mit einem höheren Priority Attribut haben Vorrang.
3. Templates ohne Priorität bekommen automatisch eine Default-Priorität. Die höchste Default-Priorität ist 0.5.

4. Diese Default Priorität errechnet sich anhand der Bedingungen oder Wildcards, die an einen Match-Regel geknüpft sind:
- Wenn mehrere Templates matchen, dann wird das am meisten spezifische zur Auswertung herangezogen.
 - Das am meisten spezifische Template wird anhand der Prioritäten berechnet.
 - Einfache Elementnamen (z.B. "para") haben Prio 0.
 - Wildcards (z.B. *, @*) haben Priorität -0.25
 - Knoten-Tests für andere Knoten (e.g. comment(), node(), etc.) haben Priorität -0.5
 - In allen anderen Fällen ist die Prio 0.5.

Beispiele:

- para -> 0
- h:* -> -0.25
- * -> -0.25
- node() -> -0.25
- contents/para -> 0.5
- contents/* -> 0.5

5. Mit einer Kommandozeilen-Option kann bei Saxon festgelegt werden, dass die Transformation abbricht, sobald es einen Konflikt bei der Regelauswertung gibt.

Import Präzedenz und Default-Regel

Wie in der obigen Sektion unter Punkt 1. angegeben, haben alle Regeln in einem importierten Stylesheet eine geringere Priorität als im importierenden Stylesheet. Diesen Umstand kann man sich zunutze machen, um eine Default-Regel einzubinden, bspw:

```
<xsl:template match="*" mode="#all"/>
```

Da sie sich in einem importierten Stylesheet befindet, hat sie geringere Priorität als alle anderen Regeln und greift nur dann, wenn für einen betretenen Knoten keine andere Match-Regel definiert ist.

Das ist z.B. praktisch, um nicht "gehandelte" Element zu identifizieren - dazu wäre die obige Regel nicht leer, sondern würde bspw. einen gelb markierten Warntext direkt in das Ausgabeformat schreiben.

Eine leere Default-Regel ist dagegen gut, wenn bspw. in einer XML-2-XML Migration automatisch Knoten im XML Baum abgetrennt werden sollen, für die keine Match-Regel existiert.

Prioritäten

Wie oben schon verdeutlicht werden alle Match-Regeln mit einer Priorität ausgestattet. Der Stylesheet-Entwickler hat die Möglichkeit diese Priorität zu überschreiben. Dazu wird das Attribut @priority an der Match-Regel verwendet. Ein Use-Case für die Prioritäten wäre bspw. folgendes Szenario:

- Die Eingabeinstanz soll in einer Vorprozessierung gefiltert werden.

- Dabei sollen Seminar-Elemente markiert werden, die nicht besonderen Bedingungen entsprechen:
 - Das Seminar-Element hat ein Feld "Ende-Datum" das abgelaufen ist.
 - Am Seminar-Element sind mehrere Dozenten angestellt, obwohl das Seminar-Element vom Type "Single" ist.
 - em Seminar-Element ist kein Dozent zugeordnet.
- Sicherlich kann es Seminar-Elemente geben, die alle drei Bedingungen erfüllen. Um das Error-Log aber nicht zu überfüllen, sollen die Filter nach ihren Prioritäten ausgeführt werden.

In Templates überführt, könnte diese Anforderung so umgesetzt werden:

```
<xsl:template match="Seminar[Ende-Datum/xs:date(.) le current-date()]"
  priority="30" mode="filter-network">
  <xsl:element name="Filtered-Seminar" namespace="{namespace-uri()}">
    <xsl:attribute name="reason">termed-seminar</xsl:attribute>
    <xsl:apply-templates select="node()|@*" mode="filter-network"/>
  </xsl:element>
</xsl:template>

<xsl:template match="Seminar[Type eq 'SINGLE' and count(dozenten/dozent) gt 1]"
  priority="20" mode="filter-network">
  <xsl:element name="filtered-Seminar" namespace="{namespace-uri()}">
    <xsl:attribute name="reason">dozenten-count</xsl:attribute>
    <xsl:apply-templates select="node()|@*" mode="filter-network"/>
  </xsl:element>
</xsl:template>

<xsl:template match="Seminar[not (dozenten/dozent)]" mode="filter-network">
  <xsl:element name="filtered-Seminar" namespace="{namespace-uri()}">
    <xsl:attribute name="reason">dozenten-missing</xsl:attribute>
    <xsl:apply-templates select="node()|@*" mode="filter-network"/>
  </xsl:element>
</xsl:template>
```

Modus Attribute

An allen Templates hat man die Möglichkeit einen selbst deklarierten Modus anzugeben. Wenn dann der XSLT Prozessor in eine bestimmte Richtung gepusht, vgl. **ERROR! Linktarget does not exist 41345870-4165-4a45-9ea5-e2c80dd1bf0a**, wird, werden nur diejenigen Regeln zur Auswertung herangezogen, die im selben Modus sind, wie der apply-templates Call.

Beispielsweise möchte man die Titel im Kapitel anders behandeln als die Kapitel im Inhaltsverzeichnis, denn im TOC sollen z.B. keine Fussnoten-Marker angezeigt werden.

In Templates formuliert würde diese Anweisung folgendermassen aussehen:

```
<xsl:template match="title" mode="toc">
  <div class="toc-entry">
    <xsl:apply-templates select="*[not(self::footnote)]"/>
  </div>
</xsl:template>

<xsl:template match="title">
  <h1>
```

```
<xsl:apply-templates/>
</hl>
</xsl:template>
```

Die Generierung des TOC könnte dann folgendermassen ablaufen:

```
<xsl:for-each select="chapter">
  <xsl:apply-templates select="title" mode="toc">
</xsl:for-each>
```

Bzgl. der Eindeutigkeit der Regelbasis kann man also auch noch anhand des Mode-Attributes Ausführungs-Gruppen bilden. Wie auch bei Angabe der Priorities kann man auf diese Weise Regeln setzen, die nie ausgeführt wurden, weil sie vllt. im Zuge einer Refactoring-Massnahme abgeklemmt und dann vergessen wurden.

Auch das mode-Attribut ist also mit Vorsicht zu geniessen und sparsam einzusetzen.

2.3 Namespaces

Wenn man XML Instanzen aus unterschiedlichen Quellen mit XSLT verarbeiten will, wird man sich wohl oder übel mit dem Thema Namespaces (NS) auseinander setzen müssen, um Konflikte in den Elementselektoren zu vermeiden.

Gerade bei hintereinandergeschalteten Transformationen kann es auch passieren, dass unerwartet ein Namespace in die Ausgabe generiert wird, den der folgende Prozessschritt nicht versteht, weil er dort nicht deklariert wurde.

Es gibt mehrere Möglichkeiten einen Namespace im Stylesheet zu deklarieren. Gehen wir davon aus, dass in einem Transformationsschritt genau eine Quelle und max. eine Konfigurationsdatei verarbeitet wird, dann kann das Stylsheet-Element bspw. so aussehen:

```
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:tektur="https://namespace-eigener-xslt-funktionen"
  xmlns="http://namespace-in-der-xml-eingabe.com/"
  xpath-default-namespace="https://namespace-der-konfigdatei.com/"
  exclude-result-prefixes="#all">
```

- Der **xsl** Namespace ist klar
- Der **xs** Namespace ist notwendig, wenn man typisiert arbeiten will. Er erlaubt das Einbinden von Datentypen nach der XML Schema Spezifikation²⁹⁾ und somit die bessere Validierung des Stylesheets zur Compile-Zeit.
- Die Deklaration eines eigenen geprefixten Namespaces erlaubt das Einbinden von eigenen XSLT Funktionen, wie z.B. auch das Einbinden der FunctX Bibliothek³⁰⁾
- Der Nicht-geprefixte Namespace ist der Default-Namespace und kann einen NS aus der Eingabe handeln

29) https://de.wikipedia.org/wiki/XML_Schema

30) <http://www.xsltfunctions.com/>

- Das Attribut **xpath-default-namespace** gibt einen weiteren NS an, der in XPATH Funktionen verwendet werden kann. In diesem Feld würde ich den NS einer Konfigurations- oder separaten Datendatei angeben.

Mehr als einen NS in der Eingabe sollte man aus meiner Sicht bei der XML Verarbeitung mit XSLT vermeiden wenn es geht. GGf. empfiehlt es sich, die Eingabe vor der Verarbeitung zu normalisieren und Elemente ggf. umzubennnen. Ansonsten kann man auch eigene Namespace-Prefixes deklarieren, wie z.B.:

```
xmlns:ext="https://www.tekturcms.de/external-tools"
```

und diese in XPATH Selektionen und Match-Regeln verwenden.

GEFAHR

Befinden sich in den Eingabedaten Namespaces, die man in den XSLT Stylesheets nicht handelt - der Namespace kann auch nur an einem ganz bestimmten Element hängen - so kann es bei der Transformation - ohne Fehlermeldung - zu unerwarteten Ergebnissen kommen.

- Deshalb sollte man die Daten im Vorfeld bzgl. Namespaces sehr genau analysieren.

Namespaces in der Eingabe werden also meistens über die Kopfdeklaration in der Stylesheetdatei gehandelt, welcher Namespace schliesslich in die Ausgabe geschrieben wird, hängt vom aktuell verarbeiteten Kontextknoten ab:

- Elemente, die man erzeugt, erhalten automatisch den Default-Namespace, wenn man nicht explizit einen NS angibt.
- Elemente, die man kopiert, transportieren den Namespace, den sie in der Eingabe hatten, wenn man dies nicht explizit verhindert.

Um diese beiden Default Einstellungen zu steuern (bzw. zu überschreiben) gibt es mehrere Möglichkeiten:

```
<xsl:element name="{local-name()}" namespace="{namespace-uri()}">
```

Hier wird ein Element mit dem un-geprefixten Namespace des Kontextknotens deklariert. Wenn der Kontextknoten keinen anderen Namespace hat, so wird hierdurch sichergestellt, dass der Default Namespace auch tatsächlich in die Ausgabe kommt.

```
<xsl:element name="meinelement" namespace="mein-namespace">
```

Hier wird ein Element mit eigener Namespace Angabe in die Ausgabe geschrieben. Einfacher geschrieben:

```
<mein-element xmlns="mein-namespace">
```

Es gibt auch ein Attribut am `xsl:copy` Element, das den Vorgang des Namespace-Kopierens steuern kann:

```
<xsl:template match="p">
  <xsl:copy copy-namespaces="no">
    <xsl:apply-templates/>
  </xsl:copy>
</xsl:template>
```

Hier wird der Namespace am `p` Element nicht in die Ausgabe geschrieben. Ggf. funktioniert diese Funktion aber mit unerwarteten Ergebnissen, deshalb sollte man sich ohne genauen Test nicht darauf verlassen.

Ebenso kann eine Default-Kopierregel verwendet werden, die es verbietet einen Namespace weiterzuvererben:

```
<xsl:template match="@* | node()">
  <xsl:copy inherit-namespaces="no">
    <xsl:apply-templates select="@* | node()" />
  </xsl:copy>
</xsl:template>
```

Namespaces in XQuery

Während XSLT dazu dienen sollte, XML Daten in andere (XML-) Formate zu transformieren, dient XQuery z.B. dazu auf einer NoSQL Datenbank Daten aus unterschiedlichen Quellen zu selektieren, zu harmonisieren und an verarbeitende Prozesse weiterzugeben.

Deshalb ist es für mich nicht so erstaunlich, dass das Namespace Konzept in XQuery irgendwie besser funktioniert.

Damit man überhaupt Daten auf einem mit Namespaces versehenen XML Dokument selektieren kann, müssen alle Namespaces am Anfang des XQuery Ausdrucks angegeben werden, das sieht so aus:

```
xquery version "1.0-ml";

import module namespace tekstur = "http://www.teturcms.de/xquery/common"
      at "common.xqy";
import module namespace mem = "http://xqdev.com/in-mem-update"
      at "/MarkLogic/appservices/utils/in-mem-update.xqy";
declare namespace local = "https://lambdawerk.com/code/alex-sandbox/1.0";
declare namespace weiredns = "https://weired-ns-in-input-data.com/weired/ns";
declare namespace xs = "http://www.w3.org/2001/XMLSchema";
```

Hier werden zuerst Funktionen aus anderen Modulen eingebunden, nämlich die in einer Datei `common.xqy` im selben Verzeichnis aus der eigenen Entwicklung, sowie die Bibliothek `mem` aus der MarkLogic Umgebung. Danach wird ein NS `local` deklariert, den man verwenden wird, wenn im weiteren Verlauf eigene XQuery Funktionen verwendet werden sollen, sowie der NS `weiredns`, der in den Eingabedaten vorhanden ist. Der NS `xs` ist analog zum XSLT Beispiel gesetzt.

3 Ausgewählte Themen

Hands-on Materiel mit Beispielquelltexten.

3.1 XSLT Uses Cases und Lösung

In diese Kapitel werden einige ausgewählte Themen zu XSLT dargestellt. Dabei geht es weder um Vollständigkeit noch um die beste/eleganteste Lösung, sondern eher um die Vorstellung eines Anwendungsszenarios mit einem potentiellen Lösungsansatz.

3.1.1 Vortransformationen

Bei einer komplexen Transformation ist es ratsam und sogar manchmal unabdingbar die Konvertierung in einzelne Stufen aufzuteilen. Das hat folgende Vorteile:

- Der Prozess ist transparenter, da die einzelnen Stufen leichter überschaubar sind.
- Die Zwischenergebnisse können für Debug-Zwecke ausgewertet werden oder dienen als Eingabe für andere Prozesse.
- Nicht-relevante oder invalide Teilbäume können aus der Eingabeinstanz gefiltert werden, um so die weitere Verarbeitung zu beschleunigen.
- Hilfskonstrukte können erzeugt werden. Diese erleichtern die weitere Verarbeitung.

Es gibt zwei Möglichkeiten, wie eine Vortransformation eingebunden werden kann:

- In einem separaten File bzw. einer XML Instanz, die vom XSLT Prozessor vor der eigentlichen Transformation aufgerufen wird um einen Zwischenstand produziert, der als Eingabe für den Haupttransformationsschritt dient.
- Innerhalb des eigentlichen XSLT Stylesheets. Hier wird das Ergebnis der Vortransformation in einer Variablen erzeugt.

Den zweiten Punkt möchte ich anhand einer Beispiel XSLT Skripts vorführen. Betrachten wir folgende Input Daten:

```
<education-system>
  <administrative-regions>
    [...]
    <administrative-region id="31" name="Bavaria">
      <shools>
        <school id="45">
          <teachers>
            <teacher id="576"/>
            <teacher id="345"/>
            <teacher id="12"/>
          </teachers>
        </school>
        <school id="36">
          <teachers>
            <teacher id="576"/>
            <teacher id="8"/>
          </teachers>
        </school>
      </shools>
    </administrative-region>
  </administrative-regions>
</education-system>
```

```

    </school>
    [...]
  </schools>
</administrative-region>
[...]
</administrative-regions>
</education-system>

```

Die erste Datei beinhaltet eine Zuordnung von Lehrern zu Schulen in verschiedenen Regierungsbezirken. Um die Daten zu den beiden referenzierten Objekten einzusehen müssen zwei weitere Dateien konsultiert werden. Die Datei, welche die Lehrer auflistet:

```

<teachers>
  [...]
  <teacher id="576">
    <first-name>Alfons</first-name>
    <last-name>Blimetsrieder</last-name>
    <subjects>
      <subject>Biology</subject>
      <subject>Math</subject>
      <subject>Sport</subject>
    </subjects>
    <suspended>2017-12-31</suspended>
  </teacher>
  [...]
</teachers>

```

Und die Datei, welche die Schulen auflistet:

```

<schools>
  [...]
  <school id="45">
    <name>Gymnasium Bad Aibling</name>
    <type>Oberschule</type>
  </school>
  [...]
</schools>

```

Um diese Daten verarbeiten zu können ist es sinnvoll, die drei Dateien in einem ersten "Resolver" Schritt zusammenzuführen und ggf. irrelevante Strukturen zu entfernen. Lehrer aus obigem Beispiel können beispielsweise suspendiert worden sein. Das folgende Skript erledigt dies mittels einer zusätzlichen Transformation in eine Variable:

```

<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  exclude-result-prefixes="#all">

  <xsl:output indent="yes" method="xml"/>

  <xsl:strip-space elements="*" />

  <xsl:param name="file-1" required="yes"/>
  <xsl:param name="file-2" required="yes"/>

```

```

<xsl:param name="file-3" required="yes"/>

<xsl:variable name="files" select="(doc($file-1), doc($file-2), doc($file-3))"/>
<xsl:variable name="bavaria-region-ids" select="(31, 58)"/>

<xsl:key name="teachers" match="teacher" use="@id"/>
<xsl:key name="schools" match="school" use="@id"/>

<xsl:template name="main">
  <xsl:variable name="resolve-result">
    <xsl:apply-templates select="$files/administrative-regions" mode="resolve"/>
  </xsl:variable>
  <xsl:apply-templates select="$resolve-result/administrative-regions"/>
</xsl:template>

<xsl:template match="administrative-region[not(@id = $bavaria-region-ids)]"
  mode="resolve"/>

<xsl:template match="school" mode="resolve">
  <xsl:copy>
    <xsl:copy-of select="key('schools',@id, $files/schools[1]/root())/node()"/>
    <xsl:apply-templates select="node()|@*" mode="resolve"/>
  </xsl:copy>
</xsl:template>

<xsl:template match="teacher" mode="resolve">
  <xsl:copy-of select="key('teachers',@id, $files/teachers[1]/root())/node()"/>
</xsl:template>

<xsl:template match="teacher[suspended/xs:date(.) le current-date()]/>

<xsl:template match="node()|@*" mode="#all">
  <xsl:copy>
    <xsl:apply-templates mode="#current"/>
  </xsl:copy>
</xsl:template>

</xsl:stylesheet>

```

Im ersten Resolve-Schritt werden die Referenzen zu den Lehrer- und Schul-Objekten aufgelöst, d.h. die Attribute des Schul-Objekts werden in die Struktur aus der ersten Datei kopiert. Die Liste der Lehrer an diesen Schul-Objekten bleibt erhalten und wird mit dem Inhalt aus der zweiten Datei bestückt.

Zusätzlich werden alle Regierungsbezirke entfernt, die nicht zu Bayern gehören, was die weitere Verarbeitung wesentlich beschleunigen wird. Lehrer die suspendiert worden sind fliegen ebenfalls raus.

3.1.2 Vererbung

Mit XSLT kann man Konstrukte nachbilden, so wie sie in anderen Programmiersprachen auch vorhanden sind. Bspw. die Vererbung. Dabei wird in einer Spezialisierung eine schon bereits getätigte Implementierung übernommen und erweitert oder eingeschränkt.

Der Vorteil dabei ist, dass man nicht alles nochmal neu schreiben muss. Das verkleinert die Redundanz, führt zu einer besseren Wartbarkeit und einer geringeren Fehleranfälligkeit.

**Beispiel:
Parameterisierung**

Gewöhnlich implementiert man ein Stylesheet für ein bestimmtes Ausgabeformat und eine Produktvariante. Schrittweise werden dann weitere Varianten und Formate hinzugefügt.

Am komfortabelsten hat man es natürlich, wenn zu Beginn der Implementierung eine vollständige Spezifikation vorliegt... Das ist aber natürlich eher selten der Fall.

Aus diesem Grund ist es wichtig, sich eine gute Strategie zu überlegen, damit die Architektur nicht in Spagetti-Code ausartet.

Eine gute Option wäre, die XSLT Import Präzedenz auszunutzen, vgl. Kapitel [ERROR! Link-target does not exist ec4e438f-b6cc-4f50-b903-bda598a9003c](#).

Angenommen es geht darum zu einem späteren Zeitpunkt weitere Parameter einzuführen. Ein Switch, wie der folgende, müsste dann an mehreren Stellen im Code aktualisiert werden.

```
<xsl:choose>
  <xsl:when test="$myParameter='this_option'">
    <!-- do this -->
  </xsl:when>
  <xsl:when test="$myParameter='that_option'">
    <!-- do that -->
  </xsl:when>
  [...]
</xsl:choose>
```

Besser ist es, wenn man ein Core-Stylesheet pflegt, das für ein Format und eine Produktvariante gut ausgetestet ist. Dieses Core-Stylesheet wird dann einfach für eine neue Variante importiert und relevante Teile werden für die neue "Spezialisierung" überschrieben. Beispielsweise könnte eine Regel zum Setzen des Headers auf jeder Seite so implementiert sein:

```
<xsl:template name="render-header">
  <!-- print logo on the left side spanning two rows-->
  <!-- print some metadata right side first row -->
  <!-- print a running header right side second row -->
</xsl:template>
```

Will man in einem neuen Format, bspw. A5, diese Logik austauschen und nur eine Zeile drucken, z.B. weil man nicht so viel Platz hat, so würde in einem "abgeleiteten" Stylesheet einfach die Regel noch einmal implementiert.

```
<xsl:choose>
<xsl:template name="render-header">
  <!-- print a running header on left side -->
  <!-- print logo on right side -->
</xsl:template>
```

Dieses Template hat nun Vorrang und wird zur Auswertung herangezogen, mit der Konsequenz, dass der Header nur einzeilig gedruckt wird. Das schöne an diesen "Named-Templates" ist auch, dass man sie innerhalb von Variablen verwenden kann:

```
<xsl:variable name="margin-width">
```

```
<xsl:call-template name="get-margin-width"/>
</xsl:variable>
```

Das Template "get-margin-width" kann in einem "Sub"-Stylesheet überschrieben werden ohne dass die Variablen-Zugriffe im Core-Stylesheet angepasst werden müssten. Eine Zuweisung, wie:

```
width="{ $margin-width }"
```

müsste nirgendwo im Code nochmal angefasst werden.

3.1.3 XSLT Streaming

Bei grossen flach strukturierten Datenmengen gibt es zwei Möglichkeiten:

1. Für einfache Sammel- und Auswertungsaufgaben schreibt man sich am besten einen kleinen Parser, z.B. mit der Python sgmlib³¹⁾.
2. Für komplexere Aufgaben, in denen man nicht an jeder Stelle über den ganzen XML Baum navigiert und sich die Werte zusammensuchen suchen muss, kann man die Streaming Funktion des Saxon XSLT Prozessors verwenden.

XSLT Streaming ist in der XSLT Version 3.0 neu hinzugekommen³²⁾ und in der kommerziellen Saxon-EE Lösung implementiert³³⁾. Bei dieser Methode wird kein Eingabebaum im Speicher aufgebaut, was zu einer drastischen Performanzsteigerung führt.

Es gibt ein paar Regeln, die man bei der Verarbeitung großer Datenmengen über die Streaming Funktionen beachten sollte:

- Bei einer XPATH Auswertung sollte nur ein einfacher Ausdruck mit höchstens einer konsumierenden Selektion gegeben sein. Konsumieren heißt, dass vom Kontextknoten aus eine Knotenmenge abwärts selektiert wird. Dagegen bleibt die Information bzgl. der Ancestor-Achse erhalten.
- Bei einer Selektion sollte man aber darauf achten nur atomarische Werte auszuwählen.
- Knotenmengen, die über die Streaming Option eingelesen wurden, können nicht einer Funktion übergeben werden. Sie sind auch nicht einer Variablen zuweisbar.
- "Crawler"-Ausdrücke, wie //section sind nicht zu verwenden, ebenso ein rekursiver Abstieg mit Selektion, wie bspw. mit einem Apply-Templates Call.

Zu Beginn der Streaming-Aktion kann man sich auf konventionelle Art und Weise Teilbäume, die nicht so performanzlastig aufgebaut werden, in einer Variablen abspeichern und im Verlauf der Streaming-Verarbeitung z.B. für einen Vergleich auswerten.

3.1.3.1 XSLT Akkumulator

Ein einfaches Streaming Stylesheet könnte z.B. so aussehen:

```
<xsl:stylesheet version="3.0"
31) https://docs.python.org/2/library/sgmlib.html
32) https://www.saxonica.com/html/documentation/sourcedocs/streaming/xslt-streaming.html
33) https://www.saxonica.com/html/documentation/sourcedocs/streaming/
```

```

xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
exclude-result-prefixes="#all">

<xsl:output method="xml" indent="yes"/>

<xsl:mode on-no-match="shallow-copy" use-accumulators="entry-count" streamable="true"/>

<xsl:accumulator name="entry-count" as="xs:integer" initial-value="0"
    streamable="yes">
    <xsl:accumulator-rule match="entry" select="$value + 1"/>
</xsl:accumulator>

<xsl:template match="/">
    <result>
        <xsl:apply-templates/>
        <count>
            <xsl:value-of select="accumulator-after('entry-count')"/>
        </count>
    </result>
</xsl:template>
</xsl:stylesheet>

```

Diese Stylesheet hat einige Besonderheiten:

Zum einen wird darin ein Default-Modus deklariert, der jeden Knoten der Eingabeinstanz über eine implizite Identity-Transformation (shallow-copy)³⁴⁾ in die Ausgabeinstanz kopiert.

Auf herkömmlichem Weg würde man dafür ein Templates wie dieses verwenden:

```

<xsl:template match="node()|@*"
    <xsl:copy>
        <xsl:apply-templates select="node()|@*" />
    </xsl:copy>
</xsl:template>

```

Zum anderen wird ein Akkumulator verwendet. Normalerweise gibt es in XSLT keine Variablen, sondern nur Konstanten, so wie das auch bei funktionalen Programmiersprachen der Fall ist.

Es gab zwar schon länger eine Saxon-Erweiterung, die die mehrmalige Zuweisung eines Wertes an eine Variable erlaubte, im Normalfall braucht man diese Eigenschaft aber nicht.

Bei der Verarbeitung sehr großer Datemengen, ist es aber unumgänglich, denn sonst würde der Laufzeitstapel schnell an seine Grenzen gelangen.

Ein Akkumulator akkumuliert Werte, wie der Name schon sagt. Das können atomare Typen sein, wie im obigen Beispiel, aber auch Datenstrukturen können aufgebaut werden, wie bspw. das Abspeichern des gerade prozessierten Teilbaums in einem Dictionary zur späteren Auswertung bzw. Gruppierung der Key-Elemente.

Auch im Akkumulator muss das streamable="yes" Property gesetzt sein, wenn er im Streaming-Modus arbeiten soll. In diesem Modus kann der Akkumulatorwert erst ausgelesen werden, wenn der untersuchte Baum vollständig durchlaufen wurde.

34) <https://www.saxonica.com/html/documentation/xsl-elements/mode.html>

Um die Unterschiede zum "normalen" XSLT Betrieb festzustellen, können im obigen Beispiel einige offensichtlich korrekte Änderungen vorgenommen werden, die der Streaming Prozessor allerdings nicht akzeptiert.

```
Cannot call accumulator-after except during the post-descent
phase of a streaming template
```

Diese Fehlermeldung erscheint, wenn man den Apply-Templates Call entfernt. Der Akkumulator wird also nur befüllt, wenn der Baum auch explizit durchlaufen wurde. Dieser Durchlauf kann auch ein reines Kopieren sein, bspw. kann man den Apply-Templates Call auch durch ein

```
<xsl:copy-of select="."/>
```

ersetzen, was gleichbedeutend mit der Mode Einstellung

```
on-no-match="deep-copy"
```

wäre. Wie man sieht hat sich in XSLT 3.0 viel bzgl. der Handhabung verschiedener Verarbeitungsmodi getan. Anstatt Default-Match Regeln zu schreiben, kann man ganz oben am Stylesheet Modus Properties setzen, die den Baumdurchlauf auf verschiedene Arten realisieren.

Die Verarbeitung großer Datenmengen ist aber mit Streaming etwas tricky und es sollte geprüft werden, ob ggf. konventionelles Performanz-optimiertes XSLT für den Anwendungsfall ausreichen würde.

3.1.3.2 XSLT Iterator

XSLT Streaming ist ziemlich tricky. Betrachten wir ein einfaches Problem.

Es soll ein kommaseparierter Report aus folgender XML Quelle generiert werden:

```
<status-report>
  <status-change>
    <billing_id>360788</dentaltrac_encounter_id>
    <claim_ids>967382,673647</claim_ids>
    <status>open</status>
    <time_stamp>2019-02-22T13:53:34.605Z</status_time>
  </status-change>
  <status-change>
    <billing_id>360788</dentaltrac_encounter_id>
    <claim_ids>967382,673647</claim_ids>
    <status>open</status>
    <time_stamp>2019-02-22T13:53:34.605Z</status_time>
  </status-change>
  [...]
</status-report>
```

Mit einer `for-each` Loop und einem Named-Template würde das so gehen:

```
<xsl:template name="main">
  <xsl:for-each select="$input-file/status-report/status-change">
    <xsl:value-of select="concat(billing_id,', ')" />
    <xsl:value-of select="concat(claim_ids,', ')" />
    <xsl:value-of select="concat(status,', ')" />
    <xsl:value-of select="concat(format-dateTime(xs:dateTime(time_stamp),
      '[Y]-[M]-[D] [H]:[m] '), '&#10;')'" />
  </xsl:for-each>
</xsl:template>
```

► *Named-Templates, die direkt über den Saxon Aufruf `saxon -it:main` aufgerufen werden, sind dann brauchbar, wenn keine eindeutige Eingabequelle vorhanden ist, bspw. weil aus mehreren Quellen eingelesen werden soll, wenn die Eingabe von einem Webservice kommt oder vom XSLT Skript selbst erzeugt wird.*

Im vorliegenden Fall wird von einer Datei eingelesen - wir brauchen also kein Named-Template. Statt der for-each Schleife können wir uns auch auf den rekursiven Abstieg des XSLT Prozessors verlassen, was den Code weiter vereinfacht:

```
<xsl:template match="/status-report/status-change">
  <xsl:value-of select="concat(billing_id,', ')" />
  <xsl:value-of select="concat(claim_ids,', ')" />
  <xsl:value-of select="concat(status,', ')" />
  <xsl:value-of select="concat(format-dateTime(xs:dateTime(time_stamp),
    '[Y]-[M]-[D] [H]:[m] '), '&#10;')'" />
</xsl:template>
```

Wollen wir große Datenmengen schnell verarbeiten mit ein paar Hundert MB, so ist es sinnvoll auf die neue XSLT3.0 Streaming Option umzuschalten, weil dadurch kein Eingabebaum in-Memory aufgebaut wird. Wie schon im Kapitel [ERROR! Linktarget does not exist 8412284f-6cf5-4f01-bb46-4b22875ccc71](#) angesprochen, gibt es dazu mehrere Möglichkeiten.

Wir betrachten hier das `xsl:iterator` ³⁵⁾ Konstrukt und stossen dabei auf einige Fallstricke. Zunächst einmal unsere Settings:

⚠ GEFAHR

32rr

- r23r232
- fwef

- Wir benutzen `xsl:source-document` in Verbindung mit dem `streamable='yes'` Attribut, um dem Prozessor mitzuteilen, dass er im Streaming Modus arbeiten soll.
- Wenn wir die Quelle über einen Parameter einlesen, dann müssen wir auch die Transformation über ein Named-Template starten.

35) <https://www.saxonica.com/html/documentation/xsl-elements/iterate.html>

Ohne zu wissen, wie XSLT Streaming genau funktioniert, setzen wir eine Reihe von `value-of select` statements in den Iterator:

```
<xsl:template name="main">
  <xsl:source-document href="{ $input-file}" streamable='yes'>
    <xsl:iterate select="status-report/status-change">
      <xsl:value-of select="concat(billing_id, ', ')" />
      <xsl:value-of select="concat(claim_ids, ', ')" />
      <xsl:value-of select="concat(status, ', ')" />
      <xsl:value-of select="concat(format-dateTime(xs:dateTime(time_stamp),
        '[Y]-[M]-[D] [H]:[m] '), '&#10; ')" />
    </xsl:iterate>
  </xsl:source-document>
</xsl:template>
```

und werden dafür prompt mit einer Fehlermeldung belohnt:

```
Static error on line 16 column 64 of report.xsl:
XTSE3430: The body of the xsl:stream instruction is not streamable
* There is more than one consuming operand: {xsl:value-of} on line 18, and
{xsl:value-of} on line 19
```

In diesem Iterator ist also nur eine "konsumierende" `value-of` Operation erlaubt. Um nur einmal zu selektieren, müssen wir - auf Kosten der Lesbarkeit - ziemlich umbauen. Eine Lösung könnte z.B. so aussehen:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  exclude-result-prefixes="xs"
  xpath-default-namespace="https://tekturcms.de/schema/status-report/1.0"
  version="3.0">

  <xsl:param name="input-file" required="yes"/>

  <xsl:output method="text"/>

  <!-- https://www.saxonica.com/html/documentation/xsl-elements/iterate.html -->

  <xsl:template name="main">
    <xsl:source-document href="{ $input-file}" streamable='yes'>
      <xsl:iterate select="status-report/status-change/*">
        <xsl:choose>
          <xsl:when test="name()='time_stamp'">
            <xsl:value-of select="concat(format-dateTime(xs:dateTime(time_stamp),
              '[Y]-[M]-[D] [H]:[m] '), '&#10; ')" />
          </xsl:when>
          <xsl:otherwise>
            <xsl:value-of select="concat(., ', ')" />
          </xsl:otherwise>
        </xsl:choose>
      </xsl:iterate>
    </xsl:source-document>
  </xsl:template>
</xsl:stylesheet>
```

Hier wird davon ausgegangen, dass das Element mit Namen 'time_stamp' als letztes in der Sequenz vorkommt und beim Auftreten (`
`) wird ein Zeilenumbruch gesetzt. Der deklarative Ansatz aus dem ersten Beispiel geht dabei verloren.

► Logisch wird beim XSLT Streaming auf einer niedrigeren Abstraktionsebene programmiert, um den Anforderungen des Prozessors gerecht zu werden.

Für eine **1.6 GB Datei** benötigt das obige Skript auf meinem Rechner gute **drei Minuten**. Der traditionelle template-match Ansatz bricht mit einer Out-of-Memory Exception ab, selbst wenn man den Java Heap Size auf 4GB einstellt.

3.1.4 Identifikation mit generate-id()

Die generate-id() Funktion gibt es in XSLT schon immer. Mit ihr kann eine Prüfsumme eines Knotens im Baum generiert werden.

Das funktioniert natürlich nur, wenn man bei der Auswertung dieses Wertes nicht den Kontext wechselt. D.h. z.B. dass ein Knoten in einem Baum, der in einer Variablen gespeichert ist, eine andere Prüfsumme bekommt, als derselbe Knoten im Kontext-Baum.

Beispiel Stückliste

Ein Anwendungsszenario wäre bspw. die Generierung einer Target-ID für ein Bauteil in einer Stückliste. Das Bauteil ist nur einmal im System erfasst, hat also eine eindeutige ID, soll aber an mehreren Stellen in die Ausgabe (Eine Dokumentation für eine Maschine) generiert werden.

Die Id an einem Element `<part id="1234">` würde somit mehrfach in die XML Eingabe für einen XSL-FO Prozessor erscheinen und ist für Referenzen unbrauchbar geworden. Deshalb ist es ratsam beim Rendern der Bauteile eine neue Id zu vergeben, das kann z.B. mit den folgenden Templates (vereinfacht) passieren:

```
<xsl:key name="parts" match="part" use="@id"/>

<xsl:template match="part" mode="content">
  <!-- Ausgabe des Bauteils im Content Bereich -->
  <fo:block id="{generate-id()}">
    <fo:external-graphic xsl:use-attribute-sets="part.img"/>
  </fo:block>
</xsl:template>

<xsl:template match="part" mode="part-list">
  <!-- Ausgabe einer Liste mit allen Verweisen an unterschiedlicher Stelle -->
  <fo:block>
    <xsl:for-each select="key('parts',@id)">
      <fo:page-number-citation ref-id="{generate-id()}" />
    </xsl:for-each>
  </fo:block>
</xsl:template>
```

Beispiel Mantel Dokument

Im Bereich EDI Datenaustausch werden große XML Dateien versendet, die man auf einzelne Transmissions aufsplitten will, um sie in einer XML Datenbank abspeichern zu können. Die Struktur einer Datenübertragung könnte folgendermassen aussehen:

```
WRAPPER1
  SEQUENZ1
  SEQUENZ2
```

```

SEQUENZ3
WRAPPER2
  SEQUENZ1
  SEQUENZ2
  SEQUENZ3
  SEQUENZ4
WRAPPER3
  SEQUENZ1
  SEQUENZ2
  CONTENT
    DATA1
    DATA2
    DATA3
    DATA4
    DATA5
  CONTENT
    DATA1
    DATA2
    DATA3
    DATA4
    DATA5
WRAPPER4
  SEQUENZ1
  CONTENT
    DATA1
    DATA2
    DATA3
    DATA4
    DATA5
[...]
```

Jedes einzelne `CONTENT` Element soll nun einen Mantel erhalten und separat in einer Datei abgelegt werden. Der "Umschlag" soll dabei alle Elemente des Rahmens der Transmission erhalten. Also alles auf der Descendant-Achse bis zum Element `WRAPPER3`, ausserdem noch die Elemente `SEQUENZ1` und `SEQUENZ2`, sowie das Element `WRAPPER4` mit Kind `SEQUENZ1`. Ohne groß auf die Performanz zu achten, könnte das recht einfach so realisiert werden:

```

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0">

  <xsl:output method="xml" indent="yes"/>
  <xsl:strip-space elements="*" />

  <xsl:template match="/">
    <xsl:apply-templates select="/WRAPPER1/WRAPPER2/WRAPPER3/CONTENT" mode="umschlag" />
  </xsl:template>

  <xsl:template match="CONTENT" mode="umschlag">
    <xsl:result-document href="{concat(@id, '.xml')}">
      <umschlag>
        <metadaten><!-- einige Metadaten --></env:metadaten>
        <nutzdaten>
          <xsl:apply-templates select="ancestor::WRAPPER1">
            <xsl:with-param name="this-id" select="generate-id()" tunnel="yes" />
          </xsl:apply-templates>
        </nutzdaten>
      </umschlag>
    </xsl:result-document>
  </xsl:template>
```

```

<xsl:template match="node()|@*">
  <xsl:copy>
    <xsl:apply-templates select="node()|@*" />
  </xsl:copy>
</xsl:template>

<xsl:template match="CONTENT">
  <xsl:param name="this-element" tunnel="yes" />
  <xsl:if test="$this-id = generate-id()">
    <xsl:copy>
      <xsl:apply-templates select="node()|@*" />
    </xsl:copy>
  </xsl:if>
</xsl:template>

</xsl:stylesheet>

```

Im rekursiven Abstieg wird im Modus "umschlag" jedes *CONTENT* Element selektiert und in einen Umschlag verpackt. Der eigentlich Inhalt des Umschlags wird generiert, indem der gesamte XML Baum über die Standard-Kopierregel in das Element *<nutzdaten>* gesetzt wird. Dabei wird aber nur derjenige *CONTENT* Abschnitt evaluiert, der zu der als Parameter übergebenen generierten Id passt.

3.1.5 Reguläre Ausdrücke

Manchmal reicht für das Durchsuchen des XML Baums XPATH nicht mehr aus und man will auf den Textknoten reguläre Ausdrücke aufrufen. Mir sind derzeit drei XPATH Funktionen bekannt, in denen man XPATH angeben kann:

- *fn:matches(subject, pattern, flags)*
- *fn:replace(subject, pattern, replacement, flags)*
- *fn:tokenize(subject, pattern, flags)*

Bei der Eingabe des regulären Ausdrucks muss man natürlich Zeichen, wie >, <, & maskieren, was den Ausdruck im Gegensatz zur herkömmlichen Nicht-XML Programmierung noch ein bisschen komplizierter macht.

Der Ausdruck, der bspw. auf alle XML Tags matched ist folgender: *<[^\>^!]+>*. Hier sind die für XML reservierten Zeichen bereits maskiert.

XSLT Analyze String

Es gibt aber auch ein eigenes XSLT Konstrukt, unabhängig von XPATH, um reguläre Ausdrücke anwenden zu können. Der Quelltext dazu sieht so aus:

```

<xsl:template match="text()[parent::xml-code]">
  <xsl:analyze-string select="." regex="&lt;[^\&gt;^!]+&gt;">
    <xsl:matching-substring>
      <b><xsl:value-of select="."/></b>
    </xsl:matching-substring>
    <xsl:non-matching-substring>
      <xsl:value-of select="."/>
    </xsl:non-matching-substring>
  </xsl:template>

```

Hier wird jeder Textknoten eines Elements

<xml-code>}} untersucht und falls ein XML-Tag enthalten ist, so wird dieses über

Anweisung in ein `` Tag gewrapped. Im nächsten Transformationsschritt wird dieses wiederum in Fettschrift dargestellt.

Das ist ein einfacher Syntax-Highlighter für XML Quelltexte. Auf diese Weise ist der Syntax-Highlighter für dieses PDF realisiert.

► Man kann auch die `xsl:analyze-string` Elemente in den `xsl:matching-substring` und `xsl:non-matching-substring` Elementen verschachteln, was natürlich noch wesentlich kompliziertere Problemstellungen erlaubt.

3.2 XQuery Use Cases und Lösung

In diese Kapitel werden einige ausgewählte Themen zu XQuery dargestellt. Dabei geht es weder um Vollständigkeit noch um die beste/eleganteste Lösung, sondern eher um die Vorstellung eines Anwendungsszenarios mit einem potentiellen Lösungsansatz.

3.2.1 Abfragen mit XQuery

Xquery führt im Publishing-Bereich ein Schattendasein. In meiner Zeit als XSL Programmierer für zwei Publishing Firmen hatte ich damit nie zu tun. Erst als ich näher an den eigentlichen Daten war und mit XML Datenbanken zu tun hatte, kam ich mit XQuery in Berührung.

Während relationale Datenbanken mit SQL abgefragt werden, verwendet man bei XML Datenbanken, wie eXist³⁶⁾ oder Marklogic³⁷⁾, XQuery als Abfragesprache.

Aber auch einzelne XML Dokumente können z.B. in Oxygen XML Editor mit dem XQuery Builder Tool³⁸⁾ oder auch per Saxon Kommandozeile abgefragt werden:

```
java -cp usr/lib/saxon/saxon.jar net.sf.saxon.Query
-s:"schulen.xml"
-qs:"/schulen/schule[id='6']"
-o:"/Users/Alex/Desktop/schule_6.xml"
```

Mit der Option `-qs` kann hier der Querystring angegeben werden.

Wie man an dem einfachen Beispiel schon sieht, ist XQuery mit XPATH verwandt. XQuery umfasst den Sprachumfang von XPATH bietet aber zusätzlich die FLOWR Syntax um mächtigere Abfragen stellen zu können. Mittels weiterer Extensions³⁹⁾ können aber auch ganze Programme erstellt werden, die weit über die Funktionalität einer "Abfragesprache" hinausgehen.

36) <http://exist-db.org/exist/apps/homepage/index.html>

37) <https://de.marklogic.com/>

38) https://www.oxygenxml.com/xml_editor/xquery_builder.html

39) <http://cs.au.dk/~amoeller/XML/querying/flwrexp.html>

XQuery Builder

Oxygen XML Editor bietet eine schöne Möglichkeit XQuery-Abfragen auf einem geladenen XML Dokument auszuführen. Dazu kann man seine Query in das betreffende Eingabefenster schreiben.

Mit dem XQuery Builder von oXygen lassen sich unkompliziert Queries testen

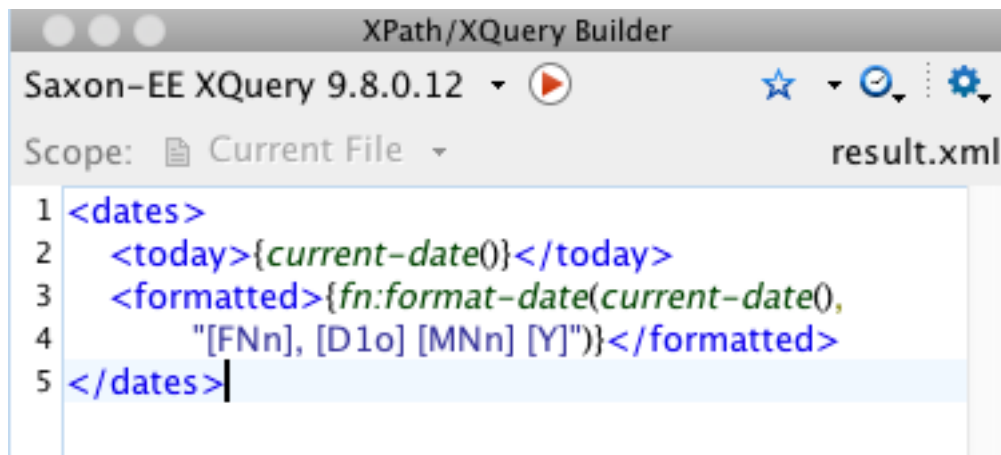


Bild 4: oXygen XQuery Builder

Mit folgendem Ergebnis:

```
<dates>
  <today>2019-01-16+01:00</today>
  <formatted>Wednesday, 16th January 2019</formatted>
</dates>
```

FLOWR Expression

FLOWR steht für *for, let, where, order by, return*. Das sind die Query-Anweisungen, die in dem Ausdruck erlaubt sind - in genau dieser Reihenfolge.

```
let $bibliothek := .
for $xin $bibliothek//buecher,
  $yin $bibliothek//autoren/autor
where starts-with($autor, 'Grass')
  and $x/@authorId = $y/@id
return $x/titel
```

In dieser Query werden die Titel aller Bücher von Grass zurückgeliefert. Bemerkenswert ist hier die Syntax.

► Normalerweise würde man zwischen den einzelnen Anweisungen einen Blockabschluss, wie ein Semikolon erwarten. Da wir aber hier funktional programmieren, ist die Sache etwas anders...

XML per XQuery

Es ist aber auch möglich XML zu erzeugen, wobei natürlich für eine Transformation XSLT vorzuziehen ist. Dazu werden Tags direkt in die Expression geschrieben, wie z.B. hier:

```
declare variable $nachname as xs:string external;
<buecher autor="{ $nachname }">
{
  let $bibliothek := .
  for $xin $bibliothek/buecher//buch,
    $yin $bibliothek/autoren//autor
  where starts-with($y, $nachname)
    and $x/@autorId = $y/@id
  order by $x/ausgabe
  return
  <buch ausgabe="{ $x/ausgabe }">
    { $x/titel }
  </buch>
}
</buecher>
```

Speichert man dieses Schnippsel in einer Datei `buecher.xquery` ab, so kann man mit der folgenden Kommandozeile auf einer `buecher.xml` Datei als Eingabe suchen:

```
java -cp usr/lib/saxon/saxon.jar net.sf.saxon.Query -t -s:buecher.xml
                                           -q:buecher.xquery
                                           -o:ergebnis.xml
                                           nachname=grass
```

Document Projection

Document Projection⁴⁰⁾ ist ein verstecktes Saxon XQuery Feature. Es funktioniert nur für eine einzige Abfrage. Das kann schon recht hilfreich sein, wenn man ein mehrere 100MB großes Dokument durchsuchen will.

Ohne Projection würde das Beispiel von oben so verarbeitet:

```
java -cp usr/lib/saxon/saxon.jar net.sf.saxon.Query -t
-s:buecher.xml
-q:buecher.xquery
-o:ergebnis.xml
-projection:off
nachname=grass
Saxon-EE 9.7.0.20J from Saxonica
Java version 1.8.0_60
Using license serial number V005095
Analyzing query from Desktop/buecher.xquery
Generating byte code...
Analysis time: 201.10095 milliseconds
Processing file:/Users/Alex/buecher.xml
Using parser com.sun.org.apache.xerces.internal.jaxp.SAXParserImpl$JAXPSAXParser
Building tree for file:/Users/Alex/buecher.xml
using class net.sf.saxon.tree.tiny.TinyBuilder
Tree builtin 3.482278ms
Tree size: 46 nodes, 58 characters, 6 attributes
Execution time: 27.137589ms
Memory used: 67031664
```

40) <http://www.saxonica.com/documentation/#!sourcedocs/projection>

Mit der Option `-projection:on` verändert sich die Ausführungszeit signifikant:

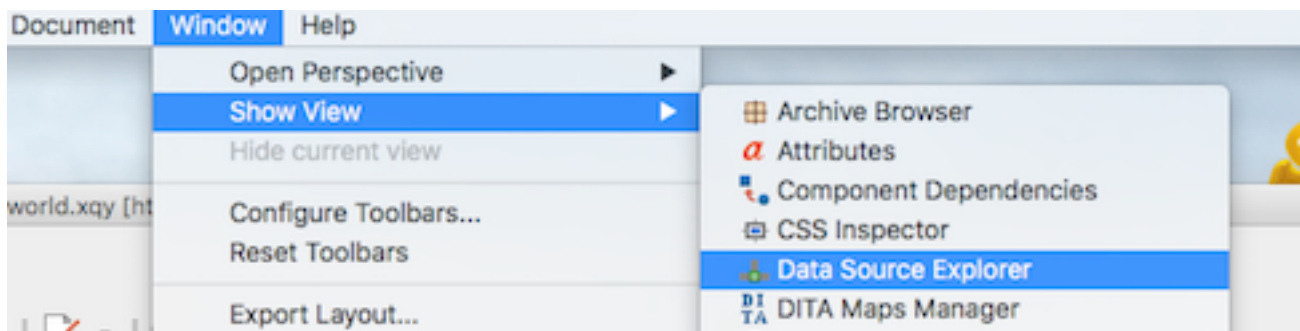
[...]

```
Document projection for file:/Users/Alex/buecher.xml
-- Input nodes 50; output nodes 27; reduction = 46%
Tree built in 3.80615ms
Tree size: 26 nodes, 58 characters, 3 attributes
Execution time: 15.83463ms
Memory used: 64339064
```

3.2.2 Connector zu Marklogic in Oxygen

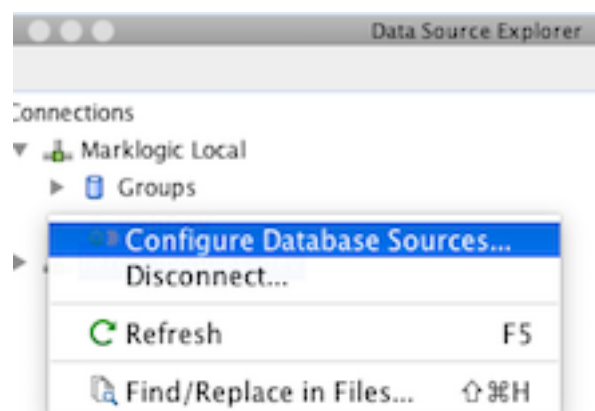
Marklogic bietet zwar auf Port 8000 per Default ein Query Console im Browser, mit der man bestimmte Sachen ausprobieren kann. Komfortabler arbeitet man aber mit einem Oxygen-Connector. Dazu öffnet man den *Data Source Explorer*:

oXygen Data Source Explorer View öffnen



und konfiguriert eine neue Datenquelle:

Neue Datenquelle in oXygen konfigurieren

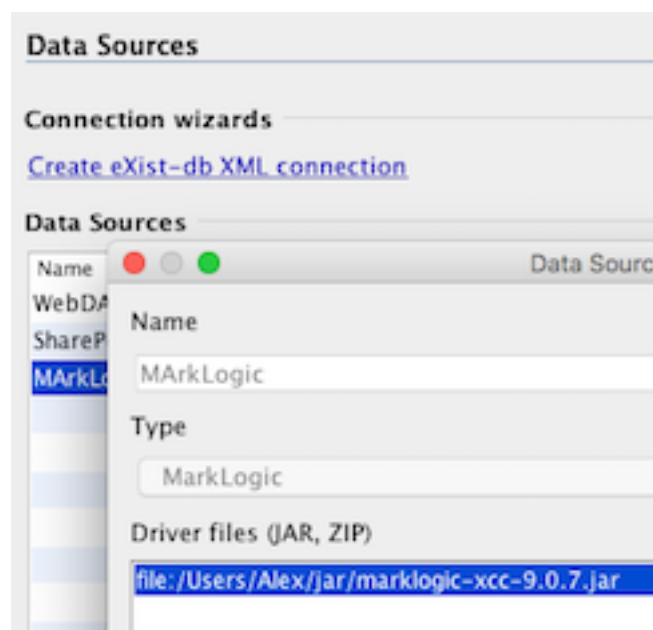


Dazu muss man den Marklogic Treiber installieren⁴¹⁾ und diesen im folgenden Screen verfügbar machen.

⚠ VORSICHT

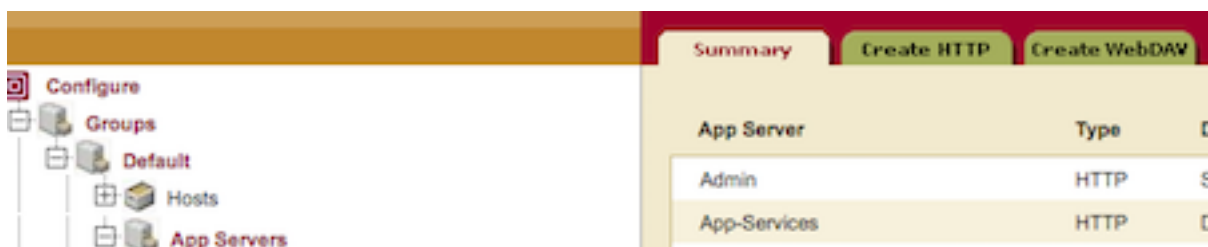
Das Jar sollte an einem soliden Ort abgespeichert werden, da hier nur ein Verweis auf diesen Ort gesetzt wird.

Marklogic Treiber auswählen



Natürlich ist auf der Serverseite auch eine Einstellung notwendig. Man wechselt als Admin in den Bereich **App Servers** und fügt einen neuen **WebDAV Server** hinzu. Ggf. muss man bei der Auswahl der Datenbank diese noch auf "automatische Directory Erzeugung" umstellen.

Wechseln in die Marklogic App Server Verwaltung



41) <https://www.oxygenxml.com/doc/versions/20.1/ug-editor/topics/configure-marklogic-datasource.html>

WebDAV in Marklogic konfigurieren

Summary Create HTTP Create WebDAV Create XDBC Create ODBC Help

ok

WebDAV server -- A WebDAV server specification.

server name MLAPIWebDav
The server name.
Required. You must supply a value for http-server-name.

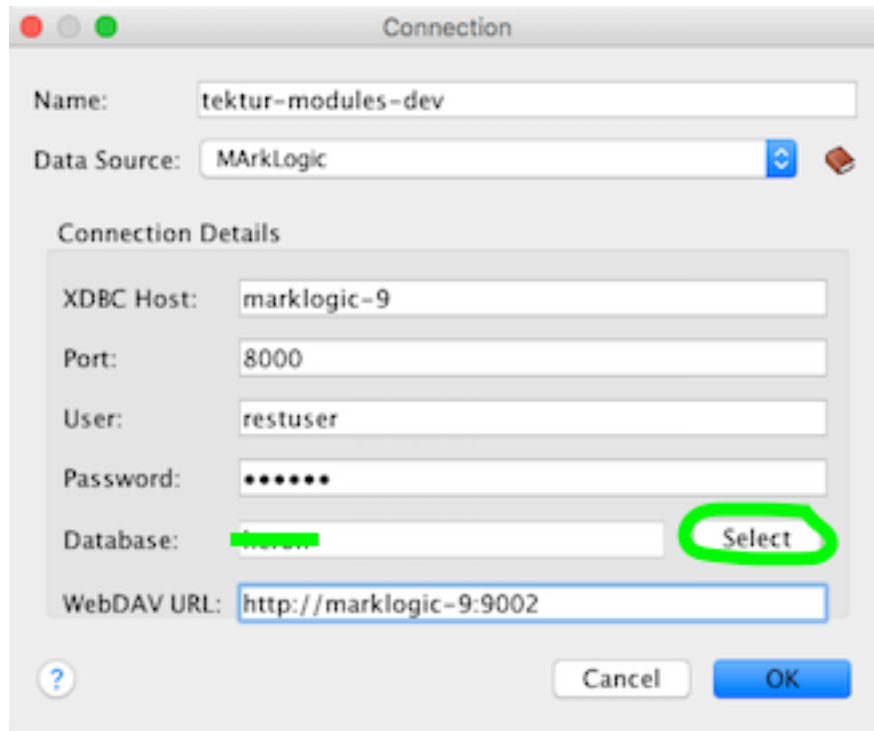
root /
The root document directory pathname.
Required. You must supply a value for root.

port 9054
The server socket bind internet port number.
Required. You must supply a value for port.

database mi-api-modules
The database name.

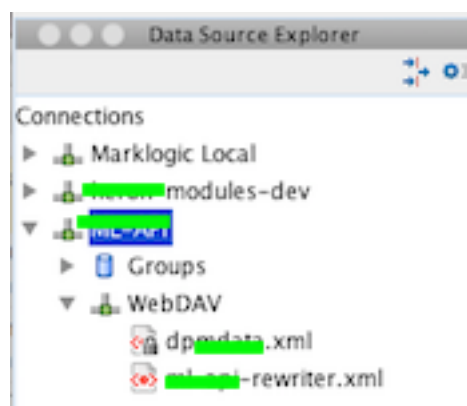
Hat man den WebDAV Server erzeugt und diesem eine bestehende oder neu angelegte Datenbank zugewiesen, dann kann man diese DB im Konfigurationsdialog der neuen Web-Dav Connection auswählen.

Konfigurieren der WebDAV Connection Einstellungen in oXygen



Schliesslich hat man im Data Source Explorer in oXygen die neue Verbindung verfügbar und kann gefühlt wie im Dateisystem mit den Files auf dem Server arbeiten.

oXygen Data Source Explorer zeigt die WebDAV Verzeichnisse auf dem Marklogic Server



3.2.3 XQuery als Programmiersprache

Erste Schritte in XQuery gehen sehr schön mit der Query Konsole auf dem Marklogic Server (Port 8000). Hat man parallel auch noch oXygen offen, kann man die Testergebnisse aus der Konsole direkt in ein XQuery Server Skript packen.

wqdwqdw

Schleifen

Die ersten 10 Dokumente auf dem Server bekommt man z.B. mit:

```
(doc())[position() lt 11]
```

Die Zahlen von 1 bis 100 mit:

```
(for $iin (1 to 100) return $i)
```

Wenn man diese beide Anweisungen untereinander in die Konsole schreibt bekommt man einen Fehler. Trennt man sie mit einem Komma - ein Tupel wird erzeugt - dann klappt es.

Dokumente in der DB anlegen

Der Befehl zum Anlegen eines Dokuments in der Marklogic DB sieht folgendermassen aus⁴²⁾:

```
xdmp:document-insert(
  "/alex-test/example-alex.xml",
  <root>Hier steht der Content</root>,
  <options xmlns="xdmp:document-insert">
    <metadata>{
      map:map() => map:with("valid-start", "2014-06-03T14:13:05.472585-07:00")
      => map:with("valid-end", "9999-12-31T11:59:59Z")
    }</metadata>
  </options>)
```

Packt man diese Instruktion in die for-Schleife oben, dann sieht das Konstrukt so aus:

```
(for $iin (1 to 10) return
  xdmp:document-insert(
    concat("/alex-test/example-alex-", $i, ".xml"),
    <root>Hier steht der Content {$i}</root>,
    <options xmlns="xdmp:document-insert">
      <metadata>{
        map:map() => map:with("valid-start", "2014-06-03T14:13:05.472585-07:00")
        => map:with("valid-end", "9999-12-31T11:59:59Z")
      }</metadata>
    </options>
  )
)
```

Dokumente kann man einer Collection zuweisen, um sie leichter finden und auswerten zu können. Das geht mit dem folgenden Befehl⁴³⁾ und diesem Schnippsel:

```
let $root :=
  <mein-test>
    <id>{$id}</id>
    <content>Hallo Welt!</content>
  </mein-test>,
  $options :=
  <options xmlns="xdmp:document-insert">
42) https://docs.marklogic.com/xdmp:document-insert
43) https://docs.marklogic.com/xdmp:document-add-collections
```

```

<permissions>{xdmp:default-permissions()}</permissions>
<collections>
  <collection>/alex-test</collection>
</collections>
</options>,
$fname := concat('/', $id, '_', '.xml'),
$td := xdmp:document-insert($fname, $root, $options)
return
[...]
```

Die Dokumente, die mit der Collection alex-test getaggt wurden, kann man sich mit der folgenden Schleife ausgeben lassen:

```

for $xin collection("/alex-test")
return
  fn:document-uri($x)
```

3.2.3.1 Funktionen und Module

Funktionen

Um bestimmte Abschnitte des XQuery Programm wiederverwendbar zu machen, stehen Funktionsdeklarationen zur Verfügung. Eine einfache Funktion wäre z.B. diese hier:

```

declare function local:wrap-header($json) {
  xdmp:add-response-header("Pragma", "no-cache"),
  xdmp:add-response-header("Cache-Control", "no-cache"),
  xdmp:add-response-header("Expires", "0"),
  xdmp:set-response-content-type('text/json; charset=utf-8'),
  $json
};
```

Sie wickelt um einen JSON String eine passende Header Information.

Damit die Funktion eingebunden werden kann, muss ein passender Namespace deklariert werden:

```

declare namespace local = 'local:';
```

Nicht nur bzgl. Wiederverwendbarkeit sind Funktionen praktisch, sondern auch um ganz elementare Konstrukte, wie `while...do` Schleifen, zu realisieren.

Dazu nutzt man, wie in der funktionalen Programmierung üblich, die Rekursion:

```

declare function local:ist-letzter-wert-in-kette($glied) {
  let $wert := local:komplizierte-berechnung($glied),
  $naechstes-glied := local:komplizierte-berechnung-der-position($glied),
  return
  if ($naechstes-glied and not($wert = 'foobar')) then
    local:durchlaufe-kette($naechstes-glied)
  else
    $wert = 'foobar'
};
```

In diesem kleinen Schnippel sind schon einige Besonderheiten von XQuery zu sehen. Variablenzuweisungen geschehen mit einem Doppelpunkt, Vergleiche dagegen nur mit einem einfachen "=". Statements werden mit einem Komma getrennt.

Module

Um eine XQuery Anwendung zu modularisieren, können einzelne Skripte in Module ausgelagert werden. Ein Modul, z.B. `common.xqy`, wird dabei über einen eigenen Namespace deklariert:

```
module namespace common = "https://www.tekturcms.de/common";
```

Dieses Modul kann dann in anderen Skripten eingebunden werden:

```
import module namespace common = "https://www.tekturcms.de/common" at "common.xqy";
```

Funktionen und Variablen werden dann mit dem Namespace geprefixt aufgerufen:

```
Funktionsaufruf: common:wrap-response-header(...)
Variablenauswertung: $common:collection-books
```

3.2.3.2 If..then..else Ausdrücke

In nicht-funktionalen Programmiersprachen sind die Schlüsselwörter `if` und `then` dazu da, um dem Compiler oder Interpreter mitzuteilen, dass eine bedingte Anweisung ausgewertet werden soll.

Was für den Nicht-funktionalen Programmierer etwas befremdlich erscheint, ist der Umstand, dass in XQuery `if..then` als Ausdrücke ausgewertet werden.

Das ist einerseits sehr praktisch, weil es richtig angewandt den Code verkürzt und damit das Wesentliche herausstellt, kann aber auch weiter zur allg. Verwirrung bzgl. des kryptischen XQuery Codes beitragen.

XQuery Server Application

Betrachten wir ein einfaches Beispiel: Wir generieren auf einer Marklogic-Webapp eine JSON Response. Da wir diesen Mechanismus an mehreren Stellen im Code einsetzen, empfiehlt es sich das Rendern des Headers in eine Funktion auszulagern.

```
declare function common:wrap-response($json)
{
  xdmp:add-response-header("Pragma", "no-cache"),
  xdmp:add-response-header("Cache-Control", "no-cache"),
  xdmp:add-response-header("Expires", "0"),
  xdmp:set-response-content-type('text/json; charset=utf-8'),
  $json
};
```

In unserem Request-Handler wird je nach Auswertung einer Variablen eine bedingte Anweisung ausgeführt, diese sieht bspw. so aus:

```
let $name := xdmp:get-request-field('name'),
    $is-afternoon := xs:time(current-dateTime()) gt xs:time('12:00:00')
return
```

```

if ($is-afternoon) then
  common:wrap-response(xdmp:unquote(concat('{"greeting":"Good Afternoon! ', $name, '!"'}'))
else
  common:wrap-response(xdmp:unquote(concat('{"greeting":"Good Morning! ', $name, '!"'}'))

```

Als prozeduraler Programmierer wäre ich mit diesem Switch voll und ganz zufrieden, der funktionale Programmier erkennt aber sofort einen Optimierungsbedarf.

Da es sich bei der bedingten Anweisung auch um einen Ausdruck handelt, der `[[code:true]]` oder `[[code:false]]` zurückgibt, können wir die gleichen Funktionsaufrufe herausziehen:

```

common:wrap-response(xdmp:unquote(
  let $is-afternoon := xs:time(current-dateTime()) gt xs:time('12:00:00')
  return
    if ($is-afternoon) then concat('{"greeting":"Good Afternoon! ', $name, '!"'}')
    else concat('{"greeting":"Good Morning! ', $name, '!"'}')
)

```

Hier wird der abstrakt denkende Programmierer aber einwenden, dass eine abstrakte Logik nicht in eine Low-Level Funktion, wie `xdmp:unquote` gewrapped werden sollte.

Das stimmt - und mehr noch, die Maskierung mit `xdmp:unquote` sollte auch noch in unsere Funktion gepackt werden. So dass der Code schliesslich so aussehen würde:

```

declare function common:render-response($json)
{
  xdmp:add-response-header("Pragma", "no-cache"),
  xdmp:add-response-header("Cache-Control", "no-cache"),
  xdmp:add-response-header("Expires", "0"),
  xdmp:set-response-content-type('text/json; charset=utf-8'),
  xdmp:unquote($json)
};
common:render-response(
  let $is-afternoon := xs:time(current-dateTime()) gt xs:time('12:00:00')
  return
    if ($is-afternoon) then concat('{"greeting":"Good Afternoon! ', $name, '!"'}')
    else concat('{"greeting":"Good Morning! ', $name, '!"'}')
)

```

Sicherlich lässt sich darüber streiten, ob nun der funktionale Ansatz besser lesbar ist als der prozedurale ganz oben.

Ich denke jeder Programmierer hat hier seinen eigenen, individuellen und bewährten Programmierstil entwickelt, den er auch beibehalten sollte.

3.2.4 SQL Views in MarkLogic

Es macht nicht immer Sinn über eine Baumstruktur zu suchen. Obwohl das in einer XML Datenbank rasend schnell geht, weil jeder Knoten des Baums initial in einen Index aufgenommen wird. So gibt es doch Anwendungsfälle bei denen man lieber eine relationale Sicht auf die Daten hätte.

In MarkLogic heisst die Lösung dazu SQL Views.

Bspw. benötigt man eine relationale Sicht auf die Daten, wenn über verschiedene Datensätze ein Report generiert werden soll.

Nehmen wir an es gibt im Datenbestand die folgenden Dokumente:

```
<k:kunde>
  <k:id>1</k:id>
  <k:name>Alex</k:name>
  <k:eMail>tekturcms@gmail.com</k:eMail>
</k:kunde>

<k:kunde>
  <k:id>2</k:id>
  <k:name>Horst</k:name>
  <k:eMail>horst@horst.de</k:eMail>
</k:kunde>

<k:kunde>
  <k:id>3</k:id>
  <k:name>Gundula</k:name>
  <k:eMail>gundl@gundula.de</k:eMail>
</k:kunde>

<b:bestellung>
  <b:id>1</b:id>
  <b:datum>02.01.2019</b:datum>
  <b:preis>99.90</b:preis>
  <kunde-id>2</kunde-id>
</b:bestellung>

<b:bestellung>
  <b:id>2</b:id>
  <b:datum>03.01.2019</b:datum>
  <b:preis>68.90</b:preis>
  <b:kunde-id>1</b:kunde-id>
</b:bestellung>
```

Will man sich alle Kunden anzeigen lassen, die eine Bestellung abgegeben haben - das sind Alex und Horst - so würde man bei einem relationalen Ansatz einen JOIN verwenden, so wie:

```
SELECT name, datum, preis
FROM kunden k
INNER JOIN bestellungen b
ON k.id = b.kunde_id
```

In einer relationalen Sicht würde uns das dann die folgende Tabelle liefern:

```
name, datum, preis
Alex, 03.01.2019, 68.90
Horst, 02.01.2019, 99.90
```

Um für MarkLogic eine SQL View zu definieren verwendet man einen Mechanismus, der da heisst: Template Driven Extraction⁴⁴⁾

44) <https://docs.marklogic.com/guide/app-dev/TDE>

Dazu werden Templates in XML deklariert und in die Template Collection eingefügt. Für unser obiges Beispiel würden wir zwei Templates brauchen, die so aussehen:

```
xquery version "1.0-ml";

import module namespace tde = "http://marklogic.com/xdmp/tde"
      at "/MarkLogic/tde.xqy";

let $sql-view-name := 'kunden-view.xml'
let $sql-view := <template xmlns="http://marklogic.com/xdmp/tde">
  <path-namespaces>
    <path-namespace>
      <prefix>k</prefix>
      <namespace-uri>https://tekturcms.de/schema/kunde/1.0</namespace-uri>
    </path-namespace>
  </path-namespaces>
  <context>/k:kunde</context>
  <collections>
    <collections-and>
      <collection>/kunden</collection>
    </collections-and>
  </collections>
  <rows>
    <row>
      <schema-name>kunden_schema</schema-name>
      <view-name>kunden_view</view-name>
      <columns>
        <column>
          <name>id</name>
          <scalar-type>string</scalar-type>
          <val>k:id</val>
          <nullable>true</nullable>
        </column>
        <column>
          <name>datum</name>
          <scalar-type>string</scalar-type>
          <val>k:datum</val>
          <nullable>true</nullable>
        </column>
        <column>
          <name>eMail</name>
          <scalar-type>string</scalar-type>
          <val>k:eMail</val>
          <nullable>true</nullable>
        </column>
      </columns>
    </row>
  </rows>
</template>
return(
  tde:template-insert(concat('/templates/', $sql-view-name),
    $sql-view, xdmp:default-permissions())
)
```

und analog für die Bestellungen:

```
[...]
  <rows>
    <row>
```

```

<schema-name>bestellungen_schema</schema-name>
<view-name>bestellungen_view</view-name>
<columns>
  <column>
    <name>id</name>
    <scalar-type>string</scalar-type>
    <val>b:id</val>
  </column>
  <column>
    <name>datum</name>
    <scalar-type>string</scalar-type>
    <val>b:datum</val>
  </column>
  <column>
    <name>preis</name>
    <scalar-type>string</scalar-type>
    <val>b:preis</val>
  </column>
  <column>
    <name>kunde_id</name>
    <scalar-type>string</scalar-type>
    <val>b:kunde-id</val>
  </column>
</columns>
</row>
[...]
```

In XQuery eingebunden könnte man dann die definierten SQL Views mit dem folgenden Befehl abfragen:

```

xdmp:sql("SELECT name, datum, preis FROM kunden_view k
        INNER JOIN bestellungen_view b ON k.id = b.kunde_id")
```

Folgende ist das komplette Beispiel für eine MarkLogic XQuery Konsolensitzung abgebildet ...

```

xquery version "1.0-ml";

declare namespace k = 'http://www.tekturcms.de/kunden';
declare namespace b = 'http://www.tekturcms.de/bestellungen';

import module namespace tde = "http://marklogic.com/xdmp/tde" at "/MarkLogic/tde.xqy";

declare function local:loadKunde($id, $name, $eMail)
{
  let $root :=
    <k:kunde>
      <k:id>{ $id }</k:id>
      <k:name>{ $name }</k:name>
      <k:eMail>{ $eMail }</k:eMail>
    </k:kunde>,
    $options :=
    <options xmlns="xdmp:document-insert">
      <permissions>{ xdmp:default-permissions() }</permissions>
      <collections>
        <collection>/kunden</collection>
      </collections>
    </options>,
    $doc := xdmp:document-insert($root, $options)
  return $doc
}
```

```

    $fname := concat('/kunden/', $id, ".xml")
    return xdmp:document-insert($fname, $root, $options)
};

declare function local:loadBestellung($id, $datum, $preis, $kunde-id)
{
    let $root :=
        <b:bestellung>
            <b:id>{ $id }</b:id>
            <b:datum>{ $datum }</b:datum>
            <b:preis>{ $preis }</b:preis>
            <b:kunde-id>{ $kunde-id }</b:kunde-id>
        </b:bestellung>,
        $options :=
        <options xmlns="xdmp:document-insert">
            <permissions>{ xdmp:default-permissions() }</permissions>
            <collections>
                <collection>/bestellungen</collection>
            </collections>
        </options>,
        $fname := concat('/bestellungen/', $id, ".xml")
    return xdmp:document-insert($fname, $root, $options)
};

declare function local:insertKundenSchema()
{
    let $sql-view-name := 'kunden-view.xml',
        $sql-view := <template xmlns="http://marklogic.com/xdmp/tde">
            <path-namespaces>
                <path-namespace>
                    <prefix>k</prefix>
                    <namespace-uri>http://www.tekturcms.de/kunden</namespace-uri>
                </path-namespace>
            </path-namespaces>
            <context>/k:kunde</context>
            <collections>
                <collections-and>
                    <collection>/kunden</collection>
                </collections-and>
            </collections>
            <rows>
                <row>
                    <schema-name>kunden_schema</schema-name>
                    <view-name>kunden_view</view-name>
                    <columns>
                        <column>
                            <name>id</name>
                            <scalar-type>string</scalar-type>
                            <val>k:id</val>
                        </column>
                        <column>
                            <name>name</name>
                            <scalar-type>string</scalar-type>
                            <val>k:name</val>
                        </column>
                        <column>
                            <name>eMail</name>
                            <scalar-type>string</scalar-type>
                            <val>k:eMail</val>
                        </column>
                    </columns>
                </row>
            </rows>
        </template>
    xdmp:insert($sql-view-name, $sql-view)
};

```

```

</template>
return
  tde:template-insert(concat('/templates/',
                             $sql-view-name), $sql-view, xdmp:default-permissions())
};

declare function local:insertBestellungenSchema()
{
  let $sql-view-name := 'bestellungen-view.xml',
      $sql-view := <template xmlns="http://marklogic.com/xdmp/tde">
    <path-namespaces>
      <path-namespace>
        <prefix>b</prefix>
        <namespace-uri>http://www.tekturcms.de/bestellungen</namespace-uri>
      </path-namespace>
    </path-namespaces>
    <context>/b:bestellung</context>
    <collections>
      <collections-and>
        <collection>/bestellungen</collection>
      </collections-and>
    </collections>
    <rows>
      <row>
        <schema-name>bestellungen_schema</schema-name>
        <view-name>bestellungen_view</view-name>
        <columns>
          <column>
            <name>id</name>
            <scalar-type>string</scalar-type>
            <val>b:id</val>
          </column>
          <column>
            <name>datum</name>
            <scalar-type>string</scalar-type>
            <val>b:datum</val>
          </column>
          <column>
            <name>preis</name>
            <scalar-type>string</scalar-type>
            <val>b:preis</val>
          </column>
          <column>
            <name>kunde_id</name>
            <scalar-type>string</scalar-type>
            <val>b:kunde-id</val>
          </column>
        </columns>
      </row>
    </rows>
  </template>
  return
    tde:template-insert(concat('/templates/',
                             $sql-view-name), $sql-view, xdmp:default-permissions())
};

local:loadKunde("1","Alex","tekturcms@gmail.com"), local:loadKunde("2","Horst","horst@horst.de"), local:
xdmp:sql("SELECT name, datum, preis FROM kunden_view k INNER JOIN
  bestellungen_view b ON k.id = b.kunde_id")

```

... mit einer schönen tabellarischen Ausgabe im unteren Panel der Query Konsole - oder als JSON:

```
[
  [
    "k.name",
    "b.datum",
    "b.preis"
  ],
  [
    "Alex",
    "03.01.2019",
    "68.90"
  ],
  [
    "Horst",
    "02.01.2019",
    "99.90"
  ]
]
```

3.3 Testing

In diesem Kapitel werden einige ausgewählte Themen zum Testing von XSLT und XQuery Programmen dargestellt. Dabei geht es weder um Vollständigkeit noch um die beste/eleganteste Lösung, sondern eher um die Vorstellung eines Anwendungsszenarios mit einem potentiellen Lösungsansatz.

3.3.1 Validierung mit Schematron

Um die Korrektheit einer XML Instanz zu prüfen, gib es verschiedene Schemata, wie XSD, RNG oder DTD, welche der Parser beim Aufbau des DOM Baums heranzieht. Eine Validierung mit Apache Xerces könnte beispielsweise als Java Code folgendermaßen angestossen werden:

```
URL schemaFile = new URL("http://host:port/filename.xsd");
Source xmlFile = new StreamSource(new File("web.xml"));
SchemaFactory schemaFactory = SchemaFactory
    .newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
try {
    Schema schema = schemaFactory.newSchema(schemaFile);
    Validator validator = schema.newValidator();
    validator.validate(xmlFile);
    System.out.println(xmlFile.getSystemId() + " is valid");
} catch (SAXException e) {
    System.out.println(xmlFile.getSystemId() + " is NOT valid reason:" + e);
} catch (IOException e) {
```

Schema Dateien können aber auch in XML Editoren eingebunden werden, um schon während der Eingabe der XML Instanz die Korrektheit zu überprüfen.

Das geht einerseits über die Angabe des Doctypes in der XML Instanz, andererseits bieten auch alle Editoren die Möglichkeit ein bestimmtes Schema explizit auszuwählen, um gegen dieses auf Anforderung zu validieren.

Schematron ist XSLT

Gilt es komplexere Businessregeln zu überprüfen, die über Syntax-, Konsistenz- und einfache Korrektheitschecks hinausgehen, empfiehlt sich eine Validierung mit Schematron Regeln.

Bei einer Schematron Validierung wird eine XML Instanz mit Hilfe eines automatisch generierten XSLT Stylesheets überprüft. Dieses kontextabhängige Stylesheet wird aus einer in der Schematron Syntax vom Autor verfassten Regelbasis, die wiederum in XML vorliegt, über ein zweites XSLT Stylesheet generiert - Dieses zweite XSLT Stylesheet ist sozusagen das eigentliche Schematron Programm.

Das folgende Diagramm veranschaulicht die Vorgehensweise anhand eines Filter-Szenarios, bei dem ein XML Dokument mit einigen ungültigen Passagen in eine gefilterte Darstellung überführt wird.

Einfacher Batch-Prozess zur Validierung mit Schematron und anschließendem Filtern der Ergebnisse

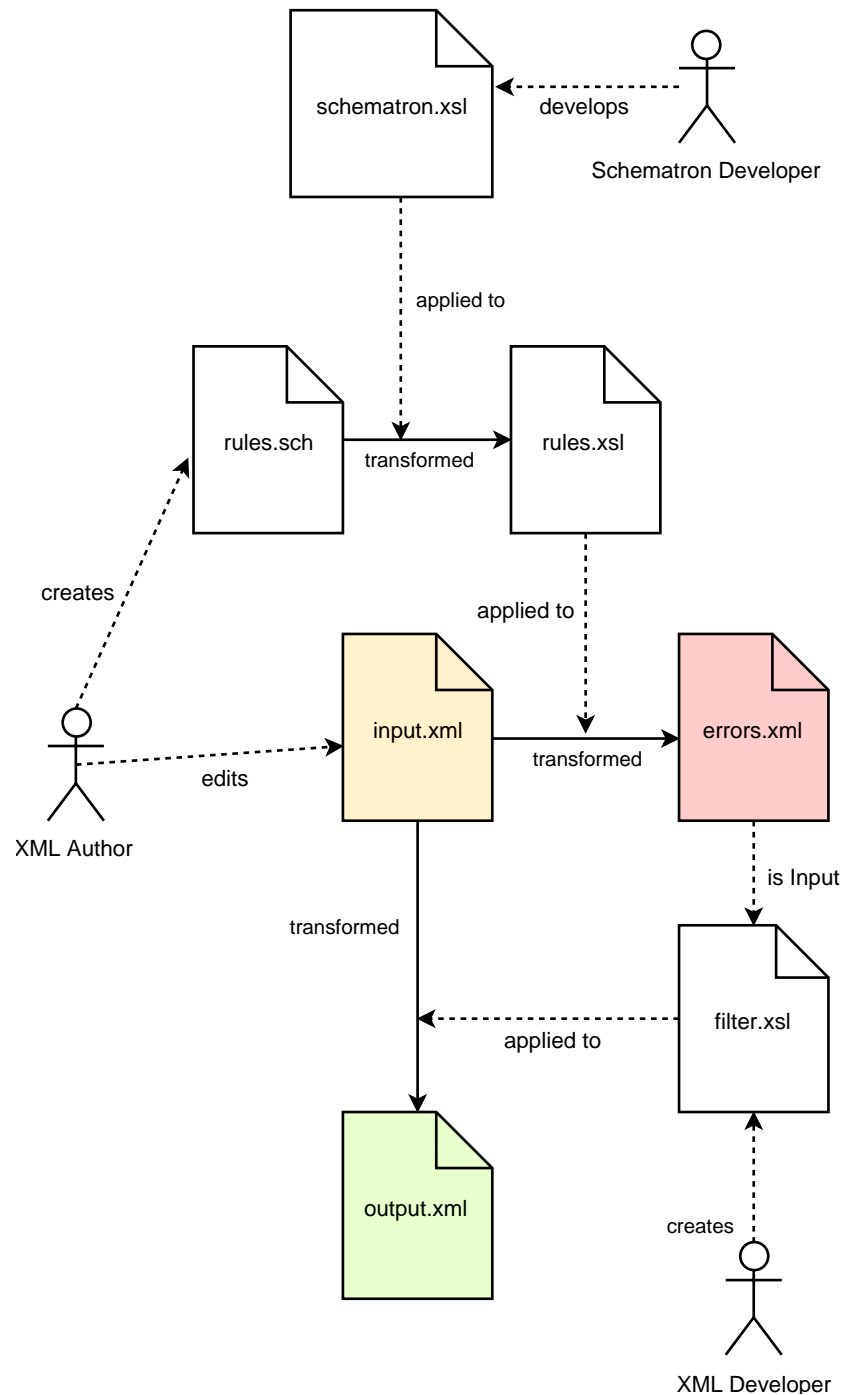


Bild 5: Schematron Validierung mit Filter

CLI Verwendung

Zu finden ist das Schematron Programm auf Github⁴⁵⁾. Dieses Repo ist etwas unübersichtlich. Der relevante Teil des Sourcecodes befindet sich unter: **schematron/code**

Um die Schematron XSLT Skripte in eine eigene XSLT Anwendung per Kommandozeile einzubinden, könnte man folgendermassen vorgehen:

- Im eigenen GIT Projekt das Schematron Projekt als Submodule referenzieren.
- Eine Regelbasis anlegen, beispielsweise **\$project_name.sch**.
- Zwei Batch-Skripte anlegen, beispielsweise **generate_schema.sh** und **validate.sh**.

Mittels des Skripts **generate_schema.sh** wird aus der Schematron Regelbasis das Schematron XSLT Stylesheet generiert. Der Inhalt dieser Batchdatei könnte zum Beispiel so aussehen:

```
saxon $script_dir/$project_name_validation.sch $script_dir/schematron/iso_dsd_include.xml
| \ saxon -s:- $script_dir/schematron/iso_abstract_expand.xml | \
saxon -s:- $script_dir/schematron/iso_svrl_for_xslt2.xml \
generate-fired-rule=false > $script_dir/$project_name_validation.xml
```

Der Prozess zum Erzeugen des projektspezifischen Validerungs-XSLT-Skripts ist dreistufig und wird über die folgenden XSLT Schritte abgearbeitet.

- iso_dsd_include.xml
- iso_abstract_expand.xml
- iso_svrl_for_xslt2.xml

Herauszufinden, was in diesen Skripten passiert, sei dem geneigten Leser selbst überlassen. Uns interessiert an dieser Stelle nur das Resultat, nämlich das XSLT Stylesheet **\$project_name_validation.xml**.

Dieses Skript wird in der Batchdatei **validate.sh** aufgerufen:

```
saxon $xml_instance_to_check.xml $script_dir/$project_name_validation.xml \
> $validation-result.xml
```

Die Ausgabe dieses Prüfprozesses ist eine XML Datei mit den Fehlern in der Eingabe-XML-Instanz, die weiterverarbeitet werden kann, beispielsweise als Filterkriterium für einen nachfolgenden Prozessschritt. Ihr Inhalt dieser Datei sieht z.B. wie folgt aus:

```
<svrl:schematron-output xmlns:svrl="http://purl.oclc.org/dsdl/svrl" [...]
  <svrl:active-pattern document="file:/Users/alex/xml_instance_to_check.xml"
    id="default" name="default"/>
  <svrl:failed-assert test="count(key('unique-ids', current()))=1">
    <svrl:text>ID is not unique!</svrl:text>
    <svrl:diagnostic-reference diagnostic="default">
      <bk:id xmlns:bk="http://tekturcms/namespaces/book">1234-5678-9</my:id>
    </svrl:diagnostic-reference>
  </svrl:failed-assert>
  [...]
```

45) <https://github.com/Schematron/schematron>

Neben den **svrl:failed-assert** Elementen, die angeben, was bei der überprüften XML-Instanz fehlgeschlagen ist, gibt es auch die Möglichkeit sich positive Ergebnisse anzeigen zu lassen - über das Element **svrl:successful-report**.

Konkret bedeutet das obige XML Schnipsel, dass unsere **id** mit dem Wert **1234-5688-9** im geprüften XML Dokument nicht eindeutig ist. Die Schematron Regelbasis, die wir zur Überprüfung angegeben haben, sieht so aus:

```
<schema xmlns:sch="http://purl.oclc.org/dsdl/schematron" [...]
  <xsl:key name="unique-ids" match="bk:id" use="."/>
  <sch:let name="date-regex" value="'^((19|2[0-9])[0-9]{2})-(0[1-9]|1[012])
                                -(0[1-9]|12[0-9]|3[01])$'"/>

  <sch:pattern id="default">
    <sch:rule context="book">
      <sch:assert id="check-book-id" role="error" test="count(key('unique-ids', bk:id))=1"
        diagnostics="default">ID is not unique!</sch:assert>
      <sch:assert id="check-book-published" role="error"
        test="matches(bk:published,$date-regex)"
    </sch:rule>
  </sch:pattern>
  <sch:diagnostics>
    <sch:diagnostic id="default">
      <xsl:element name="bk:id">
        <xsl:value-of select="bk:id"/>
      </xsl:element>
    </sch:diagnostic>
  </sch:diagnostics>
```

Neben der "successful" und "failed" Regeln ist auch die Deklaration von Funktionen und Variablen im Body der Regelbasis erlaubt. Dies ermöglicht komplexe Bedingungen, bespw. durch das Nachschlagen in einer Lookup-Tabelle abzu prüfen.

3.3.2 Erste Schritte mit Xspec

XSpec ist ein **Test-Framework**⁴⁶⁾ für XSLT, XQuery und Schematron. Um beispielsweise komplexe Schematron Regeln zu testen, hinterlegt man in einem **Test-Szenario** Erwartungswerte für positive und negative Testfälle in Form von XML Schnipseln.

```
<test-szenario>
  <testfall>
    <personen>
      <person>
        <vorname>Horst</vorname>
        <nachname>Schlämmer</nachname>
        <gewicht>100</gewicht>
      </person>
      <person>
        <vorname>Gundula</vorname>
        <nachname></nachname>
        <gewicht>60</gewicht>
      </person>
    </personen>
  </testfall>
```

46) <https://github.com/xspec>

```
</test-szenario>
```

in einer XSpec Datei *. werden **Assert- und Not-Assert-Methoden** deklariert:

```
<x:description xslt-version="2.0" xmlns:x="http://www.jenitennison.com/xslt/xspec"
  schematron="test.sch">
  <x:scenario label="ALL">
    <x:context href="test.xml"/>
    <x:expect-not-assert id="person-nachname-rule" location="//person[1]/nachname"/>
    <x:expect-assert id="person-nachname-rule" location="//person[2]/nachname"/>
  </x:scenario>
</x:description>
```

Grds. bedeutet ein Assert, dass das Mapping zwischen tatsächlichem Wert und Erwartungswert des Testfalls positiv erfüllt ist. Beim Not-Assert ist das Gegenteil der Fall. Im obigen Beispiel reichen zwei Regeln, um den Testfall vollständig abzudecken.

Wenn man Schematron Regeln mit Hilfe von XSpec testen will, dann muss man ein bisschen um die Ecke denken. Denn auch diese Regeln werden mittels Assert und Not-Assert modelliert.

```
<sch:schema xmlns:sch="http://purl.oclc.org/dsdl/schematron"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform" queryBinding="xslt2">
  <sch:pattern id="main">
    <sch:rule context="nachname">
      <sch:assert id="person-nachname-rule" role="error" test="normalize-space(.)">
        Der Nachname der Person mit ID: <sch:value-of select="@id"/> fehlt!
      </sch:assert>
    </sch:rule>
  </sch:pattern>
</sch:schema>
```

In der Schematron-Regel wird zugesichert (Assert), dass jede Person einen Nachnamen hat.

Hat sie keinen Nachnamen so wird der Bericht zum Fehlerfall in die Schematron Ergebnisdatei geschrieben. Diese Datei wertet nun Xspec aus.

► *Erscheint ein Fehler (= das Feld **nachname** ist leer), so greift bei Xspec die Assert-Regel! Das ist die umgekehrte Logik zu den Schematron Regeln.*

Als Eselsbrücke kann man ein Assert in der Xspec Datei gleichsetzen mit **Appear** und ein Not-Assert mit **Not-Appear**.

Ein Assert sichert also zu, dass sich ein Fehlerbericht in der Schematron Ergebnisdatei zum Testfall befindet. Ein Not-Assert sichert zu, dass sich kein Fehlerbericht befindet.

Wie man sich leicht vorstellen kann, sind Assert-Regeln in diesem Fall leicht zu finden, dazu muss man nur die Schematron Testregeln ins Leere zeigen lassen. Alles ist grün und alles ist gut - dem Augenschein nach.

4 Zusätzliches Know-How

Unsortierte Notizen, die für jeden XSLT-Programmierer interessant sein könnten.

4.1 GIT Einmaleins

Im Fachbereich Technische Dokumentation wird neue Technologie ein bisschen langsamer adaptiert, als in anderen Disziplinen der IT - schliesslich muss ja erst alles dokumentiert werden ...

Bei Versionsverwaltungssystemen ist das nicht anders. Doch inzwischen hat auch GIT in manchen Entwicklerbüros Einzug gehalten und verdrängt Subversion.

Ein Grund an dieser Stelle einmal zumindest die wichtigsten Befehle aufzulisten - Note to self:

```
GIT COMMANDS

CLONE
=====
git clone https://github.com/libgit2/libgit2
SIMPLE
=====
git init
git add *
git commit -m "Form validation added"
[changes]
git add [changed files] or git commit -a
git diff --cached
git branch experimental
git checkout experimental
git commit -a -m "only in the experimental branch"
git checkout master
git merge experimental
git branch -d experimental

COLLABORATION
=====
push new branch git push -u origin newLocalBranch
bob$ git clone /home/alice/project myrepo
(edit files)
bob$ git commit -a
alice$ cd /home/alice/project
alice$ git pull /home/bob/myrepo master (fetch & merge)
(fetch and not merge)
alice$ git fetch /home/bob/myrepo master
alice$ git log -p HEAD..FETCH_HEAD
(visualization only)
gitk HEAD..FETCH_HEAD
gitk HEAD...FETCH_HEAD
_
git log
git show ea14 (firs chars is enough)
git grep "headline_font_size" v2.5

PULL REQUESTS
=====
```

```
(checkout branch)
git pull origin simple-examples
(changes)
git push origin simple-examples
Create a pull request using the form on the Github page and assign it to a Reviewer
```

MERGE CONFLICTS

=====

```
git pull origin master
=> merge conflicts
```

TROUBLE SHOOTING

=====

```
git reset --hard origin/master
git tree
git status
git checkout 118886ee3f06738b53f089433078d35f4d70a8f9 vendor/error-report.xml
git log --diff-filter=D --summary
```

4.2 XML Editoren

Der XSLT Stylesheet-Entwickler wird sich gewöhnlich mit Eingabedaten beschäftigen, die entweder automatisch mittels irgendeines Prozesses erzeugt wurden, oder die durch einen menschlichen Autor mit einem XML Editor eingegeben wurden.

Aus diesem Grund ist es ganz nützlich, die wichtigsten Editoren zu kennen. Wir unterscheiden zwischen Desktopapplikationen und Webanwendungen. Ausserdem unterscheiden wir noch ob der Editor WYSIWIG (**W**hat **Y**ou **S**ee **I**s **W**hat **Y**ou **G**et) oder WYSIWYM (**W**hat **Y**ou **S**ee **I**s **W**hat **Y**ou **M**ean) unterstützt oder eine Mischung aus beidem darstellt.

**WYSIWYM
Desktop**

Editor	Beschreibung
XMetal ^[XM]	XMetal ist wahrscheinlich der am weitesten verbreitete reine WYSIWYM Editor. Er hat Schnittstellen zu COM und Java und kann daher in eigene CMS integriert werden.
Arbortext XML Editor ^[EP]	Arbortext XML Editor, früher bekannt als EPIC ist sehr betagt. Ich hatte damit im Bereich Luftfahrt/Verteidigung zu tun. Bekannter-massen ist sein Tabelleneditor etwas buggy.

**WYSIWYG
Desktop**

XMetal kann so konfiguriert werden, dass bei einer einfachen DTD der Content Bereich wie Word aussieht. Auch Code Editoren, wie OxygenXML bieten diese Möglichkeit. Das Key-Handling bei dieser Variante zeigt aber schnell, dass die UX noch weit von herkömmlichen Textverarbeitungssystem, wie Word oder OpenOffice entfernt ist.

**WYSIWYM
Online**

Editor	Beschreibung
Oxygen XML WebAuthor ^[OX]	Dieser Online-Editor verwendet auf der Serverseite dieselbe Logik, wie das Desktop Programm des Herstellers. Das führt dazu, dass bei jedem Tastendruck eine Verbindung zum Server aufgebaut wird, und die Verarbeitung langsam werden kann. Zum Betrieb und bzgl. Customizing ist einschlägiges Java-Know-How erforderlich.
FontoXML ^[FX]	FontoXML sieht schon fast aus wie Word. Neben der WYSIWYG/M Darstellung, kann auch die XML Struktur in einem Seitenpanel angezeigt werden.
XEditor ^[XE]	Xeditor benutzt XSLT Transformationen, um aus der Eingabe die Editoransicht zu generieren. Beim Abspeichern wird der umgekehrte Weg bestritten. Das mag zwar auf den ersten Blick etwas holprig erscheinen, wie aber auch Tektur beweist, funktioniert das ganze recht gut und schnell.
Xopus ^[XO]	Xopus ist wohl der älteste web-basierte XML Editor. Ich hatte damit schon 2008 zu tun, als er für ein Redaktionssystem evaluiert wurde. Wir haben uns dann für eine eigene nicht-generische Lösung basierend auf dem Webeditor CKEditor entschieden.

Das Customizing dieser Editoren erfordert einen sehr hohen Aufwand. Es müssen diverse Ressourcen angepasst werden, wie XSLT Skripte, XSD Schemas, CSS und Javascript. Das Schema wird meist über Kommandozeilentools in eine JS Repräsentation überführt.

Aus diesem Grund bieten einige Hersteller spezielle Schulungen an, wo man die Bedienung erlernen kann. Aus meiner Sicht ist das Problem "Webbasierter XML Editor" weltweit noch nicht ausreichend gelöst.

[XM] <https://xmetal.com/>

[EP] <https://www.ptc.com/en/products/service-lifecycle-management/arbortext/editor>

[OX] <https://www.oxygenxml.com/oxygen-xml-web-author/app/oxygen.html>

[FX] <https://www.fontoxml.com/>

[XE] <http://www.xeditor.com/portal>

[XO] <http://xopusfiddle.net/VT7T/3/>

Die Kosten für den Betrieb rangieren um die 1000 EUR monatl. für ein 20 Benutzer-Setup.

Glossar

C	Core-Stylesheet	In einem Stylesheet-Projekt bezeichnet das Core-Stylesheet eine bereits ausgiebig getestete Variante, die mittels Sub-Stylesheet unter Ausnutzung der XSLT Import Präferenz überschrieben wird.
D	DITA	DITA ist ein Informationsmodell für die Technische Dokumentation.
P	Parameterisierung	Bei der Parameterisierung wird ein bestehendes Stylesheet mit Parametern versehen, um für möglichst viele Produktvarianten und Ausgabeformate die gleiche Codebasis wiederverwenden zu können. Dadurch soll Redundanz eingespart werden und der Aufruf vereinfacht werden.
S	Sub-Stylesheet	Ein Sub-Stylesheet spezialisiert das Core-Stylesheet, damit Redundanz vermieden wird und somit die Wartbarkeit gewährleistet werden kann.
	SGML	SGML ist der Vorläufer von XML.
	Single Source Publishing	Beim Single Source Publishing wird aus einer XML Quelle eine Vielzahl von Ausgabeformaten erzeugt
	Structured Content Authoring	Der Content wird hierbei semantisch mittels XML Tags ausgezeichnet. Bei einem WYSIWYG Ansatz sind die meisten Tags nur optional sichtbar.
T	Topic Based Authoring	Beim Topic Based Authoring steht nicht das gesamte Buch im Vordergrund, sondern der Inhalt wird in wiederverwendbare Topics aufgeteilt, die dann in verschiedenen Publikationen wiederverwendet werden können.
	TIOBE Index	Im TIOBE Index wird jährlich die Beliebtheit von Programmiersprachen erfasst.
X	XML Datenbanken	XML Datenbanken sind NoSQL Datenbanken, d.h. "Not only SQL" oder auch tatsächlich "No SQL" wird unterstützt. Die Spezialisierung erfolgt auf XML Daten. Wie sich aber zeigt, ist eine relationale Sicht auf die Daten meistens auch notwendig, so dass DB wie Marklogic diese Funktionalität mittels Erweiterungen unterstützt.

Abbildungen

Bild 1: Pull Stylesheet, *Seite 17*

Beim "Pull" werden Elemente in der Quellinstanz selektiert und an einer passenden Stelle in der Zielinstanz eingefügt. Diese Vorgehensweise ist vergleichbar mit derer von Template-Engines, wie JSP oder ASP. Das kann in mehreren Stufen erfolgen, bis schrittweise die Quellinstanz in die finale Zielinstanz überführt wurde.

Bild 2: Push Stylesheet, *Seite 18*

Beim "Push" werden die Quelldaten schrittweise in die Zieldaten konvertiert. Diese Vorgehensweise kann explorativ erfolgen und beim Transformieren in einen Zwischenschritt entstehen Erkenntnisse, die bei der Weiterverarbeitung nützlich sind. Merke: XSLT steht für eXtensible Stylesheet Transformation.

Bild 3: Transformation des Quellbaums in den Zielbaum, *Seite 18*

Der XSLT Prozessor unternimmt einen Tiefensuchlauf und überprüft bei jedem Knoten den er betritt, ob in seiner Regelbasis eine Regel existiert, die auf diesen Knoten "matched". Dabei gibt es drei grundsätzliche Möglichkeiten, wie die Knoten des Quellbaums in den Zielbaum kopiert - oder eben nicht kopiert - werden können.

Bild 4: oXygen XQuery Builder, *Seite 38*

Mit dem XQuery Builder von oXygen lassen sich unkompliziert Queries testen

Bild 5: Schematron Validierung mit Filter, *Seite 55*

Einfacher Batch-Prozess zur Validierung mit Schematron und anschliessendem Filtern der Ergebnisse

Literaturverzeichnis

- 1 https://de.wikipedia.org/wiki/Darwin_Information_Typing_Architecture
DITA ist ein Standard im Bereich Publishing und löst ältere Dokumenttypen, wie z.B. Docbook ab. Beispielsweise ist DITA ein gutes Modell für Softwarehandbücher und zugehörige Online-Dokumentation.
- 2 <https://de.wikipedia.org/wiki/WYSIWYG>
What You See Is What You Get - Mit dieser Eingabemethode hat der Autor schon ein Bild davon, wie sein Text gedruckt werden kann.
- 3 https://en.wikipedia.org/wiki/Topic-based_authoring
Beim Topic Based Authoring wird der Content feingranular in Informationseinheiten aufgegliedert, die sich dann über Referenzen in verschiedene Publikationen einbinden lassen.
- 4 <http://www.tekturcms.de>
Das ist die private Homepage des Autors mit einer kompletten Liste seiner Hobby-Projekte seit 2000.
- 5 <https://de.wikipedia.org/wiki/TIOBE-Index>
Im TIOBE Index werden jährlich die populärsten Programmiersprachen aufgelistet
- 6 <https://bit.ly/2ARgKCJ>
- 7 https://en.wikipedia.org/wiki/ASC_X12
X12 ist ein amerikanischer Standard wie Nachrichten im Textformat zwischen Informationssystemen verschiedener Unternehmungen ausgetauscht werden. Beispielsweise werden in der Gesundheitsbranche Informationen über Arztbesuche von den Versicherungen an die Staaten geschickt.
- 8 <https://en.wikipedia.org/wiki/S1000D>
Das sehr komplexe Informationsmodell im militärischen Bereich wird z.B. über diesen Standard verwaltet. Teilweise sind die Daten noch im SGML Format vorhanden, da die verarbeitenden System aus den 80er Jahren stammen.
- 9 https://de.wikipedia.org/wiki/XSL_Transformation
Das ist die Wikipedia Seite zu XSL = XSLT, XPATH und XSL-FO
- 10 <http://www.unidex.com/turing/utm.htm>
Die Turingmaschine sollte jedem Informatiker aus den ersten Univorlesungen bekannt sein.
- 11 W3C Seiten zu The Extensible Stylesheet Language Family (XSL)
- 12 <https://www.w3.org/TR/xsl/>
- 13 <https://de.wikipedia.org/wiki/Verarbeitungsanweisung>
Die Processing Instruction wertet der Parser als Kommando aus und nicht als Teil des XML Contents
- 14 https://de.wikipedia.org/wiki/Wireless_Application_Protocol
Mittels dieser Technologie wurden Webinhalte auf Handys gespielt. Das war vor den Smartphones
- 15 <https://de.wikipedia.org/wiki/EPUB>
EPUB ist ein Dokumentformat für Ebook-Reader.
- 16 [https://de.wikipedia.org/wiki/CHM_\(Dateiformat\)](https://de.wikipedia.org/wiki/CHM_(Dateiformat))
Die alte Windows-Hilfe. Läuft immer noch im Bereich Maschinenbau auf gekoppelten Rechnern mit alter Windows Software)
- 17 <https://www.ibm.com/developerworks/library/os-echelp/index.html>
Das Hilfe-Format der Eclipse Rich Client Plattform. Eclipse wird hauptsächlich von Programmierern als Editor benutzt

- 18 <https://en.wikipedia.org/wiki/JavaHelp>
Damit wird bspw. das Java API formatiert als Webseite ausgegeben
- 19 <https://de.wikipedia.org/wiki/Framemaker>
Mit Framemaker kann man manuell gesetzte Publikationen erstellen. Über Templates lässt sich das Layout aber auch automatisieren
- 20 <https://www.i4icm.de/forschungstransfer/pi-mod/>
PI-Mod ist ein Informationsmodell, das am KIT (Uni Karlsruhe) entwickelt wird/wurde
- 21 https://de.wikipedia.org/wiki/Journal_Article_Tag_Suite
JATS ist ein sehr verbreitetes Informationsmodell im Bereich wissenschaftlicher Artikel und Fachliteratur
- 22 https://de.wikipedia.org/wiki/Text_Encoding_Initiative
- 23 <http://surguy.net/articles/client-side-svg.xml>
- 24 <http://jackrabbit.apache.org/jcr/node-type-visualization.html>
- 25 <http://argouml.tigris.org>
ArgoUML ist ein freier UML Editor
- 26 <http://butterflycode.sourceforge.net>
Damit kann man sich Code aus UML Klassendiagrammen generieren lassen, natürlich XSLT basiert
- 27 <https://www.javaworld.com/article/2073998/java-web-development/generate-javabean-classes-dynamically-with-xslt.html>
Weiterführende Lektüre zum Thema Code-Generierung mit XSLT
- 28 <http://www.saxonica.com/html/documentation/sourcedocs/streaming/>
Streaming ist eine Technik zur Verarbeitung großer XML Daten - Stichwort Big Data - mit XSLT3.0
- 29 https://de.wikipedia.org/wiki/XML_Schema
XML Schema ist der Nachfolger der DTD, ist XML basiert und erlaubt auch die Content-Validierung in einem bestimmten Umfang
- 30 <http://www.xsltfunctions.com/>
Sehr gut gegliederte Funktionsbibliothek von Priscilla Walmsley
- 31 <https://docs.python.org/2/library/sgmlib.html>
Einfacher SGML Parser der Python Standard Bibliothek
- 32 <https://www.saxonica.com/html/documentation/sourcedocs/streaming/xslt-streaming.html>
XSLT3.0 Streaming API
- 33 <https://www.saxonica.com/html/documentation/sourcedocs/streaming/>
- 34 <https://www.saxonica.com/html/documentation/xsl-elements/mode.html>
Mode Optionen in XSLT3.0
- 35 <https://www.saxonica.com/html/documentation/xsl-elements/iterate.html>
Der Iterator ist ein Konzept um XSLT Streaming zu realisieren
- 36 <http://exist-db.org/exist/apps/homepage/index.html>
Homepage der eXist XML Datenbank
- 37 <https://de.marklogic.com/>
Homepage der NoSQL/XML Datenbank MarkLogic
- 38 https://www.oxygenxml.com/xml_editor/xquery_builder.html
Tool zur einfachen Eingabe von XQuery Test-Skripten im oXygen XML Editor
- 39 <http://cs.au.dk/~amoeller/XML/querying/flwrexp.html>
XQuery Extensions für mächtigere Funktionen

- 40 <http://www.saxonica.com/documentation/#!sourcedocs/projection>
Verstecktes Saxon Feature: Dokument Projektion bei einer XQuery Abfrage
- 41 <https://www.oxygenxml.com/doc/versions/20.1/ug-editor/topics/configure-marklogic-datasource.html>
MarkLogic Datenquelle in oXygen konfigurieren
- 42 <https://docs.marklogic.com/xdmp:document-insert>
Doku zu xdmp:document-insert Funktion auf den MarkLogic Webseiten
- 43 <https://docs.marklogic.com/xdmp:document-add-collections>
Dokumentation zur xdmp:document-add-collections Funktion auf den MarkLogic Webseiten
- 44 <https://docs.marklogic.com/guide/app-dev/TDE>
Template Driven Extraction wird verwendet um in MarkLogic eine relationale Sicht auf die baumstrukturierten Daten zu setzen
- 45 <https://github.com/Schematron/schematron>
Schematron auf GitHub
- 46 <https://github.com/xspec>
XSpec auf GitHub
- [XM] <https://xmetal.com/>
Homepage des Desktop XML Editors XMetal
- [EP] <https://www.ptc.com/en/products/service-lifecycle-management/arbortext/editor>
Website zum Arbortext XML Editor
- [OX] <https://www.oxygenxml.com/oxygen-xml-web-author/app/oxygen.html>
Der oXygen XML Web Editor im Web
- [FX] <https://www.fontoxml.com/>
FontoXML Webeditor
- [XE] <http://www.xeditor.com/portal>
Homepage des webbasierten XEditors
- [XO] <http://xopusfiddle.net/VT7T/3/>
Homepage des veralteten Xopus XML Editors

Stichwortregister

A

Anwendungsgebiete

Code Generierung	15
Log- und Konfigurationsdaten	12
Migration	15
Serverseitige Konvertierung	10
Visualisierung	13
XML Webseiten	9
Assert- und Not-Assert	58

F

FLOWR Expression	38
Format- und Produktvarianten	28

K

Konzepte

Default-Regel	20
Diffing	14
Generalisierung und Spezialisierung	14, 28
Gültigkeitem	14
Import Präzedenz	20
Intelligente Querverweise	14
Match-Regeln	19
Named Template	28
Priorität	19, 20
Push vs. Pull	17
Regelauswertung	20
Regelbasis	19
Vererbung	27
Versionierung	14
Wiederverwendung	14

P

Performanzsteigerung	29
----------------------------	----

Programmierkonstrukte

Bedingte Anweisung - if..then..else	46
DB Collection	44
DB insert	44
Funktionen	45
Module	46
Schleifen - for	44
Schleifen - while..do	45

Programmiersprachen

Java	53
JSP	8
LISP	8
Metamorphosis	8
Python	29
XQuery	37

S

Schemata	53
----------------	----

Software

Marklogic	40
-----------------	----

Standards

JAT	12
PI-MOD	14
S1000D	8
SVG	13
TEI	12, 14
X12	8

T

Tektur CCMS	74
Testfall	58
Test-Framework	57
Tiefensuchlauf	18
TIOBE Index	8

Tipps und Tricks

Appear- und Not-Appear	58
Document Projection	39

Tools

GIT	59
Marklogic WebDAV Server	41

oXygen XML Editor	37, 61
oXygen-Connector	40
Schematron	58
XML Editor	60
XSpec	57

V

Verarbeitungsmethode

XSLT Streaming	31
----------------------	----

Verarbeitungsmethoden

Akkumulator	30
Filter-Szenario	54
Shallow Copy	30
Validierung	53
Vortransformation	25
XSLT Streaming	29

W

WYSIWYG und WYSIWYM	60
---------------------------	----

X

XML Konstrukte

Namespaces	22, 45, 46
Processing Instruction	9



Tekur CCMS ist ein web-basiertes Component Content Management System und befindet sich noch in der Entwicklung. Blog: www.tekturcms.de

Hier sind einige Random Features:

- Die Inhalte werden nach dem DITA Content Model eingegeben. Die Ausgabe erfolgt über ein automatisches Satzsystem.
- Grafiken können für die PDF-Ausgabe seitenbreit, spaltenbreit und in der Marginalie gesetzt werden.
- Die Breite der Marginalie ist stufenlos einstellbar; die PDF-Ausgabe ist bzgl. der Formatierung weitestgehend konfigurierbar.
- Layoutoptionen bzgl. Papierformat, Bemassung und Schriftgrößen können über einen einfachen Dialog eingestellt werden.
- TOC und mehrstufige Register werden automatisch in der PDF-Ausgabe erzeugt.
- Die Zellenbreite von CALS Tabellen kann mit der Maus eingestellt werden; Funktionen auf Zellen sind weitestgehend implementiert.
- Copy 'n Paste funktioniert Elementweise und topic-übergreifend.
- Paras, Listitems und Sections können mit den Pfeilbuttons in der Toolbar nach oben und unten verschoben werden.
- Verlinkung auf andere Topics funktioniert über Referenzen und ein Linktext wird automatisch aktualisiert, wenn sich der Topic-Titel ändert.
- Die DITA-Map kann u.a. mittels Drag 'n Drop editiert werden; Im Topic Editor gibt es an jeder Stelle ein dynamisches Kontextmenü für weitere Optionen.
- Valide DITA Strukturen können exportiert und importiert werden.
- Topics, Tasks und Maps können vom Autor an Reviewer und Approver für einen Kommentar- und Freigabeprozess überwiesen werden.