Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

ID Software Services

ETH Zürich
Benno Luthiger
STC E 13
Stampfenbachstrasse 67
8092 Zürich

benno.luthiger@id.ethz.ch

Schulung: Skripten in AEM

Inhaltsverzeichnis

1. Grundlagen 3
2. Einführung in Skriptprogrammierung 4
3. Konfiguration der Skript-Komponenten 4
4. Skript-Programmierung 6
   4.1. Request-Objekt 7
   4.2. DataSource-Objekt 7
   4.3. LabelSource-Objekt 8
   4.4. ScriptHelper Objekt 8
   4.5. Storage Objekt 10
5. JSON-Display 10
   5.1. JSON-Format 10
   5.2. Konfiguration 12
   5.3. Sortierung 14
6. Spezielle Tags für JSON-Listen 14
7. Modularisierung 16
   7.1. Skripte einbinden 16
   7.2. Gemeinsame Skripte 16
   7.3. Statische Daten einlesen 18
   7.4. Übersicht 18
8. Formulare mit ScriptNode 18
   8.1. Formular-Felder dynamisch füllen 21
9. Tipps und Tricks 23
   9.1. Debuggen 23
   9.2. Verarbeitung der Formularfelder 24
   9.3. Client-side JavaScript 24
   9.4. Client-side JavaScript mit jQuery 25

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

ID Software Services

ETH Zürich
Benno Luthiger
STC E 13
Stampfenbachstrasse 67
8092 Zürich

benno.luthiger@id.ethz.ch

Versionskontrolle

Version	Historie / Status	Datum	Autor/in	URL
01.0	Schulung	4.12.2014	Benno Luthiger	
01.1	S. 18: jQuery-Selektor muss nicht mehr escaped werden.	6.1.2015	Benno Luthiger	
01.2	S. 7: erweiterte DataSource, S. 12: JSON sortField	2.2.2015	Benno Luthiger	
01.3	S. 8: ScriptHelper getPath	26.3.2015	Benno Luthiger	
01.4	S. 19: befüllte Felder	31.3.2015	Benno Luthiger	
01.5	S. 15: neue Tags für JSON	8.4.2015	Benno Luthiger	
01.6	S. 16: zusätzliches Beispiel	24.9.2015	Benno Luthiger	
01.7	S. 16: geteilte Skripte	19.10.2015	Benno Luthiger	
01.8	S. 7: Methode DataSource.getRow() S. 13: Pfad-Parameter für JSON S. 14: sortBy-Attribut	9.11.2015	Benno Luthiger	
01.9	S. 16: Angabe zu *base* korrigiert	23.3.2016	Benno Luthiger	
01.10	S. 23: Präzisierung Proxy-Konfiguration	22.12.2016	Benno Luthiger	
01.11	S.8: ScriptHelper mapToUri	23.08.2017	Benno Luthiger	

1. Grundlagen

Die CQ5-Installation an der ETH bietet zwei Komponenten an, mit welchen Skripte eingebunden werden können.

JSON-Display-Komponente *jsondisplay*

Mit der JSON-Display-Komponente können REST-Anfragen an einen beliebigen Webservice gestellt und die Rückgaben dieser Anfragen als Tabellen dargestellt werden. Voraussetzung ist allerdings, dass die Rückgaben des Webservices als JSON-Array oder -Objekt formatiert sind.

ScriptNode-Komponente *scriptnode*

Zweck der ScriptNode-Komponente ist es, auf einer Webseite beliebige Funktionalität bereitzustellen. Ein typischer Anwendungsfall für diese Komponenten ist ein Formular, welches ja nach Benützer-Eingabe Felder ein- oder ausblendet.

In beiden Einsatzfällen soll auf der Webseite Funktionalität vorhanden sein, die (im Wesentlichen) nur auf dieser Seite gebraucht wird. Während die üblichen CQ5-Komponenten Lösungen für Anwendungsfälle bieten, welche von vielen Autoren auf verschiedenen Seiten eingesetzt werden, geht es bei den Skript-Komponenten um Fälle, wo ganz spezifische Lösungen für einen bestimmten Anwendungsfall gesucht werden.

Die Skript-Komponenten verwenden eine Mischung von html (für die Darstellung) und JavaScript (für die Verarbeitungs-Logik). Wichtig für das Verständnis ist, dass der JavaScript-Code auf dem Server interpretiert wird (Server-side JavaScript) und nicht wie bei JavaScript üblich im Browser.

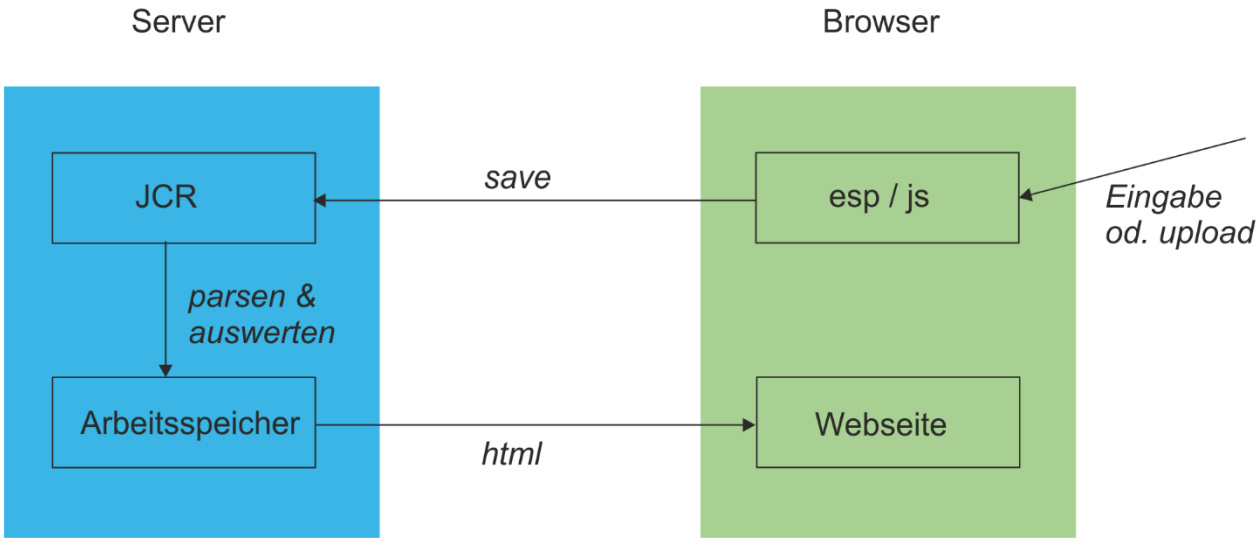

Abbildung 1: Übersicht Skript-Verarbeitung

2. Einführung in Skriptprogrammierung

Die Skript-Programmierung verbindet Darstellung (in Form von html-Anweisungen) mit Verarbeitungslogik (in Form von JavaScript-Anweisungen).

Das einfachste Skript enthält nur html-Code:

```
<h2>Hallo ETH</h2>
```

Codebeispiel 1: html

Im Skript kann der html-Code mit JavaScript angereichert werden. JavaScript-Anweisungen können als Ausdrücke oder als Scriptlets in das Skript eingefügt werden.

```
<h2>Hallo <%= labelSource.get("title") %></h2>
```

Codebeispiel 2: JavaScript-Expression

```
<h2>Test</h2>
<%
var value = true;
if (value) {
%>
      <p>Eine bedingte Anzeige</p>
<%
}
%>
```

Codebeispiel 3: html mit Scriptlet

3. Konfiguration der Skript-Komponenten

Beide Skript-Komponenten können, was die Basis-Funktionalität betrifft, auf die gleiche Weise konfiguriert werden.

Auf dem ersten Tab (Skripte) werden die Skripte verwaltet. Eines der vorhandenen Skripte muss als Einstiegspunkt deklariert werden (vgl. Abbildung 3). Dieses Skript wird beim Aufruf der Webseite abgearbeitet und kann seinerseits andere Skripte aufrufen.

Mit einem Klick auf den Link „File hinzufügen“ wird ein Dialogfenster angezeigt, mit welchem ein neues Skript erzeugt werden kann. Das Skript muss einen Namen haben und den Code, welcher auszuführen ist. Der Code kann entweder in das grosse Textfeld eingegeben oder aus dem Filesystem hochgeladen werden (Abbildung 2).

Auf dem Tab „Label“ werden die Labels verwaltet, welche im Skript verwendet werden. Die Verwendung von Labels ist dann zweckmässig, wenn die Skript-Funktionalität auf einer

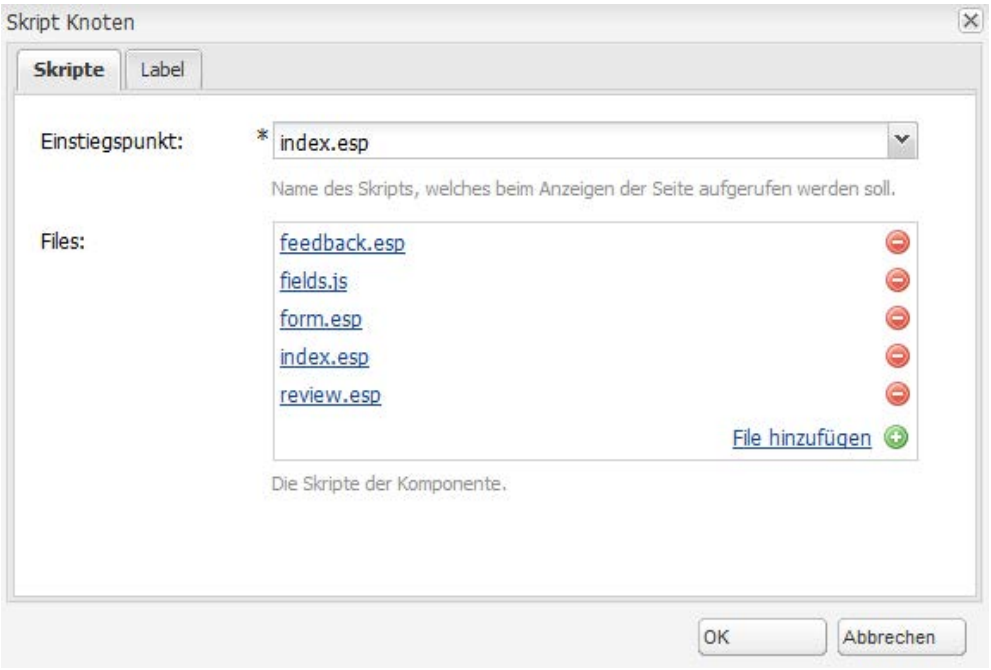

Abbildung 3: Verwaltung der Skripte

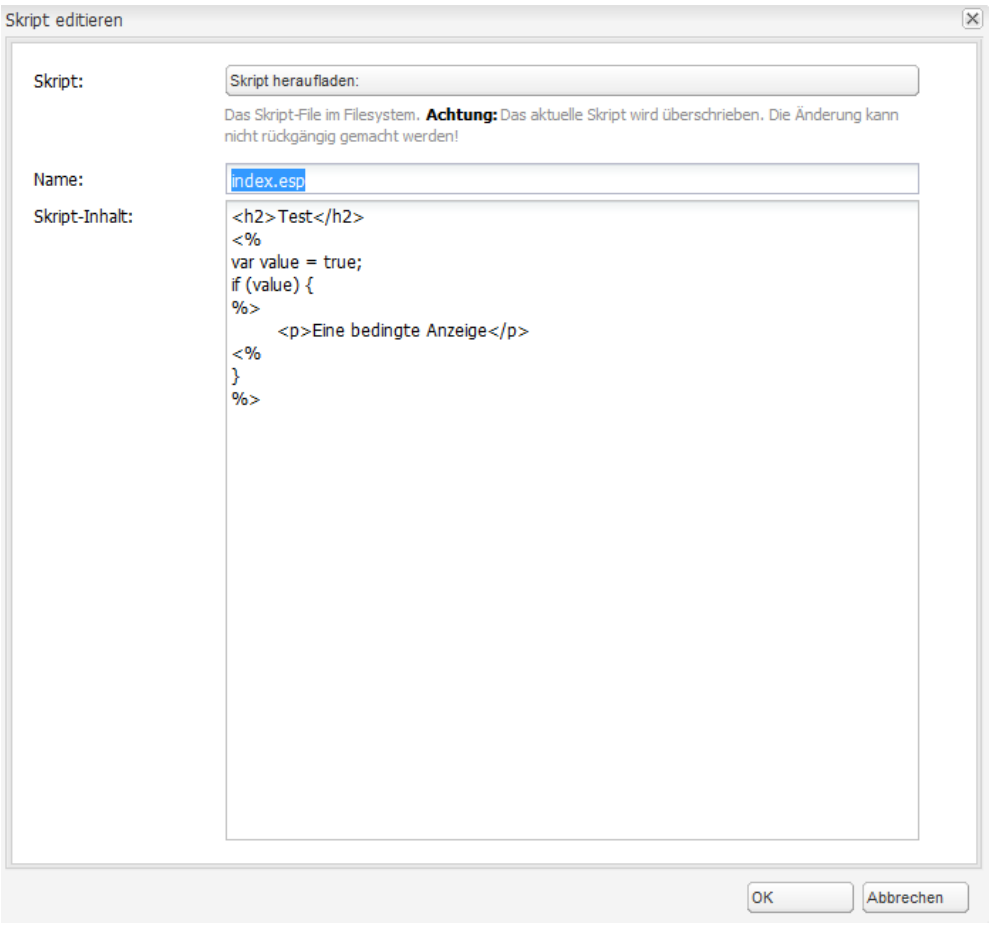

Abbildung 2: Skript editieren

Webseite eingesetzt werden soll, welche in mehreren Sprachen vorhanden ist. In diesem Fall

kann das Skript ohne Änderungen sowohl beispielsweise auf der deutschen wie auch auf der englischen Seite installiert werden, bloss das jeweils der richtige Satz an Labels hinzugefügt wird.

Die Labels müssen als Schlüssel-Label-Paare eingegeben werden (Abbildung 4). Auch die Labels können aus einem Textfile hinaufgeladen werden. In diesem Fall müssen die Schlüssel-Label-Paare jeweils auf einer Zeile stehen und das Label muss durch das Pipe-Zeichen „|“ vom Schlüssel getrennt sein (vgl. Codebeispiel 4).

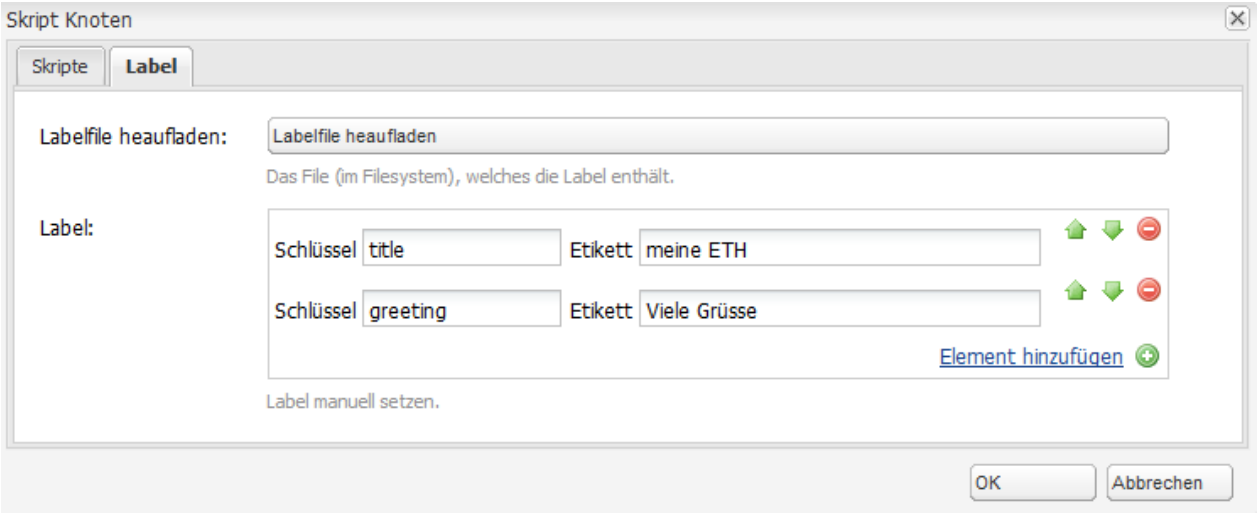

Abbildung 4: Label-Verwaltung

```
title|meine ETH

greeting|Viele Grüsse
```

Codebeispiel 4: Label-File

4. Skript-Programmierung

Damit das Skript etwas Vernünftiges machen kann, muss es mit seinem Kontext interagieren können. Zu diesem Zweck liegen im Kontext verschiedene Objekte, welche im Skript angesprochen werden können. Das wichtigste Objekt ist der *Output-Stream*, welcher mit `out` angesprochen wird. Der Output-Stream hat die Methode write(), mit welchem beliebiger Text in den Stream geschrieben wird. In unserem Fall soll der html-Code, welcher im Browser angezeigt werden soll, in den Stream geschrieben:

```
<%
out.write("<h2>Hallo ETH</h2>");
%>
```

Codebeispiel 5: Output-Stream

Die folgende Tabelle zeigt eine Übersicht über die Objekte im Kontext eines Skripts:

Objekt	Zweck, Bemerkung
out	Output-Stream, schreibt Text an Browser
request	Servlet-Request-Objekt, erlaubt Zugriff auf Parameter
response	Servlet-Response-Objekt (verwaltet Output-Stream)
dataSource	erlaubt Zugriff auf die JSON-Daten (nur JSON-Display-Komponente)
labelSource	erlaubt Zugriff auf die Labels
scriptHelper	stellt Komfortfunktionen für Skript zur Verfügung
storage	erlaubt Zugriff auf Persistenz-Bereich

Tabelle 1: Objekte im Kontext

4.1. Request-Objekt

Der Servlet-Request enthält u.a. die Parameter, mit welcher die Seite aufgerufen wurde. Das folgende Code-Beispiel fragt den Wert des Parameters *param* ab.

```
<%
out.write("<p>Parameter-Wert: " + request.getParameter("param") + "</p>");
%>
```

4.2. DataSource-Objekt

Das DataSource-Objekt ist nur im Falle der JSON-Display-Komponente vorhanden. Es erlaubt den Zugriff auf die JSON-Daten, welche vor dem Aufruf der Komponente geholt werden.

Das DataSource-Objekt verarbeitet einen Array von JSON-Objekte und gibt diese als Liste von *DataRow*-Objekten weiter. Die Funktionalität des DataSource- und DataRow-Objekts wird durch folgende Interfaces beschrieben:

```
public class DataSource {
      public Iterator<? extends DataRow> getRows();
      public Iterator<? extends DataRow> getRows(String root);
      public int getRowCount();
      public int getRowCount(String root);
      public DataRow getRow(String root, int index)
      public DataRow getObj();
      public DataSource sortBy(String fieldName);
}
```

```
public class DataRow {
      public String get(final String inKey);
      public DataRow getObj(final String inKey);
      public boolean isEmpty(final String inKey);
      public Iterator<? extends DataRow> getChildren(String inKey);
      public String toString();
}
```

Codebeispiel 6: Interface von DataSource und DataRow

Als Schlüssel für die Getter-Methoden kann auch ein Pfad in JSON-Notation angegeben werden. Wenn `row.get("title")` den Wert des Attributes *title* zurückgibt, so kann mit `row.get("sub.name")` der Wert des *name*-Attributes im Unterobjekt *sub* zurückgegeben werden.

Mit `row.toString()` kann die String-Repräsentation des JSON-Objekts geholt werden.

4.3. LabelSource-Objekt

Das LabelSource-Objekt ermöglicht den Zugriff auf die Labels der Skript-Komponente. Dieses Objekt bietet als einziges die Methode `labelSource.get()` an. Als Parameter der Methode wird der Schlüssel erwartet. Die Rückgabe ist das gesuchte Label. Gibt es kein Label mit dem übergebenen Wert, dann wird `null` zurückgegeben.

```
<%
out.write("<p>Titel: " + labelSource.get("title") + "</p>");
%>
```

Codebeispiel 7: Anzeige von Label

4.4. ScriptHelper Objekt

Die Funktionalität des ScriptHelper-Objekts wird durch folgendes Interface beschrieben:

```
public class ScriptHelper {
      public String getLanguage();
      public String getPath();
      public String getProxyUrl();
      public String scrambleMail(String inMail);
      public String scrambleMail(String inMail, String inLabel);
      public String getPostUrl();
      public FormParameters getParameters();
      public FormValidation validate(String[] inMandatory);
      public void sendMail(String[] inMailTo, String inMailFrom,
            String inSubject, String inBodyHtml, String inBodyText);
      public String mapToUri(final String path);
}
```

Codebeispiel 8: Interface von ScriptHelper

Die Methode `ScriptHelper.getLanguage()` gibt die Sprache der aktuellen Seite (d.h. der Seite, auf welcher sich das Skript befindet) zurück, z.B. „de“ oder „en“.

```
<%
out.write("<p>Sprache: " + scriptHelper.getLanguage() + "</p>");
%>
```

Codebeispiel 9: ScriptHelper - Sprache

Die Methode `ScriptHelper.getPath()` gibt den JCR-Pfad des aktuellen Skript-Knotens zurück. Diese Angabe ist nützlich, wenn im Knoten ein CSS-File abgelegt ist und dieses CSS mit der html-Link-Anweisung eingebunden werden soll. Im folgend Beispiel wird das File *my.css* mit Hilfe des ScriptHelpers eingebunden:

```
<link rel="stylesheet" type="text/css"
      href="<%=scriptHelper.getPath() %>/my.css" />
```

Codebeispiel 10: ScriptHelper - Pfad

Mit den Methoden `ScriptHelper.scrambleMail()` kann eine Mail-Adresse auf der Webseite so dargestellt werden, dass sie im erzeugten html-Code nicht offen ersichtlich ist. Das ist ein Schutz gegen Software, welche alle Webseiten einer Organisation nach verwertbaren Mail-Adressen absuchen. Im Browser wird die Mail-Adresse wie gewohnt angezeigt.

Die weiteren Methoden des ScriptHelper-Objekts sind vor allem in solchen Fällen nützlich, in welchen die Skript-Komponente zur Darstellung eines Formulars eingesetzt wird.

Die Methode `ScriptHelper.getPostUrl()` gibt die URL zurück, welche für die Formular-Aktion verwendet werden kann:

```
<form enctype="multipart/form-data" method="POST"
  action="<%=scriptHelper.getPostUrl() %>">
    <p>Input: <input type="text" value="" name="some_input" /></p>
    <p><button name="submit_form" value="send">Senden
        <span class="icon" /></button></p>
</form>
```

Codebeispiel 11: ScriptHelper - Formular-Aktion

Die Methode `ScriptHelper.getParameters()` gibt ein *FormParameters*-Objekt zurück. Dieses enthält die Werte, die der Webseiten-Besucher in das Formular eingegeben hat. Dadurch wird ein einfacher Zugriff auf die Benutzereingaben möglich.

Die Methode `ScriptHelper.validate()` ist für die Validierung der Benutzereingaben gedacht. Als Parameter erwartet diese Methode einen Array der Feldnamen, für welche eine Benutzereingabe obligatorisch ist. Zurück gibt diese Methode ein *FormValidation*-Objekt, welches Auskunft über die Validität der Benutzereingabe gibt.

Mit `ScriptHelper.sendMail()` kann aus dem Skript eine Mail verschickt werden. Dies kann beispielsweise im letzten Schritt einer Formular-Verarbeitung verwendet werden, nachdem die Benutzereingabe validiert und bestätigt worden ist.

Die Methode `ScriptHelper.mapToUri()` kann verwendet werden, um einen Pfad zu externalisieren. Dies ist beispielsweise dann notwendig, wenn ein Link auf der Webseite dargestellt wird. Wird diese Methode verwendet, ist sichergestellt, dass der Link sowohl auf der Autoren-Instanz wie auch auf der publizierten Seite korrekt funktoniert.

4.5. Storage Objekt

Das Storage-Objekt ermöglicht es, aus einem Skript heraus Werte im Repository (d.h. im Speicherbereich des CQ5-CMS‘) zu speichern. Die Funktionalität des Storage-Objekts wird durch folgendes Interface beschrieben:

```
public class Storage {
      public void put(String inKey, String inValue);
      public String get(String inKey);
}
```

Codebeispiel 12: Interface von Storage

Mit `Storage.put()` wird ein bestimmter Wert unter einem bestimmten Schlüssel gespeichert. Mit `Storage.get()` wird dieser Wert wieder aus dem Speicherbereich gelesen.

5. JSON-Display

Zweck der JSON-Display-Komponente ist es, Daten aus externen Quellen auf der Webseite darzustellen. Die Daten werden über eine URL abgeholt und aufbereitet, bevor das Skript gestartet wird. Die zu verarbeitenden Daten müssen in JSON-Format geliefert werden.

5.1. JSON-Format

Im einfachsten Fall gibt der JSON-Aufruf eine Liste von Einträgen in Form eines JSON-Arrays zurück, wobei jeder Eintrag in der Liste einem JSON-Objekt entspricht (vgl. Codebeispiel 13).

```
[
{"title":"Der Richter und sein Henker", "author": "Dürrenmatt",
      "tags":[{"value:"Schweiz"},{"value:"Drama"}]},
{"title":"Homo Faber", "author": "Frisch",
      "tags":[{"value:"Schweiz"},{"value:"Roman"}]}
]
```

Codebeispiel 13: Input für JSON-Display-Komponente als JSON-Array

Das Skript um eine solche JSON-Liste mit Hilfe des DataSource-Objekts darzustellen, kann etwa wie folgt aussehen:

```
<%=dataSource.getRowCount() %> Bücher
<ul>
<%
var rows = dataSource.getRows();
while (rows.hasNext()) {
  var row = rows.next()
  if (!row.isEmpty("title")) {
%>
  <li><%=row.get("title") %></li>
<% }
}%>
</ul>
```

Codebeispiel 14: Skript zur Darstellung eines JSON-Arrays

Oft gibt ein JSON-Aufruf nicht ein JSON-Array zurück, sondern ein JSON-Objekt, das in einem seiner Elemente die darzustellende Liste enthält. Im Codebeispiel 15 ist die Liste im Element „items“ gespeichert.

```
{
"date":"2015-01-01",
"state":"published",
"items":[
{"title":"Der Richter und sein Henker", "author": "Dürrenmatt",
      "tags":[{"value:"Schweiz"},{"value:"Drama"}]},
{"title":"Homo Faber", "author": "Frisch",
      "tags":[{"value:"Schweiz"},{"value:"Roman"}]}
]}
```

Codebeispiel 15: Input für JSON-Display-Komponente als JSON-Objekt

In diesem Fall muss im Skript dem Aufruf des DataSource-Objekts als Parameter übergeben werden, in welchem Element die Einträge für die darzustellende Liste enthalten sind. Im obigen Beispiel muss die DataSource mit dem Parameter „items“ aufgerufen werden, damit die Liste der Bücher im JSON-Objekt unter dem Schlüsse „items“ gefunden werden:

```
<%=dataSource.getRowCount("items") %> Bücher
<ul>
<%
var rows = dataSource.getRows("items");
while (rows.hasNext()) {
  var row = rows.next()
  if (!row.isEmpty("title")) {
%>
  <li><%=row.get("title") %></li>
<% }
}%>
</ul>
```

Codebeispiel 16: Skript zur Darstellung einer Liste in einem JSON-Objekt

5.2. Konfiguration

Die JSON-Display-Komponente enthält auf dem ersten Tab wie die Skript-Komponente die Funktionalität, um Skripte zu verwalten. Zusätzlich muss für diese Komponente die URL konfiguriert werden, welche für die Abfrage des JSON-Resultats verwendet werden soll.

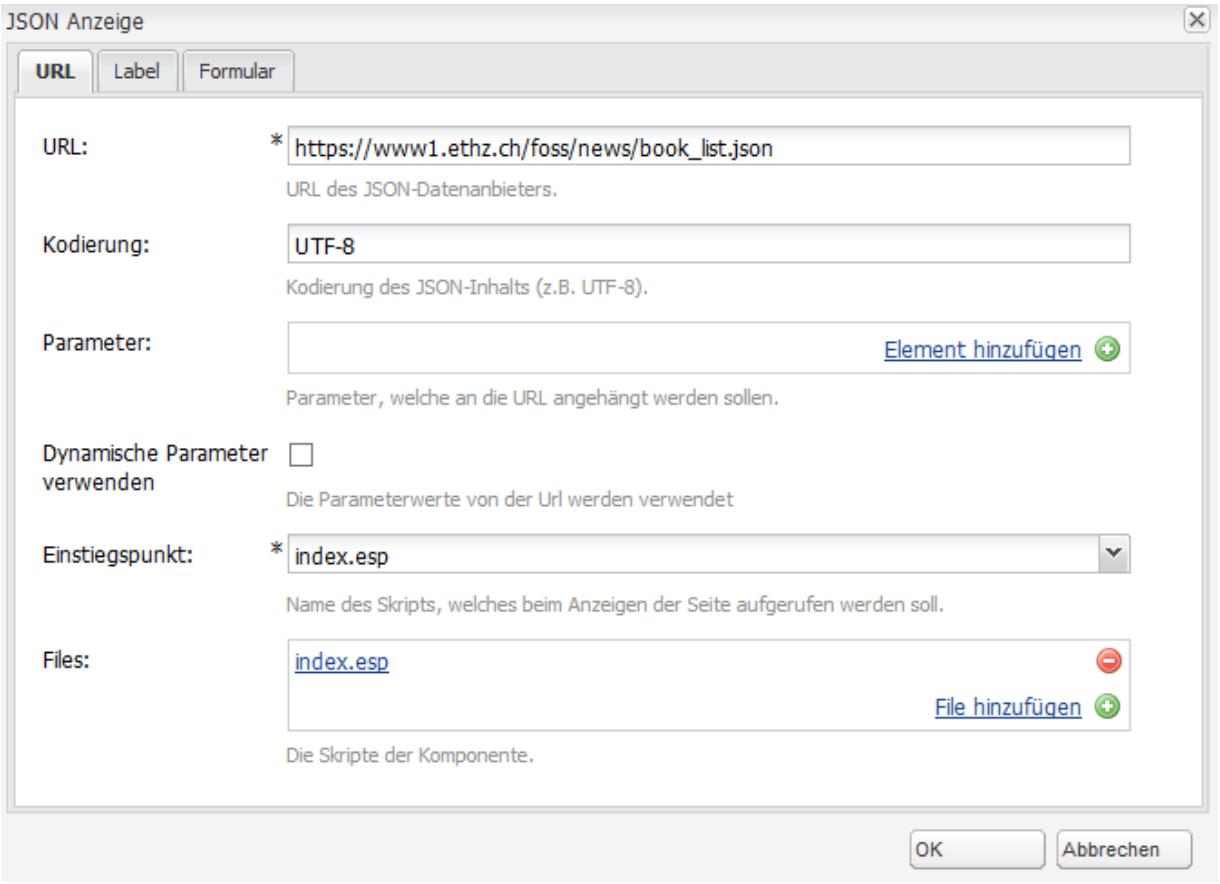

Abbildung 5: Konfiguration JSON-Display

Der URL können Parameter zugefügt werden, entweder gleich im Feld „URL“ oder über die Eingabeliste für die Parameter. Dies ist dann nützlich, wenn unter der gleichen URL je nach Parameter unterschiedliche Listen oder unterschiedliche Ansichten einer Liste abgefragt werden können.

Die URL-Parameter in dieser Form sind statisch. D.h. sie werden in der JSON-Display-Komponente konfiguriert und bleiben danach fest, bis sie anders konfiguriert werden. Oft möchte man allerdings die Parameter-Werte dynamisch festlegen. Dies kann mit der Checkbox „Dynamische Parameter verwenden“ erreicht werden. Ist diese Checkbox aktiviert, wertet die Komponente die Parameter aus, mit welcher die Seite aufgerufen wird, auf welcher die JSON-Display-Komponente eingebettet ist. Die Seiten-Parameter werden dabei wie folgt ausgewertet:

Stimmt der Name eines Parameters der Seiten-Aufrufs mit dem Namen eines Parameters in der Parameter-Liste (der Konfiguration) überein, wird sein Wert genommen und als Parameter an die URL für die JSON-Abfrage angehängt (vgl. Abbildung 6).

Im Falle von REST-Aufrufen werden Parameter bisweilen auch als Pfad-Parameter verwendet. Um die Parameter des Seitenaufrufs in dieser Weise in die JSON-URL zu verarbeiten, muss diese URL mit Platzhaltern der Form *${paramName}* ausgestattet sein.

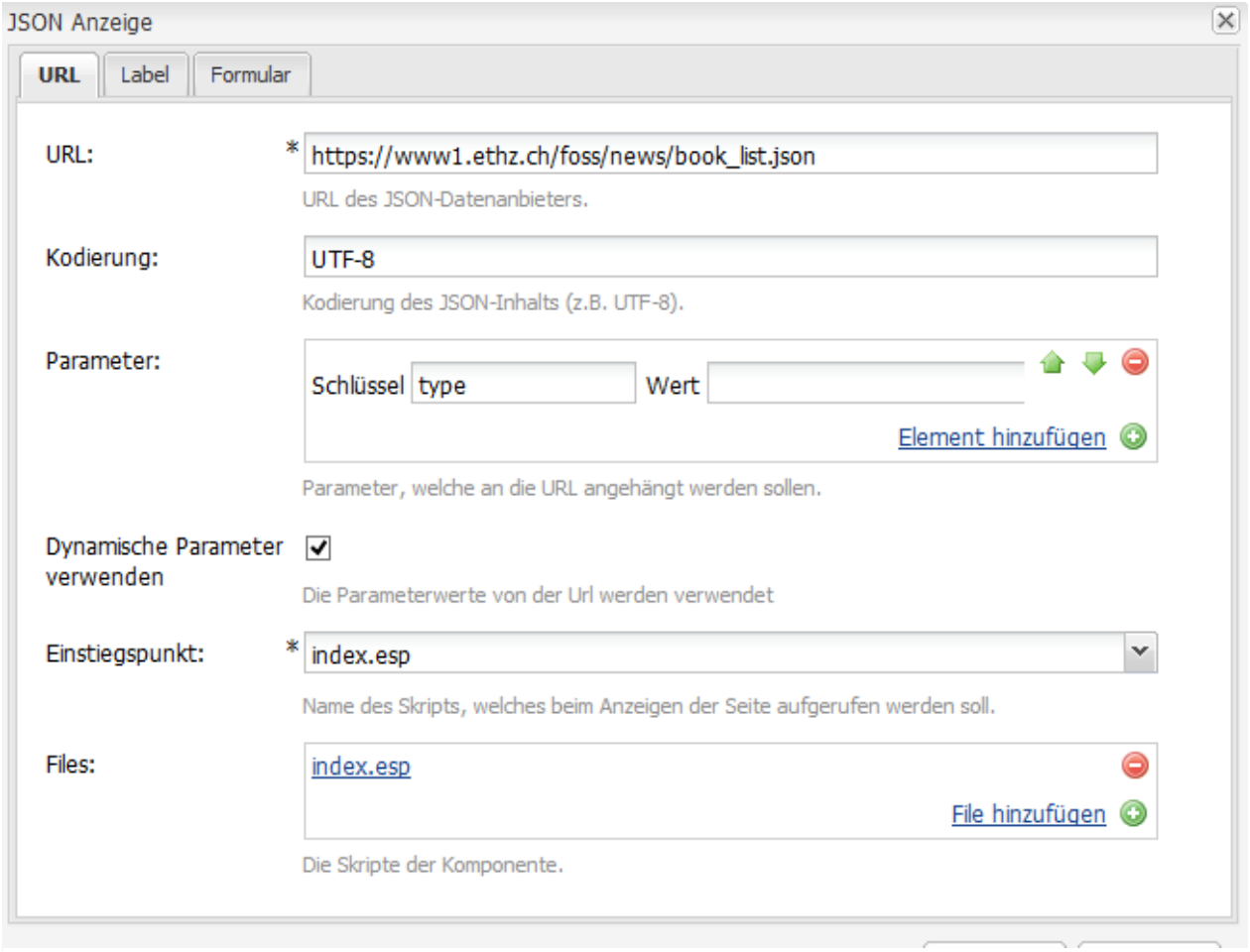

Abbildung 6: JSON-Display mit dynamischen Parametern

Beispiel1:

JSON-URL: https://www1.ethz.ch/foss/news/book_list.json
Parameter-Liste: Schlüssel: type
Seiten-Aufruf: https://www.webaut.hk.ethz.ch/content/demo.html?type=short
Dies ergibt:
JSON-Aufruf: https://www1.ethz.ch/foss/news/book_list.json?type=short

Beispiel 2:

JSON-URL: https://www1.ethz.ch/foss/*${page}*/book_list.json
Parameter-Liste: Schlüssel: type, page
Seiten-Aufruf: https://www.webaut.hk.ethz.ch/content/demo.html?type=short&page=news
Dies ergibt:
JSON-Aufruf: https://www1.ethz.ch/foss/news/book_list.json?type=short

Diese Funktionsweise kann mit folgendem Code überprüft werden:

```
<h2>Test mit dynamischem Parameter</h2>
<p>Anzahl Bücher: <i><%=dataSource.getRowCount() %></i></p>
```

Codebeispiel 17: Skript für JSON-Display mit dynamischem Parameter

Als JSON-URL kann https://www1.ethz.ch/foss/news/book_list.json verwendet werden. Mit `type=short` werden 85 Datensätze angezeigt. Mit `type=bottom` werden 177 Datensätze geliefert. In den übrigen Fällen sind es 374 Datensätze.

5.3. Sortierung

Meistens sollen die angezeigten Listen nach einem der Felder in der Liste sortiert sein. Falls die Listenelemente nicht schon in der gewünschten Reihenfolge beim JSON-Aufruf zurückgegeben werden, kann eine Sortierung beim Verarbeiten der Listenelemente innerhalb der Komponente sichergestellt werden. Dies geschieht dadurch, dass dem Seiten-Aufruf der Parameter `sortField` mit dem gewünschten Feldnamen mitgegeben wird.

6. Spezielle Tags für JSON-Listen

Um die Verarbeitung der JSON-Daten und die Darstellung dieser Daten in Listenform zu erleichtern, wurden spezielle Tags entwickelt, welche im Skript wie html-Code verwendet werden können.

Tag	Verwendung

`<ethz:iter var="row">`
`...`
`</ethz:iter>`
	Iteriert über alle Elemente der DataSource-Instanz. Die einzelnen Listenelemente werden in eine Variable geschrieben, welche im var-Attribut definiert ist.

`<ethz:iter var="row" root="items"> ... </ethz:iter>`	Iteriert über alle Elemente der DataSource-Instanz. Der Wert des root-Attributs bezeichnet das Element in der JSON-Rückgabe, welches die Liste (JSON-Array) der darzustellenden Elemente enthält.
`<ethz:iter var="row" sortBy="position"> ... </ethz:iter>`	Iteriert über alle Elemente der DataSource-Instanz, wobei die Elemente gemäss dem Feld „position“ sortiert sind.
`<ethz:iterValue value="row.name" />`	Zeigt den Wert eines bestimmten Felds eines Listenelements an.
`<ethz:iterChildren value="row.cells" var="cell"> ... </ethz:iterChildren >`	Iteriert über alle Element des Arrays, welcher im durch *value* bezeichneten Wert des Listenelements enthalten ist.
`<ethz:label value="key" />`	Zeigt das Label mit dem Schlüssel, welcher im value-Attribut definiert ist, an.
`<ethz:obj var="myObj" />`	Falls die Rückgabe der JSON-Abfrage ein einzelnes (beliebig verschachteltes) JSON-Objekt zurückgibt, kann mit diesem Tag angegeben werden, unter welchem Namen das Objekt im Skript-Kontext angesprochen werden soll. Dieser Tag ist äquivalent zur DataSource-Methode `myObj = dataSource.getObj()`.
`<ethz:value value="myObj.title" />`	Zeigt den Wert eines bestimmten Felds eines einzelnen JSON-Objekts an.

Tabelle 2: Spezial-Tags für JSON-Display

Beispiel:

```
<h2>Bücherliste</h2>
<ul>
<ethz:iter var="book">
<li>
<ethz:iterValue value="book.title" />
<i>(<ethz:label value="price" />: <ethz:iterValue value="book.price" />)</i>
</li>
</ethz:iter>
</ul>
```

Codebeispiel 18: Anzeige einer Bücherliste mit JSON-Display

Das Beispiel Codebeispiel 18 zeigt eine Bücherliste an. Ausgewertet werden JSON-Daten in Form von `[{"title":"Titel des Buchs", "price":"Preis des Buchs"}]`.

Diese Tags können mit JavaScript-Anweisungen gemischt werden. Das folgende Beispiel ergibt das gleiche Resultat:

```
<h2>Bücherliste</h2>
<ul>
<ethz:iter var="book">
<li>
<ethz:iterValue value="book.title" />
<i>(<ethz:label value="price" />: <%= book.get("price") %>)</i>
</li>
</ethz:iter>
</ul>
```

Codebeispiel 19: Bücherliste mit JSON-Display

7. Modularisierung

7.1. Skripte einbinden

Unter Umständen können Skripte gross und reich an Funktionalität werden. In solchen Fällen ist es nützlich, gewisse Teile mit gleich gearteter Funktionalität in eigene Module auszulagern. Dies fördert die Übersichtlichkeit der Skripte und erleichtert es, diese zu warten. Mit den Skript-Komponenten können beliebig viele Files verwaltet werden. Eines der Skripte wird als Einstiegspunkt markiert. Die weiteren Skripte müssen dann in geeigneter Weise in dieses Hauptskript eingebunden werden.

Zu diesem Zweck steht die Anweisung `<ethz:include src="form.esp" />` zur Verfügung. Mit der *Include*-Anweisung wird das unter *src* bezeichnete File in das aktuelle Skript eingebunden. Das *src*-Attribut kann einen Pfad enthalten. In diesem Fall muss es sich um einen relativen Pfad handeln, d.h. das File wird relativ zum aktuellen Knoten im JCR gesucht.

7.2. Gemeinsame Skripte

Eine Weiterentwicklung dieser Vorgehensweise ist, dass die Skripte, welche eingebunden werden sollen, an einem zentralen Ort verwaltet werden sollen. Diese Skripte können dann von verschiedenen SkriptNode- oder JSON-Display-Komponenten verwendet werden. Änderungen an solchen Skripten sind sofort in den Instanzen, welche diese Skripte referenzieren, sichtbar. In diesem Fall muss die Variante der Anweisung `<ethz:include src="shared.esp" base="refToShared" />` verwendet werden. Das neue Attribut *base* referenziert den JCR-Knoten, welcher die gemeinsamen Skripte enthält.

Die gemeinsamen Skripte können mit der Seite „ETHZ Skript-Verwaltung“ verwaltet werden. Erzeugen Sie eine Seite dieses Typs und öffnen Sie danach den Konfiguration-Dialog, um Skripte zu erstellen oder zu löschen, mit einem Doppelklick auf den Inhaltsbereich der Seite.

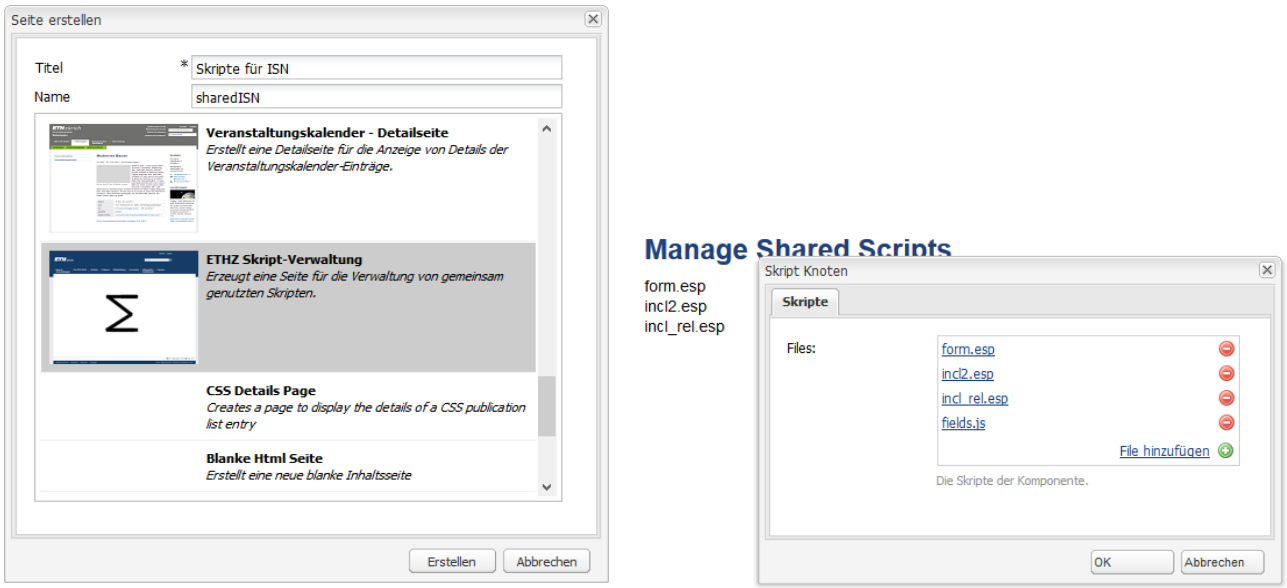

Abbildung 8: Gemeinsame Skripte verwalten

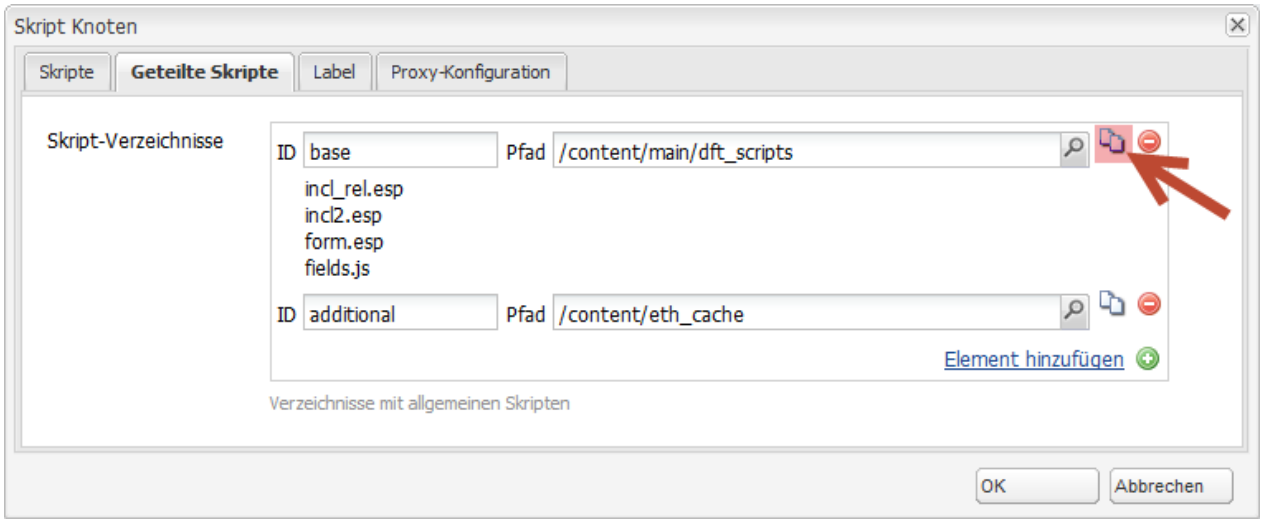

Abbildung 7: Gemeinsame Skripte referenzieren

Die Skripte, die Sie auf diese Weise erzeugen, können danach wie folgt in eine SkriptNode- oder JSON-Display-Instanz eingebunden werden. Öffnen Sie den Tab „Geteilte Skripte“ (der SkriptNode bzw. JSON-Display-Komponente) und fügen Sie eine Referenz auf die Seite mit der Skript-Verwaltung hinzu. Ein solcher Eintrag besteht aus einer ID und dem Pfad auf die gewünschte Seite. Die ID benötigen Sie für das *base*-Attribut der *Include*-Anweisung, wenn sie eines der geteilten Skripte einbinden wollen. Mit einem Klick auf das Inhalts-Symbol rechts

können Sie sich die Liste der Skripte, welche mit der referenzierten Seite verwaltet werden, anzeigen lassen. Den Skript-Namen brauchen Sie für das *src*-Attribut der *Include*-Anweisung.

7.3. Statische Daten einlesen

In bestimmten Fällen soll ein Skript Daten verarbeiten, welche sich im gleichen JCR befinden. So kann beispielsweise ein File mit Komma-separierten Werten (CSV) im JCR gespeichert sein, welche im Skript ausgewertet werden sollen. Zu diesem Zweck kann ein *Reader* verwendet werden. Die Anweisung `<ethz:reader name="lookup" src="numbers.csv" />` erzeugt einen Reader, mit welchem die Werte im File *numbers.csv* gelesen werden. Auf den Reader kann mit dem Namen *lookup* zugegriffen werden. Die Verarbeitungs-Sequenz könnte wie folgt aussehen:

```
<ethz:reader name="lookup" src="numbers.csv" />
<%
var counter = 0;
var line;
while ((line = lookup.readLine()) != null) {
      out.write("<p>" + counter++ + ": " + line + "</p>");
}
%>
```

Codebeispiel 20: Daten verarbeiten mit Reader

7.4. Übersicht

Tag	Verwendung
`<ethz:include src="form.esp" />`	Bindet das Skript mit dem angegebenen Pfad/Namen in das aktuelle Skript ein. Der Pfad ist relativ zum aktuellen Knoten der ScriptNode-Instanz im JCR.
`<ethz:include src="shared.esp" base="path/to/remote/node" />`	Bindet das Skript mit dem angegebenen Pfad/Namen in das aktuelle Skript ein. Der Pfad ist relativ zum Knoten, welcher im Attribut *base* referenziert wird.
`<ethz:reader name="lookup" src="numbers.csv" />`	Liest den Inhalt des mit *src* referenzierten Files im JCR aus und stellt diesen Inhalt als Reader unter dem angegebenen Namen in den Kontext.

Tabelle 3: Einbinden von Ressourcen

8. Formulare mit ScriptNode

Ein geeigneter Vorgehensfall für den Einsatz der ScriptNode-Komponente sind Formulare, welche sich dynamisch je nach Benutzereingabe ändern. Beispielsweise werden neue

Eingabefelder gezeigt, wenn der Benutzer in einem Auswahlfeld eine bestimmte Wahl trifft. In einem Beispiel sollen diverse der bisher vorgestellten Techniken demonstriert werden.

Das Formular soll den folgenden Verarbeitungsprozess aufweisen:

1. Als Standard-Ansicht soll das Formular angezeigt werden. Für diese Ansicht ist das Skript-Modul *form.esp* verantwortlich.

2. Hat der Benutzer die Formularfelder gefüllt und das Formular abgeschickt, müssen die Eingaben validiert werden. Ist die Benutzereingabe gültig, so soll eine Seite angezeigt werden, auf welcher die vom Benutzer eingegebenen Werte aufgelistet sind (*review.esp*). Der Benutzer soll entscheiden, ob er die Eingabe überarbeiten soll oder ob diese definitiv verarbeitet werden soll.

3. Wurde die Benutzereingabe erfolgreich verarbeitet, so soll dem Benutzer eine Erfolgsmeldung angezeigt werden. Diese Ansicht wird von Skript *feedback.esp* erzeugt.

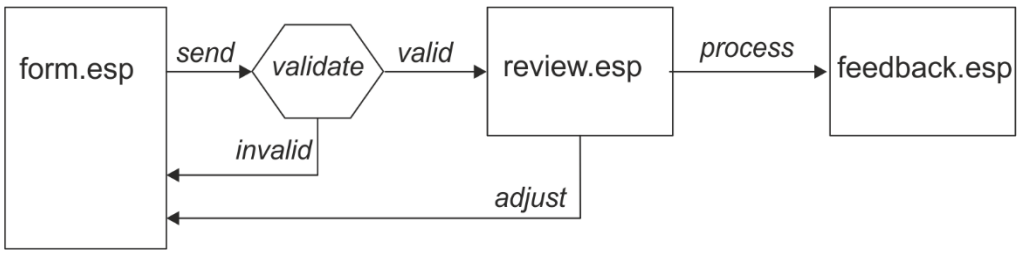

Abbildung 9: Verarbeitungsfluss von Formular

Zusammengehalten werden die Module für die Formularverarbeitung vom Skript *index.esp*. Dieses bildet den Einstiegspunkt der ScriptNode-Komponente. Dieses kann etwa wie folgt aussehen:

```
<ethz:include src="fields.js" />
<%
var params = scriptHelper.getParameters();
var submit = params.get("submit_form");
var validation = scriptHelper.validate(mandatory);
if (submit == "review") {
    if (validation.ok()) {
%>
        <ethz:include src="review.esp" />
<%
    }
    else {
%>
        <ethz:include src="form.esp" />
<%
    }
```

```
}
else if (submit == "formAdjust") {
%>
        <ethz:include src="form.esp" />
<%
}
else if (submit == "formProcess") {
%>
        <ethz:include src="feedback.esp" />
<%
}
else {
%>
    <ethz:include src="form.esp" />
<%
}
%>
```

Codebeispiel 21: Skript für Formularverarbeitung

Im ersten Scriptlet-Bereich werden mit `scriptHelper.getParameters()` die Formularparameter geholt. Im nächsten Schritt wird geprüft, ob unter den Parameter einer mit dem Namen *submit_form* gefunden wurde. Ist dies nicht der Fall, so wird das Skript *form.esp* eingebunden und angezeigt.

Diese Skript soll ein Formular anzeigen, welches mit `<form action="<%=scriptHelper.getPostUrl() %>" method="POST">` konfiguriert ist. Dieses Formular soll mit einer Schaltfläche abgeschickt werden, welches den Namen *submit_form* und den Wert *review* aufweist.

Wird dieses Formular abgeschickt, wird wiederum das Skript *index.esp* aufgerufen. Diesmal spielt die Formular-Validierung eine wichtige Rolle. Mit der Methode `scriptHelper.validate(mandatory)` wird ein Validierungs-Objekt erzeugt. Der Methode wird ein Array übergeben, welcher die Namen aller Felder enthält, welche obligatorisch sind.

Das Validierungs-Objekt besitzt folgendes Interface:

```
public class Validation {
      public boolean ok();
      public boolean fails(String inFieldName);
      public void validateAdd(String[] inAdd);
}
```

Codebeispiel 22: Interface Validierungs-Objekt

Mit `validation.ok()` wird geprüft, ob die Benutzereingabe gültig ist. In diesem Fall soll das Skript aufgerufen werden, welches dem Benutzer die Werte zur Überprüfung und Bestätigung seiner Eingabe anzeigt. Dieses Skript soll ein Formular enthalten, welches die Schaltflächen

„Anpassen“ (zum Verändern der Eingabe) bzw. „Senden“ (zur endgültigen Verarbeitung der Eingabe) aufweist. Diese Schaltflächen sollen wiederum den Namen *submit_form* haben. Der Wert der Schalfläche „Anpassen“ ist *formAdjust*, der Wert der Senden-Schalfläche *formProcess*.

Mit `validation.validateAdd()` wird das Validation-Objekt angewiesen, zusätzliche Felder zu validieren. Dies ist bei Formularen, in welchen sich die Felder dynamisch verändern, nützlich.

Meldet das Validierungs-Objekt eine ungültige Benutzereingabe, so muss das Skript *form.esp* erneut angezeigt werden, diesmal mit entsprechenden Fehlermeldungen versehen. Zu diesem Zweck wird die Methode `validation.fails()` eingesetzt:

```
<%
var validation = scriptHelper.validate(mandatory);
for (var i in mandatory) {
      var field = mandatory[i];
      if (validation.fails(field)) { %>
            <div class="error">
            <%=labelSource.get(field) %>
            <ethz:label value="lblMandatory" /></div><%
      }
}
%>
```

Codebeispiel 23: Formular - Fehlermeldungen anzeigen

Hat der Benutzer seine Eingaben auf der Bestätigungsseite mit einem Klick auf die Senden-Schaltfläche bestätigt, so wird zum Abschluss der Formularverarbeitung das Skript *feedback.esp* aufgerufen. In diesem Skript kann mit `scriptHelper.sendMail()` ein Bestätigungsmail verschickt und danach eine Erfolgsmeldung angezeigt werden.

8.1. Formular-Felder dynamisch füllen

Ein beliebter Vorgehensfall bei Formularen ist, dass man gewisse Felder bei der Anzeige des Formulars (z.B. Name, E-Mail-Adresse) befüllen möchte. Im Falle von Auswahlfeldern möchte man bisweilen die Auswahl eines zweiten Felds durch die Auswahl eines ersten Felds steuern (z.B. die Auswahl der Etagen nach Eingabe eines Gebäudes).

Für den ersten Fall kann das Parameter-Objekt aus `scriptHelper.getParameters()` verwendet werden. Normalerweise stehen in diesem Objekt die Werte, die der Benutzer im Formular eingegeben hat. Das Objekt wird dann verwendet, wenn der Benützer zur Formular-Eingabe zurückkehrt, wenn beispielsweise die Eingabe ungültig ist oder überarbeitet werden soll. Wird das Formular das erste Mal angezeigt, so sind die Felder üblicherweise leer.

Falls sich das Formular auf einer Seite befindet, für welche sich der Benützer anmelden muss, dann kann das System das Parameter-Objekt mit den Werten für den Benutzer-Namen (*name*), -Vornamen (*firstname*) und -E-Mail (*mail*) füllen. Diese Angaben werden aus dem Profil des Benützers geholt, was möglich ist, wenn sich der Benützer zuvor angemeldet hat.

Damit können die entsprechenden Felder wie folgt gefüllt werden:

```
<%
var params = scriptHelper.getParameters();
%>
<input name="name" value="<%=params.get("name") %>" type="text" />
<input name="firstname" value="<%=params.get("firstname") %>" type="text" />
<input name="mail" value="<%=params.get("mail") %>" type="text" />
```

Codebeispiel 24: Felder befüllen

Für das zweite Beispiel, wo die Auswahl-Möglichkeiten eines Auswahl-Felds durch die Eingabe eines ersten Auswahl-Felds gesteuert werden sollen, muss ein sogenannter Proxy eingerichtet werden. Die Funktionalität soll im Browser mit Hilfe von jQuery zur Verfügung gestellt werden.

```
<script type="text/javascript">
  $(document).ready(function(){
      $("select#_step1").change(function() {
          var year = $( "select option:selected" ).text();
          $.ajax({
              url: '<%=scriptHelper.getProxyUrl() %>&year='+year,
              dataType:'JSON',
              success: function(data){
                  var select = $("select#_step2");
                  $(select).empty();
                  for (var i=0; i < data.length; i++) {
                      $("<option/>").val(data[i]['firma_name'])
                           .text(data[i]['firma_text']).appendTo($(select));
                  }
                  $(select).val(data[0]['firma_name']);
                  $(select).focus();
              }
          });
      });
  });
</script>
```

Codebeispiel 25: Steuerung mit jQuery

In diesem Beispiel kann in einem Auswahlfeld mit der ID *_step1* ein Jahr ausgewählt werden. Auf diesem Feld ist ein jQuery-Handler registriert, welcher auf das *change*-Ereignis reagiert. Wird eine Auswahl getroffen, so wird das gewählt Jahr ausgewertet und mit diesem Wert eine Ajax-GET-Abfrage gestartet. Die aufgerufene URL wird mit dem *url*-Konfigurations-Parameter bestimmt. Das Problem ist nun, dass der Browser keine URLs zulässt, welche auf eine andere Domäne verweisen. Damit sollen Cross-Site-Scripting-Probleme (CORS, Cross-Origin Resource Sharing) verhindert werden.

Um dieses Problem zu umgehen, kann auf dem Skript-Knoten eine Proxy-Konfiguration erzeugt werden. Mit Hilfe dieses Proxys kann die Ajax-Abfrage an den WCMS-Server geschickt werden. Es ist in der Folge der WCMS-Server, welcher die Proxy-Konfiguration auswertet, die Abfrage mit allen Parametern an die eingestellte Adresse weiterschickt, die Antwort auswertet und per Ajax-Response an den Browser zurückschickt. Die *success*-Funktion im jQuery-Ajax-Aufruf kann in der Folge diese Antwort auswerten und, wie in diesem Beispiel, die Wertemenge eines Auswahlfelds mit der ID *_step2* befüllen.

Der entscheidende Befehl lautet `url: '<%=scriptHelper.getProxyUrl() %>'`. Auf diese Weise wird eine URL erzeugt, welche ein Servlet des WCMS-Systems aufruft. Dieses ruft seinerseits die konfigurierte Ziel-URL auf. An die Proxy-URL können mit `&name=value` beliebige Parameter angehängt werden. Zusätzlich kann mit dem Parameter `&$add$` ein zusätzlicher Pfad übergeben werden, welcher an die konfigurierte URL angehängt wird. Achtung: vor jedem Parameter muss das &-Zeichen stehen.

Beispiel:

Proxy-Konfiguration-URL:
http://my.site.org/base/path

Skript:
`'<%=scriptHelper.getProxyUrl() %>&$add$=/collection&param1=hello&param2=world'`

Daraus resultiert folgender Aufruf:
http://my.site.org/base/path/collection?param1=hello&param2=world

Die Antwort des Ziel-Systems muss zwingend in JSON-Format erfolgen.

9. Tipps und Tricks

9.1. Debuggen

Bei der Entwicklung komplexer Skripte (z.B. von dynamischen Formularen) ist es bisweilen nützlich, Klarheit über einen bestimmten Wert oder einen Zustand einer Variablen zu bekommen. Dies geschieht am einfachsten, wenn man sich mit `out.write()` den Wert auf die Seite schreiben lässt.

```
<%
var mysteriousObj = ...;
for (var i in mysteriousObj) {
      out.write("<p>Wert: " + mysteriousObj[i] "</p>");
}
%>
```

Codebeispiel 26: Debuggen in Skipt

9.2. Verarbeitung der Formularfelder

Mit `scriptHelper.getParameters()` kann im Skript auf die Formularfelder zugegriffen werden. Das Parameter-Objekt besitzt folgendes Interface:

```
public class FormParameters {
      public boolean has(String inKey);
      public String get(String inKey);
      public String[] getMulti(String inKey);
}
```

Codebeispiel 27: Parameterobjekt für Formularfelder

Mit `FormParameters.has()` kann geprüft werden, ob ein Parameter mit dem übergebenen Namen vorhanden ist. Mit `FormParameters.get()` wird der Wert des Parameters mit dem übergebenen Namen zurückgegeben. Falls kein Feld/Parameter mit den angegebenen Namen vorhanden ist, so wird ein leerer String zurückgegeben. Mit `FormParameters.getMulti()` wird ein Array von Werten zurückgegeben. Dieser Aufruf ist beispielsweise bei einer Auswahl mit mehreren Checkboxen notwendig. Im Fehlerfall wird ein leerer Array zurückgegeben.

9.3. Client-side JavaScript

Der JavaScipt-Code in den Scriptlet-Bereichen wird Serverseitig verarbeitet. Für Webseiten mit fortgeschrittener Funktionalität ist es häufig notwendig, auch clientseitig JavaScript einzusetzen. Clientseitiger JavaScript-Code wird mit der html-Anweisung `<script type="text/javascript">...</script>` eingebunden.

Beispiel:

```
<script type="text/javascript">
        function validateForm() {
            var re_date = /\\d{1,2}?\\.\\d{1,2}?\\.\\d{4}?/;
            var geburtsdatum = $('input[name="geburtsdatum"]');
            return validateField(geburtsdatum, re_date);
        }

        function validateField(field, pattern) {
            var input = field.val();
            if (!input || input.match(pattern)) {
                return true;
            }
            var cssClass = field.attr("class");
            field.removeAttr("class");
            field.attr("class", cssClass + " validation_false");
            field.after('<span class="validation_false"
                 ><%=labelSource.get("errMsgInvalid") %></span>');
            return false;
        }
```

```
    });
</script>
```

Codebeispiel 28: clientseitiger JavaScript

In diesem Beispiel wird im Browser (d.h. clientseitig) die Eingabe der Formularfelds *geburtsdatum* geprüft. Es soll sichergestellt werden, dass nur eine Eingabe mit einem korrekten Datumsformat möglich ist.

In der JavaScript-Funktion *validateForm()* wird mit der jQuery-Expression `$('input[name="geburtsdatum"]')` das Formularfeld geholt.

In der JavaScript-Funktion *validateField()* wird das Format des eingegebenen Werts überprüft und im Fehlerfall mit einer entsprechenden Meldung markiert. Zu bemerken ist, dass mit `<%=labelSource.get("errMsgInvalid") %>` ein serverseitiger Ausdruck eingewoben wird. Für die Fehlermeldung soll ein Label angezeigt werden, welches in der Skript-Komponente konfiguriert ist. Dieser Ausdruck wird auf dem Server verarbeitet. Im Browser kann das JavaScript dann wie gewünscht den Wert des Labels für die Fehlermeldung anzeigen.

9.4. Client-side JavaScript mit jQuery

Da auf den ETH-Webseiten jQuery standardmässig eingebunden wird, kann auch für die clientseitige JavaScript-Programmierung jQuery verwendet werden.