Fakultät für Informatik
und Wirtschaftsinformatik

Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Bachelor Thesis

Name: Adrian Mönnich

Thema: Redesign und Modernisierung von Indicos JavaScript-Framework

Arbeitsplatz: CERN, Genf 23

Referent: Prof. Gremminger

Korreferent: Prof. Dr. Philipp

Abgabetermin: 09.09.2011

Karlsruhe, den 10.05.2011

Der Vorsitzende des
Prüfungsausschusses

Prof. Dr. Ditzinger

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Adrian Mönnich

Karlsruhe, den 6. September 2011

Zusammenfassung

Im Rahmen dieser Bachelor Thesis soll die am CERN entwickelte *Indico*-Software um ein frei verfügbares JavaScript-Framework erweitert werden. Ein solches Framework erlaubt es, browserunabhängiger und effizienter zu entwickeln, als es in reinem JavaScript möglich wäre. Bei Indico handelt es sich um eine Webapplikation zur Planung und Verwaltung von Meetings, Konferenzen und ähnlichen Events, wobei auch die Verwaltung und Reservierung von Konferenzräumen integriert ist.

Zu Beginn werden die genutzten Technologien HTML und JavaScript vorgestellt. Danach werden sowohl das derzeitige, speziell auf Indico zugeschnittene, Framework als auch verschiedene andere Frameworks vorgestellt und anhand verschiedener Kriterien analysiert. Aufbauend auf diese Analyse werden die Vor- und Nachteile der Migration zu einem dieser Frameworks untersucht und anhand dieser ein Framework ausgewählt. Aufbauend auf dem gewählten Framework werden dann Teile von Indico migriert bzw. angepasst.

Die durch diese Thesis erarbeitete Lösung soll dabei eine Grundlage für die Nutzung von verbreitetem, gut dokumentiertem *Third Party*-Code und ein wartbares, entwicklerfreundliches System bieten, welches gleichzeitig auch benutzerfreundlicher als die aktuelle Version ist.

Abstract

The goal of this bachelor thesis is to extend the *Indico* software developed at CERN with a freely available JavaScript framework. Such a framework allows browser-independent and more efficient development than possible by using plain JavaScript. Indico is a web application to plan and manage meetings, conferences and similar events. Besides managing those, it also allows management and reservation of conference rooms.

At first the used technologies HTML and JavaScript are introduced. Then both the current, Indico-specific, framework and various other frameworks are introduced and analyzed according to certain criteria. Based on this analysis the advantages and disadvantages of migrating to one of these frameworks are analyzed and a framework is chosen. By using this framework parts of Indico will be migrated or modified.

The solution developed in this thesis is meant to be a basis for using commonly used well-documented *third party* code and a maintainable developer-friendly system, which will additionally be more user-friendly than the current version.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listingverzeichnis

1. Einleitung

JavaScript ist aus einer modernen Webanwendung inzwischen aus vielerlei Gründen nicht mehr wegzudenken. Es ermöglicht sowohl eine bessere *User Experience* als auch erhöhte Performance. So kann ein Formular schon vor dem Abschicken an den Server validiert werden; dies gibt dem Benutzer ein sofortiges Feedback - möglicherweise sogar noch während der Eingabe - und beansprucht gleichzeitig keine Server-Ressourcen. Darüber hinaus ist es auch möglich, Aktionen auszuführen, ohne dass die komplette Seite neu geladen werden muss. Dies erhöht zum einen das mögliche Arbeitstempo des Benutzers, da er mehrere Aktionen parallel starten kann, ohne jeweils darauf warten zu müssen, dass eine Aktion beendet und die Seite neu geladen ist. Zum anderen spart es Bandbreite, was insbesondere im Zeitalter des mobilen Internets wichtig ist.

Durch die Vielzahl an verschiedener Browser (Firefox, Internet Explorer, Chrome, Safari, ...) und ihren unterschiedlichen JavaScript-Engines gibt es allerdings teilweise deutliche Unterschiede darin, wie man bestimmte Aufgaben mittels JavaScript ausführt. Dazu kommen noch die unterschiedlichen Versionen, die insbesondere beim Internet Explorer fast schon als verschiedene Browser betrachtet werden können. Um bei der Entwicklung auf die Eigenheiten der Browser keine Rücksicht nehmen zu müssen, empfiehlt es sich ein JavaScript-Framework bzw. eine JavaScript-Bibliothek zu nutzen, die die wichtigsten Funktionen browserunabhängig kapselt und dabei oftmals auch Fehler in Browsern umgeht. Die meisten JavaScript-Frameworks sind dabei sehr entwicklerfreundlich gehalten, sodass sie auch unabhängig von der Browserkompatibilität dabei helfen, Zeit und Arbeit bei der Entwicklung einzusparen.

1.1. Zielsetzung

Die *Indico*-Software nutzt JavaScript um Dialogfenster anzuzeigen, Daten automatisch zu aktualisieren und Formulare bereits vor dem Abschicken zu validieren. Dabei wird derzeit ein selbstentwickeltes JavaScript-Framework genutzt, welches größtenteils weder eine Dokumentation noch eine in sich schlüssige API besitzt.

Dieses Framework soll durch ein verbreitetes und gut dokumentiertes OpenSource-Framework ersetzt werden. Sofern das alte Framework nicht vollständig ersetzt werden kann, oder der Entwicklungsaufwand dafür zu hoch wäre, ist auf eine möglichst gute Integration zu achten, sodass kein Framework Konflikte mit dem jeweils anderen verursacht und bestehender Code mit möglichst wenigen Modifikationen weiterhin funktioniert.

Dabei muss zuerst analysiert werden, welche Funktionalität das bestehende Framework zur Verfügung stellt und welche Probleme es gibt, damit bei der Wahl des neuen Frameworks insbesondere darauf geachtet werden kann, dass diese Probleme dort nicht ebenfalls vorhanden sind. Sofern das Framework User-Interface-Elemente („Widgets“) wie beispielsweise Buttons, Dialoge und Tableisten enthält, ist zu prüfen, ob diese sinnvoll genutzt und an das bestehende Design angepasst werden können.

Nachdem ein Framework ausgewählt wurde, muss dieses in die Software integriert werden. Darauffolgend müssen möglicherweise auftretende Konflikte mit bestehendem Code behoben werden. Ebenfalls kann nun analysiert werden, ob es sinnvoll ist, das alte Framework vollständig zu ersetzen oder nur durch das neue zu erweitern bzw. problematische Teile zu ersetzen und andere zu erweitern oder überhaupt nicht zu modifizieren.

1.2. Aufbau der Arbeit

Im Einleitungskapitel wird kurz auf die Aufgabenstellung eingegangen und die groben Schritte zum Ziel beschrieben. Desweiteren wird kurz auf die Firma eingegangen, an der das Projekt durchgeführt wurde. Ebenfalls wird die Software, die im Rahmen dieser Arbeit modifiziert wurde, kurz vorgestellt, sodass man sich einen Überblick darüber verschaffen kann.

Im Grundlagenkapitel werden die genutzen Technologien und ihre Besonderheiten näher betrachtet und erläutert. Nach einem kurzen Überblick über die Scriptsprache JavaScript wird ihr Objektmodell vorgestellt und mit dem anderer objektorientierter Sprachen verglichen. Anhand eines Beispiels wird demonstriert, wie sich die JavaScript-Programmierung für verschiedene Browser unterscheidet, sofern man kein Framework nutzt.

Auf die Funktionen des bestehenden Frameworks wird im dritten Kapitel kurz eingegangen. Desweiteren werden dort die in Frage kommenden Frameworks vorgestellt,

anhand von verschiedenen Gesichtspunkten analysiert und mit dem aktuellen Framework verglichen.

Im vierten Kapitel wird die Entscheidung für das jQuery-Framework in Kombination mit der Underscore.js-Bibliothek begründet, nachdem die Vor- und Nachteile der Migration auf die verschiedenen Frameworks diskutiert wurden.

Das fünfte Kapitel geht auf die eigentliche Migration und die damit verbundenen Schritte, Entwicklungen und Probleme ein.

Im sechsten Kapitel wird der aktuelle Stand mit der ursprünglichen Aufgabenstellung verglichen und kurz auf mögliche zukünftige Erweiterungen eingegangen.

1.3. CERN

Beim CERN (Europäische Organisation für Kernforschung), handelt es sich um eine Forschungseinrichtung im Schweizer Kanton Genf. Der Hauptaufgabenbereich liegt in der Erforschung von Grundlagen der Physik, wobei der weltgrößte Teilchenbeschleuniger *LHC* zum Einsatz kommt.

Neben der physikalischen Forschung wird am CERN auch Software entwickelt, die zwar in erster Linie zur internen Nutzung dient, jedoch oftmals auch als *Open Source* der Allgemeinheit zur Verfügung gestellt wird. Diese Software erstreckt sich über viele Bereiche der IT, so werden am CERN beispielsweise Steuersysteme für den Teilchenbeschleuniger, Business-Software für die Verwaltung des Personals, Grid-Lösungen für die verteilte Datenspeicherung und Webanwendungen für die Archivierung und Indizierung von Medien entwickelt.

Die Abteilung *IT-UDS-AVC*[1] am CERN ist zuständig für die Verwaltung und Einrichtung von video- und telefonbasierten Konferenzsystemen, Aufzeichnung und Onlinestreaming von Vorträgen, Meetings und Konferenzen und die Software zu deren Planung. Ebenfalls im Zuständigkeitsbereich der Abteilung sind die Informationsbildschirme, die in verschiedenen Gebäuden des CERN aufgebaut sind und beispielsweise den Status des *LHC*[2] und für die Mitarbeiter relevante Neuigkeiten anzeigen.

[1] User and Document Services - Audio Visual & Conference Rooms
[2] Large Hadron Collider, der Teilchenbeschleuniger am CERN

1.4. Indico

1.4.1. Überblick

Indico[3] ist eine von der Abteilung *IT-UDS-AVC* am CERN entwickelte Webapplikation, die zum Planen und Organisieren von Events dient. Diese Events können sowohl einfache Vorträge als auch Meetings und Konferenzen mit mehreren Sessions und Beiträgen sein. Die Applikation unterstützt außerdem externe Benutzerauthentifizierung, *Paper Reviewing*[4], elektronische Sitzungsprotokolle und ein Archiv für die in einer Konferenz benutzten Materialien. [Ind11]

Im Laufe der Zeit wurden immer mehr Funktionen hinzugefügt, sodass diese Featureliste schon lange nicht mehr aktuell ist. Inzwischen enthält Indico u.a. ein Buchungs- und Reservierungssystem für Konferenzzimmer, sodass beim Erstellen eines Events sichergestellt werden kann, dass der gewünschte Raum auch verfügbar ist und nicht gerade anderweitig benutzt wird. Ebenfalls in Indico integriert sind die am CERN genutzten Audio- und Videokonferenzsysteme, sodass diese vollautomatisch eingerichtet und aktiviert werden können, sofern diese Systeme im gewählten Raum verfügbar sind. Eines der neuesten Features ist die Möglichkeit, sich die Konferenzzimmer auf einer *Google Maps*-basierten Karte anzeigen zu lassen und anhand der Nähe zu einem bestimmten Gebäude auszuwählen.

1.4.2. Aufbau

Events in Indico sind Teil einer Baumstruktur: Auf der Top-Level-Ebene finden sich ausschließlich Kategorien, die jeweils wieder Kategorien oder beliebige Events (Konferenzen, Meetings und Vorträge) enthalten können. Ein Event wiederum kann diverse Elemente enthalten; welche das sind, hängt vom Typ des Events ab, so kann z.B. nur eine Konferenz ein Registrierungsformular besitzen.

[3] Integrated Digital Conference - http://indico-software.org/
[4] Im Rahmen eines *Call for Papers* oder *Call for Abstracts* bei einer Konferenz

Die folgende Auflistung beinhaltet die wichtigsten Bestandteile von Events:

- Audio-/Videokonferenzen
- *Call for Abstracts*
- Chaträume
- Evaluation
- Materialien
- *Paper Reviewing*
- Registrierung
- Zeitplan

1.4.3. Architektur

Der serverseitige Code von Indico ist in Python geschrieben und nutzt ZODB[5] als Datenbank. Der Code ist in einer Multi-Tier-Architektur organisiert: Die Anfrage des Webservers wird über WSGI[6] an die Applikation weitergegeben, in der die *Request Handler (RH)*-Ebene die Geschäftlogik ausführt. Sie prüft die Zugangsberechtigung des Benutzers, verarbeitet GET- bzw. POST-Daten, baut die Datenbankverbindung auf und führt letztendlich auch die gewünschte Aktion aus. Um eine HTML-Seite auszugeben, nutzt der *Request Handler* die *Web Pages (WP)*-Ebene. Dort werden diverse Aktionen ausgeführt, welche die Anzeige der Daten beeinflussen - zum einen werden die benötigten JavaScript- bzw. CSS-Packages geladen, zum anderen werden, falls vorhanden, Tabs als aktiv markiert oder Daten sortiert. Die eigentliche Erzeugung der HTML-Datei übernimmt die *Components (W)*-Ebene. Jede Klasse dieser Ebene repräsentiert ein Template mit demselbem Namen. In der Regel werden in dieser Ebene nur die aus der WP-Ebene übergebenen Daten an das Template weitergereicht und falls nötig aufbereitet.

[5] http://www.zodb.org
[6] Web Server Gateway Interface

Wie Abbildung 1 verdeutlicht, hat sich Indico im Laufe der Zeit verändert: Statt dem obsoleten *mod_python*[7] wird inzwischen WSGI verwendet.

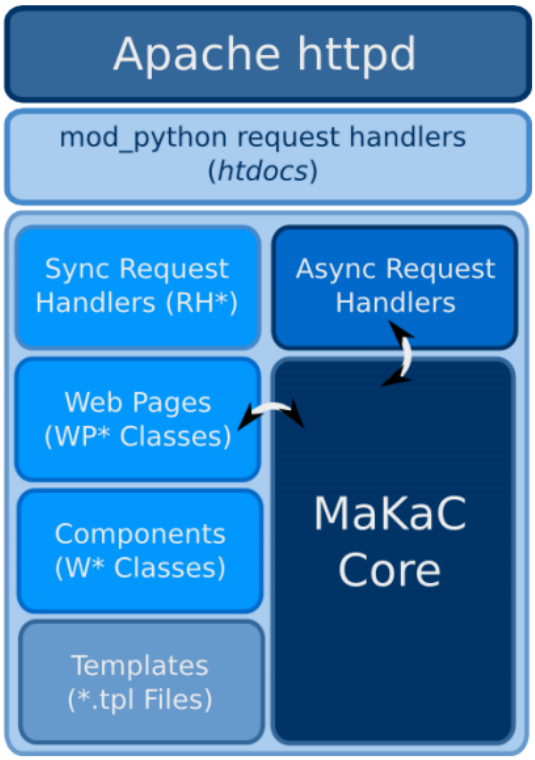

Abbildung 1: Die Architektur von Indico [Fer08]

[7] http://www.modpython.org

2. Grundlagen

2.1. HTML

2.1.1. Geschichte

HTML wurde zusammen mit dem ersten Webbrowser und -server 1990 von Tim Berners-Lee am CERN entwickelt, um Wissenschaftlern zu ermöglichen, digitale Dokumente auszutauschen und durch Querverweise miteinander zu verknüpfen. Da am CERN bereits ein SGML-Dialekt für andere Zwecke benutzt wurde, wurde HTML am ISO-Standard SGML[8] angelehnt, auf dem auch das 1998 vorgestellte XML[9] basiert.

Erstmalig durch ein RFC[10] spezifiziert wurde HTML 1995 in der Version 2.0[11]. 1997 wurde HTML 3.2 als *W3C Recommendation* veröffentlicht[12]. Nur zwei Jahre danach wurde HTML 4.01 ebenfalls als *W3C Recommendation* veröffentlicht[13] und wird auch heute noch aktiv verwendet, obwohl es bereits Nachfolger gibt.

Einer der Nachfolger von HTML 4.01 ist XHTML[14], eine auf XML basierte Version von HTML. Diese Version wurde entwickelt, da XML es ermöglicht, eigene Elemente und Attribute zu definieren und seitens vieler Entwickler der Bedarf nach einer solchen Erweitertungsmöglichkeit bestand. Darüber hinaus ermöglichen die Einschränkungen von XML, einfache Parser zu entwickeln oder einen bestehenden XML-Parser zu nutzen und XHTML-Dokumente in andere Formate zu transformieren; beispielsweise mittels XSLT[15]. [Pem00]

Die neueste HTML-Version ist HTML5 und basiert auf Elementen von HTML 4.01 und XHTML 1.1. Sie befindet sich zwar noch immer in Entwicklung, jedoch werden viele Neuerungen bereits heute von den verbreiteten Browsern unterstützt. Ne-

[8] Standard Generalized Markup Language
[9] Extensible Markup Language
[10] Request for Comments
[11] http://tools.ietf.org/rfc/rfc1866.txt
[12] http://www.w3.org/TR/REC-html32
[13] http://www.w3.org/TR/html401/
[14] eXtensible HyperText Markup Language
[15] Extensible Stylesheet Language Transformations

ben der Einführung neuer Features insbesondere im Multimedia-Bereich spezifiziert HTML5 die Verarbeitung von Dokumenten mit Syntaxfehlern. [W3C11] Diese Dokumente wurden zuvor von fast allen Browsern angezeigt, allerdings gab es gravierende Unterschiede zwischen den verschiedenen Browsern, sodass Seiten oftmals für einzelne Browser optimiert wurden und in anderen Browsern nahezu unbenutzbar waren. Durch die Vorgaben von HTML5 werden Dokumente mit Syntaxfehlern zwar nicht wie in Programmiersprachen mit einer Fehlermeldung abgelehnt - eine radikale Lösung, die eine Vielzahl von Websites unbenutzbar machen würde - jedoch sollen sie in jedem aktuellen Browser identisch aussehen. Anders als HTML 4 und XHTML basiert HTML5 weder auf SGML noch auf XML, sondern definiert eigene Parsingregeln, was unter anderem für die Verarbeitungsregeln bei Syntaxfehlern notwendig ist. Es ist jedoch trotzdem möglich und erlaubt, XML-valides HTML5 zu schreiben, sodass man einen XML-Parser benutzen kann. Diese Variante von HTML5 bezeichnet man auch als *XHTML5*.

Abbildung 2: Das offizielle HTML5-Logo (http://www.w3.org/html/logo/)

2.1.2. Elemente

Ein HTML-Dokument setzt sich aus einigen grundlegenden Bestandteilen zusammen.

Am Anfang des Dokuments steht der *Doctype*. Er teilt dem Parser und der Rendering-Engine mit, um welche HTML-Version es sich handelt.

```
1 <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
    Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
```

Listing 1: Doctype von HTML 4.01

Der Rest des Dokuments ist aus *Tags* aufgebaut, die wiederum *Attribute* besitzen und Text oder weitere Tags enthalten können. Auf der obersten Ebene eines HTML-Dokuments befindet sich immer das <**html**>-Tag. Dieses enthält zwei Tags, <**head**> und <**body**>.

Im *HEAD*-Abschnitt des Dokuments finden sich neben Seitentitel und Metadaten verschiedene Referenzen auf externe Daten, die eingebunden werden sollen. Dabei kann es sich sowohl um Stylesheets handeln als auch um Scripts.

Der *BODY*-Abschnitt enthält den eigentlichen Inhalt des Dokuments. Dieser besteht aus Text, der durch Tags in Absätze, Tabellen, usw. unterteilt werden kann. Ebenfalls mithilfe von Tags lassen sich Bilder, Videos und Audiodateien einbinden.

```
1  <!DOCTYPE HTML>
2  <html>
3      <head>
4          <title>Beispiel</title>
5      </head>
6      <body>
7          <!-- Hier ist das eigentliche Dokument -->
8          <h1>Testabschnitt</h1>
9          <p>Dies ist nur ein Beispiel.</p>
10         <img src="test.png" alt="Testbild">
11     </body>
12 </html>
```

Listing 2: Ein HTML5-Dokument

Genau wie in Programmiersprachen können auch in HTML Kommentare eingefügt werden, die vom Browser ignoriert werden. Eine Ausnahme bilden dabei die *Conditional Comments*[16] im Microsoft Internet Explorer. Diese enthalten spezielle Scriptelemente, die den Kommentar abhängig von der Browserversion ganz normal als Kommentar behandeln oder aber seinen Inhalt wie regulären HTML-Code ausführen, sodass einzelne Elemente nur in bestimmten Browserversionen verarbeitet werden. Da die Syntax so gewählt ist, dass diese Kommentare in anderen Browsern entweder ungültige Tags oder reguläre Kommentare sind, eignen sie sich nicht nur

[16] http://msdn.microsoft.com/en-us/library/ms537512%28v=vs.85%29.aspx

zur Unterscheidung zwischen Internet Explorer-Versionen sondern auch dazu, Code ausschließlich im Internet Explorer auszuführen bzw. nicht auszuführen.

2.1.3. Document Object Model (DOM)

Das W3C[17] definiert das DOM als „a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of [HTML] documents. The document can be further processed and the results of that processing can be incorporated back into the presented page.“ [HWW09]

Es handelt sich beim DOM also um eine standardisierte Schnittstelle, um auf die Elemente eines HTML-Dokuments zuzugreifen und sie zu modifizieren. Das Dokument selbst wird dabei durch ein *Document*-Objekt repräsentiert, welches Methoden und Eigenschaften zum Zugriff auf die enthaltenen Elemente enthält. Beispielsweise sucht `getElementById()` nach einem Element mit einer bestimmten ID und die `documentElement`-Eigenschaft eines Elements verweist das <**`html`**>-Element. Neben den Zugriffsmethoden stellt das DOM-Interface auch Methoden zur Manipulation des Dokuments zur Verfügung. Diese reichen über das einfache Erstellen eines neuen Elements über das Entfernen eines Elements samt untergeordneten Elementen bis zum Einfügen eines Elements an einer bestimmten Position im Dokument.

Da jedes Element (außer dem Dokument selbst) genau ein Elternelement und beliebig viele Kindelemente besitzt, spricht man auch von dem *DOM-Tree*. Jeder Knoten in diesem Baum besitzt neben den bereits genannten Methoden und Eigenschaften weitere Attribute, die Informationen über den jeweiligen Knoten enthalten. Dadurch lässt sich beispielsweise leicht herausfinden ob ein Knoten ein Element ist, ob es sich um reinen Text handelt oder ob es sich gar um ein weiteres Dokument handelt.

2.2. JavaScript

2.2.1. Geschichte

JavaScript wurde 1995 von Brendan Eich für den *Netscape Navigator 2.0* entwickelt. Die Sprache war auch in allen darauf folgenden Versionen enthalten und wurde von Microsoft unter dem Namen *JScript* im *Internet Explorer 3.0* implementiert.

[17] World Wide Web Consortium, http://www.w3.org

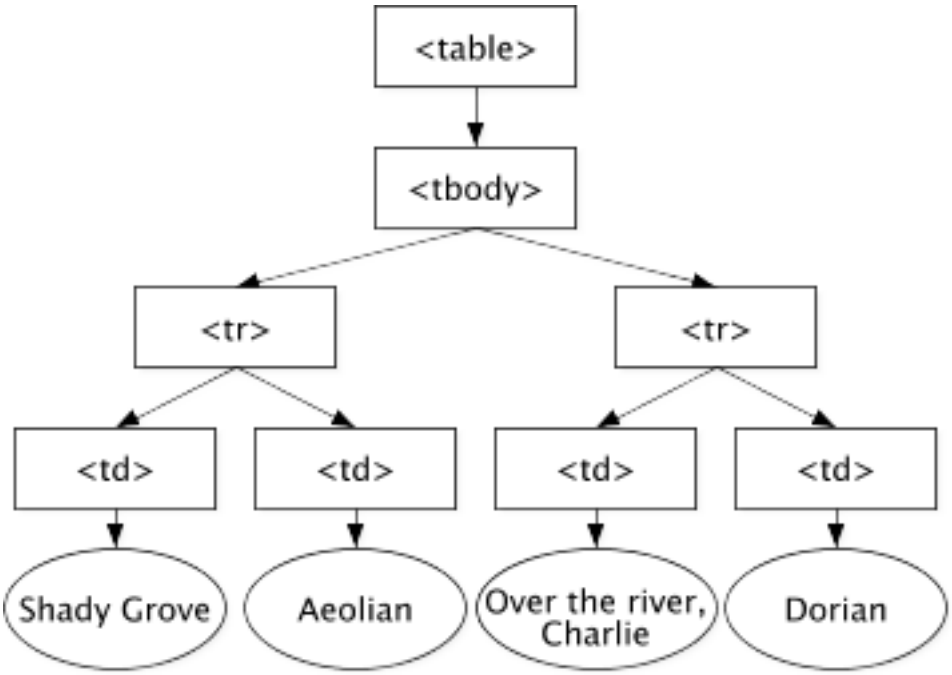

Abbildung 3: Der DOM-Tree einer HTML-Tabelle [NCH+04]

Ab 1996 begann die Standardisierung der Sprache im Rahmen eines ECMA-Standards, welcher im Juni 1997 erschienen ist. Im April 1998 wurde dieser durch die ISO zum internationalen Standard *ISO/IEC 16262* zugelassen.

Im Laufe der Jahre wurde der ECMAScript-Standard immer mehr erweitert und unter anderem reguläre Ausdrücke, Exception-Handling und verbesserte Stringfunktionen hinzugefügt. Die aktuelle Ausgabe 5.1 des ECMAScript-Standards entspricht dem internationalen Standard *ISO/IEC 16262:2011*. [ECM11]

Auch zum Zeitpunkt der Veröffentlichung dieser Arbeit wird der ECMAScript-Standard und damit die Sprache JavaScript weiterentwickelt. Jeder moderne Browser unterstützt die Sprache, allerdings nicht zwangsläufig alle Sprachelemente, die in der aktuellsten Spezifikation enthalten sind.

2.2.2. Anwendungsgebiete

Der bekannteste und häufigste Verwendungszweck von JavaScript ist Web-Scripting innerhalb von Browsern. Dies ist nicht weiter verwunderlich ist, da die Sprache ursprünglich für diesen Zweck entwickelt und lange Zeit ausschließlich in diesem Kontext benutzt wurde. Dabei stellt der Webbrowser die Host-Umgebung, also die Objekte und Funktionen zu Kommunikation mit der Applikation, in der die Scripts

ausgeführt werden. Im Webbrowser sind solche Objekte beispielsweise das Browserfenster (`window`), Popupfenster (Rückgabewert von `window.open()`), die Browserhistory (`history`), das aktuelle HTML-Dokument (`document`). Ebenfalls vom Browser bereitgestellt werden Objekte, die die einzelnen HTML-Elemente repräsentieren, also zum Beispiel Hyperlinks, Formulare und Bilder. Darüber hinaus bietet der Browser Methoden, um über verschiedene Ereignisse wie Klicks, Mausbewegungen und Änderungen an Formularfeldern benachrichtigt zu werden. [ECM11, Kap. 4.1]

Ein weiteres Anwendungsgebiet ist *Server-Side JavaScript* (*SSJS*). Dabei wird JavaScript direkt auf den (Web-)Server ausgeführt und kann direkt auf Datenbanken und Dateien (des Servers) zugreifen. Während *SSJS* erstmalig 1996 im *Netscape Enterprise HTTP Server* zum Einsatz kam, erlebte es erst fast 15 Jahre später seinen Durchbruch. Mit *node.js*[18] existiert ein quelloffenes, plattformunabhängiges Framework, welches leichtgewichtig ist und dank einer eventbasierten Architektur auch bei einer sehr großen Anzahl gleichzeitiger Clientverbindungen noch performant ist, eine Eigenschaft die in klassischen prozess- oder threadbasierten Webservern wie *Apache* oftmals nicht gegeben ist. Die serverseitige Host-Umgebung unterscheidet sich logischerweise stark von der eines Browsers, da ein Webserver weder mit Fenstern noch HTML-Dokumenten arbeitet. Dafür kennt er u.a. HTTP-Requests und Formular*daten*. Die Ereignisse, auf die Scripte reagieren können, sind dabei primär netzwerkbezogen, zum Beispiel die neue Verbindung eines Clients oder der vorzeitige Abbruch eines Seitenaufrufs. Es ist allerdings durchaus möglich, dass mit entsprechendem Code Teile der Browserumgebung serverseitig nachgebildet werden. So ist es denkbar, ein HTML-Dokument nicht als String sondern als DOM-Tree darzustellen. In diesem Fall könnten dieselben Methoden wie im Browser zur Verfügung gestellt werden, damit Teile des Codes sowohl client- als auch serverseitig nutzbar sind.

2.2.3. Programmierparadigmen

Das Programmierparadigma einer Programmiersprache ist „die Sichtweise auf und den Umgang mit den zu verarbeiteten Daten und Operationen“. [HV06, Kap. 1.3.1] In JavaScript kann man man sich zwischen drei dieser Paradigmen entscheiden:

Imperative/prozedurale Programmierung

Bei der imperativen Programmierung wird eine Folge von Anweisungen sequen-

[18] http://nodejs.org

tiell abgearbeitet. Zur Kapselung und Wiederverwendung von Funktionalität werden Funktionen genutzt. [HV06, Kap. 1.3.1]

Da die Host-Umgebung in JavaScript Objekte zur Verfügung stellt und auch Datentypen wie Strings und Arrays Objekte sind, lässt sich zwar nicht rein prozedural programmieren, es kann jedoch vollständig auf die Nutzung eigener Objekte verzichtet werden. Da dadurch viel Komfort, wie die Nutzung assoziativer Arrays (die in JavaScript simple Objekte sind), verloren geht, ist dieses Paradigma nicht zu empfehlen. Es kann aber durchaus sinnvoll sein, Funktionen außerhalb von Objekten zu definieren, sofern sie alleinstehend sind und man keine Namespaces verwendet werden: JavaScript unterstützt zwar keine Namespaces, allerdings sind Funktionen sog. *First-Class-Objekte*, d.h. sie können in Variablen gespeichert werden und haben noch weitere Eigenschaften, auf die später näher eingegangen wird. Aufgrund der Möglichkeit, Funktionen in Variablen zu speichern, kann man Namespaces einfach durch assoziative Arrays simulieren, indem man die Funktionen in solchen Arrays speichert und sie daher nicht mehr im globalen Kontext liegen, sondern nur über das Array aufrufbar sind.

Objektbasierte Programmierung

Eine objektbasierte Programmiersprache unterstützt Objekte, kennt im Gegensatz zu einer objekt*orientierten* Programmiersprache allerdings keine Klassen. [HV06, Kap. 1.3.1]

Nachdem zuvor schon Objekte erwähnt wurden, die von der Host-Umgebung zur Verfügung gestellt werden, ist es nur logisch, dass der Entwickler auch selbst Objekte definieren kann. Dies geschieht anders als in klassischen objektorientierten Programmiersprachen wie Java oder C++ nicht über Klassen, sondern erfolgt hier über Objekte und Konstruktoren (die wiederum ganz normale Funktionen sind). Da das Objektsystem von JavaScript einzigartig ist und einige Besonderheiten besitzt, wird im Verlauf dieses Kapitels näher darauf eingegangen.

Funktionale Programmierung

Eine rein funktionale Programmiersprache basiert nicht auf Wertzuweisungen sondern benutzt ausschließlich Funktionsdefinitionen, die eine Eingabe in eine Ausgabe transformieren. [HV06, Kap. 1.3.1]

JavaScript ist allerdings nicht direkt eine funktionale Programmiersprache und damit erst recht keine rein funktionale Programmiersprache. Dadurch, dass

Funktionen *First-Class-Objekte* sind und sowohl anonyme Funktionen als auch Closures unterstützt werden, kann man in JavaScript sehr einfach funktionale Elemente mit objektorientierten bzw. prozeduralen Elementen mischen. Die prominentesten aus anderen funktionalen Programmiersprachen bekannten Methoden sind `forEach()`, um eine Funktion alle Elemente einer Liste anzuwenden, `map()`, um eine Liste mittels einer Funktion in eine neue Liste zu transformieren und `filter()`, um nur die Elemente aus einer Liste zu übernehmen, die von der Filter-Funktion durch die Rückgabe von **`true`** akzeptiert wurden.

2.2.4. First-Class-Objekte

In JavaScript handelt es sich bei Funktionen um sogenannte *First-Class-Objekte*. Dies bedeutet an sich nur, dass es sich bei Funktionen um Objekte handelt. So kann zum Beispiel eine Funktion wie jedes andere Objekt Attribute und Methoden, die wiederum Funktionen sind, besitzen. Details zu den verschiedenen Objekt-Typen einschließlich Funktionsobjekten finden sich in [ECM11, Kap. 8.6].

First-Class beschreibt Eigenschaften die man bei Objekten im Allgemeinen erwartet, im Zusammenhang mit Funktionen allerdings durchaus ungewöhnlich sind:

- Sie können in Variablen und anderen Datenstrukturen (beispielsweise Arrays) gespeichert werden.

```
1 var foo = someFunction;
```

Listing 3: Zuweisung einer Funktion an eine Variable

- Sie können Funktionsparameter sein.

```
1 var roots = [4, 9, 16].map(Math.sqrt);
```

Listing 4: Erstellen einer neuen Liste mittels einer Transformationsfunktion

- Sie können Rückgabewert einer Funktion sein.
- Sie können zur Laufzeit erstellt werden.

```
1 function makeFunction() {
2     function newFunction() {}
3     return newFunction;
4 }
```

Listing 5: Eine Funktion die eine neue Funktion erstellt und zurückgibt

- Sie sind nicht an einen Namen gebunden.

```
1 function makeFunction() {
2     return function() {};
3 }
```

Listing 6: Erstellen einer anonymen Funktion

2.2.5. Native Objekte

Wie zuvor erwähnt, sind Strings und Arrays Objekte. Diese sind jedoch nicht in JavaScript selbst implementiert, sondern in der Sprache der JavaScript-Engine, also beispielsweise C oder C++. Ihr Verhalten ist durch die ECMAScript-Spezifikation [ECM11] vollständig definiert. Dies bedeutet, dass es prinzipiell sicher ist, gewisse Operationen mit diesen Objekten auszuführen, um sie z.B. mit eigenen Methoden zu erweitern. Sofern ein natives Objekt eine Operation nicht unterstützt, wird eine entsprechende Fehlermeldung ausgegeben.

2.2.6. Host-Objekte

Bei Host-Objekten handelt es sich um von der Host-Umgebung zur Verfügung gestellte Objekte, die in der Regel nicht in JavaScript, sondern in der Sprache der Host-Umgebung oder der JavaScript-Engine geschrieben sind. Diese Objekte sind auf die Umgebung zugeschnitten in der das Script läuft - im Browser werden z.B. die DOM-Elemente durch Host-Objekte repräsentiert. Während native Objekte durch die ECMAScript-Spezifikation definiert sind, gibt es für Host-Objekte lediglich einige Anforderungen und Einschränkungen, die notwendig sind, damit die JavaScript-

Engine korrekt mit ihnen arbeiten und die Codeausführung optimieren kann: Alle internen Objekteigenschaften müssen definiert und gewisse Invarianten erfüllt sein.

Das Verhalten dieser Objekte ist jedoch nicht spezifiziert: „Host objects may implement these internal methods in any manner unless specified otherwise“ [ECM11, Kap. 8.6.2]; die „internal methods“ sind beispielsweise die Getter und Setter von Objekteigenschaften. Darüber hinaus dürfen auch die Auswirkungen von internen Eigenschaften erweitert werden: „Host objects may support these internal properties with any implementation-dependent behaviour as long as it is consistent with the specific host object restrictions stated in this document.“ [ECM11, Kap. 8.6.2]

Ein Beispiel für das undefinierte Verhalten von Host-Objekten findet sich im Blog[19] eines ehemaligen Entwicklers eines JavaScript-Frameworks, welches im dritten Kapitel noch näher betrachtet wird:

```
1 document.createElement('p').offsetParent; // "Unspecified
    error."
2 new ActiveXObject("MSXML2.XMLHTTP").send; // "Object
    doesn't support this property or method"
```

Listing 7: Verhalten von Host-Objekten im Internet Explorer

In diesem Beispiel wird auf nicht vorhandene Eigenschaften bzw. Methoden von zwei verschiedenen Host-Objekten lesend zugegriffen. Native und benutzerdefinierte Objekte würden in diesem Fall einfach `undefined` zurückgeben statt eine Exception auszulösen. Es ist auch durchaus möglich, dass eine Wertzuweisung keine Fehlermeldung bzw. Exception auslöst, sondern stillschweigend ignoriert wird und die Eigenschaft des betroffenen Objekts ihren alten Wert beibehält.

2.2.7. Closures

Normalerweise unterscheidet man in der Programmierung zwischen lokalen Variablen, die nur innerhalb einer Funktion sichtbar und lebendig sind, globalen Variablen, die überall sichtbar sind und Membervariablen, die nur über ein Objekt sichtbar sind. Wenn keine Funktionen zur Laufzeit definiert werden, reichen diese Variablentypen auch aus. Wenn man jedoch Funktionen zur Laufzeit definiert, besteht die

[19] http://perfectionkills.com/whats-wrong-with-extending-the-dom

Möglichkeit, dass dies in einer Funktion geschieht, die bereits lokale Variablen besitzt. In der inneren Funktion sind diese Variablen nun ebenfalls verfügbar. Daher dürfen sie vom Garbage Collector nicht gelöscht werden, obwohl es *funktionslokale* Variablen sind und die Funktion, in der sie definiert wurden, bereits verlassen wurde. Die Kombination aus einer Funktion und ihrer Umgebung (also den außerhalb definierten Variablen) bezeichnet man als *Closure*. Die innerhalb der Funktion verfügbaren Variablen, die weder lokal noch global sind, bezeichnet man als *nichtlokale* Variablen.

Im folgenden Beispiel wird eine Closure genutzt, um eine Funktion zu erstellen, die eine Variable in ihrer Nutzung insofern einschränkt, dass sie nicht direkt gelesen oder geschrieben werden kann sondern nur mittels der zurückgegebenen Funktion ausgelesen werden kann, wobei ihr Wert sich bei jedem dieser Vorgänge um `1` erhöht.

```
1 function makeIncrementer(val) {
2     return function() {
3         return ++val;
4     }
5 }
```

Listing 8: Beispiel für eine Closure

Eine mögliche Nutzung von Closures ist also die Kapselung von Variablen; ein weiterer Anwendungsbereich wird später zusammen mit dem Objektmodell von JavaScript vorgestellt.

2.2.8. Objekte, Konstruktoren und Prototypen

Anders als objektorientierte Sprachen wie Java oder C++ ist JavaScript objekt*basiert*. Dies bedeutet, dass JavaScript keine Klassen besitzt wie bereits bei der Vorstellung der Programmierparadigmen erwähnt wurde. Um dennoch ein neues Objekt zu erstellen, was in etwa der *Instanz* einer Klasse entspricht, benutzt man den **new**-Operator und eine Konstruktorfunktion `F`. Der Operator erstellt ein neues Objekt `obj` und weist seiner internen Eigenschaft *[[Prototype]]* den Wert der Eigenschaft `F.prototype` zu, sofern er existiert und ein Objekt ist. Danach wird `F` ausgeführt, wobei **this** `===` `obj` ist. Der Rückgabewert des **new**-Operators ist `obj`.

Es besteht also eine Möglichkeit, ein neues Objekt zu erstellen und mittels einer Funktion auf dieses Objekt zuzugreifen bzw. es danach einer Variable zuzuweisen.

Da ein Objekt ohne Methoden allerdings kaum mit einer Klasse vergleichbar ist, muss JavaScript logischerweise eine Möglichkeit bieten, einem Objekt Methoden zuzuweisen. Eine Möglichkeit wäre, sie wie in Listing 9 im Konstruktor dem Objekt hinzuzufügen, was dank der *First-Class*-Eigenschaften von Funktionen auch ohne Weiteres möglich ist. Allerdings hat diese Herangehensweise einen entscheidenden Nachteil: Für jedes Objekt wird eine neue Funktion erstellt, d.h. bei vielen Objekten ist diese Lösung sehr speicherineffizient.

```
1 function SomeClass {
2     this.doStuff = function() { };
3 }
```

Listing 9: Funktionszuweisung im Konstruktor

Für eine effizientere Lösung ist es wichtig zu wissen, was beim Zugriff auf eine Objekteigenschaft geschieht: Die JavaScript-Engine führt die interne Methode *[[GetProperty]]* des Objekts aus. Diese Methode prüft, ob das Objekt selbst die gewünschte Eigenschaft besitzt und gibt sie ggf. zurück. Ansonsten wird überprüft, ob die *[[Prototype]]*-Eigenschaft des Objekts `null` ist. Falls sie das ist, gibt *[[GetProperty]]* `undefined` zurück. Existiert jedoch ein Prototyp, so wird die *[[GetProperty]]*-Methode des Prototypen ausgeführt und ihr Rückgabewert zurückgegeben. Es werden also alle Prototypen rekursiv durchlaufen, bis entweder die Eigenschaft gefunden wurde oder ein Objekt ohne Prototyp erreicht wurde.

Da Methoden ebenfalls Eigenschaften sind, wird durch die Nutzung der *prototype chain* sowohl der erhöhte Speicherverbrauch verhindert, als auch Vererbung und Überschreiben von Methoden ermöglich, da die Funktion nur noch in einem einzigen Objekt, dem Prototypen, enthalten ist.

```
1 function SomeClass { }
2 SomeClass.prototype.doStuff = function() { };
```

Listing 10: Funktionszuweisung im Prototypen

Zur Vererbung weist man dem Prototypen der Konstruktorfunktion einfach ein (neues) Objekt zu, welches die entsprechenden Funktionen enthält, und fügt die neuen Funktionen danach diesem Objekt hinzu.

```
1 function Parent() { }
2 Parent.prototype.say = function() { print('Hi'); }
3 function Child() {
4     this.greeting = 'Hallo';
5 }
6 Child.prototype = new Parent();
7 Child.prototype.greet = function() { print(this.greeting);
    }
8 c = new Child();
9 c.greet(); // Hallo
10 c.say(); // Hi
11 Child.prototype.say = function() {
12     Parent.prototype.say.call(this);
13     this.greet();
14 }
15 c.say(); // Hi Hallo
```

Listing 11: Prototypische Vererbung

Listing 11 demonstriert Vererbung und Überschreiben von Funktionen. Bei `this` handelt es sich um eine spezielle Variable, die jeweils auf den aktuellen Objektkontext zeigt. Dies ist bei Funktionen, die direkt mittels `func();` aufgerufen werden das globale Objekt - im Browser ist dies `window`; im Fall eines Methodenaufrufs über `obj.method();` ist `this === obj`. Dabei ist zu beachten, dass eine innere Funktion ein neues `this` besitzt; falls das der äußeren Klasse genutzt werden soll, muss es unter einem anderen Namen in einer Closure gesichert werden. Häufig wird dazu wie in Listing 12 der Name `self` genutzt. Um eine überschriebene Methode des Elternobjekts aufzurufen, kann man nicht `this.method();` nutzen, da `this.method` ja auf die neue Funktion zeigt. Stattdessen muss über `ParentObject.prototype.method` auf die Methode zugreifen. Wenn man diese jedoch direkt ausführen würde, wäre `this === ParentObject.prototype`, was eigentlich nie gewollt ist. Daher nutzt man die `call(thisArg, ...)`-Methode, die jede Funktion besitzt und einem ermöglicht, den Wert von `this` innerhalb der

aufgerufenen Funktion manuell festzulegen.

```
1 Test.prototype.func = function() {
2     var self = this;
3     return function() {
4         self.doSomething();
5     }
6 };
```

Listing 12: Sichern von this mittels einer Closure

Verglichen mit klassenbasierten objektorientierten Sprachen ist das Objektsystem von JavaScript sehr flexibel, da alle Operationen, vom Hinzufügen von Methoden zu einer „Klasse“ bis zum Erstellen einer neuen „Klasse“, zur Laufzeit möglich sind. Diese Flexibilität hat jedoch den Nachteil, dass statische Analysen und aus IDEs[20] bekannte Funktionen wie Autovervollständigung von Methoden und Eigenschaften nicht immer möglich sind, da der Code für solche Funktionen in der Regel nicht ausgeführt sondern nur anaylsiert wird.

2.2.9. Browserspezifische Unterschiede

Aufgrund der verschiedenen Browser-Rendering-Engines, die meist auch unterschiedliche JavaScript-Engines besitzen, gibt es trotz vorhandenen Standards wie [ECM11] und [NCH+04] Unterschiede, die bei der JavaScript-Programmierung beachtet werden müssen, sofern man kein Framework nutzt, welches die entsprechenden Operationen abstrahiert bzw. kapselt. Listing 13 zeigt ein Beispiel für die unterschiedlichen Funktionen, die man nutzen muss, um auf Events zu reagieren, sofern man auch den Internet Explorer 8 (oder älter) unterstützen möchte. Wie sich dieselbe Funktionalität durch ein Framework realisieren lässt, ist in Listing 14 gezeigt.

[20] Integrated Development Environment; z.B. Eclipse oder Visual Studio

```
1 function handleClick(event) {
2     if(!event) {
3         event = window.event;
4     }
5     // [...]
6 }
7
8 if(el.addEventListener) {
9     el.addEventListener('click', handleClick, false);
10 }
11 else if (el.attachEvent) { // IE <=8
12     el.attachEvent('onclick', handleClick);
13 }
```

Listing 13: Event-Handling ohne Abstraktion

```
1 $(el).bind('click', function(event) {
2     // [...]
3 });
```

Listing 14: Event-Handling mit jQuery

3. JavaScript-Frameworks

3.1. Vergleichskriterien

Um die Frameworks miteinander vergleichen zu können, müssen einige Kriterien festgelegt werden, anhand derer sich alle Frameworks messen lassen.

DOM-Zugriff

JavaScript-Frameworks kapseln den Zugriff auf Elemente in der Regel über Funktionen, die ein oder mehrere Elemente anhand eines CSS-Selektors suchen und zurückgeben. Dabei können neben den nativ implementierten CSS-Selektoren auch weitere Selektoren unterstützt werden.

DOM-Manipulation

Häufig möchte man mit JavaScript neue HTML-Elemente erstellen oder vorhandene Elemente verändern. Dabei kann das Framework den Entwickler unterstützen, indem es z.B. das Verknüpfen von Elementen und die Veränderung von CSS-Attributen vereinfacht.

DOM Traversal

Unter *DOM Traversal* versteht man das Navigieren durch den DOM-Tree, ausgehend von einem bestimmten Element. Dabei gibt es sowohl einige Standardfunktionen, die man in jedem Framework erwarten kann, als auch Komfortfunktionen.

Events

Wie bereits in Abschnitt 2.2.9 gezeigt, unterscheiden sich die Eventsysteme der Browser teilweise. Daher ist dies ein Bereich, in dem ein Framework sowohl die Browserunterschiede verbergen als auch den Komfort erhöhen sollte.

Objektsystem

Manche JavaScript-Frameworks bringen ein eigenes Objektsystem mit, sodass die Arbeit mit Prototypen und Konstruktoren vereinfacht wird und beispielsweise Mixin-Objekte unterstützt werden oder der Aufruf einer überschriebenen Method der Parent-Klasse komfortabel möglich ist.

Hilfsfunktionen

Die meisten JavaScript-Frameworks bieten neben der bereits genannten Funktionalität weitere Funktionen, die oftmals einen funktionalen Programmierstil vereinfachen oder komfortabler machen. Allerdings sind nicht nur die funktionalen Funktionen, sondern auch jegliche anderen Hilfsfunktionen betrachtenswert.

UI-Elemente

Einige JavaScript-Frameworks enthalten UI-Elemente, um z.B. Dialoge oder Buttons zu erzeugen. Die Frameworks unterscheiden sich dort sowohl im Umfang als auch in der Anpassbarkeit.

Kompatibilität

Bei der Integration in Indico ist es von Vorteil, wenn das Framework möglichst wenig Potenzial besitzt, Konflikte zu verursachen. Dies kann durch verschiedene Ansätze erreicht werden.

Sonstige Features

Viele JavaScript-Frameworks haben neben den üblichen Features zusätzliche Funktionen, die in anderen Frameworks nicht vorhanden sind.

Performance

Die unterschiedlichen Frameworks sind bei verschiedenen Operationen wie beispielsweise dem Zugriff auf Elemente anhand eines CSS-Selektors unterschiedlich schnell. Allerdings spielt die Performance im Rahmen dieser Arbeit nur eine untergeordnete Rolle, da Indico nirgends extrem viele Aktionen auf einmal ausführt und Performanceunterschiede somit für den Endbenutzer nicht spürbar sind, was auch den immer schnelleren JavaScript-Engines zu verdanken ist.

Dokumentation

Da mehrere Entwickler mit dem Framework arbeiten müssen und starke Fluktuation herrscht, da oftmals Studenten für 3 bis 12 Monate an Indico arbeiten, ist eine gute Dokumentation wichtig, da es nicht sehr produktiv ist, wenn man erst den Quellcode des Frameworks lesen und verstehen muss, um es benutzen zu können. Insbesondere ist eine Dokumentation hilfreich, wenn sie für alle Funktionen des Frameworks Beispielcode enthält.

Lizenz

Die meisten Frameworks sind unter einer OpenSource-Lizenz verfügbar. Da

Indico unter der GNU GPL[21] steht, ist auf Kompatibilität mit dieser Lizenz zu achten. Abbildung 4 bietet einen kurzen Überblick über die verbreitetsten Open Source-Lizenzen und zeigt die Kompatibilität: „To see if software can be combined, just start at their respective licenses, and find a common box you can reach following the arrows.“ [Whe07]
Da Indico unter der GPLv2+, d.h. „either version 2 of the License, or (at your option) any later version“, lizenziert ist, sind alle Open Source-Lizenzen außer der *Affero GPL*[22] kompatibel.

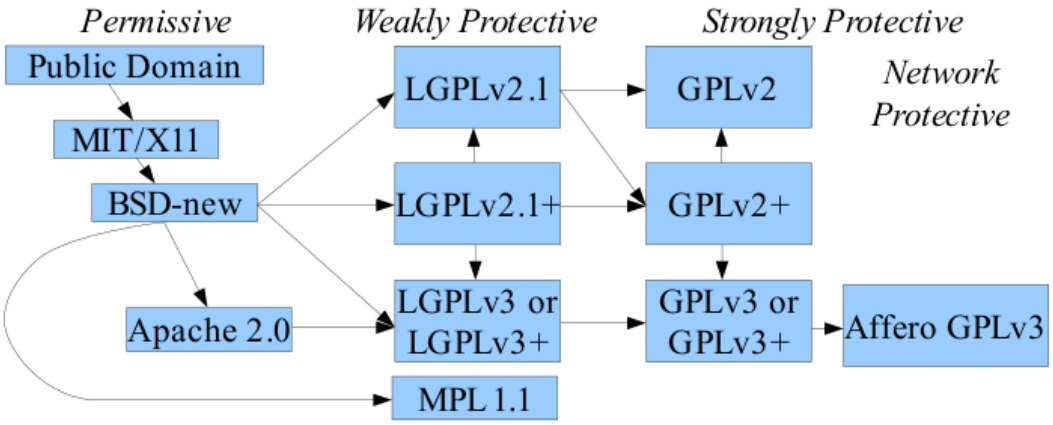

Abbildung 4: Kompatibilitätsdiagramm div. Open Source-Lizenzen [Whe07]

3.2. Indico

Das derzeit in Indico verwendet Framework besteht aus einem Loader-Script und ca. 50 JavaScript-Dateien, die das eigentliche Framework enthalten. In den HTML-Seiten von Indico wird das Loader-Script jedoch nur im Entwicklungsmodus eingebunden; auf dem Produktivsystem werden die einzelnen JavaScript-Dateien in einer einzelnen Datei zusammengefasst und komprimiert.

Das Framework ist in mehrere Module aufgeteilt. Im *Core* werden Interfaces, Iteratorfunktionen, zusätzliche Stringfunktionen und diverse Helferfunktionen implementiert. Ebenfalls in diesem Modul befindet sich das Objektsystem, welches in Indico genutzt wird. Das *Data*-Modul enthält ein *Data Binding*-Framework, Funktionen

[21] http://www.gnu.org/licenses/gpl-2.0.txt
[22] GPL mit der Erweiterung, dass auch bei einem Netzwerkzugriff auf das laufende Programm eine „Verbreitung“ stattfindet und der Quellcode zugänglich gemacht werden muss.

um serverseitige Aktionen via JSON-RPC[23] auszuführen und diverse Funktionen zum Verarbeiten von diverser Datentypen wie `Date`-Objekten und JSON.

Das größte und im weiteren Verlauf dieser Arbeit wichtigste Modul ist das *UI*-Modul. Es enthält die in Kapitel 2 erwähnte Kapselung der teilweise browserspezifischen Methoden um auf das Dokument zuzugreifen und Komfortmethoden zur Erzeugung von DOM-Elementen. Darüber hinaus hat das Modul verschiedene Submodule: *Draw* abstrahiert das Erstellen von SVG-Grafiken bzw. im Internet Explorer VRML-Grafiken. *Extensions* und *Styles* erweitern einige Objekte des *UI*-Moduls um nützliche Methoden. Das *Widgets*-Submodul enthält Funktionen, um HTML-Elemente mit häufig benutzen Funktionen bzw. Eventhandlern zu verknüpfen oder mehrere HTML-Elemente in einer bestimmten Art und Weise zusammenzufügen.

DOM-Zugriff

Das Indico-Framework stellt drei Methoden zur Verfügung, um einzelne DOM-Elemente zu selektieren. Die Funktion `$E(id)` sucht ein Element anhand der eindeutigen ID und gibt dieses in einen Wrapper verpackt zurück; alternativ kann auch ein DOM-Elemente an die Funktion übergeben werden. `$N(name)` gibt eine Liste aller Elemente zurück, die den passenden Namen haben. Neben diesen beiden Funktionen gibt es mit `getElementsByClassName(class)` eine weitere Funktion, die alle Elemente mit der angegebenen CSS-Klasse zurückgibt. Eine Funktion, um Elemente anhand eines CSS-Selektors zu finden, existiert nicht. Das Wrapper-Objekt `XElement`, in welches alle drei Funktionen die gefundenen DOM-Elemente verpacken, enthält neben der Referenz auf das DOM-Element diverse Hilfsmethoden, um die Arbeit mit dem Element zu erleichtern.

DOM-Manipulation

Indico besitzt ein globales Objekt `Html`, welches Methoden für die meisten HTML-Elemente besitzt. Diese akzeptieren beliebig viele Argumente, wobei das erste Argument jeweils ein Objekt ist, welches die Attribute enthält, die das Element nach dem Erstellen besitzen soll. Beim `style`-Attribut ist statt der bei anderen Attributen benötigten Strings oder Zahlen auch ein Objekt zulässig, welches die gewünschten CSS-Eigenschaften enthält.

Die übrigen Argumente werden zu den Child-Elementen des neuen Elements; es kann sich dabei sowohl um DOM-Elemente als auch um `XElement`-Objekte handeln. `XElement`-Objekte handeln.

[23] Remote Procedure Call via AJAX+JSON

Zum Verändern der Eigenschaften von Elementen muss über das `dom`-Attribut des Wrappers direkt auf das DOM-Element zugegriffen werden; die Manipulation des DOM-Trees selbst ist durch Methoden wie `detach()`, `append()`, `insert()`, `remove()` und `replaceWith()` direkt über den Wrapper möglich. Er enthält ebenfalls eine Hilfsmethode `ancestorOf()` um zu überprüfen, ob ein Element ein anderes Element enthält, wobei es sich nicht um ein direktes Unterelement handeln muss.

DOM Traversal

`XElement`-Objekte besitzen einige wenige Methoden, um durch den DOM-Tree zu navigieren. Mit `each()` wird eine Callbackfunktion für jedes Child-Element aufgerufen, `getParent()` gibt das Parent-Element im Wrapper zurück. Methoden für den Zugriff auf Geschwisterelemente fehlen ebenso wie Methoden um ausgehend von einem Element ein anderes Element zu suchen.

Events

`XElement` stellt einige Funktionen zum Registrieren von Eventhandlern zur Verfügung. Für einige häufig genutze Events wie *click*, *change* und *keypress* gibt es spezielle Funktionen; für alle anderen Events gibt es die Funktion `observeEvent()`, welche ein Callback für ein beliebiges Event registrieren kann. Statt die DOM-Funktion `addEventListener()` zu nutzen, wird hierbei jedoch die `onEVENTNAME`-Eigenschaft des DOM-Elements benutzt mit dem Nachteil, dass der Aufruf mehrerer Handler vom Framework gehandhabt werden muss, da die Eigenschaft jeweils immer nur auf eine Funktion zeigen kann. Das Framework stellt sicher, dass das erste Argument der Funktion jeweils das Event-Objekt ist, jedoch zeigt **`this`** nicht auf das Element. Dadurch, dass das Event-Objekt unverändert weitergegeben wird, sind auch zusätzliche Hilfsfunktionen notwendig, um browserunabhängig auf das Zielelement des Events oder die gedrückte Taste im Falle eines *KeyPress*-, *KeyDown*- oder *KeyUp*-Events zuzugreifen.

Objektsystem

Das Objektsystem des Indico-Frameworks simuliert Klassen und Mixins, wobei es sich bei letzteren ebenfalls um theoretisch eigenständig verwendbare Klassen handelt. Objekte werden mit der Funktion `type(name, mixins, members, ctor)` erstellt, wobei `name` der Name der Klasse und `mixins` eine Liste der Mixin-Klassen ist. `members` ist ein Objekt, welches alle Funktionen enthält, die Memberfunktionen der Klasse werden sollen. `ctor` ist die Konstruktorfunktion. Bei der Verwendung von Mixins werden die Kon-

struktoren dieser Klassen nicht ausgeführt; diese müssen gegebenenfalls mit `this.NameDesMixins(...)` manuell ausgeführt werden. Der Aufruf von Methoden eines Mixins, die durch eine Memberfunktionen überschrieben wurden, ist nur über den Prototypen des Mixins möglich:
`this.<mixin>.prototype.<func>.call(this, ...)`

Hilfsfunktionen

Mit Funktionen wie `each`, `map` und `filter` bietet das Indico-Framework einige Standardfunktionen zur funktionalen Programmierung. Darüber hinaus enthält es viele Klassen bzw. Mixins, um *Observer*, *Getter* und *Setter* zu ermöglichen, wobei letztere jeweils durch einen *Observer* überwacht werden können. Es werden außerdem diverse Stringfunktionen zur Verfügung gestellt, wobei einige davon als globale Funktionen implementiert sind und andere dem `String`-Prototypen hinzugefügt wurden. Dies ist ein Zeichen dafür, dass mehrere Entwickler am Framework gearbeitet haben, ohne sich an einheitliche Standards zu halten.

UI-Elemente

Indico enthält einige UI-Widgets, die teilweise im Framework enthalten sind oder darauf basieren und außerhalb des Frameworks erweitert werden. Da diese Erweiterungen größtenteils allgemein gehalten sind, kann man sie zum Framework dazuzählen. Das Dialog-Widget ermöglicht Inline-Popups mit Titel, Inhalt und Buttons; zur Verwendung erstellt man eine Klasse, die von der Dialogklasse abgeleitet ist und eine `draw()`-Methode implementiert, welche sowohl den Inhalt als auch die Buttons generiert. Letztere werden ebenfalls durch ein Widget erstellt, welches jedoch nur den Button erstellt und einen *click*-Handler hinzufügt. Die Positionierung innerhalb des Dialogs findet jeweils innerhalb der *draw*-Methode statt, wodurch viel *duplicate code* entsteht. Neben dem Button-Widget werden auch die meisten anderen Formularelement von HTML durch Widgets repräsentiert. Ebenfalls enthalten sind Funktionen zur Erzeugung und Validierung von HTML-Formularen. Diese sind allerdings nicht komplett fehlerfrei: so wird bei einer Gruppe von Radiobuttons im Fehlerfall nur der erste Button als fehlerhaft markiert. Darüber hinaus enthält das Indico-Framework ein Kalenderwidget, welches jedoch nur ein Wrapper für einen *Third-Party*-JavaScript-Kalender ist.

Abbildung 5: Das Kalenderwidget von Indico

Kompatibilität

Da es sich um das aktuelle Framework von Indico handelt, gibt es logischerweise keine Kompatibilitätsprobleme mit Indico. Allerdings ist das Risiko von Konflikten mit anderen Frameworks relativ groß, da es sehr viele globale Funktionen besitzt und teilweise kein `var` verwendet wird, wodurch Variablen, die eigentlich lokal sein sollten, global werden. Auf der anderen Seite nutzt Indico keine einbuchstabigen Funktionsnamen, wodurch das Konfliktrisiko mit einigen Frameworks wieder verringert wird. Details dazu finden sich in den folgenden Abschnitten bei den jeweiligen Frameworks.

Sonstige Features

Das Indico-Framework enthält *Data Binding*-Funktionen, die es ermöglichen, beispielsweise den Inhalt eines Formularfeldes mit dem eines Objekts, welches *Getter* und *Setter* implementiert, automatisch zu synchronisieren. Während AJAX und die Unterstützung verschiedener Datentypen wie JSON und XML ein Standardfeature von JavaScript-Frameworks ist, unterstützt das Indico-Framework ausschließlich JSON-RPC. Da das Framework auf Indico zugeschnitten ist und dort nur JSON-RPC benutzt wird, ist dies jedoch kein Problem. Ein weiteres Feature ist die Erzeugung von SVG-Grafiken, jedoch wird dieses nicht mehr benutzt.

Dokumentation

Das JavaScript-Framework von Indico besitzt keine externe Dokumentation. Einige Funktionen sind jedoch im JavaDoc-Stil kommentiert, wobei die Funktionsbeschreibungen dort nicht sehr ausführlich sind. Viele Funktionen - ins-

besondere nachträglich hinzugefügte - sind jedoch nicht kommentiert, obwohl man ihren Zweck nicht auf den ersten Blick erkennt. Abgesehen vom Indico-Code, der die meisten im Framework vorhandenen Funktionen nutzt, existiert kein Beispielcode.

Lizenz

Da das Framework speziell für Indico entwickelt wurde und auch nicht *standalone* verfügbar ist, steht es genau wie Indico selbst unter der GPL.

3.3. Prototype

Bei Prototype handelt es sich um eine kompakte JavaScript-Klassenbibliothek; das Release besteht nur aus einer ca. 35 KB großen JavaScript-Datei. Es wurde ursprünglich für das *Ruby on Rails*-Framework[24] entwickelt, ist aber auch als *standalone*-Bibliothek weit verbreitet. Sie kann von http://www.prototypejs.org heruntergeladen werden.

Prototype zielt darauf ab, Webentwicklung zu vereinfachen und insbesondere die Benutzung von AJAX sehr einfach zu machen. Durch das ausgereifte und komfortable Objektmodell fördert es die im Web-Bereich eher wenig genutzte objektorientierte Programmierung.

In Indico wird Prototype zusammen mit *Scriptaculous* bereits an einigen Stellen benutzt, um Drag&Drop zu ermöglichen.

DOM-Zugriff

Prototype bietet mehrere Methoden, um auf DOM-Elemente zuzugreifen. Die Funktion `$(id...)` gibt das DOM-Element mit der angegebenen ID zurück. Falls mehrere IDs angegeben wurden, gibt die Funktion ein Array zurück. Statt einer ID kann auch ein DOM-Element übergeben werden, womit die Funktion sehr flexibel ist. Bei allen gefundenen Elementen wird sichergestellt, dass diese mit den Prototype-spezifischen Erweiterungen versehen wurden: Prototype nutzt keinen Wrapper sondern modifiziert den Prototyp des `HTMLElement`-Objekts des Browsers. Da der Internet Explorer diese Modifikation jedoch nicht zulässt, fügt die `$`-Funktion diese Erweiterungen ggf. dem jeweiligen Objekt hinzu. Das Fehlen eines Wrappers hat den Nachteil, dass Funktionsnamen nicht mit denen nativer Funktionen kollidieren sollten und prinzipiell zwischen

[24] http://rubyonrails.org/

einem einzelnen Element und mehreren Elementen unterschieden werden muss, da bei letzterem ein Array notwendig ist, während bei ersterem ein Array dem Komfort abträglich wäre; der Entwickler müsste dann jeweils `elem[0]` nutzen, um auf das Element zuzugreifen. Diese Unterscheidung schadet oftmals der Lesbarkeit des Codes, da keine Prototype-Methode direkt auf alle Elemente einer Liste angewendet werden kann; man muss sie entweder selbst durchlaufen oder die `invoke()`-Methode des Arrays nutzen, um die Funktion auf jedes Element anzuwenden. Da der Funktionsname jedoch als String an diese Methode übergeben werden muss, macht dies den Code unübersichtlicher und schlechter durchsuchbar, da man nun nicht mehr einfach nach `func(` suchen kann, wenn man die Aufrufe von *func* finden möchte.

Sofern eine Suche anhand der ID nicht ausreichend ist, kann man auch die Funktion `$$(selector...)` nutzen. Diese sucht Elemente anhand eines CSS-Selektors und gibt grundsätzlichen ein Array zurück, da abgesehen von ID-Selektoren der Form `#id` beliebig viele Elemente gefunden werden können. Zum selektorbasierten Suchen benutzt Prototype die *Sizzle*-Middleware[25].

DOM-Manipulation

Das `Element`-Objekt enthält nicht nur alle Methoden, die den `HTMLElement`-Objekten hinzugefügt werden, sondern dient auch der Erstellung neuer Elemente. Um mit Prototype ein neues HTML-Element zu erstellen, nutzt man den `Element(tagName[, attributes])`-Konstruktor. Dabei werden der Name des Elements als String und die optionalen Attribute als Objekt übergeben. Da letztere unverändert als Attribute des neuen Elements übernommen werden, dürfen die Werte ausschließlich Strings und Zahlen sein. Wenn das neue Element nicht leer sein soll, kann man die `update()`-Methode nutzen, um ihm einen Inhalt zuzuweisen. Dieser kann entweder ein HTML-String, ein HTML-Element oder ein Objekt sein, welches entweder eine `toHTML()`-Methode oder `toElement()`-Methode besitzt.

Da Prototype keinen Wrapper nutzt, kann auf Element-Eigenschaften direkt zugegriffen werden. Wenn explizit der Zugriff auf ein Attribut gewünscht ist, kann man die Methoden `readAttribute()` bzw. `writeAttribute()` nutzen. Für den Zugriff auf diverse Eigenschaften wie der CSS-Klasse (hinzufügen, entfernen, *contains*), der Sichtbarkeit und dem Wert und *Disabled*-Flag von Formularelementen stellt Prototype ebenfalls Methoden zur Verfügung.

[25] http://sizzlejs.com

Darüber hinaus können auch CSS-Eigenschaften über ein Objekt gesetzt werden, sodass mehrere Eigenschaften auf einmal geändert werden können. Ein insbesondere für grafiklastige oder pixelgenau arbeitende Webapplikationen nützliches Feature ist die `measure()`-Methode. Diese gibt ein Objekt zurück, welches Zugriff auf alle Größen- und Positionsinformationen bietet.

Auch für die Manipulation des DOM-Trees bietet Prototype diverse Funktionen. Neben den Standardfunktionen bietet Prototype dort auch Komfortfunktionen wie `wrap()`, womit ein Element durch ein neues Element umschlossen wird und das neue Element zurückgegeben wird.

DOM Traversal

Prototype bietet mit der `select(selector...)`-Methode einen komfortablen Weg, mittels eines CSS-Selektors ein unterhalb von einem anderen Element liegendes Element zu finden. Zur einfachen Navigation innerhalb der DOM-Struktur stellt Prototype ebenfalls Methoden zur Verfügung, wobei teilweise mehrere Methoden für fast denselben Zweck existieren. Die Methoden `adjacent()` und `siblings()` geben die Geschwisterelemente eines Elements zurück, der einzige Unterschied ist, dass `adjacent()` ein Filtern der Elemente durch einen CSS-Selektor erlaubt.

Events

Prototype bietet zwei Möglichkeiten, Events zu registrieren. Die klassische Möglichkeit ist über die `observe(eventName, handler)`-Methode, wobei `handler` für alle Events ausgeführt werden, die das Element erreichen, also Events die vom Element selbst oder einem enthaltenen Elemente ausgehen. Da die Methode das Element selbst zurückgibt, kann der Eventhandler nur entfernt werden, indem man ihn anhand des Eventnamens und der Handlerfunktion identifiziert. Dazu dient die Methode `stopObserving([eventName][, handler])`, wobei beide Argumente optional sind. Ist ein Argument nicht angegeben, werden alle entsprechenden Eventhandler entfernt.

Der neue und auch komfortablere Weg ist über die Methode `on(eventName[, selector], handler)`. Sofern ein CSS-Selektor angegeben wurde, wird aus dem Eventhandler ein *Delegate*, d.h. er reagiert nur noch auf Events, die von einem enthaltenen Element ausgelöst wurden, welches dem CSS-Selektor entspricht. Ein weiterer Vorteil dieser Methode ist ihr Rückgabewert. Sie gibt ein Objekt zurück, welches zwei Methodem `start()` und `stop()` enthält, wobei letztere den Eventhandler entfernt und erstere ihn wieder hinzufügt.

In beiden Fällen stellt Prototype sicher, dass `this` auf das ursprüngliche Element zeigt, wobei sich bei einem *Delegate* darüber streiten lässt, ob `this` nicht eher auf das Element zeigen sollte, von dem das Event ausgeht. Als Parameter erhält die Callbackmethode jeweils das durch Prototype erweiterte Event-Objekt: Das Zielelement ist browserunabhängig über die `target`-Eigenschaft verfügbar und die `stop()`-Methode verhindert sowohl das Aufsteigen des Events im DOM-Tree als auch das Ausführen der Standardaktion (z.B. das Laden einer neuen Seite bei einem Link). Eine browserunabhängige Möglichkeit, nur eine dieser Aktionen zu verhindern, fehlt jedoch. Sofern `on()` zum Registrieren des Eventhandlers genutzt wurde, wird das Zielelement in einem zusätzlichen Parameter an da Callback übergeben.

Objektsystem

Das Objektsystem von Prototype bietet Klassen, Vererbung und Mixins, wobei eine Klasse beliebig viele Mixins aber nur eine Parent-Klasse besitzen kann. Eine neue Klasse erstellt man, indem man den Rückgabewert der Funktion `Class.create([superclass][, methods...])` einer Variable zuweist. Es handelt sich dabei um die Konstruktorfunktion der neuen Klasse. Sofern man von einer anderen Klasse erben möchte, übergibt man diese Klasse als erstes Argument. Alle weiteren Argumente sind JavaScript-Objekte, die Funktionen enthalten. Diese werden der neuen Klasse hinzugefügt, wobei bereits vorhandene Methoden überschrieben werden. Benutzt man Mixins, sollte man diese also vor dem Objekt übergeben, welches die neuen Funktionen der Klassen enthält. Beim Überschreiben von Methoden der Parent-Klasse nutzt Prototype den dynamischen Character von JavaScript voll aus: Sofern das erste Argument der neuen Methode den Namen `$super` hat, zeigt dieses auf die überschriebene Methode, sodass sie einfach und ohne zusätzlichen Code aufgerufen werden kann.

Hilfsfunktionen

Prototype stellt keine globalen Hilfsfunktionen zur Verfügung. Es erweitert die Prototypen und Konstruktoren bestehender Objekte, sodass über diese Objekte auf die zahlreichen Hilfsfunktionen von Prototype zugegriffen werden kann.

UI-Elemente

Prototype enthält keine UI-Widgets. Es wird allerdings oftmals zusammen mit Scriptaculous[26] verwendet wird. Diese Erweiterung erlaubt sortierbare Listen,

[26] http://script.aculo.us

Autovervollständigung von Textfeldern, Inline-Editierung von Texten/Werten, Drag&Drop von Elementen und ein *Slider*-Widget zur Auswahl eines numerischen Werts.

Kompatibilität

Prototype wird bereits in einigen Teilen von Indico genutzt, daher ist es an sich mit dem übrigen Code von Indico kompatibel, obwohl sowohl Indico als auch Prototype einige `$X`-Funktionen nutzen, wobei `$A` sogar in beiden Frameworks definiert wird, allerdings überschreibt die `$A`-Funktion von Indico die aus Prototype, wobei es keine Konflikte gibt, da beide Funktionen denselben Zweck haben und auch von den Parametern her identisch sind. Neben diesem Konflikt gibt es keine weiteren Probleme, obwohl Prototype sehr invasiv ist: Wie auch das Indico-Framework, definiert es mehrere globale Objekte mit „Standardnamen“ wie *Event* oder *Element*. Darüber hinaus erweitert es die Prototypen von nativen Objekten wie `Array`, `Number` und `String`. Indem der Prototyp von `Object` nicht erweitert wird, beugt Prototype den damit verbundenen Problemen vor. Eine solche Erweiterung würde z.B. das Verhalten von `for(var elem in object)` verändern. Neben nativen Objekten erweitert Prototype auch Host-Objekte wie `HTMLElement`. Dies ist jedoch aufgrund des größtenteils undefinierten Verhaltens dieser Objekte riskant.

Sonstige Features

Die meisten Methoden von Prototype kann man sowohl als Methode als auch als Funktion aufrufen, wobei bei letzterem das Objekt als Funktionsargument übergeben wird. Prototype enthält selbst keine Funktionen zur Animation von Elementen, jedoch bietet Scriptaculous entsprechende Funktionalität.

Dokumentation

Prototype besitzt eine ausführliche Online-Dokumentation. Zu jeder Funktion gibt es neben der Beschreibung auch ein kurzes Codebeispiel, allerdings sind diese Beispiele statisch und können nicht ausgeführt werden, ohne den Code manuell in eine HTML-Datei zu kopieren. Der Quellcode von Prototype ist ausführlich kommentiert und in Module unterteilt, die jedoch im Release in eine einzelne Datei zusammengefasst sind.

Lizenz

Prototype steht unter der MIT-Lizenz[27].

[27] http://www.opensource.org/licenses/mit-license.php

Ähnlich wie das Indico-Framework bietet Prototype aus fast jedem Bereich etwas, wobei JSON-RPC und *Data Binding* nicht unterstützt werden. Die Erstellung und auch der Zugriff auf DOM-Elemente ist in Prototype deutlich sauberer gelöst als in Indico, da nur selten auf native DOM-Eigenschaften und -Methoden zurückgegriffen werden muss und damit einheitlich Prototype-Code verwendet werden kann statt einer Mischung aus Framework-Funktionen und nativen Funktionen. Die Eventsysteme von Indico und Prototype sind relativ ähnlich, allerdings bietet Prototype insbesondere durch das auf das Element zeigende `this` mehr Komfort. Anders als in Indico können bei Prototype keine Klassen als Mixins genutzt werden, sondern nur Objekte, daher ist es nur schwer möglich, Mehrfachvererbung zu erzeugen. Man könnte zwar jedem Mixin eine Funktion zuweisen die man als Konstruktor nutzt, allerdings würden sich diese Funktionen der einzelnen Mixins überschreiben, sodass neben der Parent-Klasse jeweils nur ein solches Mixin genutzt werden könnte. Über die *Enumerable*- und *Function*-Funktionen bietet Prototype die auch in Indico vorhandenen Funktionen zur funktionalen Programmierung, wobei sie bei Prototype als Instanzmethoden realisiert sind statt mit dem jeweiligen Objekt als Parameter aufgerufen zu werden. Bei UI-Elementen kann Prototype nicht mit Indico mithalten, da selbst mit Scriptaculous wichtige Elemente wie Dialogfenster nicht unterstützt werden. Die Dokumentation von Prototype unterscheidet sich positiv von der nur minimalen Dokumentation des Indico-Frameworks. Die MIT-Lizenz, die Prototype nutzt, ist mit der GPL kompatibel, sodass es lizenztechnisch keine Konflikte gibt.

3.4. jQuery

jQuery ist eine leichtgewichtiges JavaScript-Bibliothek, die auf komfortablen DOM-Zugriff und AJAX spezialisiert ist. Dabei wird insbesondere das *Chaining* von Methoden genutzt, d.h. die meisten Methoden geben das aktuelle Objekt zurück, sodass Aufrufketten wie `obj.m1().m2()` möglich sind. Auf der Website http://jquery.com wird jQuery auch als „designed to change the way that you write JavaScript“ bezeichnet. Mit einer einzigen, ca. 30 KB (minimiert und *gzip*-komprimiert) großen Datei ist jQuery sehr kompakt.

DOM-Zugriff

In jQuery findet der Zugriff auf das DOM über die `$(...)`-Funktion statt. Sofern man als Parameter einen CSS-Selektor übergibt, werden alle durch diesen Selektor ausgewählten Element in einem neuen jQuery-Objekt verpackt und dieses wird zurückgegeben. Optional kann als zusätzlicher Parameter der

Kontext in Form eines DOM-Elements, DOM-Dokuments oder jQuery-Objekts angegeben werden, um dieses Element als Basis der Suche zu verwenden. Statt eines Selektors kann man auch ein DOM-Element oder ein Array solcher Elemente übergeben, um aus selbigen ein neues jQuery-Objekt zu erzeugen. Wenn man `$()` ohne Parameter aufruft, wird ein leeres jQuery-Objekt erstellt.

Um an in einem jQuery-Objekt enhaltene DOM-Objekte zu kommen, kann man die Arraysyntax `obj[index]` nutzen, um das das jeweilige Element zu erhalten. Alternativ kann man auch die `get([index])`-Methode nutzen. Sofern ein Index angegeben wird, gibt sie das jeweilige Element zurück, ansonsten ein Array aller im jQuery-Objekt enthaltenen Elemente.

jQuery nutzt *Sizzle* zur Suche von Elementen anhand eines CSS-Selektors.

DOM-Manipulation

Zum Erstellen neuer Elemente nutzt man in jQuery ebenfalls die `$`-Funktion, indem man den gewünschten HTML-Code als Parameter übergibt. Dabei kann es sich sowohl um ein einzelnes Tag als auch eine beliebig komplexe HTML-Struktur handeln. Als zusätzlichen Parameter kann man entweder ein DOM-Dokument angeben, in dem die Elemente erstellt werden sollen, oder aber ein Objekt, welches die Attribute, die dem neuen Element zugewiesen werden sollen, als Strings enthält.

Bei der Verwendung von jQuery ist es eigentlich niemals notwendig, direkt auf DOM-Elemente zuzugreifen, da sowohl der Zugriff auf die Attribute eines Elements mittels `attr(name[, value])` als auch der Zugriff auf seine Eigenschaften mittels `prop(name[, value])` möglich ist. Elementspezifische CSS-Eigenschaften lassen sich durch die `css()`-Methode hinzufügen bzw. ändern. Diese akzeptiert entweder Eigenschaft und Wert in separaten Parametern oder aber ein Objekt, welches mehrere Eigenschaften und Werte enthält. Dabei kann der Name sowohl in der CSS-Schreibweise *words-with-dashes* (z.B. `background-color`) als auch in der DOM-Schreibweise *lowerCamelCase* (z.b. `backgroundColor` angegeben werden. Bei numerischen Eigenschaften kann anstelle eines absoluten Werts auch ein Offset in der Form `+=number` angegeben werden, um *number* zum aktuellen Wert zu addieren.

Für häufig genutzte Manipulationen, wie dem Ein- und Ausblenden von Elementen, stellt jQuery komfortabel nutzbare Funktionen zur Verfügung, sodass man dabei Entwicklungszeit einspart. Die Funktionen `hide()` und `show()` akzeptieren einen optionalen *duration*-Parameter, der aus dem sofortigen Aus-

und Einblenden eine Animation macht. Alternativ können Elemente auch mit den *slide*-Funktionen ein- und ausgelbendet werden, wobei sie dabei am oberen Rand der Seite hinaus- bzw. hineinbewegt werden.

Zur Veränderung der Struktur des DOM-Trees - also dem Einfügen, Entfernen und Umsortieren von Elementen - bietet jQuery viele Funktionen, die eigentlich alle sinnvollen Möglichkeiten abdecken. Neben der inzwischen relativ verbreiteten `wrap()`-Funktion enthält jQuery mit `unwrap()` auch das genaue Gegenteil dieser Funktion, d.h. das Entfernen des Parent-Elements eines Elements und dessen Einfügen an dieser Stelle. Das Einfügen von Elementen kann in beide Richtungen geschehen: zusätzlich zu Methoden wie `append(elem)`, die das übergebene Element in das im jQuery-Objekt enthaltene Element einfügen, existieren mit Methoden wie `appendTo(elem)` Methoden, die das im jQuery-Objekt enthaltene Element in das übergebene Element einfügen. Der komplette Inhalt von Elementen lässt sich durch `empty()` löschen; beim Ersetzen durch neuen Inhalt gibt es mit `html()` und `text()` sowohl die Möglichkeit, HTML einzufügen, als auch einen String als Plaintext einzufügen, sodass möglicherweise enthaltener HTML-Code nicht ausgeführt wird.

Wenn ein jQuery-Objekt mehrere Element enthält, hängt das Verhalten von der aufgerufenen Funktion ab. Sofern es möglich und sinnvoll ist, wirkt sich die Funktion auf jedes Element aus. Bei anderen Operationen wie beispielsweise dem lesenden Zugriff mittels `attr()` wird ausschließlich auf das erste Element zugegriffen.

DOM Traversal

Wie bereits zuvor erwähnt, kann der `$`-Funktion ein Kontext übergeben werden, um ausgehend von einem Element anhand eines Selektors andere Elemente zu suchen. Neben dieser Möglichkeit bietet jQuery diverse Methoden um Elemente ausgehend von einem Element auszuwählen. Alle diese Methoden akzeptieren einen CSS-Selektor zum Filtern der gefundenen Elemente. Neben den Geschwisterelementen, wobei diese auf Elemente vor bzw. nach dem aktuellen Element eingeschränkt werden können, können mit diesen Methoden auch die Parent-Elemente selektiert werden, wobei dort je nach Funktion das direkte Parent-Element, alle Parent-Elemente oder alle Parent-Elemente bis zu einem bestimmten Element zurückgegeben werden. Mit `closest(selector)` bietet jQuery eine Komfortmethode, um sich ausgehend vom aktuellen Element im DOM-Tree nach oben durchzuarbeiten, bis das gewünschte Element gefunden wurde. Bei diesem kann es sich auch um das aktuelle Element selbst

handeln.

Der Inhalt von jQuery-Objekten lässt sich darüber hinaus modifizieren: man kann die Elementliste anhand eines CSS-Selektors filtern, bestimmte Elemente entfernen, die vor einer Suche enthaltenen Elemente der Elementliste wieder hinzufügen und einzelne Elemente hinzufügen. Auch das Reduzieren eines jQuery-Objekts auf ein einzelnes Element ist möglich. Dabei kann es sich z.B. um das Element mit einem bestimmten Index oder um das erste bzw. letzte Element handeln.

Eine sehr spezielle, aber für die absolute Positionierung von Elementen sehr nützliche Methode ist `offsetParent()`. Diese sucht das erste übergeordnete Element, welches positioniert ist und damit das Referenzelement für die absolute Positionierung darstellt.

Events

jQuery bietet verschiedene Möglichkeiten, Eventhandler zu registrieren. Der einfachste und komfortabelste Weg ist, die dem Eventnamen entsprechende Methode eines jQuery-Objekts aufzurufen und dieser die Handlerfunktion als Parameter zu übergeben. Eine weitere Möglichkeit ist über die `bind()`-Methode; dabei übergibt man Eventname und -handler als separate Parameter oder in einem Objekt, welches mehrere Eventhandler enthalten kann. Um dieselbe Funktion für mehrere Events zu benutzen können die jeweiligen Eventnamen mit Leerzeichen getrennt angegeben werden. Der große Vorteil bei der Benutzung von `bind()` ist jedoch, dass man die Events in Namespaces unterteilen kann indem man `'eventname.something'` als Eventname verwendet. Dies hat den Vorteil dass man diese Events separat auslösen oder entfernen kann, ohne andere Events dasselben Typs auszulösen oder zu entfernen. Darüber hinaus kann man mit dieser Funktion beliebige Events erstellen, d.h. nicht nur die standardmäßig unterstützen DOM-Events. Neben den regulären DOM-Events unterstützt jQuery die Internet Explorer-spezifischen Events *mouseenter* und *mouseleave*, welche emuliert werden, wenn sie nicht vom Browser unterstützt werden.

Mit `delegate(selector, eventType, handler)` erlaubt jQuery die Delegation von Events an ein übergeordnetes Element. Dies hat den Vorteil, dass nicht alle von `selector` ausgewählte Elemente bereits zum Zeitpunkt der Eventhandler-Registrierung existieren müssen und z.B. bei einer Liste mit vielen Elementen nur ein einziger Eventhandler registriert werden muss, was

die Performance erhöht. Sofern man kein Basis-Element kennt, sondern Events von Elementen an einer *beliebigen* Position im Dokument verarbeiten möchte, kann man die `live`-Methode nutzen; ihre Parameter sind mit denen von `bind()` identisch. Intern erstellt diese Methode einfach ein *Delegate* mit dem Dokument selbst als Basis-Element. Der Nachteil dieser Art der Eventverarbeitung ist jedoch, dass das Element bis zum Basis-Element aufsteigt und erst dort ein weiteres Aufsteigen verhindert werden kann.

Zum Löschen von Eventhandlern bietet jQuery mit `unbind()`, `die()` und `undelegate()` Funktionen für die verschiedenen Eventhandlertypen. Dabei entsprechen die Parameter der Funktionen ihren Gegenstücken, wobei die Handlerfunktion jeweils optional ist, sodass man auch alle Events eines Typs löschen kann. Zusätzlich zu diesen Parametern kann man bei `unbind()` und `undelegate()` auch `'.namespace'` angeben, um alle Events aus dem angegebenen Namespace zu löschen.

Um Events manuell auszulösen bietet jQuery, genau wie bei der regulären Registrierung von Eventhandlern, zwei Möglichkeiten. Zum einen kann man die Shortcut-Methode die dem Eventnamen entspricht ohne Parameter aufrufen, zum anderen kann man die Methode `trigger(eventType[, params])` nutzen. Diese erlaubt neben der Möglichkeit, einen Namespace anzugeben, die Übergabe weiterer Parameter an den Eventhandler. Durch `trigger()` ausgelöste Events verhalten sich wie durch Benutzerinteraktion ausgelöste. Sie steigen im DOM-Tree auf und lösen die Standardaktion des Browsers aus (z.B. Laden einer neuen Seite). Wenn dieses Verhalten nicht erwünscht ist, bietet die Funktion `triggerHandler()` die Möglichkeit, nur per jQuery registrierte Eventhandler auszulösen und den Rückgabewert des letzten Handlers dieses Events zu verarbeiten.

An die Eventhandler-Funktionen übergibt jQuery jeweils ein Event-Objekt und im Falle einer manuellen Auslösung des Events alle zusätzlich angegebenen Parameter. Beim Event-Objekt handelt es sich um ein von jQuery erstelltes Objekt, in welches die Eigenschaften des ursprünglichen Event-Objekts kopiert wurden. Dabei stellt jQuery sicher, dass browserspezifische Eigenschaften wie der gedrückten Taste bei einem Tastatur-Event oder der Mausposition bei einem Maus-Event normalisiert werden und immer über dieselben Eigenschaften verfügbar sind. Darüber hinaus enthält das Event-Objekt browserunabhändige Methoden, um die weitere Verarbeitung des Events zu beeinflussen und beispielsweise das Ausführen der Standardaktion zu verhindern.

```
1 $('a.do-not-click').click(function(e) {
2     e.preventDefault();
3 });
```

Listing 15: Blockieren des Click-Events einiger Links via jQuery

Objektsystem

jQuery enthält kein Objektsystem.

Hilfsfunktionen

jQuery bietet eine Vielzahl an Funktionen, die jeweils über `$.funcName` verfügbar sind. Neben den üblichen funktionalen Funktionen wie `map()` und `each` bietet jQuery auch Funktionen, um eine Variable auf einen bestimmten Typ zu testen und mit `trim()` auch eine Funktion, um Whitespace am Anfang oder Ende eines Strings zu entfernen. Auch ein JSON-Parser ist enthalten, der jedoch nur genutzt wird, wenn der verwendete Browser JSON nicht nativ unterstützt.

Mit `$.browser` besitzt jQuery ein Objekt, welches Informationen zum verwendeten Browser und dessen Version enthält. Dieses ist jedoch als *deprecated* markiert, da es im Falle von browserspezifischen Problemen und Features sinnvoller ist, auf das Vorhandensein ebendieser zu prüfen. Dazu besitzt jQuery das `$.support`-Objekt. Es enthält Informationen zu allen Browserfeatures, die jQuery selbst benötigt. Die Tests, ob die jeweiligen Features vorhanden sind müssen beim Laden jeder Seite ausgeführt werden, weshalb die Anzahl möglichst gering gehalten wird.

UI-Elemente

jQuery selbst enthält keinerlei Widgets, allerdings existiert mit *jQuery UI*[28] eine Erweiterung, die diverse Widgets und UI-bezogene Features wie Drag&Drop und vergrößerbare, verkleinerbare, auswählbare und sortierbare Elemente enthält. Die Widgets bieten Dialogfenster, Buttons, Kalender, Fortschrittsbalken, Tabs und Slider. Neben diesen Standardwidgets gibt es noch ein *Accordion*-Widget, das ein „Akkordion“-Menü bzw. -Panel ermöglicht. Ebenfalls enthalten ist ein *Autocomplete*-Widget, um HTML-Textfelder mit Autovervollständigung zu versehen.

[28] http://jqueryui.com

Abbildung 6: Der jQuery UI Datepicker

Erwähnenswert ist die einfache Anpassbarkeit der jQuery UI-Widgets - mit dem *ThemeRoller*[29] gibt es eine Webapplikation, über die man mit wenigen Klicks ein neues Farbschema für alle Widgets zusammenstellen kann und auch direkt sieht, wie die Widgets mit dem neuen Theme aussehen. Neben den Farben lassen sich auch viele weitere Aspekte per CSS anpassen - entweder global für alle Widgets oder über die widgetspezifischen CSS-Klassen `.ui-<widgetname>` nur für einzelne Widgets.

Codeseitig erweitert jedes Widget das jQuery-Objekt um eine Methode, über die das Widget sowohl erstellt als auch verändert oder wieder entfernt werden kann. Wenn die Methode ohne Parameter aufgerufen wird, erstellt sie das Widget mit Standardeinstellungen; alternativ kann auch ein Objekt als Parameter übergeben werden, um eigene Einstellungen zu benutzen. Für alle anderen Operationen übergibt man dieser Methode als ersten Parameter den Funktionsnamen und in den weiteren Parametern die funktionsspezifischen Parameter, beispielsweise `dialog('open')` um einen zuvor erstellten Dialog zu öffnen.

Kompatibilität

jQuery exportiert nur zwei globale Funktionen: `jQuery` und `$`, wobei es sich bei letzterer um einen Alias für `jQuery` handelt. Falls `$` bereits anderweitig verwendet wird, kann man die `jQuery.noConflict()`-Funktion nutzen, um das ursprüngliche `$` wiederherzustellen. Um trotzdem nicht immer `jQuery` statt `$` schreiben zu müssen, kann man seinen Code einfach in eine Funktion wrappen, die `$` als Parameter zur Verfügung stellt.

[29] http://jqueryui.com/themeroller/

```
1 jQuery.noConflict();
2 (function($) {
3     // $ === jQuery
4 })(jQuery);
```

Listing 16: Verwendung von $.noConflict

Sonstige Features

Wie bereits in der einführenden Beschreibung erwähnt macht jQuery extensiven Gebrauch vom *Method Chaining*. Dies erspart dem Programmierer Arbeit, da er viele Methoden auf ein Objekt anwenden kann ohne den Objektnamen für jeden Aufruf neu schreiben zu müssen. Ebenfalls erhöht es die Performance, da nur einmal ein *variable lookup* für die Objektvariable notwendig ist. Bei Methoden, die sowohl Getter als auch Setter sein können, ist allerdings nur die Setter-Variante *chainable*, da der Getter einen Wert zurückgeben muss, der nicht dem ursprünglichen Objekt entspricht. Ein weiteres Feature in diesen Bereich ist die interne Verkettung von jQuery-Objekten. Wenn ein jQuery-Objekt, welches Element enthält, gefiltert wird oder Elemente hinzugefügt werden, wird intern ein neues Objekt erstellt, sodass die alte Elementliste erhalten bleibt. Mit der `end()`-Methode kann man in der Aufrufkette zum vorherigen Objekt zurückspringen. Allerdings ist dabei zu beachten, dass der Code lesbar und verständlich bleibt. Es kann sinnvoll sein, eine neue Aufrufkette zu beginnen statt stur zu versuchen, möglichst viele Aktionen innerhalb derselben Kette auszuführen.

Die meisten Funktionen in jQuery, die Werte verändern, akzeptieren statt eines Werts auch eine Funktion. Diese wird durch jQuery mit dem Index des Elements innerhalb des jQuery-Objekts und dem aktuellen Wert als Parameter aufgerufen und der Rückgabewert wird als Wert benutzt, sofern er nicht `undefined` ist. Innerhalb dieser Funktion zeigt **`this`** auf das jeweilige DOM-Element.

Ein anderes Feature von jQuery ist das `Deferred`-Objekt. Es ermöglicht die Registrierung von Callbacks, wobei diese in drei verschiedenen Kategorien registriert werden können: *done*, *fail* und *always*. Das Deferred-Objekt kann entweder *fail* oder *done* zu einer beliebigen Zeit auslösen, es gilt dann als *resolved*. Nach dem Ausführen der Callbacks werden automatisch die *always*-Callbacks aufgerufen. Sobald das Objekt *resolved* ist kann sich sein Status nicht mehr

ändern, d.h. es ist ausgeschlossen, dass sowohl die *fail*-Callbacks als auch die *done*-Callbacks ausgeführt werden. Der große Vorteil des Deferred-Objekts ist, dass Callbacks auch registriert werden können, wenn das Objekt bereits *resolved* ist. Die entsprechende Callbackfunktion wird in diesem Fall sofort ausgeführt. Da externer Code oftmals nur die Möglichkeit haben soll, Callbacks zu registrieren, diese aber nicht selbst auslösen können soll, bietet das Deferred-Objekt mit der `promise()`-Methode eine Möglichkeit, nur die entsprechenden Methoden zurückzugeben, sodass das Deferred-Objekt selbst niemals den Sichtbarkeitsbereich der erstellenden Funktion verlassen muss. Diese Funktionalität wird beispielsweise von den AJAX-Funktionen von jQuery genutzt; sie geben ein solches *Promise*-Objekt zurück.

jQuery kann einfach mit eigenen Funktionen erweitert werden. Dazu ist der Prototype der jQuery-Objekte über `jQuery.fn` erreichbar, sodass man diesem eigene Funktionen hinzufügen kann. Um jQuery unabhängig von einem bestimmten jQuery-Objekt zu erweitern - beispielsweise um eine Funktion hinzuzufügen, die keine HTML-Elemente verwendet - kann man eine Funktion auch direkt dem globalen `jQuery`-Objekt hinzufügen.

Dokumentation

jQuery besitzt eine ausführliche Online-Dokumentation, die nach Aufgabenbereichen wie *Events*, *Traversing*, *Manipulation* und *forms* gegliedert ist. Jede Funktionsbeschreibung besitzt mindestens ein Codebeispiel; wenn eine Funktion je nach Parameter unterschiedliche Aktionen ausführt, gibt es für jede Funktionalität ein eigenes Beispiel. Diese Beispiele werden meist beim Laden der Seite automatisch ausgeführt, sodass man direkt sieht, wie sich eine Funktion auswirkt. Bei einigen Funktionen wie beispielsweise Animationen und Effekten muss der Beispielcode durch einen Klick explizit ausgeführt werden. Dies macht Sinn, da bei solchen Funktionen nicht der Endzustand interessant ist sondern der Übergang. Die komplette Dokumentation kann inklusive Beispielcode auch als XML-Datei heruntergeladen werden, sodass man sie beispielsweise in einer IDE, die das Format unterstützt, importieren könnte.

Während die Produktivversion von jQuery minimiert ist, kann man auf der jQuery-Website eine lesbare Entwicklungsversion herunterladen. Dabei handelt es sich um eine ca. 230 KB große JavaScript-Datei die von der Funktionalität her mit der Produktivversion identisch ist. Sofern man lieber einzelne, nach Aufgabenbereich unterteilte, Dateien möchte, kann man sich auch den

„Quellcode“ auf der GitHub-Seite[30] von jQuery herunterladen. Unabhängig davon welche Entwicklungsversion man herunterlädt, ist der enthaltene Code kommentiert, wobei es sich bei den Kommentaren strikt um technikbezogene Hinweise handelt. Diese sind als Dokumentation nicht brauchbar, was dank der Online-Dokumentation aber auch nicht notwendig ist.

Lizenz

jQuery ist sowohl unter der MIT-Lizenz als auch der GPL verfügbar; jeder Entwickler hat die Möglichkeit die Lizenz zu nutzen, die besser zu seiner Software bzw. dessen Lizenz passt.

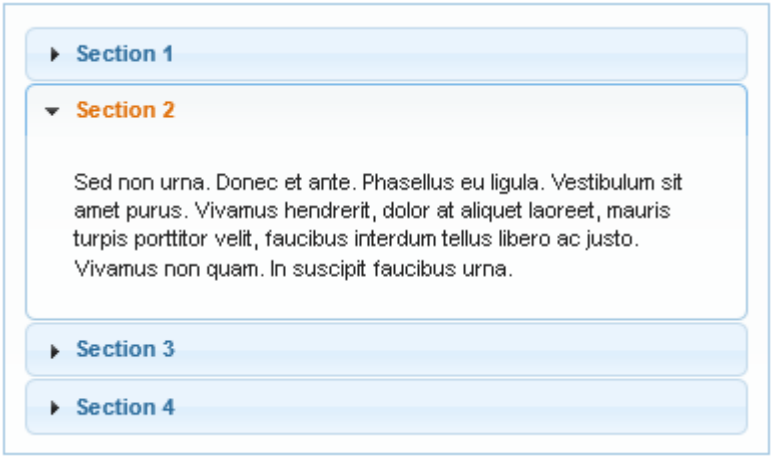

Abbildung 7: Das Akkordion-Menü von jQuery UI

jQuery ist auf DOM-Operationen und AJAX spezialisiert, wobei letzteres allgemein gehalten ist, sodass JSON-RPC selbst implementiert werden müsste. Der Zugriff auf DOM-Elemente ist in jQuery deutlich komfortabler als in Indico, da man durch die Nutzung von CSS-Selektoren flexibler ist und es oftmals nicht notwendig ist, automatisch generierte IDs zu vergeben, um ein Element später noch referenzieren zu können: In jQuery nutzt man die Methoden zum Durchsuchen des DOM-Trees um beispielsweise ausgehend vom angeklickten Element die dazugehörige Tabellenzeile zu finden. Beim Verändern von Elementen führt jQuery ebenfalls: die Möglichkeit, Objekte und Funktionen als Wert zu nutzen erhöht die Flexibilität extrem und auch die große Menge an Funktionen zum Verändern der DOM-Struktur ist beim Erzeugen dynamischer Elemente wie Formularen in Dialogfenstern und deren Validierung nützlich. Das Eventsystem von jQuery ist dank der Namespaces sehr flexibel und das vollständig normalisierte Event-Objekt vereinfacht die browserunabhängige Entwicklung. Ein Nachteil, der durch das *Method Chaining* aber unumgänglich

[30] https://github.com/jquery/jquery

ist, ist das Fehlen eines Eventhandler-Objekts, welches eine komfortable Möglichkeit bietet, das Event wieder zu löschen. Man kann eine solche Funktion aber leicht selbst erstellen, indem man beim Registrieren des Events Eventname und Funktion in einer Closure speichert und damit den entsprechenden Eventhandler löscht. Ein Objektsystem ist in jQuery nicht vorhanden, daher müsste dort auf das bestehende System oder eine separate Bibliothek zurückgegriffen werden. Genau wie das Indico-Framework unterstützt auch jQuery Standardmethoden zur funktionalen Programmierung, wobei nicht alle in Indico verwendeten Funktionen auch in jQuery vorhanden sind. Sowohl Indico als auch jQuery enthalten ein UI-Framework, wobei sie sich in einigen grundlegenden Aspekten unterscheiden: Die Widgets von Indico sind darauf ausgelegt, sowohl Struktur als auch Inhalt selbst zu generieren, während jQuery UI dazu gedacht ist, bestehenden HTML-Code zu erweitern. Dies hat den Vorteil, dass auch ohne JavaScript eine gewisse Grundfunktionalität vorhanden ist, was in Indico derzeit nicht der Fall ist und auch nicht vorgesehen ist. Abgesehen von diesem Nachteil enthält jQuery UI mit den Widgets *Dialog* und *Tabs* die beiden Widgets, die für Indico am wichtigsten sind; das in Indico vorhandene Dialog-Widget hat insbesondere beim Anzeigen modaler Dialogfenster einige Probleme. Ein weiterer Pluspunkt für jQuery bzw. jQuery UI ist die Anpassbarkeit der Widgets. Da das Design von Indico einfach gehalten ist, sollten die Widgets nicht durch bunte Balken hervortreten sondern sich sauber in das Design integrieren. Durch die widgetübergreifenden CSS-Klassen ist dies problemlos möglich. Da Prototype derzeit auf einigen Seiten von Indico integriert ist, steht dessen `$`-Funktion mit der gleichnamigen Funktion aus jQuery im Konflikt. Indico nutzt derzeit an einigen Stellen *Preloader*-Mixins, die zuerst einen oder mehrere AJAX-Requests ausführen und danach das Objekt selbst initialisieren. Dabei könnte das *Deferred*-Objekt von jQuery nützlich sein. *Data Binding* wird von jQuery nicht unterstützt. Die ausführliche Dokumentation von jQuery ermöglicht es insbesondere auch neuen Entwicklern, sich schnell einzuarbeiten, während man beim JavaScript-Framework von Indico viel vorhandenen Code studieren muss, um sich mit seinen Funktionen vertraut zu machen. Beide Lizenzen von jQuery sind mit für die Nutzung in Indico geeignet.

3.5. Classy

Bei Classy handelt es sich um eine JavaScript-Bibliothek, um Klassen zu implementieren. Dabei orientiert sie sich am Objektmodell von Python. Classy kann von http://classy.pocoo.org/ heruntergeladen werden.

DOM-Funktionen

Da Classy eine auf auf Klassen/Objekte spezialisierte Bibliothek ist, bietet sie keinerlei DOM-Funktionen.

Objektsystem

Das Objektmodell von Classy orientiert sich an dem der Scriptsprache Python. Jede Klasse erbt von einer Basisklasse. In Python ist das bei *new-style*-Klassen `object`, in Classy ist es `Class`. Um eine neue Klasse zu definieren, erstellt man also eine neue Klasse, die von `Class` erbt, indem man deren statische Methode `$extend(props)` aufruft, wobei `props` ein Objekt ist, das die neue Klasse definiert: die Methode `__init__()` wird zum Konstruktor der neuen Klasse, alle weiteren Methoden werden zu Instanzmethoden der Klasse.

Um einer Klasse Mixins hinzuzufügen, definiert man beim Erstellen der Klasse ein Array `__include__`, welches die Mixins enthält. Bei diesen handelt es sich um JavaScript-Objekte, die Funktionen enthalten. Ein weiteres optionales Objekt, welches man beim Erstellen der Klasse übergeben kann, ist `__classvars__`. Die darin gespeicherten Werte sind innerhalb der Klasse über `this.$class.<name>` verfügbar und nicht instanzspezifisch.

Sofern beim Erstellen einer Klasse eine Methode der Parent-Klasse überschrieben wurde, ist diese Methode in `this.$super` verfügbar, sodass sie komfortabel aufgerufen werden kann.

Genau wie in Python ist auch bei Classy der `new`-Operator optional. Dies wird dadurch erreicht, dass in der Konstruktorfunktion überprüft wird, ob `this === window` ist. In diesem Fall erstellt die Funktion ein neues Objekt und gibt dieses zurück.

Hilfsfunktionen

Classy bietet neben dem Objektsystem keine weitere Funktionalität.

Kompatibilität

Classy exportiert ein globales Objekt `Class`. Sofern dieser Name mit einem bestehenden Objekt kollidiert, kann die ursprüngliche Zuweisung mit `Class.$noConflict()` wiederhergestellt werden und der Rückgabewert dieser Funktion entweder einer anderen Variable zugewiesen oder als Parameter einer anonymen Funktion genutzt werden, um innerhalb dieser `Class` benutzen zu können.

Dokumentation

Classy besitzt eine Online-Dokumentation, in der die Benutzung anhand von Beispielen beschrieben wird und jede Funktion dokumentiert ist. Der Quellcode ist an Stellen, bei denen es zum Verständnis notwendig ist, kommentiert.

Lizenz

Classy steht unter der BSD-Lizenz[31].

Classy implementiert ausschließlich ein Klassensystem, weshalb auch nur dieser Bereich mit Indico verglichen werden kann. Sowohl Indico als auch Classy ermöglichen die Definition von Klassen und das Erben von einer anderen Klasse. Classy hat dabei den Vorteil, dass überschriebene Methoden komfortable aufgerufen werden können, ohne den Namen der ursprünglichen Klasse im Aufruf verwenden zu müssen. Dies bedeutet aber gleichzeitig, dass man jeweils nur eine gleichnamige Parent-Methode aufrufen kann. Dies stellt in Classy aber kein Problem dar, da es keine Mehrfachvererbung unterstützt, sondern nur eine Parent-Klasse und mehrere Mixins, die ausschließlich Funktionen enthalten. Dabei überschreiben gleichnamige Funktionen innerhalb von Mixins die vorherige Funktion. Dadurch, dass Classy zwischen Mixins und der Parent-Klasse unterscheidet, ist keine echte Mehrfachvererbung möglich. Der manuelle Aufruf einer als Konstruktor verwendeten Funktion ist problematisch, da jedes Mixin einen einzigartigen Namen für diese Funktion benutzen müsste. Allerdings wäre es durchaus möglich, diese Funktion genau wie das Mixin zu nennen, sodass der Unterschied zum Indico-Code, der den Konstruktor einer Parent-Klasse mittels `this.NameOfTheClass()` aufruft, nicht so gravierend wäre. Der große Nachteil gegenüber dem Objektsystem von Indico besteht jedoch darin, dass Mixins nicht eigenständig benutzt werden können. Sie sind reine Sammlungen von Funktionen. Kompatibilitätsprobleme zwischen Indico und Classy bestehen nicht, da der Name `Class` nirgendwo verwendet wird. Classy ist deutlich besser dokumentiert als das Objektsystem von Indico, allerdings gibt es aufgrund der starken Nutzung von Klassen in Indico extrem viele Codebeispiele des aktuellen Objektsystems. Die BSD-Lizenz ist mit der von Indico genuzten GPL kompatibel.

3.6. Underscore.js

Underscore.js ist eine insbesonders auf funktionale Programmierung ausgerichtete JavaScript-Bibliothek. Sie hat zum Ziel möglichst viele Hilfsmittel anzubieten und

[31] http://www.opensource.org/licenses/BSD-3-Clause

dabei sowohl die funktionale Programmierung zu vereinfachen als auch *Method Chaining* zu ermöglichen.
Underscore.js kann von der Website http://documentcloud.github.com/underscore/ heruntergeladen werden.

DOM-Funktionen

Underscore.js ist eine auf funktionale Programmierung ausgerichtete Bibliothek. Da sie explizit als „tie to go along with jQuery“ bezeichnet wird, enthält sie selbst keine DOM-Funktionen.

Objektsystem

Underscore.js enthält kein Objektsystem.

Hilfsfunktionen

Die Funktionen von Underscore.js sind alle über das globale Objekt _ verfügbar, wobei sie in fünf Kategorien unterteilt sind: *Collections*, *Arrays*, *Functions*, *Objects* und *Utility*.

Collections enthält Funktionen, die sowohl mit Arrays als auch mit Objekten arbeiten können. Dabei handelt es um Standardfunktionen wie `filter()`, `all()` und `map()`, aber auch um Komfortfunktionen wie `pluck()`, welche aus einer Liste mit Objekten jeweils eine bestimmte Eigenschaft extrahiert.

Arrays enthält Funktionen, die ausschließlich bei Arrays sinnvoll sind, da sie entweder sortierte Elemente oder Zugriff über einen fortlaufenden Index voraussetzen. Neben Standardfunktionen wie `indexOf()` und `range()` gibt es dort auch Funktionen zum Verknüpfen oder Vergleichen von Arrays; beispielsweise `zip()` und `union()`.

Functions enthält Funktionen, die mit Funktionen arbeiten[32]. Während viele dieser Funktionen in JavaScript-Bibliotheken und -Frameworks üblich sind, so z.B. `bind()` zum Fixieren von **`this`** und optional auch von Argumenten (*Currying*), enthält die Kategorie auch komplexere Funktionen wie `memoize()` zum Cachen des Rückgabewerts einer Funktion bei gleichbleibenden Aufrufparametern oder `debounce()` und `throttle()` um viele kurz hintereinander erfolgende Funktionsaufrufe zu reduzieren.

Objects enthält Funktionen, die ausschließlich mit Objekten arbeiten. Darunter sind beispielsweise Funktionen, die bestimmte Elemente (Keys, Werte

[32] „Function (uh, ahem) Functions“ um die Underscore.js-Dokumentation zu zitieren

oder Funktionen) aus einem Objekt extrahieren und Funktionen, die ein Objekt mit Eigenschaften eines anderen Objekts erweitern. Ebenfalls enthalten sind `is<something>()`-Funktionen, die überprüfen, ob ein Objekt einen bestimmten Typ hat. Diese nutzen *Duck Typing*, d.h. es wird überprüft ob sich das Objekt wie ein bestimmter Typ verhält und falls dem so ist wird angenommen, dass es ein Objekt dieses Typs ist.

Utilities enthält neben der bereits aus vielen anderen Libraries bekannten `noConflict`-Funktion eine Funktion, um Underscore.js mit eigenen Funktionen zu erweitern, einen ID-Generator der einzigartige IDs zurückgibt und eine einfache Template-Engine.

Gleichzeitig handelt es sich bei _ auch um eine Funktion, die ein Objekt in einem Wrapper-Objekt einschließt, welches *Chaining* ermöglicht. Eine solche Aufrufkette beginnt in Underscore.js immer mit `_(obj).chain()`. Danach können alle Funktionen aus Underscore.js als Instanzmethoden des Chain-Objekts aufgerufen werden, wobei der erste Funktionsparameter, der ja normalerweise das OBjekt wäre, nicht mehr angegeben werden muss. Um am Ende aus dem Chain-Wrapper wieder ein Array bzw. Objekt zu erhalten, besitzt es die Methode `value()`.

Kompatibilität

Underscore.js exportiert ausschließlich ein globales Objekt _. Die Chance, dass dieser Name in bestehendem Code verwendet wird ist zwar relativ hoch, allerdings existiert solch ein Variablenname in der Regel nur innerhalb von Funktionen, wodurch der Konflikt ignoriert werden kann, sofern keine Funktionen von Underscore.js innerhalb dieser Funktion genutzt werden sollen. Ein weiterer häufiger Verwendungszweck des Namens _ für eine Funktion ist bei der Internationalisierung. Dort nutzt man oftmals `_('something')` um Text zu lokalisieren. In diesem Fall muss man die `_.noConflict()`-Funktion nutzen und Underscore.js unter einem anderen Namen zugänglich zu machen und die ursprüngliche Variable bzw. Funktion wiederherzustellen.

Dokumentation

Underscore.js enthält eine ausführliche Online-Dokumentation mit einer kurzen Beschreibung und einem Beispiel für jede Funktion. Die Beispiele sind nicht ausführbar, allerdings sind Ausgabe bzw. Rückgabewerte des Beispielcodes jeweils angegeben. Neben der Dokumentation existiert auch eine ausführlich kommentierte Version den Quellcodes, die alle internen Abläufe erklärt.

Lizenz

Underscore.js steht unter der MIT-Lizenz.

Underscore.js bietet sehr viele Hilfsfunktionen, die auch in Indico vorhanden sind. Da einige Funktionen wie `each`, `filter` und `map` in der aktuellsten ECMAScript-Spezifikation standardisiert sind und von manchen Browsern bereits nativ unterstützt werden, nutzt Underscore.js diese nativen Funktionen, sofern sie verfügbar sind. Da sie in C oder C++ implementiert sind, erhöht dies die Performance. Damit ist Underscore.js in der Lage, einen Großteil der in Indico vorhandenen Hilfsfunktionen durch performantere und besser dokumentierte Funktionen zu ersetzen. Der Name _ kollidiert auf globaler Ebene mit keinen anderen Variablen/Funktionen. In einzelnen Funktionen wird _ als temporäre lokale Variable genutzt. Man könnte diese Variablen aber ohne Weiteres umbenennen, was unabhängig von Underscore.js sinnvoll wäre, da _ nicht wirklich aussagekräftig ist und `tmp` für eine temporäre Variable ebenso gut geeignet ist. Die MIT-Lizenz von Underscore.js ist mit der GPL kompatibel.

4. Auswahl eines Frameworks

4.1. Migrationspfade

Sowohl die einzelnen Frameworks als auch die Kombination mehrerer Frameworks haben jeweils spezifische Vor- und Nachteile. Im Folgenden werden sowohl der vollständige Umstieg auf ein bestimmtes Framework als auch Kombinationen mehrerer Frameworks vorgestellt und bewertet.

4.1.1. Vollständige Migration zu Prototype

Da Prototype bereits in einigen Bereichen von Indico integriert ist, muss es nur global eingebunden werden; Konflikte sind dabei eher unwahrscheinlich. Danach muss der Code nach und nach vom alten System auf Prototype umgestellt werden. Dies bedeutet, dass ca. 240 Klassen in der Struktur angepasst werden müssen. Sofern Mehrfachvererbung genutzt wird, müssen die jeweiligen Klassen redesignt werden, um das Objektsystem von Prototype nutzen zu können. Um das Generieren von HTML-Elementen auf Prototype umzustellen, müssen ca. 2000 Aufrufe von `Html.*`-Funktionen angepasst werden und, da der `XElement`-Wrapper entfällt, alle darauf aufbauenden Funktionen modifiziert werden. Da JSON-RPC-Aufrufe in Indico über eine zentrale Funktion laufen, kann diese relativ einfach auf die AJAX-Funktionen von Prototype umgestellt werden. Bei der Migration von Hilfsfunktionen wie `each()` oder `curry()` kann aufgrund der größtenteils identischen Parametern mit *Search&Replace* eines modernen Editors gearbeitet werden oder es können unter den alten Funktionsnamen Wrapper bereitgestellt werden, sodass dabei nur wenige Codestellen geändert werden müssen. Eine größere Hürde bei der Migration ist das *Data Binding*-Framework von Indico. Da Prototype keine derartige Funktionalität beinhaltet, muss dieses unter Verwendung moderner JavaScript-Technologien neu entwickelt werden.

Zusammenfassend kann man sagen, dass es sich bei dieser Migration um ein extrem aufwändiges und zeitintensives Unterfangen handelt. Sie führt aber am Ende zu sauberem Code, der sowohl leicht wartbar ist als auch für neue Entwickler einen einfachen Einstieg ermöglicht.

4.1.2. Vollständige Migration zu jQuery und Classy

Um jQuery zu benutzen, muss es zunächst global eingebunden werden und dabei auftretende Konflikte behoben werden. Dazu sollte Prototype möglichst früh entfernt und Prototype-basierter Code angepasst werden, was aufgrund der ähnlichen Paradigmen der beiden Frameworks relativ einfach möglich ist. Da jQuery kein Objektsystem besitzt, Indico aber weiterhin objektorientiert aufgebaut sein soll, wird neben jQuery die Classy-Bibliothek eingebunden. Wie Prototype unterstützt allerdings auch Classy nur eine echte Parent-Klasse und Mixin-Objekte, weshalb ein Redesign der Objektstruktur von Indico notwendig wird. Bei der Migration der DOM-Zugriff und -Manipulationen kann entweder einfach auf jQuery gewechselt werden oder aber die Codestruktur angepasst werden, damit der Code die Vorteile von jQuery vollständig ausnutzt, z.B. durch *Method Chaining* und die Nutzung von Komfortfunktionen wie `wrap()` und `closest()`. Die funktionalen Hilfsfunktionen können teilweise direkt durch jQuery-Äquivalente ersetzt werden. Dabei reicht es oftmals aus, `func` durch `$.func` zu ersetzen. Diverse in jQuery nicht vorhandenen Funktionen müssen dabei neu implementiert werden, sofern sie nicht unabhängig vom restlichen Indico-Framework funktionsfähig sind. jQuery unterstützt genau wie Prototype kein *Data Binding*, weshalb dort eine Eigenentwicklung notwendig ist. Allerdings bietet jQuery durch `val([newValue])` eine Methode um den Wert eines Formularelements auszulesen und zu ändern. An einigen Stellen könnte man diese Funktion an ein bestimmtes Element binden und als *Accessor* an eine andere Funktion übergeben. Wenn man diese Funktionen in einem Objekt speichert, bietet dieses ein sauberes Interface für den Zugriff auf die Werte. Änderungen dieser wirken sich logischerweise direkt auf das zugehörige Formularelement aus.

Der Komplettumstieg auf jQuery ist genau wie der Umstieg auf Prototype sehr aufwändig und zeitintensiv, führt aber ebenfalls zu sauberem und gut wartbarem Code. Da jQuery derzeit eines der beliebtesten Frameworks ist, besteht darüber hinaus eine große Chance, dass ein neuer Entwickler bereits Erfahrung damit hat.

4.1.3. Vollständige Migration zu jQuery und Prototype

Theoretisch wäre es möglich, sowohl jQuery als auch Prototype einzubinden, wobei in diesem Fall die `$`-Funktion von jQuery unter einem anderen Namen - z.B. `$j` - aufgerufen werden müsste, da Prototype ebenfalls `$` nutzt. Da beide Frameworks

einen ähnlichen Funktionsumfang besitzen, gibt es keinen sinnvollen Grund, beide zu benutzen.

4.1.4. Erweiterung durch Prototype

Neben der vollständigen Migration auf ein Framework besteht auch die Möglichkeit, das bestehende Framework vorerst beizubehalten und nur durch Prototype zu erweitern, was ja dadurch, dass einige Seiten teilweise Prototype nutzen, bereits der Fall ist. Neben der globalen Einbindung von Prototype werden dabei die Teile von Indico auf Prototype umgestellt, bei denen es sich besonders anbietet. Ein solcher Teil ist das Eventsystem. Für die Umstellung muss lediglich der Code des Indico-Frameworks bearbeitet werden. Daneben bietet es sich an, die Hilfsmethoden zum browserunabhängigen Zugriff auf bestimmte Eigenschaften des Event-Objekts zu entfernen, sofern Prototype diese bereits normalisiert. Ein weiterer Bereich, in dem die Umstellung auf Prototype sinnvoll ist, ist Drag&Drop; teilweise wird dort sogar schon Prototype genutzt. Bei Hilfsfunktionen, die sowohl im Indico-Framework als auch in Prototype mit derselben *Signatur*, d.h. denselben Aufrufparametern und demselben Rückgabedatentyp, existieren, bietet es sich an, diese Funktionen aus Indico zu entfernen und entweder alle Aufrufe dieser zu ändern, sodass die Prototype-Funktion benutzt wird, oder eine Variable mit demselben Namen wie alten Funktion zu definieren und darin eine Referenz auf die Prototype-Funktion zu speichern. DOM-Zugriff bieten sich ebenfalls zum Migrieren an, allerdings muss dabei der Aufwand mit dem Nutzen verglichen werden. Es gibt über 1200 Aufrufe der `$E`-Funktion, wobei danach oftmals noch `XElement`-Methoden verwendet werden. Diese müssten alle angepasst werden. Dasselbe gilt für die über 2000 Aufrufe der `Html.*`-Funktionen zur Erstellung neuer DOM-Elemente. Unabhängig davon, ob man diesen Bereich migriert oder nicht, sollte neuer Code ausschließlich Prototype nutzen, damit die Verwendung der alten Funktionen nicht zunimmt. Durch das Beibehalten der alten Funktionen ist es jedoch notwendig, dass neu geschriebener Code überprüft, ob es sich bei einem Objekt bereits um ein (Prototype-erweitertes) DOM-Element handelt oder um ein `XElement`-Objekt und in letzterem Fall durch `$(obj.dom)` das enthaltene DOM-Element mit den Prototype-spezifischen Erweiterungen versieht und dieses statt des Wrappers benutzt.

Bei der Erweiterung des bestehenden Frameworks durch Prototype lässt sich viel Entwicklungszeit einsparen, da man nur wenige Klassen umschreiben muss und bestehenden Code größtenteils weiterverwenden kann. Indem man neuen Code aus-

schließlich auf Prototype aufbaut, verbessert sich die Codequalität im Laufe der Zeit automatisch. Dies lässt sich noch beschleunigen, wenn jeder Entwickler, der bestehenden Code modifiziert, diesen auf Prototype umstellt. Durch die Mischung von altem und neuem Code ist die Gesamtqualität des Codes natürlich schlechter, ermöglicht Entwicklern aber dennoch, effizient zu arbeiten, da sie bei der Entwicklung neuer Funktionen größtenteils sauber dokumentierte Funktionen nutzen können und sich nur in Legacy-Code einlesen müssen, sofern keine Alternative existiert.

4.1.5. Erweiterung durch jQuery

Statt das bereits vorhandene Prototype-Framework zu nutzen, besteht auch die Möglichkeit, jQuery zum bestehenden Code hinzuzufügen und diesen damit zu erweitern. Dabei sollte wie auch bei der vollständigen Migration Prototype möglichst früh entfernt werden, um den Konflikt zwischen den beiden Frameworks aufzulösen. Da Prototype derzeit primär für einige Events und Drag&Drop genutzt wird, ist der betroffene Code einfach zu migrieren, da Events durch jQuery selbst und Drag&Drop durch jQuery UI unterstützt werden. Danach ist zu überprüfen, welche Teile von Indico auf jQuery umgestellt werden sollten und welche nicht. Das Eventsystem kann zwar relativ einfach umgestellt werden, allerdings bietet es sich dort an, den bestehenden Code nicht zu verändern sondern nur in neuem Code jQuery zur Eventregistrierung zu nutzen, sodass man die Vorteile von jQuery vollständig nutzen kann statt die jQuery-Methoden in den vorhandenen Funktionen einzubauen, die beispielsweise kein *Chaining* ermöglichen. Da jQuery UI ein Dialog-Widget enthält, sollte die Dialogklasse von Indico auf jeden Fall angepasst werden, um dieses Widget zu nutzen. Dabei ist es sinnvoll, die Kompatibilität mit bestehendem Code möglichst beizubehalten, da diese Klasse eine der am häufigsten genutzen Klassen in ganz Indico ist. Neben den Dialogen ist es auch sinnvoll, das Tab-Widget zu nutzen, da Tabs oftmals innerhalb von Dialogen verwendet werden und ein einheitliches Aussehen erwünscht ist. Bei der Migration von DOM-Zugriffen ist, wie auch beim Umstieg auf Prototype, der Aufwand mit dem Nutzen zu vergleichen. Ebenfalls zu beachten ist die Kompatibilität mit bestehenden Funktionen: Eine Funktion, die jQuery nutzt, muss überprüfen, ob es sich bei einem Objekt um ein DOM-Element bzw. jQuery-Objeklt handelt oder ob ein `XElement`-Wrapper übergeben wurde und in diesem Fall mittels `$(obj.dom)` das enthaltene Element in ein jQuery-Element kopieren. Auch in die andere Richtung muss auf Kompatibilität geachtet werden, indem Funktionen keine jQuery-Objekte sondern DOM-Objekte zurückgeben. Dank des Arrayzugriffs auf jQuery-Objekte ist dies jedoch problemlos möglich.

Indico durch jQuery zu erweitern hat neben dem im Vergleich zur Komplettmigration niedrigen Entwicklungsaufwand verschiedene Vorteile. Mit jeweils einer einzigen Zeile Code lässt sich die Kompatibilität zwischen jQuery-basiertem Code und Legacy-Code gewährleisten, wodurch neuer Code ohne Schwierigkeiten jQuery nutzen und gleichzeitig auf Legacy-Funktionen zugreifen kann. In solchen Fällen bietet es sich allerdings an, zu überprüfen, wie aufwändig es wäre, die Funktion auf jQuery umzustellen. Ein weiterer Vorteil ist die große Menge an jQuery-Plugins[33], die ohne Konflikte mit bestehendem Code zu riskieren eingebunden werden können. In der Regel stehen sie unter einer mit der GPL kompatiblen Open Source-Lizenz.

4.1.6. Zusatz: Underscore.js

Im Indico-Framework existieren diverse unsauber programmierte und/oder fehleranfällig implementierte Hilfsfunktionen. Darunter sind einige, die weder in Prototype noch in jQuery enthalten sind. Die Chance ist jedoch groß, dass eine äquivalente Funktion in der Underscore.js-Bibliothek verfügbar ist. Der Nachteil dabei ist jedoch, dass dadurch einige Funktionen mehrfach verfügbar sind, zum einen in Prototype bzw. jQuery, zum anderen in Underscore.js. Daher ist in diesem Fall auf eine entsprechende Dokumentation zu achten, damit Entwickler wissen, welche Funktion sie bevorzugen sollen, falls sie redundant vorhanden ist.

4.2. Entscheidung für ein Framework

Nach der Analyse der diversen Frameworks bzw. Bibliotheken und der verschiedenen Migrationspfade sind genügend Informationen vorhanden, um eine Entscheidung treffen zu können. Bei dieser Entscheidung muss allerdings auch bedacht werden, dass sie dauerhaft ist, sobald die Migration begonnen hat und entsprechend weit fortgeschritten ist, da es nicht wirtschaftlich wäre, nach kurzer Zeit erneut die Technologie zu wechseln und man damit auch die übrigen Entwickler zwingen würde, sich wieder mit einem neuen Framework auseinanderzusetzen.

Die Idee, das Indico-Framework vollständig zu ersetzen, ist zwar diskussionswürdig, kann aber gerade bei einem von einer Vielzahl an Benutzern aktiv verwendeten System, bei dem es häufig kleinere Fehler-Reports, aber auch Feature-Requests gibt,

[33] http://plugins.jquery.com

nicht wirklich in die Tat umgesetzt werden, da ein Entwickler nicht langfristig ausschließlich an einer solchen Migration arbeiten kann. Die große Zahl zu ändernder Klassen und Funktionen würde möglicherweise sogar die Mitarbeit weiterer Entwickler benötigen. Daher stellt sich nur die Frage, durch welches Framework Indico erweitert werden soll.

Nach Abschätzen der Vor- und Nachteile der beiden Frameworks Prototype und jQuery bietet jQuery einige Vorteile gegenüber Prototype, obwohl dieses bereits teilweise in Indico integriert ist. Mit den Widgets *Dialog*, *Tab* und *Datepicker* deckt jQuery UI drei in Indico häufig genutzte Widgets ab und weitere UI-Elemente können durch das Widgetsystem von jQuery UI einfach und wiederverwendbar entwickelt werden. Durch den komfortablen Zugriff auf die in HTML5 eingeführten `data-`Attribute, die unabängig von der Browserunterstützung von HTML5 über jQuery zugänglich sind, kann auf unübersichtlichen Inline-JavaScript-Code verzichtet werden. Die `$`-Funktion ist intuitiver als die drei in Prototype vorhandenen Funktionen zum Zugriff bzw. dem Erstellen von Elementen. Daher ist die Entscheidung auf **jQuery** gefallen.

Zusätzlich zu jQuery wird **Underscore.js** integriert, da es einige Funktionen implementiert, die durch die Entfernung von Prototype verloren gehen und auch davon abgesehen eine nützliche Funktionssammlung enthält, die die zukünftige Entwicklung vereinfachen kann.

5. Migration zu jQuery

Bei der Migration zu jQuery sind einige Schritte am Anfang zwingend notwendig, während andere optional sind und in relativ beliebiger Reihenfolge ausgeführt werden können. Im Folgenden werden die Migrationsschritte in der Reihenfolge beschrieben, wie sie durchgeführt wurden.

5.1. Vorbereitung

In der Vorbereitungsphase wird jQuery eingebunden und dabei auftretende Konflikte behoben. Es findet zu diesem Zeitpunkt noch keine Migration statt.

5.1.1. Einbinden von jQuery

Um JavaScript-Dateien in Indico einzubinden, fügt man nicht wie üblich <**script**>-Tags in einem HTML-Template ein, sondern erweitert die Funktion `getJSFiles()` der Basisklasse für HTML-Seiten `WPBase`, sodass sie die notwendigen Dateien einbindet. Dazu bietet sich eine separate Funktion an, da neben jQuery selbst auch jQuery UI und Underscore.js eingebunden werden müssen. Diese Liste wird später noch durch diverse jQuery-Plugins erweitert. Da Indico im Produktivmodus einzelne JavaScript-Dateien zu komprimierten Paketen „kompiliert“, muss zusätzlich zur Python-Funktion die Konfigurationsdatei von *jsmin* angepasst werden, damit die neuen Dateien in einem solchen Paket zusammengefasst werden.

```
1 def _includeJQuery(self):
2     info = HelperMaKaCInfo().getMaKaCInfoInstance()
3     files = ['underscore', 'jquery', 'jquery-ui']
4     if info.isDebugActive():
5         return ['js/jquery/%s.js' % f for f in files]
6     else:
7         return ['js/jquery/jquery.js.pack']
```

Listing 17: Einbinden von jQuery und Underscore.js in Indico

5.1.2. Beheben von Konflikten

Nachdem jQuery und Underscore.js eingebunden sind, muss überprüft werden, ob es Konflikte gibt. Ein Konflikt tritt durch das gleichzeitige Vorhandensein von jQuery und Prototype auf. Dieser Konflikt ist bereits aus der Analysephase bekannt und hat zur Folge, dass auf den Seiten, die Prototype nutzen, diverse JavaScripts nicht mehr korrekt funktionieren. Der Konflikt entsteht dadurch, dass jQuery und Prototype beide `$` für sich beanspruchen und daher das zuletzt eingebundene Framework die vorherige Variable überschreibt. Da jQuery nach Prototype eingebunden wird, ist dieses Problem einfach zu lösen - mittels `var $j = jQuery.noConflict();` wird die Prototype-Funktion wiederhergestellt und jQuery via `$j` verfügbar gemacht. Dabei handelt es sich allerdings um eine temporäre Lösung, um Prototype so lange beizubehalten, bis der darauf basierende Code vollständig migriert wurde. Sie hat den Vorteil, dass der Prototype-Code teilweise zeilenweise migriert und währenddessen immer wieder auf korrekte Funktion geprüft werden kann.

5.1.3. Migration von Prototype-basiertem Code

Einige Bereiche des *Room Booking*-Systems von Indico nutzen Prototype zum Registrieren von Events. Darüber hinaus wird Prototype dort zur Formularvalidierung genutzt: Mithilfe des Objekts `Form.Observer` werden die Formulare alle 400ms auf Änderungen überprüft und wenn nötig erneut validiert. Die häufige Prüfung auf Änderungen ist weder effizient noch notwendig. Formularelemente können über die `change`-, `click`- und `keyup`-Events bei jeder Änderung validiert werden ohne mehrmals pro Sekunde alle Formularelemente auf Änderungen zu überprüfen. Um den Validierungscode mit jQuery möglichst effizient zu machen, werden die Events nicht für alle Formularelemente registriert, sondern nur für das Formular selbst, wobei ein *Delegate* genutzt wird, um speziell auf Events der enthaltenen Formularelemente zu reagieren. Listing 18 zeigt die Registrierung der Events; im *submit*-Event wird das Formular erneut validiert und, sofern es nicht gültig ist, das Abschicken des Formulars verhindert und eine Fehlermeldung ausgegeben.

```
1 $j('#bookingForm').delegate(':input', 'keyup change',
    function() {
2     forms_are_valid();
3 }).submit(function(e) {
4     if (!forms_are_valid(true)) {
```

```
5         e.preventDefault();
6         alert($T('There are errors in the form. Please
              correct the fields with red background.'));
7     };
8 });
```

Listing 18: Formularvalidierung via jQuery

Ebenfalls Änderungsbedarf besteht bei den in Listing 19 verwendeten Inline-Events. Da die betroffenen Formularelemente bereits IDs besitzen, kann ihnen über jQuery mühelos ein Eventhandler zugewiesen werden. Dadurch können alle Scripts an derselben Stelle im HTML-Code stehen, was den Code lesbarer macht. Unabhängig davon muss der Code geändert werden, da die dort verwendete $-Funktion aus Prototype stammt.

```
1 <input id="onlyBookings" type="checkbox" onchange="if
    (this.checked) $('onlyPrebookings').checked = false;"/>
2 <input id="onlyPrebookings" type="checkbox" onchange="if
    (this.checked) $('onlyBookings').checked = false;" />
```

Listing 19: Inline-Eventhandler

An vielen Stellen ist es von Vorteil, dass Prototype und jQuery gewisse Ähnlichkeiten besitzen; Listing 20 zeigt die Änderungen eines von Prototype zu jQuery migrierten Codeblocks.

```
1 - $('blockedRooms').setValue(Json.write(roomGuids));
2 + $j('#blockedRooms').val(Json.write(roomGuids));
```

Listing 20: Ähnlichkeit zwischen jQuery und Prototype

Neben dem *Room Booking*-System verwendet auch der *Badge Editor*, ein Tool zum Erstellen von Namensschildern für Konferenzen, das Prototype-Framework. Dort wird es primär zur Unterstützung von Drag&Drop eingesetzt.

Nebem dem Konvertieren des Prototype/Scriptaculous-basierten Drag&Drop-Systems bietet sich dort eine vollständige Migration an, da eine Mischung aus Indico- und

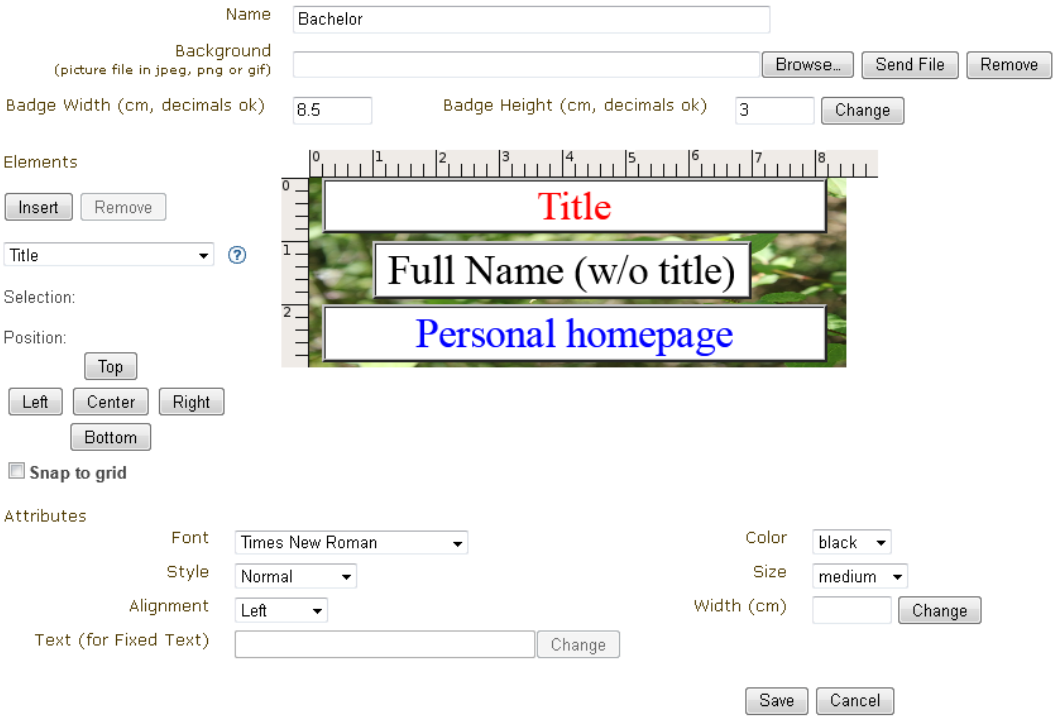

Abbildung 8: Der Indico Badge Editor

Prototype-Code benutzt wird, die ausschließlich DOM-Operationen ausführt. Wie auch auf dem *Room Booking*-Seiten bietet sich auch hier an, ungültigen und unsauberen Code zu korrigieren. In diesem Fall beinhaltet dies die Benutzung korrekter IDs (Leerzeichen sind dort nicht zulässig) und das Entfernen von Inline-Eventhandlern. Die Uploadfunktion für Hintergrundbilder erstellt einen unsichtbaren IFrame für das Uploadformular, damit die Seite beim Abschicken das Formulars nicht neugeladen werden muss. Für diesen Use Case bieten sich zwei Lösungen an, die beide besser als die bestehende sind: Durch die Integration des *jQuery Form*-Plugins lässt sich der IFrame-Code in jQuery-Plugin auslagern, sodass der Code übersichtlicher gehalten wird und gleichzeitig die Kompatibilität mit allen Browsern bestehen bleibt. Sofern letzteres nicht notwendig ist, kann man auch die Fähigkeit moderner Browser nutzen, über AJAX Dateien hochzuladen. Aufgrund der Zielgruppe von Indico, die zu einem nicht zu vernachlässigenden Teil veraltete *Internet Explorer*-Versionen nutzt, ist dies jedoch nicht praktikabel. Da in der aktuellen Phase jedoch das Ziel Prototype zu entfernen im Vordergrund steht und das Form-Plugin auch noch nicht eingebunden ist, bleibt der IFrame-Code zunächst erhalten.

An dieser Stelle sind alle anhand der `$`-Funktion leicht aufzufindenden Codestellen, die Prototype nutzen, angepasst. Durch die Erweiterung der Prototypen nativer Objekte gibt es allerdings noch Prototype-Funktionen, die nicht über `$` oder eine der

globalen Prototype-Klassen gefunden werden können. Um diese aufzuspüren kann man entweder eine Liste aller in Prototype enthaltenen Methoden erstellen und danach suchen oder aber nach der Entfernung von Prototype testen, wo es Probleme gibt. Da Prototype nur auf relativ wenigen Seiten eingebunden wird, bietet sich letztere Methode aufgrund des geringeren Aufwands an, obwohl dabei nicht sichergestellt ist, dass alle Problemstellen gefunden werden. Dafür wäre parallel dazu eine *Code Coverage*-Analyse notwendig, die sich jedoch ausschließlich auf die Codeteile erstrecken dürfte, die möglicherweise Prototype nutzen, da ansonsten mangels automatischer Tests möglichst alle JavaScripts in Indico manuell ausgeführt werden müssten, obwohl ein großer Teil garnicht in der Lage ist, Prototype zu nutzen.

5.1.4. Entfernen von Prototype

Ähnlich wie beim Einbinden von jQuery muss auch beim Entfernen von Prototype die Methode `getJSFiles()` modifiziert werden. Da Prototype nicht global integriert ist, überschreiben verschiedene Unterklassen von `WPBase` diese Methode. Falls dies ausschließlich der Integration von Prototype dient, kann die Methode einfach entfernt werden; falls weitere JavaScripts eingebunden werden, muss die Methode entsprechend angepasst werden.

Die im letzten Abschnitt erwähnten Tests zeigen, dass außer einer undefinierten Funktion `size()` keine Probleme auftreten. Bei der Funktion handelt es sich um eine generische Funktion, um die Anzahl der Elemente in einem *Enumerable*-Objekt, also einem Array, einem Objekt oder einer Argumentliste, zu bestimmen. Während JavaScript bei Arrays über die `length`-Eigenschaft direkten Zugriff auf die Anzahl der Elemente bietet, existiert eine solche Eigenschaft bei Objekten nicht, da es zwei verschiedene Möglichkeiten gibt, die Anzahl zu bestimmen: Zum einen kann man ausschließlich die Eigenschaften zählen, die das Objekt selbst besitzt, zum anderen kann man auch alle über die Prototypenkette erreichbaren Eigenschaften mitzählen. Da Underscore.js mit `_.size()` eine solche Funktion enthält, kann `someObject.size()` einfach durch `_.size(someObject)` ersetzt werden.

Zusammen mit Prototype kann auch die `$j`-Kompatibilitätsschicht entfernt werden, da `$` nun unbenutzt ist. Dazu wird der Aufruf von `jQuery.noConflict` entfernt und per dateiübergreifendem *Suchen&Ersetzen* jedes Vorkommen von `$j` durch `$` ersetzt.

5.2. Migration

5.2.1. Entwicklung des DateRange-Widgets

Das *Room Booking*-Modul von Indico ermöglicht es bestimmten Benutzern Konferenzzimmer über eine gewisse Zeit zu blockieren, sodass diese nur von ausgewählten Gruppen reserviert werden können. Beim Erstellen einer solchen Blockade müssen sowohl Start- als auch Enddatum ausgewählt werden. Dazu werden zwei Textfelder benutzt, die jeweils beim Klick auf den nebenstehenden Button einen Kalender zur Auswahl des Datums öffnen. Diese Datumsfelder werden durch ein Indico-Widget erstellt. Komfortabler ist es allerdings, statt der Textfelder dauerhaft zwei Kalender anzuzeigen. Da das von Indico verwendete Kalenderwidget mehrere gleichzeitig sichtbare Kalender nicht unterstützt und es von Vorteil ist, kein separates *Third-Party*-Script für den Kalender zu benötigen, bietet es sich hier an, den *Datepicker* von jQuery UI zu nutzen.

5.2.1.1. Funktionsanalyse des Datepickers

Der Datepicker wird in der Regel mit einem Textfeld verknüpft, sodass er sich beim Fokussieren des Felds automatisch öffnet und nach Auswahl eines Datums wieder schließt. Es ist allerdings auch möglich, ihn mit einem <**div**> zu verknüpfen. In diesem Fall ist er dauerhaft sichtbar und stellt das ausgewählte Datum lediglich über seine API zur Verfügung. Viele Eigenschaften des Datepickers können individuell konfiguriert werden. Insbesondere ist es möglich, Daten in der Vergangenheit bzw. vor oder nach einem bestimmten Datum zu sperren und die auswählbaren Jahre zu beschränken. Das Widget bietet auch eine Option, um mehrere Monate gleichzeitig anzuzeigen. Hierbei sind allerdings nur mehrere aufeinanderfolgende Monate möglich und es kann unabhängig von der Anzahl der angezeigten Monate nur ein Datum ausgewählt werden. Daher muss das DateRange-Widget zwei separate Datepicker nutzen.

5.2.1.2. Anforderungen

Das DateRange-Widget soll zwei Datepicker nebeneinander anzeigen und ausschließlich die Auswahl gültiger Daten zulassen. Die ausgewählten Daten sollen optional in

versteckten Formularfeldern gespeichert werden, sodass sie beim Absenden des zugehörigen Formulars mitgesendet werden. Es soll möglich sein, bereits beim Erstellen des Widgets ein Start- und Enddatum anzugeben, das standardmäßig ausgewählt ist. Das Datum soll über die Datepicker jeweils änderbar sein, sofern das Widget nicht deaktiviert ist. Um den Benutzerkomfort zu erhöhen, soll jedes gültige Datum auswählbar sein, sodass es immer möglich ist, mit zwei Klicks die gewünschten Daten auszuwählen, falls Jahr und Monat bereits ausgewählt sind. Es soll eine Option geben, um die Auswahl von Daten in der Vergangenheit zu verhindern. Um das Widget so allgemein wie möglich zu halten, soll es allerdings möglich sein, beliebige Datepicker-Optionen zu verändern und zwar sowohl für beide Datepicker als auch nur für einzelne.

5.2.1.3. Implementierung

Da das DateRange-Widget vollständig auf jQuery basieren soll und keinerlei Funktionen aus dem alten Indico-Framework benötigt, bietet es sich an, es als *UI Widget* zu realisieren. Dabei handelt es sich um eine Sammlung von Funktionen, die über eine einzelne jQuery-Funktion zugänglich gemacht werden, und verschiedene Basisfunktionen standardmäßig enthalten. Innerhalb der Funktionen eines Widgets zeigt `this` auf das Widget-Objekt und `this.element` enthält das jQuery-Element, in welchem das Widget erstellt wurde. Wenn ein Widget über ein jQuery-Objekt erstellt wird, das mehrere Elemente enthält, werden separate Widgets erstellt, sodass `this.element` immer exakt ein Element enthält. Einzelne Funktionen im Widget werden durch jQuery automatisch aufgerufen. So wird `_init()` bei jedem Aufruf der Widgetfunktion ausgeführt. `_create()` hingegen wird nur dann ausgeführt, wenn die Widgetfunktion ein neues Widget erstellt.

Im DateRange-Widget wird die `_create`-Funktion genutzt, um die Datepicker zu erstellen und zu konfigurieren. Sollen versteckte Formularfelder verwendet werden, werden sie ebenfalls in dieser Funktion erstellt. Um die geforderte Flexibilität bei der Konfiguration der einzelnen Datepicker zu erreichen, wird die Methode `$.extend()` von jQuery genutzt. Sie erweitert das erste übergebene Objekt mit den Eigenschaften der übrigen Objekte, wobei bereits vorhandene Eigenschaften überschrieben werden. Wie in Listing 21 zu sehen ist, wird als erstes das frühestmögliche Datum festgelegt. Da es unabhängig von `allowPast` möglich sein soll, ein anderes frühestes Datum vorzugeben, werden die zusätzlichen Datepicker-Optionen erst danach festgelegt, damit sie `minDate` überschreiben können. Zuletzt werden die

Optionen für das Standarddatum und das versteckte Formularfeld übergeben. Diese können nicht überschrieben werden, da beide durch die API des DateRange-Widgets ausreichend konfigurierbar sind.

```
1 self.startPicker.datepicker($.extend({
2     minDate: self.options.allowPast ? null : 0
3 }, self.options.pickerOptions,
    self.options.startPickerOptions, {
4     altField: self.startDateField || '',
5     defaultDate: self.options.startDate
6 }));
```

Listing 21: Datepicker-Konfiguration im DateRange-Widget

Abbildung 9: Das jQuery-basierte DateRange-Widget

Um ungültige Daten zu verhindern, wird ein Eventhandler für die *onSelect*-Events der beiden Datepicker registriert. In diesem wird im Falle eines ungültigen Datums, d.h. `startDate > endDate`, das Datum des Datepickers, der gerade nicht verändert wurde, auf das des anderen Datepickers gesetzt. Eine andere Möglichkeit ist, in diesem Eventhandler jeweils `minDate` und `maxDate` des anderen Datepickers anzupassen, sodass ausschließlich gültige Daten ausgewählt werden können. Dies verringert jedoch den Benutzerkomfort, da ein an sich gültiges Datum in diesem Fall u.U. nicht auswählbar ist, ohne zuvor das andere Datum anzupassen.

5.2.2. Migration der Dialogfenster

Indico nutzt Dialogfenster, im Code normalerweise als *Popup* bzw. *ExclusivePopup* bezeichnet, um Informationen und Formulare in einem separaten Fenster darzustellen, das allerdings auf der Seite integriert ist. Während ein solches Fenster geöffnet ist, wird diese Seite durch ein graues Overlay als inaktiv dargestellt und lässt keine Benutzerinteraktion zu. Diese Fenster haben den Vorteil, dass sie fest mit der Seite verknüpft sind und, da es sich nicht um neue Browserfenster handelt, nicht durch Popup-Blocker blockiert werden können.

5.2.2.1. Aktuelle Situation

Dialogfenster werden durch die Klasse `ExclusivePopup` erstellt. Diese generiert die HTML-Struktur des Dialogs und seines Inhalts. Neben `ExclusivePopup` gibt es mit `ExclusivePopupWithButtons` eine davon abgeleitete Klasse, die am unteren Ende des Dialogs eine Buttonzeile erstellt. Um ein Dialogfenster zu erstellen, leitet man eine neue Klasse von einer der gerade genannten Klassen ab und überschreibt dort die `draw`-Methode und ggf. den Konstruktor. In `draw()` generiert man die DOM-Elemente, die im Dialog angezeigt werden sollen und, sofern man `ExclusivePopupWithButtons` nutzt, den Inhalt der Buttonleiste. Beides wird danach an die `draw`-Methode der Parent-Klasse übergeben. Teilweise werden in von `ExclusivePopup` abgeleiteten Dialogfenstern Buttons erstellt welche zwar problemlos funktionieren, sich jedoch im Aussehen von der korrekten Implementierung unterscheiden.

```
1 draw: function() {
2     var self = this;
3     var okButton = Html.input('button',
          {style:{marginRight: pixels(3)}},
          $T(this.buttonTitle));
4     okButton.observeClick(function(){
5         self.close();
6         self.handler(true);
7     });
8
9     var cancelButton = Html.input('button',
          {style:{marginLeft: pixels(3)}},
```

```
        $T(this.cancelButtonTitle));
10      cancelButton.observeClick(function(){
11          self.close();
12          self.handler(false);
13      });
14
15      return
           this.ExclusivePopupWithButtons.prototype.draw.call(
16          this,
17          this.content,
18          Html.div({}, okButton, cancelButton));
19  }
```

Listing 22: Indico-Code zum Erstellen eines einfachen Dialogfensters

In Listing 22 sieht man die Implementation eines einfachen „OK/Abbrechen“-Dialogs. Für beide Buttons werden manuell HTML-Buttons erstellt und mit CSS der Abstand zwischen ihnen festgelegt. Ähnlicher Code ist für alle Dialogfenster mit Buttons notwendig, da das Indico-Framework keine Möglichkeit bietet, Dialog-Buttons in einer abstrakten Art und Weise zu erstellen.

5.2.2.2. Gewünschte Situation

Beim Erstellen von Dialogfenstern, mit oder ohne Buttons, handelt es sich um eine Standardaufgabe. Während der Workflow für ein Dialogfenster ohne Buttons bereits frei von *duplicate code* ist, besteht bei den Buttons Änderungsbedarf. Ein Button kann mit vier Eigenschaften charakterisiert werden: Titel, Aktion, Position und Zustand. Titel, Aktion und Zustand sind dabei buttonspezifisch während die Position bzw. Reihenfolge der Buttons dialogspezifisch ist. Daher bietet es sich an, die Buttons in einem Array zu speichern, wobei es sich bei jedem Button um ein Objekt oder Array handelt. Da der Zustand (aktiv/inaktiv) eines Buttons in der Regel per Code nachträglich geändert wird, muss dieser nicht unbedingt schon bei der Definition des Buttons angegeben werden; die meisten Buttons sind dauerhaft aktiv. Damit gibt es pro Button noch zwei Attribute. Ein Array bietet sich also an, da der Entwickler dabei nicht immer die Namen der Eigenschaften angeben muss.

Die `draw`-Funktion enthält bei komplexeren Dialogen oftmals einen Großteil der Logik des jeweiligen Dialogs. Um sie zu entschlacken bietet es sich an, die Buttonde-

finition in eine separate Funktion auszulagern. Dies dient auch dem Zweck, voneinander unabhängige Codeabschnitte zu trennen. Listing 23 zeigt dabei den Code des jQuery-basierten Dialogs. Die `draw`-Funktion reicht nun ausschließlich den (an den Konstruktor übergebenen) Inhalt des Dialogfensters an die Parent-Funktion weiter; `_getButtons()` gibt ein Array mit den Buttons zurück, wobei ein Button jeweils ein 2-Tupel aus Titel und Callbackfunktion ist. Die Variable **`self`** wird genutzt, da innerhalb des Callbacks **`this`** nicht mehr auf die aktuelle Objektinstanz, sondern auf das globale Objekt `window` zeigt.

```
1  draw: function() {
2      return
          this.ExclusiveWithButtonsPopup.prototype.draw.call(
3           this, this.content);
4  },
5  _getButtons: function() {
6      var self = this;
7      return [
8          [$T(this.buttonTitle), function() {
9              self.close();
10             self.handler(true);
11         }],
12         [$T(this.cancelButtonTitle), function() {
13             self.close();
14             self.handler(false);
15         }]
16     ];
17 }
```

Listing 23: jQuery-Code zum Erstellen eines einfachen Dialogfensters

5.2.2.3. Anpassung der ExclusivePopup-Klasse

Im ursprünglichen Code erstellt die `draw`-Funktion der `ExclusivePopup`-Klasse sowohl das graue Overlay als auch das Dialogfenster samt Titelzeile und „X“-Button zum Schließen. Da das Dialog-Widget von jQuery UI diese Aufgaben übernimmt, muss die neue Funktion lediglich ein leeres <**`div`**>-Element erstellen und es mit

der jQuery-Methode `dialog()` in einen Dialog umwandeln. Danach muss nur noch der Inhalt des Dialogs eingefügt werden, wobei eine Kompatibilitätsschicht dafür sorgt, dass auch `XElement`-Objekte des Indico-Frameworks als Dialoginhalt übergeben werden können. Wie auch das Datepicker-Widget, akzeptiert der jQuery UI-Dialog ein Objekt mit verschiedenen Optionen. Um Unterklassen zu ermöglichen, „ihren“ Dialog zu beeinflussen, können die Optionen durch das Überschreiben einer weiteren Funktion beeinflusst werden. Dies ist insbesondere für die Klasse `ExclusivePopupWithButtons` notwendig, da sie dem Dialog Buttons hinzufügen muss.

Aufgrund der einfachen Erweiterbarkeit der Dialog-Optionen besteht die erweiterte Dialogklasse `ExclusivePopupWithButtons` primär aus Code, um das Button-Array in das vom Dialog-Widget benötige Format zu konvertieren.

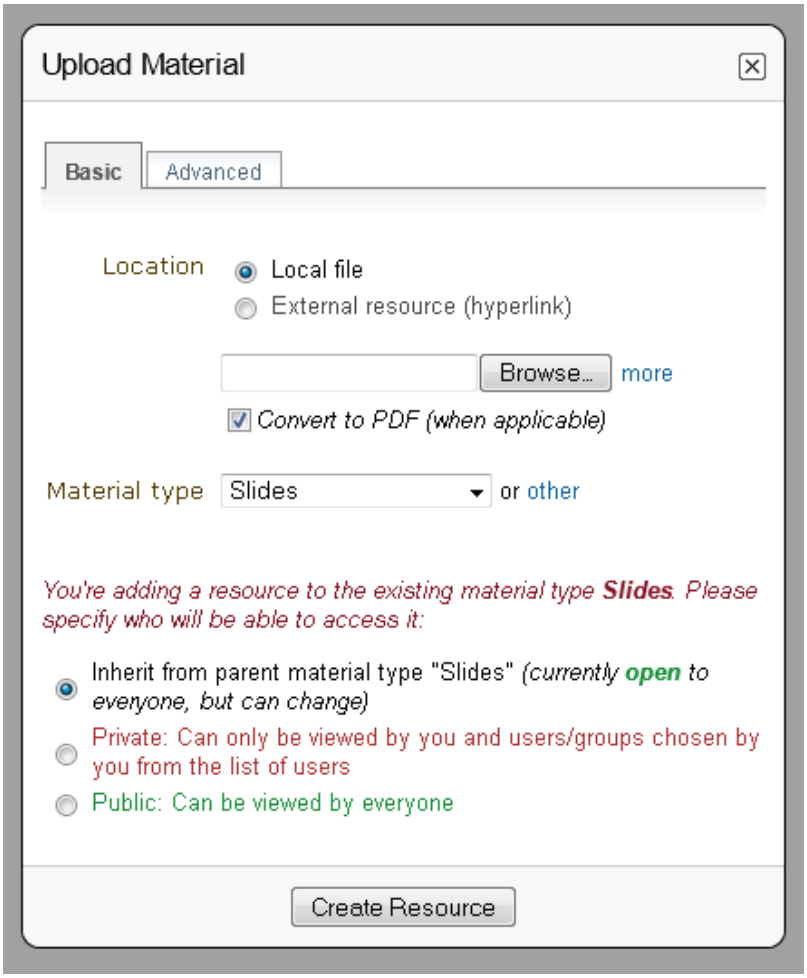

Abbildung 10: Klassisches Dialogfenster in Indico

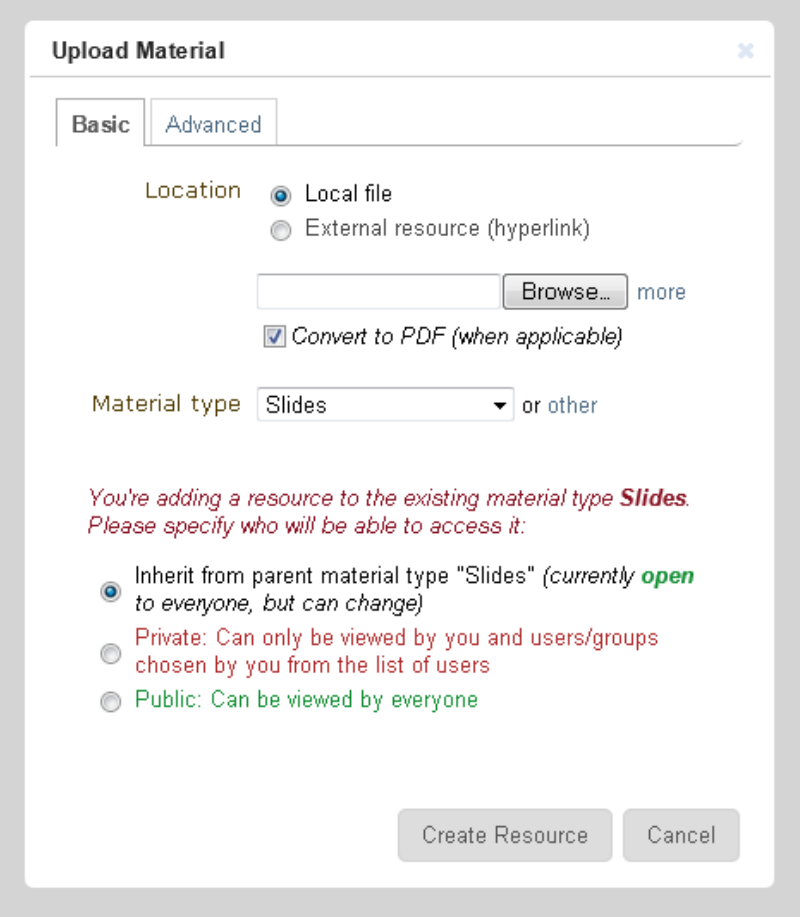

Abbildung 11: jQuery UI-basiertes Dialogfenster in Indico

5.2.3. Migration der statischen Tabs

Indico verwendet zwei Arten von Tabs, was primär damit zusammenhängt, dass in Indico ursprünglich kein JavaScript verwendet wurde. Bei den „dynamischen“ Tabs handelt es sich um eine Klasse, an die für jedes Tab Titel und Generatorfunktion übergeben werden und daraus ein Tab-Widget erzeugt wird. Die „statischen“ Tabs hingegen werden im serverseiten Python-Code erzeugt und laden beim Auswählen eines Tabs eine neue Seite. Um Designanpassungen möglichst einfach zu machen, ist darauf zu achten, dass nicht mehrere Codestellen dieselbe Aufgabe ausführen. Tabs sollten also ausschließlich an einer Stelle generiert werden. Da das Tab-Widget aus jQuery UI verwendet werden soll, bietet es sich an, dieses auch für die statischen Tabs zu nutzen. Dazu gibt es prinzipiell zwei Möglichkeiten: Man kann das Template, das für die statischen Tabs genutzt wird, so anpassen, dass es dieselbe Struktur und CSS-Klassen wie die jQuery-Tabs verwendet. Dies vereinheitlicht zwar das Aussehen und CSS-Änderungen betreffen alle Tabs, allerdings kann es zu Problemen führen

falls sich die Struktur in einer zukünftigen Version von jQuery UI ändert. Daher ist es sinnvoll, im Template ausschließlich die vom Tab-Widget benötigte Struktur zu erzeugen und danach per JavaScript das eigentliche Tab-Widget zu erzeugen.

Dazu wird der HTML-Code der statischen Tabs mit einer speziellen CSS-Klasse versehen, sodass er via JavaScript einfach gefunden werden und in ein Tab-Widget umgewandelt werden kann. Normalerweise ist vorgesehen, dass der Inhalt von Tabs entweder bereits vorhanden ist oder per AJAX nachgeladen wird. Um das klassische Verhalten zu erreichen, muss sowohl verhindert werden, dass ein AJAX-Request abgesendet wird, als auch dass versucht wird, einen nicht vorhandenen Tab-Container anzuzeigen. Dies lässt sich am einfachsten dadurch erzielen, dass die Links der einzelnen Tabs als Linkziel die dem Tab zugehörige Seite enthalten und keinerlei *click*-Events ausgeführt werden. Für ersteres wird beim Tab-Link in einem `data`-Attribut die Ziel-URL gespeichert, da das Tab-Widget das Linkziel verändert; nach dem Erstellen des Widgets wird die gespeicherte URL wieder als `href` zugewiesen. Um das Tab-Widget daran zu hindern, den Tab-Inhalt per AJAX zu laden, werden einfach die *click*-Events gelöscht - wegen der Nutzung von Namespaces betrifft dies ausschließlich die durch das Tab-Widget hinzugefügten Eventhandler.

Der Code, um die Links anzupassen, ist in Listing 24 zu sehen. **`this`** zeigt dabei jeweils auf den äußersten Container des Tab-Widgets, sodass nur die Links innerhalb eines direkten Child-Elements des Containers mit der Klasse `ui-tabs-nav` modifiziert werden.

```
1 $('> .ui-tabs-nav a', this).each(function() {
2     var $this = $(this);
3     $this.attr('href', $this.data('href'));
4     $this.unbind('click.tabs');
5 });
```

Listing 24: Umwandlung der Tabs in reguläre Links

5.3. Probleme bei der Migration

Die Migration zu jQuery lief größtenteils unproblematisch ab, die einzigen Probleme gab es durch die Verwendung von HTML-Elementen an Stellen, an denen man

reinen Text erwarten würde, insbesondere in Dialogtiteln. Da diese direkt an das jQuery UI-Dialog-Widget durchgereicht werden, müssen sie entweder reiner Text oder aber ein DOM-Element bzw. jQuery-Objekt sein. Aufgrund der Verwendung der `Html.*`-Funktionen wurden jedoch `XElement`-Wrapper übergeben, sodass es je nach Browser zu einer Fehlermeldung oder einem leeren Titel kam. Um diesen Fehler auszuschließen, wäre es nützlich, wenn jQuery beliebige Objekte unterstützen würde, sofern sie eine Methode enthalten, die HTML oder ein DOM-Element zurückgibt. Allerdings hat das Fehlen dieses Features auch seine Vorteile, da unsauberer Code so sofort auffällt und überarbeitet werden kann.

6. Fazit

Im Rahmen dieser Bachelorarbeit am CERN wurden nicht nur zwei der bekanntesten JavaScript-Frameworks unter die Lupe genommen, sondern auch die *Internals* des Indico-Frameworks. Bei allen Frameworks haben sich sowohl Stärken als auch Schwächen gezeigt, wobei sich jQuery gegenüber Prototype insbesondere durch die enthaltenen UI-Widgets behaupten konnte.

Tatsächlich auf jQuery umgestellt wurden bisher nur einige Teile von Indico, wobei mit den Dialogfenstern und Tabs Elemente ausgesucht wurden, die an möglichst vielen Stellen zum Einsatz kommen. Im Verlauf der Arbeit hat sich neben dem Primärziel, das bestehende Framework zu ersetzen oder zu erweitern, ein weiteres Ziel herauskristallisiert: jQuery so früh wie möglich, d.h. sobald Prototype entfernt und Konflikte behoben waren, in die von allen Entwicklern genutzten Version zu integrieren, damit sowohl bei Neuentwicklungen als auch beim nicht frameworkbezogenen Refactoring bestehender Elemente jQuery und jQuery-Plugins verwendet werden können.

Der Entwicklungszweig, in dem Dialoge und Tabs auf jQuery umgestellt werden, befindet sich auch nach Abschluss dieser Arbeit noch in Entwicklung, da noch kleinere Anpassungen am Design und einige Bugfixes notwendig sind.

Neben den bereits migrierten Bereichen von Indico gibt es noch weitere, bei denen sich ein, zumindest DOM-Operationen betreffend, vollständiger Umstieg auf jQuery lohnen würde. Einer dieser Bereiche ist die Validierung von Formularelementen. Dort würde sich die Nutzung des jQuery Validation-Plugins[34] anbieten, wobei dabei aufgrund der dynamischen Erzeugung vieler Formulare und den teilweise komplexen, d.h. nicht auf ein einzelnes Feld beschränkten, Validierungsregeln weitere Analysen notwendig werden, um festzustellen, ob dieses Plugin tatsächlich für den gewünschten Zweck geeignet ist.

Wenn langfristig der Anteil an jQuery-basiertem Code deutlich gestiegen ist, bietet es sich auch an, über eine vollständige Migration nachzudenken.

[34] http://bassistance.de/jquery-plugins/jquery-plugin-validation/

Literaturverzeichnis

[ECM11] ECMA. *ECMA-262: ECMAScript Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, 5.1 edition, June 2011.

[Fer08] Pedro Ferreira. Indico developer's guide for the newbie. `http://indico.cern.ch/materialDisplay.py?materialId=slides&confId=37937`, 2008.

[HV06] Peter A. Henning and Holger Vogelsang. *Handbuch Programmiersprachen: Softwareentwicklung zum Lernen und Nachschlagen*. Hanser Verlag, 2006.

[HWW09] Philippe Le Hégaret, Ray Whitmer, and Lauren Wood. Document object model (DOM). W3C architecture domain, W3C, January 2009. `http://www.w3.org/DOM/`.

[Ind11] Indico. Indico project homepage. `http://indico-software.org/`, 2011. [18. August 2011].

[NCH+04] Gavin Nicol, Mike Champion, Philippe Le Hégaret, Jonathan Robie, Lauren Wood, Arnaud Le Hors, and Steve Byrne. Document object model (DOM) level 3 core specification. W3C recommendation, W3C, April 2004. http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407.

[Pem00] Steven Pemberton. XHTML™ 1.0: The extensible hypertext markup language - a reformulation of HTML 4 in XML 1.0. first edition of a recommendation, W3C, January 2000. `http://www.w3.org/TR/2000/REC-xhtml1-20000126`.

[W3C11] W3C. HTML5. Editor's draft, W3C, August 2011. `http://dev.w3.org/html5/spec/Overview.html`.

[Whe07] David A. Wheeler. The Free-Libre / Open Source Software (FLOSS) license slide. `http://www.dwheeler.com/essays/floss-license-slide.html`, September 2007. [27. August 2011].

A. Anhang

A.1. Screenshots von Indico

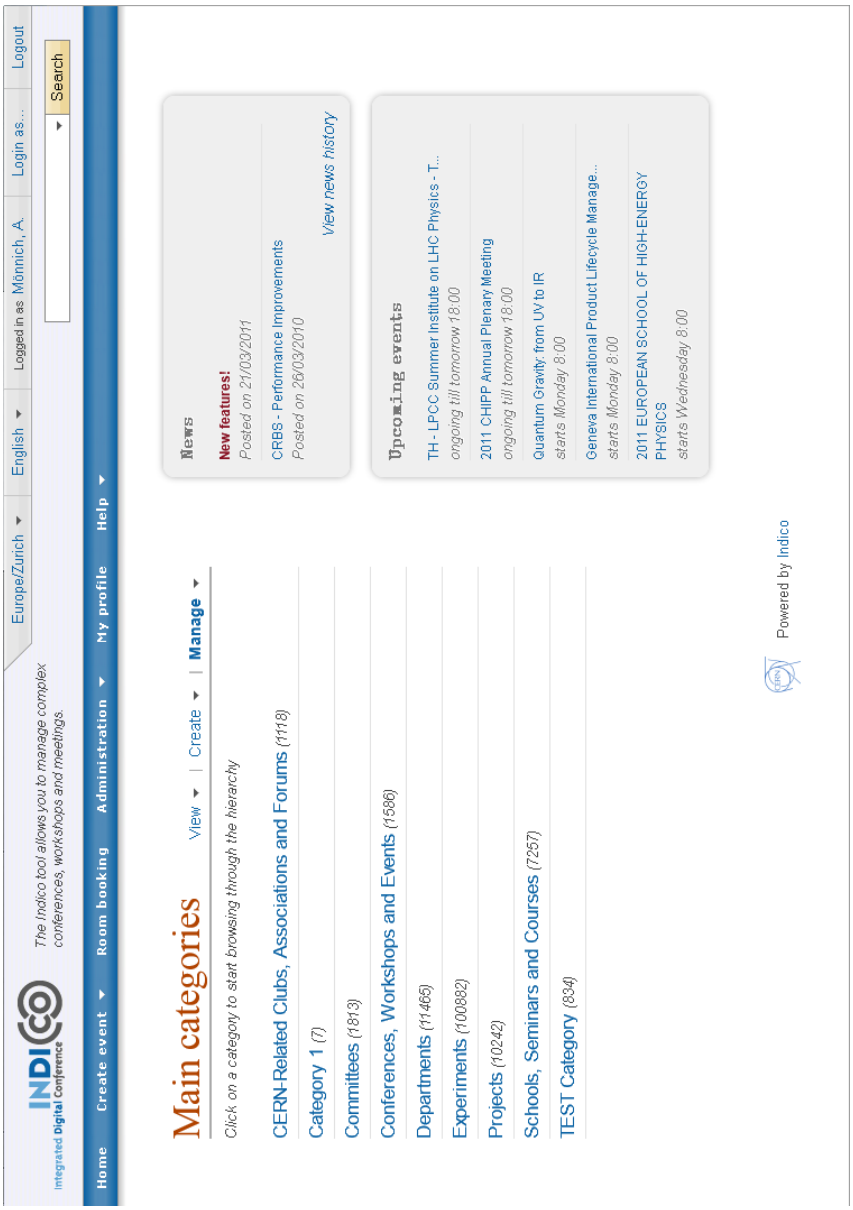

Abbildung A.1: Die Startseite von Indico

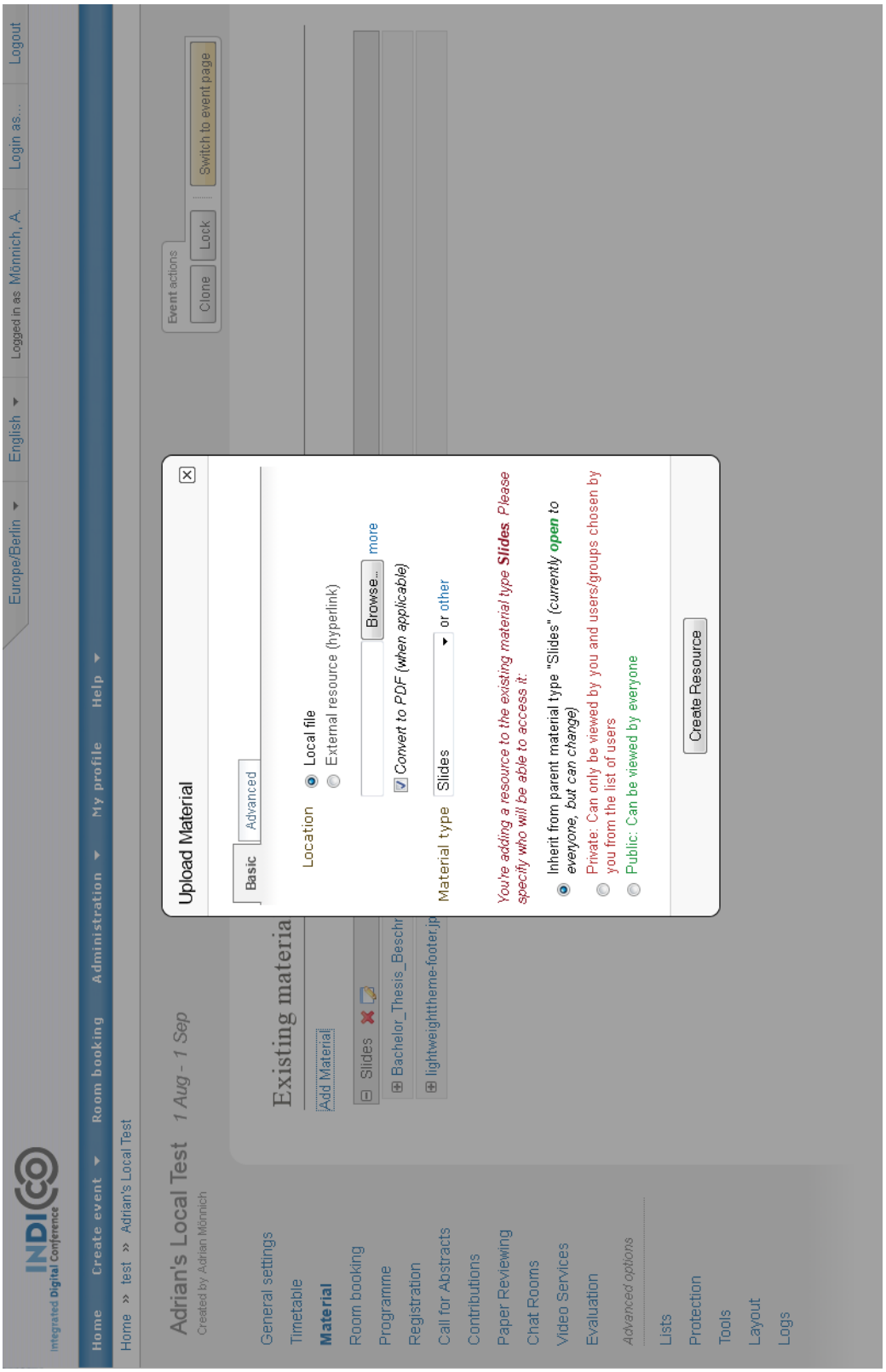

Abbildung A.2: Ein klassischer Dialog im Materialeditor von Indico

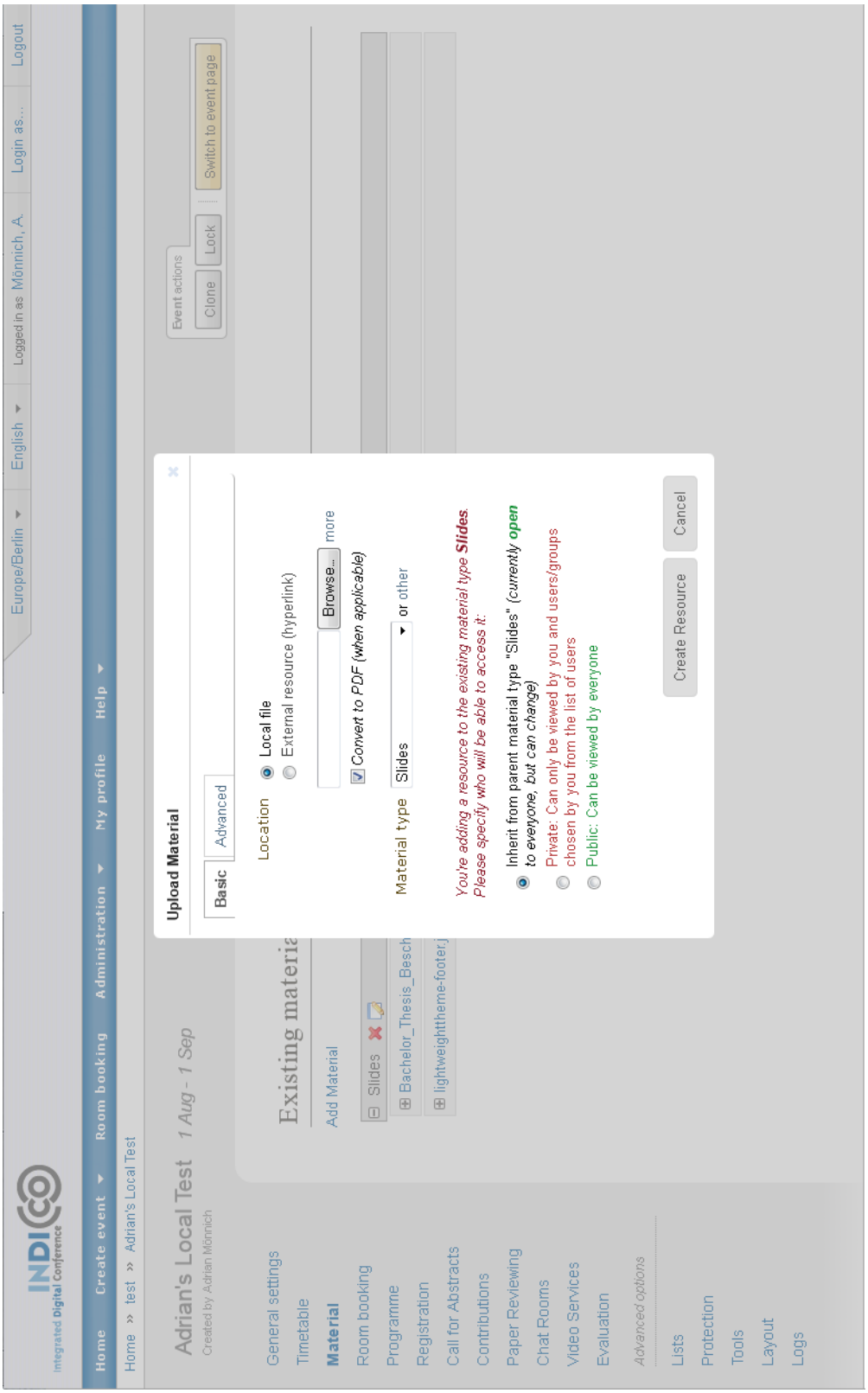

Abbildung A.3: Ein jQuery-Dialog im Materialeditor von Indico

A.2. Quellcodes

A.2.1. Quellcode des DateRange-Widgets

```
1  (function($, undefined) {
2    $.widget('cern.daterange', {
3      // Default options
4      options: {
5        disabled: false,
6        allowPast: false,
7        useFields: true,
8        fieldNames: ['startDate', 'endDate'],
9        labels: ['', ''],
10       labelAttrs: {},
11       startDate: null,
12       endDate: null,
13       pickerOptions: {
14         changeMonth: true,
15         changeYear: true,
16         firstDay: 1,
17         showOtherMonths: true,
18         selectOtherMonths: true,
19         unifyNumRows: true,
20         yearRange: '-0:+2'
21       },
22       startPickerOptions: {},
23       endPickerOptions: {}
24     },
25
26     _create: function() {
27       var self = this;
28
29       // Create the hidden fields containing the dates
30       if(self.options.useFields) {
31         self.startDateField = $('<input/>', {
32           type: 'hidden',
```

```
33            name: self.options.fieldNames[0],
34            value: self.startDate || ''
35          }).appendTo(self.element);
36          self.endDateField = $('<input/>', {
37            type: 'hidden',
38            name: self.options.fieldNames[1],
39            value: self.endDate || ''
40          }).appendTo(self.element);
41        }
42        else {
43          self.startDateField = self.endDateField = null;
44        }
45
46        // Create the markup for the inline widget
47        self.container = $('<div/>', { style: 'display:
            table; width: 100%;' });
48        self.startDateContainer = $('<div/>', { style:
            'float: left', align: 'center'
            }).appendTo(self.container);
49        self.endDateContainer = $('<div/>', { style: 'float:
            left; margin-left: 10px;', align: 'center'
            }).appendTo(self.container);
50        $('<span/>', self.options.labelAttrs)
51          .html(self.options.labels[0])
52          .appendTo(self.startDateContainer);
53        $('<span/>', self.options.labelAttrs)
54          .html(self.options.labels[1])
55          .appendTo(self.endDateContainer);
56        self.startPicker =
            $('<div/>').appendTo(self.startDateContainer);
57        self.endPicker =
            $('<div/>').appendTo(self.endDateContainer);
58
59        // We have to do attach the container to the
            document before creating the pickers because e.g.
            chrome otherwise gives them zero height
60        self.element.addClass('ui-daterange')
```

```
61          .append(self.container);
62
63        // Create the date pickers
64        // The picker options first get a default minDate,
             then the global picker options, then the
             picker-specific options, then our non-overridable
             options
65        self.startPicker.datepicker($.extend({
66          minDate: self.options.allowPast ? null : 0
67        }, self.options.pickerOptions,
             self.options.startPickerOptions, {
68          altField: self.startDateField || '',
69          defaultDate: self.options.startDate
70        }));
71        self.endPicker.datepicker($.extend({
72          minDate: self.options.allowPast ? null : 0
73        }, self.options.pickerOptions,
             self.options.endPickerOptions, {
74          altField: self.endDateField || '',
75          defaultDate: self.options.endDate
76        }));
77        self.pickers = self.startPicker.add(self.endPicker);
78
79        // Prevent nonsense ranges
80        self.pickers.datepicker('option', 'onSelect',
             function() {
81          if(self.startPicker.datepicker('getDate') >
               self.endPicker.datepicker('getDate')) {
82            if(this == self.startPicker[0]) {
83              self.endPicker.datepicker('setDate',
                   self.startPicker.datepicker('getDate'));
84            }
85            else {
86              self.startPicker.datepicker('setDate',
                   self.endPicker.datepicker('getDate'));
87            }
88          }
```

```
89          self.options.startDate =
               self.startPicker.datepicker('getDate');
90          self.options.endDate =
               self.endPicker.datepicker('getDate');
91        });
92        // Copy current date in case it was not set before
93        self.options.startDate =
             self.startPicker.datepicker('getDate');
94        self.options.endDate =
             self.endPicker.datepicker('getDate');
95        // Disable the date picker if necessary
96        if(self.options.disabled) {
97          self.pickers.datepicker('disable');
98        }
99      },
100
101     destroy: function() {
102       this.element.removeClass('ui-daterange')
103       if(this.startDateField) {
104         this.startDateField.remove();
105       }
106       if(this.endDateField) {
107         this.endDateField.remove();
108       }
109       this.pickers.datepicker('destroy');
110       this.container.remove();
111       $.Widget.prototype.destroy.apply(this, arguments);
112     },
113
114     _setOption: function(key, value) {
115       if(key == 'disabled') {
116         if(value) {
117           this.pickers.datepicker('disable');
118         }
119         else {
120           this.pickers.datepicker('enable');
121         }
```

```
122       }
123       else if(key == 'allowPast') {
124         this.pickers.datepicker('option', 'minDate', value
              ? null : 0);
125       }
126       else if(key == 'startDate') {
127         this.startPicker.datepicker('setDate', value);
128       }
129       else if(key == 'endDate') {
130         this.endPicker.datepicker('setDate', value);
131       }
132
133       $.Widget.prototype._setOption.apply(this, arguments);
134     },
135
136     getStartPicker: function() {
137       return this.startPicker;
138     },
139
140     getEndPicker: function() {
141       return this.endPicker;
142     },
143
144     getDates: function() {
145       return [this.options.startDate,
            this.options.endDate];
146     }
147   });
148 })(jQuery);
```

Listing A.1: DateRange-Widget

A.2.2. Quellcode der ExclusivePopup-Klasse

```
1 type("ExclusivePopup", ["Printable"], {
2   open: function() {
```

```
3       this.draw();
4       this.canvas.dialog('open');
5     },
6
7     draw: function(content, customStyle, popupStyle) {
8       customStyle = customStyle || {};
9       if(!content) {
10        content = '';
11      }
12      else if(content.dom) {
13        // support indico XElement objects
14        content = content.dom;
15      }
16
17      if(popupStyle === undefined) {
18        popupStyle = customStyle;
19      }
20      var container = $('<div class="exclusivePopup"/>')
21      container.css(popupStyle).append(content);
22
23      this.showCloseButton = !!this.title; // show if there
          is a title
24      this._makeCanvas();
25      this.canvas.empty().css(customStyle).append(container);
26      this.dialogElement.css(customStyle);
27    },
28
29    close: function() {
30      if(this.isopen) {
31        this.canvas.dialog('close');
32      }
33    },
34
35    _getDialogOptions: function() {
36      // Override to set additional dialog options
37      return {};
38    },
```

```
39
40  _makeCanvas: function() {
41    if(!this.canvas) {
42      var opts = $.extend(true, {
43        autoOpen: false,
44        draggable: true,
45        modal: true,
46        resizable: false,
47        closeOnEscape: true,
48        title: this.title,
49        minWidth: '250px',
50        minHeight: 0,
51        open: $.proxy(this._onOpen, this),
52        close: $.proxy(this._onClose, this),
53        beforeClose: $.proxy(this._onBeforeClose, this)
54      }, this._getDialogOptions());
55      this.canvas = $('<div/>').dialog(opts);
56    }
57    if(!this.dialogElement) {
58      this.dialogElement = this.canvas.dialog('widget');
59    }
60    this.buttons =
         this.dialogElement.find('.ui-dialog-buttonset
         button');
61  },
62
63  _onBeforeClose: function(e) {
64    // Close button clicked
65    if(e.originalEvent && $(e.originalEvent.currentTarget)
66        .hasClass('ui-dialog-titlebar-close')) {
67      if(isFunction(this.closeHandler) &&
           !this.closeHandler()) {
68        return false;
69      }
70    }
71    // Escape key
```

```
72      else if(e.keyCode && e.keyCode ===
            $.ui.keyCode.ESCAPE) {
73        e.stopPropagation(); // otherwise this triggers
              twice for some reason
74        if(this.closeHandler === null ||
              !this.showCloseButton) {
75          // Ignore escape if we don't have a close button
76          return false;
77        }
78        if(isFunction(this.closeHandler) &&
              !this.closeHandler()) {
79          return false;
80        }
81      }
82    },
83
84    _onOpen: function(e) {
85      this.isopen = true;
86      if(this.closeHandler === null ||
            !this.showCloseButton) {
87        this.dialogElement.find('.ui-dialog-titlebar-close')
88          .hide();
89        if(!this.title) {
90          this.dialogElement.find('.ui-dialog-titlebar')
91            .hide();
92        }
93      }
94
95      if(this.postDraw() === true) {
96        // refresh position
97        var pos = this.canvas.dialog('option', 'position');
98        this.canvas.dialog('option', 'position', pos);
99      }
100   },
101
102   postDraw: function() {
```

```
103     // Override if you need to do something after
            displaying the dialog
104   },
105
106   _onClose: function(e, ui) {
107     this.isopen = false;
108     this.canvas.dialog('destroy');
109     this.canvas.remove();
110     this.canvas = this.dialogElement = null;
111     this.buttons = $();
112   }
113
114 }, function(title, closeButtonHandler, printable,
     showPrintButton, noCanvas) {
115   this.title = any(title, null);
116
117   // Called when user clicks the close button, if the
        function
118   // returns true the dialog will be closed.
119   this.closeHandler = any(closeButtonHandler, positive);
120   // the close button will be enabled in draw() so if that
        method is overridden it will not be drawn
121   this.showCloseButton = false;
122
123   // The maximum allowed height, used since it doesn't look
124   // very nice it the dialog gets too big.
125   this.maxHeight = 600;
126
127   // Decides whether the popup should be printable. That
        is, when the user
128   // clicks print only the content of the dialog will be
        printed not the
129   // whole page. Should be true in general unless the
        dialog is containing
130   // something users normally don't want to print, i.e.
        the loading dialog.
131   this.printable = any(printable, true);
```

```
132
133   // Whether to show the print button or not in the title
134   // Note: the button will only be shown if the popup
        dialog has a title.
135   // and is printable.
136   this.showPrintButton = any(showPrintButton && title &&
        printable, false);
137
138   this.buttons = $();
139   if(!noCanvas) {
140     this._makeCanvas();
141   }
142 });
```

Listing A.2: ExclusivePopup-Klasse

A.3. DOM-Interfaces

A.3.1. Das Node-Interface

```
1  interface Node {
2    // NodeType
3    const unsigned short ELEMENT_NODE = 1;
4    const unsigned short ATTRIBUTE_NODE = 2;
5    const unsigned short TEXT_NODE = 3;
6    const unsigned short CDATA_SECTION_NODE = 4;
7    const unsigned short ENTITY_REFERENCE_NODE = 5;
8    const unsigned short ENTITY_NODE = 6;
9    const unsigned short PROCESSING_INSTRUCTION_NODE = 7;
10   const unsigned short COMMENT_NODE = 8;
11   const unsigned short DOCUMENT_NODE = 9;
12   const unsigned short DOCUMENT_TYPE_NODE = 10;
13   const unsigned short DOCUMENT_FRAGMENT_NODE = 11;
14   const unsigned short NOTATION_NODE = 12;
15
```

```
16  readonly attribute DOMString nodeName;
17            attribute DOMString nodeValue;
18
19  readonly attribute unsigned short nodeType;
20  readonly attribute Node parentNode;
21  readonly attribute NodeList childNodes;
22  readonly attribute Node firstChild;
23  readonly attribute Node lastChild;
24  readonly attribute Node previousSibling;
25  readonly attribute Node nextSibling;
26  readonly attribute NamedNodeMap attributes;
27  readonly attribute Document ownerDocument;
28
29  Node insertBefore(in Node newChild, in Node refChild)
30  Node replaceChild(in Node newChild, in Node oldChild)
31  Node removeChild(in Node oldChild)
32  Node appendChild(in Node newChild)
33  boolean hasChildNodes();
34  Node cloneNode(in boolean deep);
35  void normalize();
36  boolean isSupported(in DOMString feature, in DOMString
       version);
37  readonly attribute DOMString namespaceURI;
38  attribute DOMString prefix;
39
40  readonly attribute DOMString localName;
41  boolean hasAttributes();
42  readonly attribute DOMString baseURI;
43
44  // DocumentPosition
45  const unsigned short DOCUMENT_POSITION_DISCONNECTED =
       0x01;
46  const unsigned short DOCUMENT_POSITION_PRECEDING = 0x02;
47  const unsigned short DOCUMENT_POSITION_FOLLOWING = 0x04;
48  const unsigned short DOCUMENT_POSITION_CONTAINS = 0x08;
49  const unsigned short DOCUMENT_POSITION_CONTAINED_BY =
       0x10;
```

```
50   const unsigned short
        DOCUMENT_POSITION_IMPLEMENTATION_SPECIFIC = 0x20;
51
52   unsigned short compareDocumentPosition(in Node other)
53   attribute DOMString textContent;
54
55   boolean isSameNode(in Node other);
56   DOMString lookupPrefix(in DOMString namespaceURI);
57   boolean isDefaultNamespace(in DOMString namespaceURI);
58   DOMString lookupNamespaceURI(in DOMString prefix);
59   boolean isEqualNode(in Node arg);
60   DOMObject getFeature(in DOMString feature, in DOMString
        version);
61   DOMUserData setUserData(in DOMString key, in DOMUserData
        data, in UserDataHandler handler);
62   DOMUserData getUserData(in DOMString key);
63 };
```

Listing A.3: DOM-Node-Interface (DOM Level 3 Core Specification)

A.3.2. Das Document-Interface

```
1  interface Document : Node {
2    readonly attribute DocumentType doctype;
3    readonly attribute DOMImplementation implementation;
4    readonly attribute Element documentElement;
5    Element createElement(in DOMString tagName)
6    DocumentFragment createDocumentFragment();
7    Text createTextNode(in DOMString data);
8    Comment createComment(in DOMString data);
9    CDATASection createCDATASection(in DOMString data)
10   ProcessingInstruction createProcessingInstruction(in
        DOMString target, in DOMString data)
11   Attr createAttribute(in DOMString name)
12   EntityReference createEntityReference(in DOMString name)
13   NodeList getElementsByTagName(in DOMString tagname);
```

```
14   Node importNode(in Node importedNode, in boolean deep)
15   Element createElementNS(in DOMString namespaceURI, in
        DOMString qualifiedName)
16   Attr createAttributeNS(in DOMString namespaceURI, in
        DOMString qualifiedName)
17   NodeList getElementsByTagNameNS(in DOMString
        namespaceURI, in DOMString localName);
18   Element getElementById(in DOMString elementId);
19   readonly attribute DOMString inputEncoding;
20   readonly attribute DOMString xmlEncoding;
21   attribute boolean xmlStandalone;
22   attribute DOMString xmlVersion;
23   attribute boolean strictErrorChecking;
24   attribute DOMString documentURI;
25   Node adoptNode(in Node source)
26   readonly attribute DOMConfiguration domConfig;
27   void normalizeDocument();
28   Node renameNode(in Node n, in DOMString namespaceURI, in
        DOMString qualifiedName)
29 };
```

Listing A.4: DOM-Document-Interface (DOM Level 3 Core Specification)