Hochschule Offenburg

Medien und Informationswesen

Bachelor-Thesis

Konzeption und Implementierung einer generischen Autorisierungsebene für AngularJS

Wintersemester 2015

Virtual Identity AG

Autor:
Felix Heck

Betreuer:
Prof. Dr. Tom Rüdebusch
Dipl. Informatiker Frank Rittinger

Abstract

Bei der Entwicklung von Single-page-Anwendungen wird gelegentlich eine zusätzliche clientseitige Authorisierungsebene benötigt, die die *User Experience* der Anwendung begünstigt, indem Elemente des *User Interface* rollenbasiert ein- und ausgeblendet werden. Da den Single-page-Anwendungen unterschiedliche Content-Management-Systeme zugrunde liegen, konnte bisher keine einheitliche Autorisierungsebene realisiert werden. Diese Arbeit hat zum Ziel, hinsichtlich der Problemstellung eine generische Implementierung zu konzipieren und zu realisieren, sodass diese unkompliziert in Anwendungen auf Basis des Frameworks AngularJS integriert werden kann.

Dazu werden zunächst grundlegende Techniken und Architekturstile erläutert, im Kontext der Single-page-Anwendungen analysiert und bewertet. Anschließend folgt die Definition von Prämissen sowie die Anforderungsspezifikation. Der Vergleich und die Bewertung zweier bestehender Module verdeutlichen die Notwendigkeit eines generischen Moduls. Im Rahmen der Konzeption werden mögliche Lösungsansätze analysiert. Daraufhin wird die Funktionalität, das Zusammenwirken der benötigten Komponenten sowie die Maßnahmen zur Qualitätssicherung konzipiert. Im weiteren Verlauf werden besondere Aufgabenstellungen der Implementierung näher betrachtet und entsprechende Strategien erläutert. Der Prototyp wird daraufhin in ein reales Projekt integriert und anhand dessen evaluiert. Für die in der Evaluation identifizierten Schwachstellen werden Lösungsansätze konzipiert und einer davon implementiert. Zuletzt wird die Arbeit reflektiert und Ausblicke hinsichtlich der Weiterentwicklung und Distribution gewährt.

Das Ergebnis der Arbeit ist eine prototypische Implementierung auf Basis des *Web Application Frameworks* AngularJS, welche bereits als Open Source Modul (`https://github.com/felixheck/uXess`) öffentlich verfügbar und in realen Web-Anwendungen einsetzbar ist.

Stichworte: Autorisierung, AngularJS, Direktive, Single-page-Anwendung, Modul, User Experience, User Interface, generisch, clientseitig, rollenbasiert

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Abkürzungsverzeichnis

AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
CLI	Command-line Interface
CMS	Content-Management-System
CSS	Cascading Style Sheets
DI	Dependency Injection
DOM	Document Object Model
DRY	Don't Repeat Yourself
HATEOAS	Hypermedia as the Engine of Application State
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IIFE	Immediately-Invoked Function Expression
JSON	JavaScript Object Notation
JWT	JSON Web Token
KISS	Keep It Simple Stupid
MVC	Model View Controller
MVVM	Model View ViewModel
MVW	Model View Whatever
npm	Node Package Manager
REST	Representational State Transfer

SEO	Search Engine Optimization
UI	User Interface
URI	Uniform Resource Identifier
UX	User Experience
WAF	Web Application Framework
WWW	World Wide Web
XML	Extensible Markup Language
YAGNI	You Ain´t Gonna Need It

Kapitel 1

Einleitung

Der theoretische Teil der Konzeption und Implementierung einer generischen Autorisierungsebene in AngularJS gliedert sich wie folgt: eine Einführung in die Thematik; die Analyse der Systemanforderungen; die Betrachtung bestehender Ansätze und Systeme; die Konzeption und Implementierung eines Prototyps; dessen Auswertung und Evaluation sowie ein abschließendes Résumé.

Einleitend wird zunächst die Theorie hinsichtlich derjenigen Technologien und Konzepte hergeleitet, welche für die Realisierung des Prototyps erforderlich sind. Daraufhin werden, in Absprache mit Frontend-Entwicklern des Betriebs, die Anforderungen an das neue System herausgearbeitet und bereits bestehende Konzepte in Anbetracht dieser Anforderungen analysiert und bewertet. Jene Anforderungen dienen als Grundlage für die Konzeption des Prototyps, der im Anschluss implementiert und in ein bestehendes Projekt integriert wird. Durch die definierten Anforderungen und den realen Einsatz der Implementierung, lässt sich der Prototyp präzise bewerten. Im Anschluss wird für die offenen Punkte, welche aus der Evaluation resultieren, jeweils ein denkbarer Handlungsansatz oder eine Implementierung herausgearbeitet. Im Zuge eines abschließenden Résumés, wird die Thesis reflektiert und ein perspektivischer Ausblick gewährt.

1.1 Problemstellung

Bei der Entwicklung von modernen Single-page-Anwendungen – Applikationen, welche durch dynamisches Laden der erforderlichen Ressourcen [1] die *User Experience* (UX) erhöhen sollen [2] – bedarf es gelegentlich einer Autorisierungsebene. Während der eigentliche Autorisierungsprozess aus Sicherheitsgründen

auf dem Server stattfindet, da dort die Manipulation von Datenbeständen weit schwerer als im Client ist, wird ein weiterer aufseiten des Clients benötigt. Diese zweite Autorisierung zielt abermals auf die Steigerung der UX und die dadurch bedingte Reduktion von Frustrationseffekten bei Nutzern ab. Hierfür müssen zugleich Elemente des *User Interface* (UI) auf Basis der Nutzerrechte ein- bzw. ausgeblendet werden, um den Nutzern nur jene Elemente zu bieten, mit denen sie ohne daraus resultierende Fehlermeldungen interagieren können. Darüber hinaus müssen nicht-autorisierte Anfragen sowohl server- als auch clientseitig verarbeitet werden, um den Anwender visuell verständlich über die fehlenden Rechte zu informieren.

Im Agenturalltag werden zur Erstellung jener Anwendungen diverse Content-Management-Systeme (CMS) eingesetzt, welche im Bezug auf die Schnittstellen und das Rechtemanagement differieren. Infolge der Varianzen konnte bisher keine generische, clientseitige Autorisierungsebene realisiert werden. Durch die redundante Programmierung werden weitere Aufwände impliziert, welche für den Betrieb sowohl von zeitlicher als auch finanzieller Bedeutung sind.

1.2 Zielsetzung

Ziel der Arbeit ist es, die in Kapitel 1.1 erwähnten, zusätzlichen Aufwände durch die Entwicklung einer prototypischen Anwendung so zu reduzieren, dass die clientseitige Autorisierung – sprich das rollenbasierte Ein-/Ausblenden von UI-Elementen – für Frontend-Entwickler möglichst unkompliziert umzusetzen ist. Die Erstellung einer vollständig universellen Lösung stellt dabei den Idealfall dar, da hierbei der Aufwand am signifikantesten reduziert wird.

Jene Implementierung soll in Form eines generischen Moduls oder Middleware umgesetzt und dabei insbesondere in Webanwendungen auf Grundlage des *Web Application Frameworks* (WAF) AngularJS angewandt werden. Ob die Lösung server- oder clientseitig realisiert wird und welche konkreten Techniken dabei ihren Einsatz finden, bleibt dabei offengehalten. Diese Entscheidungen werden im Rahmen der Konzeption getroffen.

Kapitel 2

Theoretische Grundlagen

In diesem Kapitel werden aktuelle Technologien und Konzepte – welche für die Konzeption und Realisierung des Prototyps herangezogen werden – vorgestellt, um ein grundlegendes Verständnis zu vermitteln. Neben kurzen Erläuterungen werden sowohl Vorteile als auch Nachteile betrachtet und die Entscheidung für die jeweilige Technologie im Kontext des Prototyps begründet.

2.1 Webanwendungen

In den letzten Jahren haben sich Webanwendungen und deren Architekturen fortschreitend entwickelt. Dabei werden inzwischen traditionelle Anwendungen vermehrt durch Single-page-Anwendungen substituiert. Nachfolgend werden die wesentlichen Unterschiede, einschließlich den Vorzügen und Mankos, herausgearbeitet und veranschaulicht.

2.1.1 Traditionelle Anwendungen

In klassischen Webanwendungen kommen in der Regel mehrere Webseiten zum Einsatz, welche auf *Hypertext Markup Language* (HTML) basieren und durch Hyperlinks miteinander verknüpft sind. Der Server generiert die HTML-Seiten, indem zugrunde liegende Daten in Templates integriert werden, und übermittelt diese über das *Hypertext Transfer Protocol* (HTTP) an den Client. Daneben ist der Server für die wesentliche Geschäftslogik und Zugriffskontrolle verantwortlich. Die ausgelieferten Seiten können bei Bedarf vom Client unter Zuhilfenahme von JavaScript interaktiv gestaltet werden. So lassen sich beispielsweise Formulare vor dem Übermitteln an den Server validieren [3].

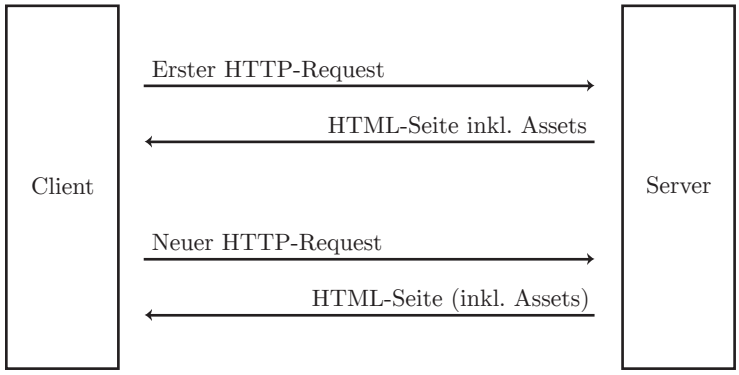

Abbildung 2.1: Traditionelle Client-Server-Kommunikation

Beim Aufruf einer dieser Webseiten wird die gesamte HTML-Datei inklusive aller benötigten Assets geladen; dieses Szenario tritt ebenfalls ein, sollte eine weitere Webseite dieser Anwendung angefordert werden (vgl. Abbildung 2.1) [4]. Aufgrund dieses Verhaltens werden benötigte Ressourcen und Daten oft redundant geladen. Der Einsatz diverser Caching-Techniken lässt gewisse Ressourcen wie Assets für einen bestimmten Zeitraum speichern [5], wodurch lediglich die HTML-Seite selbst bei jedem Request vollständig gerendert und geladen werden muss. Die HTTP-Requests, welche die einzelnen Seiten anfordern, werden in den traditionellen Anwendungen meist synchron ausgeführt; während der Server den Request verarbeitet, ist keine Interaktion möglich [3].

2.1.2 Single-page-Anwendungen

Der Ansatz der Single-page-Anwendungen (SPA) versucht die Schwächen der traditionellen Webanwendungen zu umgehen, indem Aufgaben innerhalb der Beziehung zwischen Client, Server und Datenbeständen neu verteilt werden. Im Zuge dieser Neuverteilung wird zudem die Komplexität vom Server zum Client verschoben.

Der Server dient bei modernen, modularen Anwendungen vorrangig als Schnittstelle zwischen Datenbank und Webanwendung, der Validierung von Requests und der Zugriffskontrolle. Beim ersten Request wird eine statische HTML-Seite samt aller erforderlichen Assets ausgeliefert [2]. Für weitere Requests werden ausschließlich zusätzlich benötigte Ressourcen dynamisch nachgeladen und in das UI integriert (vgl. Abbildung 2.2). *Asynchronous JavaScript and XML* (AJAX) bildet derweil den Kern für diese Dynamik; das asynchrone Ausführen

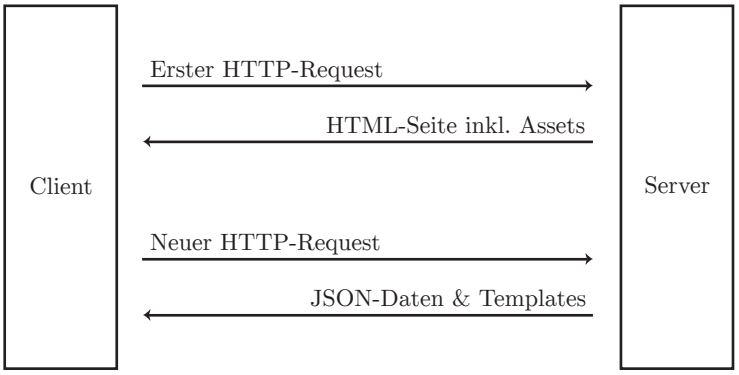

Abbildung 2.2: Client-Server-Kommunikation in SPAs

von Requests ermöglicht es, auch während der serverseitigen Verarbeitung mit dem UI der Anwendung zu interagieren. Die Datenintegration und die Anwendungslogik sind somit überwiegend clientseitig untergebracht [3].

Die asynchrone Verarbeitung der Requests bildet den Kern der SPAs, welche unter anderem das Ziel verfolgen, durch das Vermeiden redundanter Requests, performantere Webanwendungen hervorzubringen. Durch dynamisches Nachladen von Daten und Templates, werden die UIs anwenderfreundlicher gestaltet. Diese beiden Umstände resultieren in der Steigerung der UX [2].

Neben den Stärken von SPAs gibt es auch einige Schwachpunkte, welche von Entwicklern entsprechend berücksichtigt werden müssen und insbesondere im Rahmen der *Search Engine Optimization* (SEO) deutlich werden. Grund hierfür ist, dass die Crawler der Suchmaschinen üblicherweise kein JavaScript ausführen und somit für diese nur reines HTML-Markup detektierbar ist [1]. Da es jedoch bereits von Suchmaschinen und Entwicklern Ansätze gibt, das Problem zu umgehen [6, 7], fällt dieser Punkt nur geringfügig ins Gewicht. Ein weiterer technischer Nachteil ist dabei, dass das serverseitige Fehlerlogging für die eigentliche Geschäftslogik als defizitär angesehen werden kann; die Identifikation und das Beseitigen von Fehlern wird dadurch erschwert. Obwohl sich Fehler mit JavaScript über die Konsole der meisten Browser ausgeben lassen [8], sind diese nur temporär auf dem jeweiligen Client einsehbar. Dieses Problem lässt sich durch clientseitiges, persistentes Logging umgehen, welches die Fehler lokal speichert [9] und überträgt, sobald die Anwendung dazu imstande ist. Jedoch sollte die Anwendung vorweg mit ausführlichen Tests, welche die Anwendung samt eventueller Fehlerquellen weitestmöglich abdecken, untersucht werden [8].

2.2 AngularJS

Für die Implementierung von Single-page-Anwendungen existieren einige *Web Application Frameworks*. Die Idee hinter diesen ist, eine Palette von wiederverwendbaren Funktionen und Klassen zu bieten, welche es Entwicklern ermöglicht, in kurzer Zeit funktionale Anwendungen zu erstellen [3]. Die am weitesten verbreiteten Frameworks im Bereich moderner Single-page-Anwendungen sind: Ember.js, Backbone.js und AngularJS [1].

Nachfolgend wird lediglich auf AngularJS und dessen Vor- und Nachteile eingegangen. Die erwähnten Alternativen – Ember.js und Backbone.js – werden nicht weiter berücksichtigt; die Gründe gegen deren Einsatz und für AngularJS werden in Kapitel 2.2.2 näher erläutert.

AngularJS ist ein von Google und einer Community verwaltetes Framework mit unzähligen Komponenten [3], dessen Entwickler das Ziel verfolgen, die Umsetzung von modularen und testbaren Webanwendungen zu simplifizieren. Das Framework ist Open Source und unterliegt der MIT-Lizenz. Im Vergleich zu den übrigen Frameworks verfolgt AngularJS ein Konzept, das die konventionellen Architekturmuster *Model View Controller* (MVC) und *Model View ViewModel* (MVVM) kombiniert [3, 8].

Die Implementierung unter Berücksichtigung des MVC-Musters sorgt innerhalb einer Anwendung für die Separation von Daten (Model) und deren Darstellung (View), bei der beide Komponenten über eine weitere bedient werden. Die dritte Komponente (Controller) enthält die Logik der Anwendung. Das MVVM-Muster, bei welchem die Daten über bidirektionales Data-Binding an die jeweiligen UI-Elemente gebunden werden, stellt eine Abwandlung hiervon dar [8]. Die Kombination beider Architekturmuster wird auch als *Model View Whatever* (MVW) bezeichnet [10].

2.2.1 Vor- und Nachteile

Die Kombination der beiden Architekturmuster, insbesondere das integrierte bidirektionale Data-Binding, ist eines der signifikantesten Merkmale. Daneben deckt AngularJS viele weitere Funktionen ab, welche üblicherweise bei der Entwicklung von SPAs benötigt werden, wie die Manipulation des *Document Object Model* (DOM) oder das clientseitige Routing [3].

Die Verwendung von *Dependency Injection* (DI) erlaubt die Abhängigkeiten eines Objektes in Form von Parametern zu injizieren. Im Vergleich zu regulären Abhängigkeiten ist es somit ohne weiteres möglich, zu Testzwecken sogenannte

Mock-Objekte zu übergeben, das Verhalten komplexer Objekte zu simulieren und das Testen von Modulen zu vereinfachen [1].

Zudem wird die Programmlogik von AngularJS in den Template-Dateien mit einfachem HTML in Form von Attributen oder eigenen Elementen kombiniert, wodurch diesbezüglich eine steilere Lernkurve als bei den alternativen Frameworks zu erwarten ist. Obendrein wird das Projekt von einer großen Community und Google stetig weiterentwickelt [1, 11]. Die Community ist unter anderem für die hohe Anzahl an entwickelten Modulen und Erweiterungen für AngularJS verantwortlich [12], welche die Bedeutung des Projekts stärken.

Der erhebliche Funktionsumfang des Frameworks kann dabei sowohl als Vor- als auch Nachteil angesehen werden. Dadurch, dass viele Vorgänge größtenteils eigenständig koordiniert werden, kann der Entwickler vereinzelt den Überblick darüber verlieren, welche Prozesse im Hintergrund ablaufen. Nach mehreren funktionellen Erweiterungen und Fehlerbehebungen der ersten Version, wird momentan eine zweite entwickelt. Diese beiden Versionen weisen stellenweise große Unterschiede auf, sodass eine Aktualisierung der mit AngularJS umgesetzten Projekte nur über Umwege möglich sein wird [1]. Hierfür ist nicht zuletzt der Einsatz des modernen ECMAScript 6 verantwortlich [13].

2.2.2 Bewertung des Frameworks

Trotz genannter Nachteile und dem anstehenden *Major Release*, wird bei der Umsetzung des Prototyps AngularJS in der Version 1.4.7 eingesetzt. Diese Entscheidung lässt sich auf zwei grundlegende Argumente zurückführen: einerseits wird in der Agentur nahezu ausschließlich AngularJS eingesetzt und andererseits wird in der professionellen Entwicklung für Endkunden auf keine Software zurückgegriffen, welche sich noch in der Entwicklungsphase befindet.

Die Frontend-Entwickler des Betriebs haben sich dazu entschieden, sich auf ein System zu fokussieren, um die betriebsinterne Lernkurve zu optimieren. Die Wahl fiel dabei – in erster Linie aufgrund der relativ großen Community und Google – auf AngularJS. Wie zuvor erwähnt, hängen diese mit der großen Anzahl an Modulen zusammen. Google wird dabei als Sicherheit für eine langfristige Pflege des Projektes gesehen. Eine Einstellung der Weiterentwicklung wäre, bedingt durch den damit notwendigen Systemwechsel, für die Agentur und deren Kunden mit finanziellen Aufwänden verbunden. Letzten Endes war auch das integrierte, bidirektionale Data-Binding entscheidend, da dieses die Arbeit der Entwickler ungemein vereinfacht.

Abgesehen davon, dass sich AngularJS 2.0 noch im Alpha-Entwicklungsstadium befindet und nicht für den Produktiveinsatz empfohlen wird, gibt es weitere

Gründe gegen dessen Verwendung [11]. Insbesondere werden ältere Versionen von Browsern, wie dem Internet Explorer, nicht weiter unterstützt [14]. Da diese jedoch teils noch immer von Nutzern verwendet werden, stellt Version 2.0 für viele Projekte und Kunden keine Alternative dar. Zudem wären aufgrund der ausgeprägten Unterschiede zwischen beiden *Major Releases* anspruchsvolle Upgrades bestehender Projekte nötig. Diese Aufwände müssen in Relation zu den für Kunden meist nur geringfügigen Mehrwert gesetzt werden.

Die alternativen Frameworks Backbone.js und Ember.js finden nur noch in einzelnen Projekten Verwendung, sodass diese keine Option darstellen.

2.3 Node.js

Um die Entwicklung von Anwendungen zu vereinfachen, kann auf serverseitige Technologien wie *Node.js* [15] zurückgegriffen werden. Bei *Node.js* handelt es sich um eine freie Laufzeitumgebung auf Basis von Googles V8 Engine zum Entwickeln von serverseitigen JavaScript-Anwendungen. Die Laufzeitumgebung ist auf den am meisten verbreiteten Plattformen einsetzbar [16].

Quellcode 2.1: Einfacher Webserver mit *Node.js* [15]

```
1 var http = require('http');
2
3 http.createServer(function (req, res) {
4   res.writeHead(200, {'Content-Type': 'text/plain'});
5   res.end('Hello World\n');
6 }).listen(1337, '127.0.0.1');
```

Node.js zeichnet sich insbesondere durch das verwendete I/O-Modell aus, das ereignisgesteuert und nicht-blockierend ist. Dies ermöglicht die Entwicklung von skalierbaren Webanwendungen. Bei Betrachtung des Quellcode 2.1 wird der asynchrone und ereignisgesteuerte Charakter deutlich, welcher für die hohe Skalierbarkeit mitverantwortlich ist. In diesem Beispiel wird in wenigen Zeilen ein Webserver eingerichtet, der unter `http://127.0.0.1:1337/` erreichbar ist. Bei jedem Herstellen einer Verbindung – in diesem Fall das Ereignis – wird die *Callback*-Funktion als dazugehörige Aktion aufgerufen; besteht keine Verbindung, findet in *Node.js* keine weitere Aktivität statt [15]. Die Laufzeitumgebung kann somit als äußerst effizient angesehen werden.

2.3.1 Node Package Manager

Der *Node Package Manager* (npm) ist die übliche Paketverwaltung für *Node.js*, wird bei dessen Installation als globale Abhängigkeit installiert und besteht aus zwei Komponenten: einem *Command-line Interface* (CLI) zur Verwaltung der globalen und lokalen Abhängigkeiten sowie einer Paketdatenbank. Über die Paketdatenbank lassen sich die Module Dritter suchen, herunterladen und installieren [16–18]. Die Installation kann global und lokal erfolgen; es wird zwischen *Dependencies* und *Development Dependencies* unterschieden.

Für die ausführliche Beschreibung der zu entwickelnden Anwendung sowie der projektspezifischen Konfiguration der Paketverwaltung wird die JSON-Datei `package.json` eingesetzt. Als deskriptive Eigenschaften werden unter anderem Name, Kurzbeschreibung, Version, Stichwörter und Informationen zum Autor festgelegt (vgl. Quellcode 2.2).

Quellcode 2.2: `package.json` – Descriptive Eigenschaften

```
1 "name": "<name>",
2 "version": "<major.minor.patch>",
3 "description": "<description>",
4 "keywords": [],
5 "author": {
6   "name": "<prename lastname>",
7   "email": "<email>",
8   "url": "<url>"
9 }
```

Module, die für den Produktiveinsatz vonnöten sind, sind als allgemeine Abhängigkeit zu definieren; für den Entwicklungsprozess benötigte Module werden als *Development Dependency* installiert. Dafür werden bei der Installation die entsprechenden Optionen `--save` und `--save-dev` verwendet, welche die Module in der `package.json` adäquat hinterlegen (vgl. Quellcode 2.3). Durch die klare Trennung sind die Abhängigkeiten für weitere Entwickler und Anwender nachvollziehbar dem Verwendungszweck zuzuordnen.

Zudem werden die erforderlichen Versionen von *Node.js* und npm in der Rubrik `engines` aufgeführt. Diese Informationen geben an, ab welchen Versionen die Entwicklungsumgebung funktionsfähig ist. Da im Rahmen der Konzeption und Implementierung keine Kompatibilitätsprüfung mit älteren Versionen vorgesehen ist, werden die aktuell verwendeten Versionsnummern hinterlegt.

Quellcode 2.3: `package.json` – Exemplarische Abhängigkeiten

```
"devDependencies": {
  "angular-mocks": "^1.4.7"
},
"dependencies": {
  "angular": "^1.4.7",
  "angular-animate": "^1.4.7"
},
"engines": {
  "node": "0.12.7",
  "npm": "^2.11.3"
}
```

Neben serverseitigen lassen sich auch clientseitige Pakete mit npm verwalten. Darüber hinaus stellt *Node.js* gewisse Funktionen zur Verfügung, welche die Entwicklung von clientseitigen JavaScript-Modulen simplifizieren können, wie das automatisierte Ausführen bestimmter Aufgaben [19]. Die Aufgaben werden in Form von Konsolenbefehlen definiert (vgl. Quellcode 2.4) und können im Anschluss ausgeführt werden.

Quellcode 2.4: `package.json` – Exemplarische Skript-Definition

```
"scripts": {
  "start": "node server.js"
}
```

2.3.2 Vor- und Nachteile

Die Laufzeitumgebung *Node.js* ist sehr leichtgewichtig, sprich es lässt sich eine hohe Funktionalität mit wenig Programmieraufwand umsetzen. Die asynchrone Verarbeitung von Requests sowie das ereignisgesteuerte, nicht-blockierende Modell sorgen für eine hohe Effizienz, welche besonders bei Echtzeit-Anwendungen zum Tragen kommt [15]. Auch die große Open Source Community, welche eine immense Anzahl an Modulen [18] – in den möglichen Sprachen C, C++ und JavaScript [15] – beisteuert, ist vorteilhaft. Nachteilig dagegen ist, dass aufgrund der Verwendung eines einzelnen Prozessor-Hauptkerns die vertikale Skalierbarkeit beschränkt ist. Diese Problematik lässt sich jedoch mit entsprechenden Modulen umgehen, indem stattdessen horizontal skaliert wird [15, 20].

2.3.3 Bewertung der Laufzeitumgebung

Wie bereits in Kapitel 2.3.1 erläutert wurde, stellt npm hilfreiche Funktionen bereit, welche die clientseitige Entwicklung simplifizieren. Durch den Einsatz von *Node.js* lässt sich somit der komplette Entwicklungsprozess mit JavaScript durchführen. Die Verwendung einer einzigen Sprache erlaubt es, *Best Practices* und *Design Patterns* sowohl im server- als auch im clientseitigen Quellcode identisch umzusetzen; es entstehen Synergieeffekte, welche sowohl die Entwicklung der Lösung als auch eines eventuell notwendigen Backends erleichtern.

2.4 Representational State Transfer

Um die Kommunikation zwischen Client und Server zu vereinfachen, werden deren Schnittstellen vereinheitlicht, indem sich die Anwendungen und *Web Services* an Paradigmen wie *Representational State Transfer* (REST) orientieren. Bei REST handelt es sich um einen abstrakten Architekturstil [5], der erstmals in Roy Fieldings Disseration [21] benannt und beschrieben wird. Die Dissertation behandelt unter anderem die Architekturstile netzwerkbasierter Software hinsichtlich ihrer Skalierbarkeit. REST liegen dabei die Konzepte der Ressourcen und deren Repräsentationen zugrunde:

> „The server sends a representation describing the state of a resource [and the] client sends a representation describing the state it would like the resource to have.“ [22, S. 32]

Diese beiden Konzepte werden um Verknüpfungen zwischen den Ressourcen und *Uniform Resource Identifier* (URI) ergänzt [23]. Im Anschluss werden die architektonischen Grundsätze von REST betrachtet; HTTP-basierte, REST-konforme Schnittstellen erläutert sowie der Architekturstil bewertet und dessen Einsatz in Bezug auf AngularJS begründet.

2.4.1 Architektonische Grundsätze

Im Rahmen der Dissertation werden sechs grundsätzliche Bedingungen aufgestellt, welche im Zusammenhang mit der Skalierbarkeit des Webs stehen: das Client-Server-Modell, die zustandslose Kommunikation, das Caching, die uniforme Schnittstelle, das Schichtenmodell sowie der Ansatz des *Code-on-Demand* [21]. Hiermit konforme Schnittstellen (API, *Application Programming Interface*) werden als RESTful APIs bezeichnet. Nachfolgend werden die einzelnen Bedingung kurz näher erläutert.

Client-Server-Modell

Das Client-Server-Modell bildet mit der Separation von UI und Daten die Basis sowohl für die Simplifizierung des Servers als auch die plattformübergreifende Portabilität des UIs. Beide können aufgrund der Entkopplung unabhängig voneinander entwickelt werden, solange die Prinzipien der uniformen Schnittstelle eingehalten werden [21].

Zustandslose Kommunikation

Eine zustandslose Kommunikation wird durch Requests beschrieben, in welchen alle notwendigen und kontextbezogenen Informationen enthalten sind. Obwohl hierdurch auch redundante Informationen übertragen werden, wirkt sich dieser Ansatz insbesondere auf die Skalierbarkeit aus. Es wird jedoch auch die Zuverlässigkeit und Transparenz der Kommunikation optimiert [21, 24].

Caching

Das Prinzip des Cachings besagt, dass alle Daten einer Anwendung als *cacheable* oder *non-cacheable* deklariert werden müssen [21]. Das Cachen erlaubt, dass Daten für eine gewisse Dauer wiederverwendet werden können. Die Interaktion zwischen Client und Server wird somit signifikant reduziert und das System effizienter gestaltet [5].

Uniforme Schnittstelle

Die uniforme Schnittstelle ist bedeutender Bestandteil von REST und unterscheidet den Architekturstil von anderen. Das *Uniform Interface* ist dabei die Schnittstelle zwischen Client, Server und allen Intermediären. Für die konsequente Umsetzung einer einheitlichen Schnittstelle müssen weitere Prinzipien erfüllt werden: die eindeutige Identifikation von Ressourcen, die Verwendung von Repräsentationen, selbstbeschreibende Nachrichten und *Hypermedia as the Engine of Application State* (HATEOAS) [21].

Die weltweit eindeutige Identifikation von Ressourcen erfolgt über URI [5]. URI stellt eine standardisierte und weit verbreitete Syntax bereit [25]. Dabei kann jegliche Art von Information als Ressource angesehen werden, wie Dokumente, Binärdaten oder eine Sammlung von weiteren Ressourcen [26]. Alle Ressourcen einer Anwendung werden mit einem URI versehen, unabhängig davon um welchen Informationstyp es sich handelt [22].

Für den Client ist es in erster Linie irrelevant, in welcher Form die Ressource auf dem Server vorliegt, da dieser lediglich eine Repräsentation sendet. Die Repräsentation stellt den aktuellen Zustand einer Ressource maschinenlesbar [22] und für den Client sinnvoll [23] dar. Dabei können mehrere Repräsentationen für eine Ressource existieren, welche sich beispielsweise hinsichtlich des Datenformats unterscheiden [22].

Neben den eigentlichen Daten sind in Repräsentationen beschreibende Metadaten enthalten, welche unter anderem über das Datenformat und den Zeitpunkt der letzten Änderung Auskunft geben [21]. Diese Metadaten sind derweil meist im Header des HTTP-Responses enthalten.

Repräsentationen von Ressourcen lassen sich mittels *Hypermedia* auch anwendungsübergreifend verknüpfen. Neben dem Verknüpfen von Informationen ist es möglich, die Zustände von Anwendungen zu steuern, indem eine Applikation zum einen den Verlinkungen folgt und zum anderen die Repräsentationen Informationen über ihre weitere Verwendung beinhalten [5,24]; diese Beschreibungen werden auch als *Hypermedia Controls* bezeichnet [5].

Schichtenmodell

Durch den Einsatz der uniformen Schnittstelle und des Client-Server-Modells ist es möglich, beliebige Komponenten zwischen Client und Server zu schalten. Diese Komponenten sind meist für Aufgaben verantwortlich, welche diverse Netzwerke gemeinsam haben; hierzu zählen meist Funktionen für die Sicherheit oder das Caching [24].

Code-on-Demand

Der optionale Ansatz des *Code-on-Demand* erlaubt den Transfer von Code in Form eines Skriptes oder Plug-Ins auf den Client sowie die dortige Ausführung [24]. Der Funktionsumfang des Client lässt sich hierdurch erweitern, ohne dessen Komplexität bereits im Vorhinein zu erhöhen [21].

2.4.2 RESTful Web APIs

Im *World Wide Web* (WWW) setzen RESTful APIs meist das zustandslose Anwendungsprotokoll HTTP ein [5], welches speziell für die Übertragung von Repräsentationen im Web konzipiert wurde [21]. Das Protokoll definiert acht

Methoden, mit welchen jeweils unterschiedliche Operationen ausgeführt werden können [26]. In Tabelle 2.1 befinden sich vier der ursprünglichen HTTP-Methoden zusammen mit `PATCH`, welches nachträglich eingeführt wurde [27]. Die beiden Methoden `HEAD` und `OPTIONS` werden für das Erkunden einer API verwendet, da diese über die HTTP-Header der Repräsentationen und die für die Ressource verfügbaren Operationen Auskunft geben. Die verbleibenden `TRACE` und `CONNECT` werden dabei überwiegend von Proxies eingesetzt [22].

Tabelle 2.1: Grundlegende Methoden einer RESTful Web API [22]

HTTP-Verb	**CRUD**	**Beschreibung**
`GET`	Read	Erhalte eine Repräsentation der Ressource
`POST`	Create	Erstelle eine neue Ressource auf Basis der übergebenen Repräsentation
`PUT`	Update	Ersetze den Zustand einer Ressource mit der übergebenen Repräsentation
`PATCH`	Update	Editiere selektiv den Zustand einer Ressource basierend auf der übergebenen Repräsentation
`DELETE`	Delete	Lösche die Ressource

Die sich in Tabelle 2.1 befindlichen HTTP-Methoden besitzen Vorgaben und Charakteristika, welche sie jeweils von anderen unterscheiden. HTTP gibt vor, dass `GET` „sicher“ sein muss und bedeutet in diesem Kontext, dass mit dessen Hilfe lediglich Repräsentationen abgefragt werden dürfen. Nebeneffekte, wie Änderungen am Zustand einer Ressource, sind nicht gestattet. Weiterhin existieren die idempotente Methoden `GET`, `PUT` und `DELETE`. Diese zeichnen sich dadurch aus, dass lediglich beim ersten Aufruf mit einem URI Nebeneffekte auftreten dürfen. Eine wiederholte Anwendung darf zu keinen weiteren Nebeneffekten führen. Die beiden letzten Methoden `POST` und `PATCH` haben, gemäß ihrer Spezifikation, nicht frei von Nebeneffekten zu sein [5].

Den HTTP-Methoden aus Tabelle 2.1 lassen sich die fundamentalen Datenbankoperationen *create*, *read*, *update* und *delete* (CRUD) zuweisen [28]. Die Operationen fassen die von den HTTP-Methoden angestoßenen Prozesse zusammen.

Die Ressourcen und deren Repräsentationen werden in RESTful Web APIs oft in sogenannten *Collections* gruppiert. Diese *Collections* werden mit einem eigenen URI angesprochen; ein beispielhafter URI für eine Sammlung wäre `http://api.service.com/v1/items`. Einer Sammlung von Ressourcen stehen dabei in der Regel die HTTP-Methoden `GET` und `POST` zur Verfügung. Diese ermöglichen das Auflisten aller Elemente dieser Sammlung und das Hinzufügen

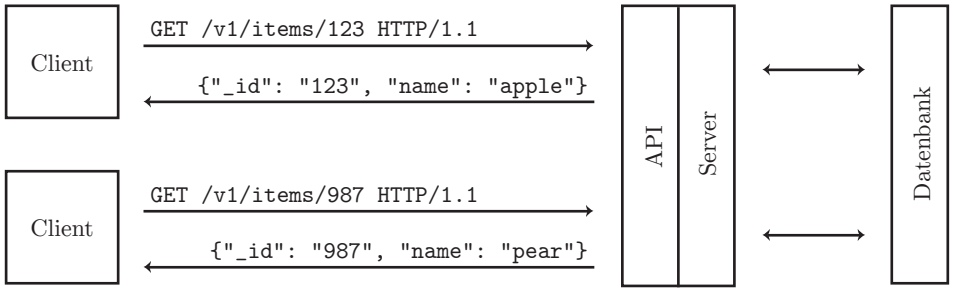

Abbildung 2.3: Kommunikation mit einer RESTful Web API

eines neuen Elementes. Weitere Operationen zum Löschen oder Ersetzen einer Sammlung sind möglich aber nicht üblich. Einzelne Ressourcen einer *Collection* werden dabei über einen untergeordneten URI angesprochen, welcher mit einer eindeutigen ID versehen ist: `http://api.service.com/v1/items/:id`. Hierfür stehen üblicherweise die HTTP-Methoden `GET`, `PUT`, `PATCH` und `DELETE` zur Wahl. Die beiden übrigen Methoden, welche für die Clients von Nutzen sind, `HEAD` und `OPTIONS`, sind für alle URIs verfügbar [22, 24].

In Web APIs wird zur Übermittlung von Repräsentationen oft auf *JavaScript Object Notation* (JSON) als *Internet Media Type* zurückgegriffen [24]. Dessen Eigenschaften und Vorzüge im Vergleich zu *Extensible Markup Language* (XML) werden in Kapitel 2.5 beschrieben.

In Abbildung 2.3 wird die beispielhafte Kommunikation mehrerer Clients mit einer RESTful Web API aufgezeigt. Hierbei wird jeweils ein HTTP-Request vom Typ `GET` an die Schnittstelle gesendet. Die URIs der Requests differieren lediglich hinsichtlich der ID; es werden die Repräsentationen zweier Ressourcen der *Collection* `items` angefragt. Die Abbildung verdeutlicht zudem das Zusammenspiel von Client, Server und Datenbank. Die Details zur Kommunikation mit der Datenbank können vernachlässigt werden, da für die Clients nur die Schnittstelle von wesentlicher Bedeutung ist.

2.4.3 Bewertung des Architekturstils

Representational State Transfer weist als Architekturstil einige Vorzüge auf, mit welchen teils Nachteile einhergehen. Anknüpfend werden die architektonischen Grundsätze aus Kapitel 2.4.1 bezüglich ihrer positiven und negativen Auswirkungen untersucht und bewertet. Zudem wird der Einsatz von RESTful APIs in Bezug auf AngularJS begründet.

Das Client-Server-Modell, die Separation von Zuständigkeiten und die einheitliche Schnittstelle ermöglichen einen modularen Aufbau von Systemen. Solche modularen Systeme sind einfach zu warten und wiederzuverwenden [5]; dabei resultiert die Wiederverwendbarkeit überwiegend aus der uniformen Schnittstelle. Des Weiteren wird durch die Verwendung gemeinsamer Standards eine Interoperabilität – sprich die Möglichkeit zur Kommunikation zwischen gänzlich unterschiedlich implementierten Systemen – impliziert [5]. Die Implementierung von RESTful Services kann somit plattformunabhängig erfolgen. Das Schichtenmodell, der Verzicht auf das serverseitige Speichern von Sitzungsinformationen und das Caching erlauben es, Performanz und Skalierbarkeit eines Netzwerkes zu erhöhen [5].

Die zuvor erwähnte Interoperabilität setzt gemeinsame Standards voraus. Eine Abweichung von diesen Standards innerhalb eines Systems führt zu einem Zusammenbruch dessen Kommunikation [24]. Das *Uniform Interface* hat ebenfalls zur Folge, dass Informationen in einem standardisierten Format übermittelt werden; eine ineffiziente Datenverarbeitung wird in Kauf genommen. Durch die zustandslose Kommunikation werden gewisse Daten oft redundant geladen, jene Redundanzen können sich negativ auf die Leistung des Netzwerkes auswirken. Diese Problematik wird unter anderem durch Caching zu Ungunsten der Zuverlässigkeit kompensiert; gecachete und aktuelle Daten können teils signifikant differieren. Zu guter Letzt sind Datenüberhänge und eventuelle Latenzen, welche mit dem Schichtenmodell einhergehen, zu nennen [21].

Die Vorzüge Modularität, Interoperabilität, Performanz und Skalierbarkeit stehen der reduzierten Zuverlässigkeit, der redundanten Kommunikation und den Beschränkungen durch Standards gegenüber. Da die Standards URI, HTTP und auch JSON im Web weit verbreitet sind, schränken diese ein System nicht schwerwiegend ein. Die redundante Kommunikation zwischen Client und Server kann durch entsprechend konfiguriertes Caching minimiert werden; ebenso die mit Caching einhergehenden Diskrepanzen, auf welche bereits in Kapitel 2.4.1 eingegangen wurde [5]. Die genannten Vorzüge überwiegen demnach die negativen Aspekte.

Insgesamt ist REST, als Architekturstil für eine Schnittstelle im Web, positiv zu bewerten. Auch für die Entwicklung von Single-page-Anwendungen ist eine serverseitige, REST-konforme Schnittstelle bestens geeignet, da auf AngularJS basierende Anwendungen über den integrierten HTTP-Client `$http` verfügen. Dieser stellt die für API-Zugriffe benötigten Funktionen bereit und vereinfacht die Handhabung im Fehlerfall. Die Verarbeitung von Responses wird zudem durch ähnliche Syntax von JavaScript und JSON erleichtert [3, 29].

2.5 JavaScript Object Notation

Für den Transfer von Repräsentationen werden, wie in Kapitel 2.4.2 bereits beschrieben, diverse Datenformate benötigt. *JavaScript Object Notation* (JSON) ist ein Datenaustauschformat, welches aus einer Untermenge von JavaScript entstand und sich an dessen *Object* Syntax orientiert [29]. Trotz der Nähe zu JavaScript, kann der Einsatz von JSON unabhängig von der Plattform oder Programmiersprache erfolgen; das Format bildet Daten anhand eines allgemeingültigen Konzeptes ab [29]. So ähnelt die Syntax nicht nur JavaScripts *Object*, sondern auch dem *Dictionary* aus Python [22].

Wie im Quellcode 2.5 zu sehen ist, sind die aus JavaScript bekannten Typen *String*, *Number*, *Array* und *Object* gestattet. Zusätzlich zu diesen, dürfen die booleschen Werte `true` und `false` sowie die leere Menge `null` in einem JSON-Objekt verwendet werden [30]. Die Gemeinsamkeiten in der Schreibweise ermöglichen eine simple Integration in JavaScript-basierten Anwendungen [24].

Quellcode 2.5: Beispielhaftes JSON-Objekt

```
{
  "name": "John Doe",
  "age": 42,
  "address": {
    "street": "123 Main St",
    "city": "Anytown",
    "country": "USA"
  },
  "jobs": [
    "carpenter",
    "cashier"
  ]
}
```

Im Vergleich zu XML gilt JSON als leichtgewichtig und übersichtlich [24]. Diese Eigenschaft begründet sich – wie in Quellcode 2.6 zu erkennen ist – dadurch, dass in XML *Tags* eingesetzt werden. Diese *Tags* umschließen weitere Daten und sind somit redundant; es entsteht ein Datenüberhang und das Dokument wird unübersichtlich. Trotz diverser Vorteile von XML gegenüber JSON, wie standardisierte Schemata und die Validierung von Dokumenten, findet es nur noch selten in JavaScript-basierten Anwendungen seinen Einsatz [5]. Es ist anzumerken, dass spezielle Objekte mit JSON nur über Umwege und mit Ein-

bußen serialisiert werden können [24]. JSON wird aufgrund der Lesbarkeit, der geringen Größe und der leichten Integration dennoch als Datenaustauschformat bevorzugt.

Quellcode 2.6: Beispielhaftes XML-Dokument

```
<?xml version="1.0" encoding="UTF-8"?>
<person>
   <name>John Doe</name>
   <age>42</age>
   <address>
      <street>123 Main St</street>
      <city>Anytown</city>
      <country>USA</country>
   </address>
   <jobs>
      <job>carpenter</job>
      <job>cashier</job>
   </jobs>
</person>
```

2.6 Zusammenfassung

In diesem Kapitel wurden die grundlegenden Techniken und Architekturstile, die im Rahmen der Konzeption und Implementierung des Prototyps zum Einsatz kommen, vorgestellt. Zu den untersuchten Technologien und Konzepten zählen Single-page-Anwendungen und das Framework AngularJS; die serverseitige Komponente *Node.js* und der *Node Package Manager*; REST und damit konforme Schnittstellen für die Client-Server-Kommunikation sowie JSON als Datenaustauschformat. Nach einer Betrachtung der jeweiligen Vor- und Nachteile, wurden diese im Kontext der SPAs analysiert und abschließend bewertet. Diese Bewertung ergab, dass die jeweiligen Vorteile die Nachteile überwiegen und aufgrund dessen die Technologien im Prototyp gerechtfertigt sind.

Kapitel 3

Analyse und Spezifikation der Systemanforderungen

Als Grundlage für die Recherche nach bestehenden Lösungen und die Konzeption des Prototyps müssen Prämissen und Anforderungen definiert werden. In diesem Kapitel werden die Grundvoraussetzungen des Systems dargelegt, die Anforderungen an die Funktionalität und Charakteristik der prototypischen Anwendung spezifiziert sowie priorisiert. Dabei bilden konkrete Projektanforderungen und Gespräche mit Frontend-Entwicklern der Agentur die Grundlage für die Anforderungsanalyse.

3.1 Prämissen des Systems

Bei der Konzeption und Entwicklung der angestrebten Autorisierungsebene sind Gegebenheiten zu berücksichtigen, welche sich auf vorhandene Features oder das zugrunde liegende Backend beziehen. Im Anschluss werden Annahmen, Abhängigkeiten und Einschränkungen ausgeführt und deren Bedeutung für den Prototyp erläutert.

In den Single-page-Anwendungen der Agentur werden die *Template Partials* – separate Template-Dateien für den wiederkehrenden Einsatz – bereits vorab in JavaScript konvertiert. Dies wird über serverseitige Skripte realisiert, welche meist von sogenannten *Task Runnern* verwaltet werden. Deren Eigenschaften und Funktionen werden in Kapitel 5.2.3 näher ausgeführt. Die vorab geladenen *Partials* werden im *Template Cache* des AngularJS-Anwendung gespeichert und unterbinden, dass diese während der Laufzeit über HTTP-Requests angefordert

werden müssen; die Kommunikation zwischen Client und Server lässt sich hierdurch reduzieren. Dies bedeutet für die Autorisierungsebene, dass die *Partials* nicht direkt vom Server an die Nutzerrechte angepasst werden können, da sich jene mit dem Initialisieren der Anwendung beim Client befinden.

Die Sichtbarkeit von UI-Elementen wird mittels einer rollenbasierten Autorisierung reguliert. Nutzerspezifische Anpassungen des UIs sind dabei nicht Teil der generischen Autorisierungsebene; jene werden unabhängig davon in der weiteren Geschäftslogik behandelt.

Das jeweilige serverseitige Backend – eine Anwendung, mit welcher sich Datenbestände und Nutzerrechte verwalten lassen – wird von einem CMS gebildet. Es ist davon auszugehen, dass das Backend dem Client lediglich sichere Daten liefert, welche den Rollen des jeweiligen Nutzers entsprechen. Die generische Lösung dient somit, wie in Kapitel 1.1 beschrieben, in erster Linie der Steigerung der UX und weniger der Systemsicherheit. Dennoch wird das Zusammenspiel aus Sicherheit und UX in der anschließenden Betrachtung der Funktionalität kurz thematisiert und in der Konzeption weiterführend untersucht. Als Schnittstelle zwischen Single-page-Anwendungen und Backend dient dabei in vielen Fällen eine RESTful API mit JSON als Datenaustauschformat. Der Aufbau der entsprechenden HTTP-Responses kann im Zuge des Prototyps angepasst und erweitert werden.

3.2 Anforderungsspezifikation

Im Zuge der Anforderungsspezifikation werden die Anforderungen hinsichtlich der Funktionalität, Charakteristika, Architektur und Qualität spezifiziert. Auf dieser Beschreibung bauen der Vergleich gegenwärtiger Lösungen und die spätere Konzeption des Prototyps auf.

3.2.1 Funktionale Anforderungen

Aus Sicht der Anwender sollte die generische Autorisierungsebene mindestens den aktuellen Funktionsumfang abdecken. Dieser umfasst das rollenbasierte Ein- und Ausblenden der UI-Elemente und wird technisch mittels *Cascading Style Sheets* (CSS) realisiert. Es ist somit möglich, den DOM-Baum einer Anwendung zu durchsuchen und sich mit nur geringen Eingriffen Zugang zu ausgeblendeten Elementen zu verschaffen. Auch wenn die Annahme besteht, dass vom Server nur sichere Daten gesendet werden, könnten die UI-Elemente auf weiterführende Inhalte hindeuten. Indem jene Elemente – sollten dem Nutzer

entsprechende Rollen fehlen – so weit wie möglich aus dem DOM-Baum entfernt werden, können die Hürden für Dritte und die Sicherheit der Anwendung erhöht werden. Beim Wechsel der Sichtbarkeit sollten zudem die Animationen des AngularJS-Moduls *ngAnimate* [31] unterstützt werden. Das Modul dient dazu, die Anpassungen im UI für den Nutzer angenehm zu gestalten, sollten sich die Rollen dynamisch ändern; obwohl es nicht in allen Projekten zwingend benötigt wird, wird die Möglichkeit dennoch bereit gestellt. Es ist der Standard für Animationen im AngularJS-Umfeld.

Eine Änderung der Rollen sollte zudem eine direkte, dynamische Modifikation der UIs zur Folge haben, ohne dass die Änderungen erst mit dem Aktualisieren der Seite oder dem Laden von *Template Partials* wirksam werden. Dadurch wird der weiteren Verfügbarkeit trotz fehlender Rollen vorgebeugt.

Falls die Seite oder *Template Partials* geladen werden, müssen entsprechende UI-Elemente ausgeblendet oder aus dem DOM entfernt werden, bevor diese für den Nutzer sichtbar sind. Diese Anforderung dient der UX, indem man einem Aufflackern von Texten oder grafischen Elementen entgegengewirkt. Darüber hinaus wird der Nutzer nicht über ihm verwehrte Inhalte informiert.

Neben der schlichten Angabe von erforderlichen Rollen kann mit Hilfe diverser Autorisierungsmodi die rollenbasierte Steuerung der UI-Elemente erleichtert werden. Die Entwickler können – durch die Auswahl an mehreren, gebräuchlichen Strategien – auf eigene Anwendungslogik verzichten und die eigenen Aufwände reduzieren. Dabei werden jeweils eine der folgenden Bedingungen geprüft und die UI-Elemente entsprechend gehandhabt:

- Ist dem Nutzer eine der übergebenen Rollen zugewiesen?
- Ist dem Nutzer keine der übergebenen Rollen zugewiesen?
- Sind dem Nutzer alle übergebenen Rollen zugewiesen?

3.2.2 Nicht-funktionale und architektonische Anforderungen

Die nicht-funktionalen Anforderungen definieren Ansprüche insbesondere an die Architektur der Implementierung und den Funktionsumfang als Ganzes. In Bezug auf den Funktionsumfang der Implementierung werden zudem diverse Designprinzipien eingeführt.

Die entscheidendsten Charakteristika eines generischen Ansatzes sind dessen Universalität und Modularität. Die Universalität definiert sich durch eine einfache Integration einer Lösung in bestehende Anwendungen, möglichst ohne

hierbei manuelle Eingriffe vornehmen zu müssen. Die Modularität einer Implementierung wird erreicht, indem die Abhängigkeiten zu weiteren Modulen reduziert und die Zuständigkeiten getrennt werden; durch die Reduktion von Abhängigkeiten wird außerdem die Universalität unterstützt.

Trotz des generischen Charakters ist es stellenweise notwendig, die Implementierung projektspezifisch anzupassen. Eine fakultative Konfiguration ermöglicht es Frontend-Entwicklern, an zentraler Stelle Anpassungen vornehmen zu können. Eingriffe in den Quellcode der Implementierung werden durch die Konfiguration vermieden.

Um den modularen Charakter zu stärken, soll die Lösung zudem nur die geforderte Funktionalität enthalten. Der auf das Wesentliche konzentrierte Funktionsumfang – gemäß der gebräuchlichen Designprinzipien *You Ain´t Gonna Need It*, *Keep It Simple Stupid* und *Don't Repeat Yourself* – hilft, die Implementierung überschaubar zu halten [32, 33].

3.2.3 Qualitative Anforderungen

Um eine hohe Qualität der Implementierung sicher zu stellen, werden mehrere Anforderungen definiert. Diese beziehen sich auf den Quellcode, dessen Dokumentation und Testabdeckung sowie zusätzliche Erläuterungen.

Der Quellcode sollte übersichtlich strukturiert sein, um die Lesbarkeit zu erhöhen und es Entwicklern einfacher zu machen, diesen nachzuvollziehen. Der Einsatz von *Best Practices* [8] und den obengenannten Prinzipien helfen dabei, wiederkehrende Probleme auf bewährte Weise zu lösen. Durch die Einhaltung dieser Richtlinien wird weiterhin die Wiederverwendbarkeit gesteigert.

Des Weiteren sollte der Quellcode mittels Kommentaren ausführlich dokumentiert sein, um diesen nachvollziehbar zu gestalten. Die Dokumentation von Schnittstellen und Objekten erfolgt anhand der Auszeichnungssprache *JSDoc* des gleichnamigen *API Documentation Generators* [34]. Diese sieht vor das zu beschreibende Objekt mit diversen Informationen zu versehen und jenes somit zu spezifizieren. Der Quellcode 3.1 zeigt die exemplarische Dokumentation des Konstruktors `Person`. Einzeilige Kommentare stehen für die Begründung der Existenz einzelner Anweisungen und nicht zu Erklärungszwecken zur Verfügung. Der Grund hierfür ist, dass der Quellcode selbsterklärend sein sollte.

Um nach Änderungen neue Fehlerfälle zeitsparend identifizieren und beheben zu können, werden Modultests benötigt. Diese Modultests sollten die Implementierung weitestmöglich abdecken. Bei der Wahl der *Test Runner* und *Test Frameworks* sind jeweils weitverbreitete Anwendungen zu wählen.

Die Integration und Weiterentwicklung der Lösung wird simplifiziert, indem eine verständliche und prägnante Anleitung für die Einrichtung zur Verfügung gestellt wird. Eine zusätzliche Demo kann unter Verwendung der wichtigsten Funktionen den Umfang der Implementierung veranschaulichen, sodass sich der Nutzer nicht direkt mit dem Quellcode auseinandersetzen muss.

Quellcode 3.1: Beispielhafte Dokumentation mit JSDoc

```
/**
 * @constructor
 * @public
 *
 * @param {string} name First and last name
 * @param {number} age Age in years
 * @param {Object.<string>} address Address inclusive country
 * @param {Array.<string>} jobs List of jobs
 */
function Person(name, age, address, jobs) {}
```

3.3 Priorisierung der Anforderungen

Die in Kapitel 3.2 definierten Anforderungen werden zugunsten der Übersichtlichkeit zunächst kategorisiert und priorisiert (vgl. Tabelle 3.1). Die zeitlich begrenzten Ressourcen sind dabei ursächlich für die Priorisierung. Die Umsetzbarkeit der Anforderungen orientiert sich am jeweiligen Funktionsumfang und den damit verbundenen Aufwände. Manche optionalen Anforderungen ergänzen die obligatorischen und setzen diese somit voraus.

3.4 Zusammenfassung

In diesem Kapitel wurden die Prämissen des Systems aufgezeigt, die Anforderungen an das neue System erläutert sowie diese letztlich priorisiert. Bei der Betrachtung der Prämissen wurden sowohl architektonische als auch funktionale Gegebenheiten berücksichtigt. Die definierten Anforderungen umfassen unterschiedliche Aspekte der Funktionalität und Charakteristika des Systems. Abschließend wurden die Anforderungen priorisiert und aus Gründen der Übersichtlichkeit kategorisiert.

Tabelle 3.1: Systemanforderungen und deren Priorisierung

Priorität	**Anforderung**
	Rollenbasiertes Handling des UIs
Obligatorisch	Ein-/Ausblenden der Elemente
Optional	Manipulation des DOMs
Optional	Unterstützung von Animationen
Optional	Dynamische Modifikation
Optional	Frühzeitige Intervention
	Diverse Autorisierungsmodi
Obligatorisch	Ist dem Nutzer eine der übergebenen Rollen zugewiesen?
Optional	Ist dem Nutzer keine der übergebenen Rollen zugewiesen?
Optional	Sind dem Nutzer alle übergebenen Rollen zugewiesen?
	Allgemeingültigkeit & Modularität
Obligatorisch	Hohe Universalität der Lösung
Obligatorisch	Geringe Abhängigkeiten
Obligatorisch	Trennung der Zuständigkeiten
Obligatorisch	Einfache Integration in bestehende Systeme
Optional	Fakultative Konfiguration
Optional	Konzentrierter Funktionsumfang
	Einfache Wart- und Testbarkeit
Obligatorisch	Ausführliche Dokumentation
Obligatorisch	Strukturierter Code
Obligatorisch	Einhaltung von *Best Practices* und Designprinzipien
Optional	Verständliche Einrichtungsanleitung
Optional	Exemplarische Demo
Optional	Hohe Testabdeckung

Kapitel 4

Untersuchung bestehender Konzepte

Das Problem der rollenbasierten Handhabung von UI-Elementen wird bereits von anderen Lösungen und Ansätzen angegangen. Für eine aussagekräftige Betrachtung wird zuerst ein Versuchsaufbau beschrieben, nach welchem die jeweiligen Implementierungen untersucht werden. Im Kontext dieses Aufbaus werden Kriterien definiert, welche der Bewertung und dem abschließenden Vergleich dieser Lösungen dienen.

4.1 Untersuchungsaufbau

Der Untersuchungsaufbau beschreibt wie sich die Untersuchung der jeweiligen Lösungen strukturiert, um diese aussagekräftig vergleichen und bewerten zu können. Der Aufbau ist lediglich für einsatzfähige Implementierungen gültig und besteht aus drei Schritten.

Im ersten Schritt wird der Quellcode der Implementierung unter Anbetracht der nachfolgend definierten Kriterien analysiert. Anschließend wird, falls eine Demo bereitgestellt wird, diese ebenfalls untersucht und die Funktionalität getestet. Sollte keine Demo zur Verfügung stehen, wird ein sogenanntes *Minimal Working Example* mit den grundlegenden Funktionen implementiert. Durch das Testen der Demo beziehungsweise des *Minimal Working Examples* wird die Funktionsweise und Zuverlässigkeit der Implementierung ermittelt. In einem abschließenden Schritt werden die Ergebnisse der Analyse von Quellcode und Demo bewertet.

4.2 Untersuchungskriterien

Die selektierten Lösungen werden anhand von Kriterien untersucht und bewertet, denen die in Kapitel 3.2 definierten Anforderungen zugrunde liegen. Die Untersuchungskriterien werden benötigt, um die umfassenden und vielseitigen Lösungen nach einem einheitlichen Schema bewerten zu können.

Da das rollenbasierte Ein-/Ausblenden der UI-Elemente die Kernfunktionalität bildet, ist dieses das bedeutendste Untersuchungskriterium. In erster Linie ist dabei entscheidend, ob die UI-Elemente lediglich mittels CSS ausgeblendet oder diese vollständig aus dem DOM entfernt werden; das Entfernen aus dem DOM wird bevorzugt. Des Weiteren wird eine Unterstützung des AngularJS- Moduls *ngAnimate* positiv bewertet.

Das UI sollte nach dem Ändern der Rollen dynamisch modifiziert werden, ohne dass die Seite oder *Template Partials* neu geladen werden müssen. Um die UX zu erhöhen, sollte zudem die Sichtbarkeit der UI-Elemente reguliert werden, bevor diese für den Nutzer sichtbar sind. Dadurch wird vermieden, dass Elemente Aufflackern oder der Nutzer über ihm vorenthaltene Informationen in Kenntnis gesetzt wird.

Ein weiteres Untersuchungs- und Bewertungskriterium bildet die Auswahl an mehreren Autorisierungsmodi. Neben der Anzahl an verfügbaren Strategien ist auch deren Charakteristik und Funktionsweise zu bewerten. Im Idealfall werden die in Kapitel 3.2 beschriebenen Modi abgedeckt; diese prüfen, ob dem Nutzer eine, keine oder alle der übergebenen Rollen zugewiesen sind.

Die Universalität und Modularität einer Lösung wird mittels der Untersuchung von Teilaspekten bewertet. So sollte die Lösung möglichst wenig Abhängigkeiten besitzen und die Zuständigkeiten innerhalb der Implementierung strikt trennen. Es wird zwischen Abhängigkeiten für den produktiven Einsatz und die Entwicklung unterschieden; die sogenannten *Development Dependencies* fallen nicht ins Gewicht, da diese lediglich den Entwicklungsprozess vereinfachen. Die einfache Integration in Projekte ist ein weiterer Indikator für die Allgemeingültigkeit und eine hohe Modularität der Lösung. Eine optionale Konfiguration, welche projektspezifische Anpassungen ermöglicht, gilt als weiterer Benefit.

Die Implementierung sollte lediglich die gewünschte Funktionalität enthalten und zusätzliche Aspekte weitestmöglich vernachlässigen. Es wird dabei geprüft in welchem Umfang weitere Features implementiert sind. Darüber hinaus ist für die Untersuchung auch deren Zweck relevant. Auf die Bewertung wirken sich zusätzliche Funktionen negativ aus, insbesondere wenn diese keinen unmittelbaren Bezug zur Kernfunktionalität aufweisen.

Der Quellcode wird hinsichtlich seiner Struktur und Dokumentation untersucht und bewertet. Im Konkreten wird geprüft, inwieweit bei diesem aktuelle *Best Practices* und Designprinzipien eingehalten werden. Die Verfügbarkeit und Verständlichkeit einer Einrichtungsanleitung stellt ein weiteres Bewertungskriterium dar. Falls die Lösung mit Modultests versehen ist, wird deren prozentuale Quellcode-Abdeckung ermittelt; eine hundertprozentige Testabdeckung bildet das anzustrebende Optimum.

Tabelle 4.1: Untersuchungskriterien und deren Ausprägungen

Untersuchungskriterium	**Ausprägung**	**Optimum**[1]
Funktionale Kriterien		
Rollenbasiertes Handling des UIs	DOM / CSS	DOM
Unterstützung von Animationen	Ja / Nein	Ja
Dynamische Modifikation	Ja / Nein	Ja
Frühzeitige Intervention	Ja / Nein	Ja
Mehrere Autorisierungsmodi	Ja / Nein	Ja
Architektonische Kriterien		
Grad der Abhängigkeit	Gering / Mittel / Hoch	Gering
Trennung von Zuständigkeiten	Gering / Mittel / Hoch	Hoch
Einfache Systemintegration	Ja / Nein	Ja
Fakultative Konfiguration	Ja / Nein	Ja
Konzentrierter Funktionsumfang	Ja / Nein	Ja
Qualitative Kriterien		
Strukturierter Code	Ja / Nein	Ja
Ausführliche Dokumentation	Ja / Nein	Ja
Best Practices und Designprinzipien	Ja / Teils / Nein	Ja
Verständliche Einrichtungsanleitung	Ja / Nein	Ja
Testabdeckung	Prozentsatz / n. a.	100%

Für die jeweils abschließende Bewertung werden die Untersuchungsergebnisse übersichtlich in einer Tabelle zusammengefasst. Zu Gunsten eines besseren Überblicks werden die Kriterien entsprechend der Anforderungsspezifikation in Kapitel 3.2 kategorisiert. Diese Tabelle umfasst die zu untersuchenden Kriterien mit der entsprechenden Ausprägung. Die Tabelle 4.1 stellt die Vorlage dar, welche die jeweils möglichen Ausprägungen sowie den Optimalwert enthält.

[1] Die anzustrebende Ausprägung des Untersuchungskriteriums

4.3 Untersuchung selektierter Lösungen

Bei der Recherche nach Implementierungen, welche das gegebene Problem teils oder vollständig abdecken, waren lediglich zwei AngularJS-Module aufzufinden. Des Weiteren wurden Weblog-Beiträge identifiziert, die das Thema behandeln und Lösungsvorschläge skizzieren. Im Folgenden werden die Module ausführlich anhand des Untersuchungsaufbaus und der definierten Kriterien analysiert sowie die grundlegende Idee der Weblog-Beiträge kurz erörtert.

4.3.1 Lally Elias' ngAA

Bei *ngAA* handelt es sich um ein AngularJS-Modul des Software-Entwicklers Lally Elias. Dieses Modul unterliegt der MIT-Lizenz und kann unter anderem über die Plattform *GitHub* in der aktuellen Version bezogen werden [35].

Untersuchung

Die Analyse des Quellcodes und der bereitgestellten Demo hat ergeben, dass die UI-Elemente lediglich mit CSS ausgeblendet werden. Hierfür wird dem Element die Klasse `ng-hide` hinzugefügt, welche aufgrund der gleichnamigen AngularJS-Direktive zur Verfügung steht. Die Klasse ist standardmäßig mit `display: none !important` ausgestattet, entsprechende Elemente werden aus dem Elementfluss entfernt. Die Animationen des Moduls *ngAnimate* werden unterstützt. Der Entwickler greift für die Umsetzung der Animationen auf Teile der Direktiven `ng-hide` und `ng-show` zurück.

Die UI-Elemente lassen sich mit Hilfe von drei Direktiven ausblenden, die jeweils einen Autorisierungsmodus abbilden: `show-if-has-permit`, `show-if-has-permits` und `show-if-has-any-permits`. Die ersten beiden Strategien prüfen, ob dem Nutzer alle übergebenen Rollen zugewiesen sind. Sie unterscheiden sich dadurch, dass `show-if-has-permit` nur eine Rolle übergeben werden kann und `show-if-has-permits` eine Liste auswertet. Diese sind somit hinsichtlich ihrer Prüfung redundant. Die dritte Strategie nimmt ebenfalls eine Liste entgegen, prüft jedoch nur, ob eine dieser Rollen zugewiesen ist.

Keiner der Modi ist mit einem *Event Listener* versehen, sodass Änderungen bezüglich der Rollen erst mit dem anschließenden Laden von Templates oder dem Aktualisieren der Seite wirksam werden. Zudem werden alle UI-Elemente nach dem Laden einer Seite zunächst angezeigt und erst im Anschluss rollenbasiert ausgeblendet. Beide Umstände resultieren in Einbußen hinsichtlich der UX und sind zu vermeiden.

Das Modul besitzt neben *ngAnimate* noch weitere Abhängigkeiten: *JSON Web Token* (JWT) [36], *AngularUI Router* [37] und *ngStorage* [38]. Hierbei handelt es sich um Module, deren Funktionalität nicht im direkten Bezug zum Handling der UI-Elemente steht, sondern die vielmehr die Anwendung hinsichtlich Authentifizierung und Autorisierung erweitern. Darüber hinaus bestehen auch *Development Dependencies*, welche für die Automatisierung von Aufgaben, das Testen und die Paketverwaltung verantwortlich sind; hierzu zählen unter anderem npm [18], *Bower* [39], *Grunt* [40], *Karma* [41], *Mocha* [42] und *Chai* [43]. Die einzelnen Komponenten werden in Kapitel 5 ausführlich betrachtet.

Die Implementierung trennt die Zuständigkeiten voneinander, indem für diese eigene Komponenten verwendet werden. Durch die strikte Trennung ist den unterschiedlichen Dateien ein einzelner Zweck zugewiesen. Die Komponenten decken Funktionen hinsichtlich der JWT, der Authentifizierung und der DOM-Manipulation ab. Wiederholt benötigte Funktionen sind in einer weiteren Datei ausgelagert. Die Authentifizierung lässt sich als einzige Komponente projektspezifisch konfigurieren.

Der Funktionsumfang des Moduls umfasst drei zusätzliche Aspekte: Authentifizierung, Autorisierung und *Route Interception*. Letzteres überträgt dabei den rollenbasierten Zugriff auf die Routen, also die Seiten oder URIs einer Anwendung. Das Modul *ngAA* bildet den vollumfänglichen Authentifizierungs- und Autorisierungsprozess samt dem Speichern von Sitzungsinformationen ab. Der Funktionsumfang ist klar definiert, jedoch beinhaltet dieser Funktionen, die in diesem Kontext nicht benötigt werden.

Um die Einrichtung zu vereinfachen und den Funktionsumfang zu verdeutlichen, wird sowohl eine ausführliche Anleitung als auch eine Demo bereitgestellt. Die Anleitung deckt die Installation, die Einrichtung, alle öffentlich verfügbaren Funktionen sowie die Konfigurationsmöglichkeiten ab. In der Demo sind lediglich einige der Funktionen enthalten, wie die Authentifizierung und das Handling der Routen. Dennoch stellt jene einen guten Überblick dar.

Der Quellcode selbst ist dagegen nur kaum dokumentiert. Bei der Dokumentation bleiben insbesondere Funktionen und weitere komplexe Komponenten weitestgehend unberücksichtigt. Die Kommentare beschränken sich dabei auf die Beschreibung einzelner Anweisungen. Aufgrund der mangelnden und heterogenen Schnittstellen-Dokumentation ist der Quellcode an manchen Stellen nicht vollkommen verständlich.

Trotz der Defizite hinsichtlich der Dokumentation, lässt sich der Quellcode meist nachvollziehen, da dieser übersichtlich strukturiert ist. Das DRY Prinzip wird dabei erfolgreich angewandt, indem redundante Ausdrücke ausgelagert werden. Zudem werden die Namensräume einzelner Komponenten mittels *Immediately-*

Invoked Function Expression (IIFE) getrennt. Jedoch wird bei der Nomenklatur gegen AngularJS-Richtlinien und *Best Practices* [44] verstoßen. Zum einen ist das Präfix `$` für AngularJS selbst reserviert [45], wurde jedoch wiederholt für eigenen Variablen verwendet. Zum anderen werden Direktiven nicht mit einem eigenen Präfix versehen. Beide Richtlinien schützen vor Kollisionen mit künftigen HTML-Standards oder anderen Modulen.

Für die Anwendung sind auf *Mocha* und *Chai* basierende Modultests verfügbar. Diese werden von *Karma*, einem sogenannten *Test Runner*, verwaltet und ausgeführt. Die definierten Modultests decken mit 86,76 Prozent die meisten Teile der Implementierung ab.

Bewertung

Tabelle 4.2: Lally Elias' *ngAA* – Untersuchungsergebnisse

Untersuchungskriterium	**Ausprägung**
Funktionalität	
Rollenbasiertes Handling des UIs	CSS
Unterstützung von Animationen	Ja
Dynamische Modifikation	Nein
Frühzeitige Intervention	Nein
Mehrere Autorisierungsmodi	Ja
Architektur	
Grad der Abhängigkeit	Hoch
Trennung von Zuständigkeiten	Hoch
Einfache Systemintegration	Ja
Fakultative Konfiguration	Ja
Konzentrierter Funktionsumfang	Nein
Qualität	
Strukturierter Code	Ja
Ausführliche Dokumentation	Nein
Best Practices und Designprinzipien	Teils
Verständliche Einrichtungsanleitung	Ja
Testabdeckung	86,76%

Obwohl das Modul *ngAA* einige Vorzüge – wie die Auswahl an mehreren Autorisierungsmodi, die strikte Trennung von Zuständigkeiten, die verständliche Einrichtungsanleitung sowie die hohe Testabdeckung – mit sich bringt, fällt die Gesamtbewertung negativ aus.

Für das negative Urteil sind insbesondere Kriterien hinsichtlich der Kernfunktionalität und der Universalität verantwortlich: die Regulation mit CSS, der große Funktionsumfang, die vielen Abhängigkeiten und die spärliche Schnittstellendokumentation. Zudem führen die Defizite bei der dynamischen Modifikation und der frühzeitigen Intervention zu einer Minderung der UX, obwohl diese eigentlich positiv beeinflusst werden sollte. Aus genannten Gründen stellt das Modul einen Ansatz, aber keine Lösung für das Problem dar.

4.3.2 maybenulls angular-authz

Das AngularJS-Modul *angular-authz* ist vom *Apache Shiro Security Framework* [46] inspiriert. Es unterliegt der MIT-Lizenz und kann unter anderem über die Plattform *GitHub* in der aktuellen Version bezogen werden [47].

Untersuchung

Um die Funktionsweise des Moduls besser nachvollziehen zu können, wurde zunächst ein *Minimal Working Example* implementiert. Im Anschluss daran wurde die Demo zusammen mit dem Quellcode analysiert. Aus dieser Untersuchung geht hervor, dass entsprechende UI-Elemente nach ihrer Kompilierung aus dem DOM entfernt werden. Das Modul *ngAnimate* und dessen Animationen werden von *angular-authz* nicht unterstützt.

Die UI-Elemente lassen sich mit Hilfe von zwei Autorisierungsmodi aus dem DOM entfernen: `has-permission` und `has-not-permission`. Die Direktiven akzeptieren jeweils eine Rolle in Form einer Zeichenkette, welche auch das Resultat einer *Expression* sein können. Beide Modi sind nahezu identisch, sodass `has-not-permission` die übrige Funktion lediglich negiert.

In keinem der Modi werden *Event Listener* eingesetzt, sodass das Ändern von Rollen erst mit dem Laden von Templates oder dem Aktualisieren der Seite wirksam wird. Das Ein-/Ausblenden der UI-Elemente erfolgt nachdem diese dem Nutzer angezeigt werden; dies ist auf die Konfiguration und Funktionsweise der Direktiven zurückzuführen. Die frühzeitige Intervention und die Defizite hinsichtlich der dynamischen Modifikation reduzieren die UX.

Der Funktionsumfang des Moduls umfasst kaum weitere Aspekte. Hierzu zählt zum einen die Möglichkeit sogenannte *Wildcards* für die Angabe der Rollen zu verwenden. Mit Hilfe der *Wildcards* können über Muster ganze Zweige von Rollen-Hierarchien zugewiesen. Zum anderen können eigene *Resolver* kreiert und verwendet werden, mit denen sich eigene Muster auflösen lassen. Der Funktionsumfang ist somit eindeutig definiert und kompakt gehalten. Die zusätzlich implementierte Funktionalität ist für die Lösung des gegebenen Problems nicht nötig, aber hilfreich.

Abgesehen von *Development Dependencies* besitzt die Lösung keine weiteren Abhängigkeiten. Zum Testen werden *Karma* und *Jasmine* [48] verwendet. Zudem kommen *Bower*, npm und *Grunt* zur Paket- und Aufgabenverwaltung zum Einsatz. Die geringen Abhängigkeiten sind dabei positiv zu bewerten.

Die Zuständigkeiten des Moduls – die Autorisierung, das Auflösen von Rollenangaben und die DOM-Manipulation – werden voneinander getrennt und den entsprechenden Komponenten zugewiesen. Die Autorisierung lässt sich umfangreich konfigurieren: es können eigene *Resolver* und die Rollen des Nutzers zugewiesen werden. Die Konfiguration des Moduls kann vor, während und nach der Initialisierung erfolgen.

Eine ausführliche Dokumentation erleichtert die Einrichtung, Konfiguration und Verwendung des Moduls. Insbesondere die unterschiedlichen Anwendungsmöglichkeiten werden anschaulich ausgeführt. Auch wenn keine Demo bereit gestellt wird, ist die Implementierung eines *Minimal Working Example* mittels der Anleitung einfach umzusetzen.

Der Quellcode ist frei von dokumentierenden Kommentaren. Durch den Verzicht auf Kommentare sind sowohl Anweisungen als auch die Schnittstellen und Logik einzelner Funktionen schwer nachzuvollziehen.

In der Implementierung werden sich wiederholende Anweisungen in Funktionen ausgelagert. Unter anderem resultiert dies in einem gut strukturierten Quellcode und der Einhaltung des DRY Prinzips. Zudem werden die AngularJS-Nomenklatur-Richtlinien berücksichtigt. Nichtsdestotrotz wird gegen mehrere *Best Practices* verstoßen: auf IIFEs wird verzichtet, es werden keine modulspezifischen Präfixe verwendet und nicht alle Variablen werden zu Beginn eines Gültigkeitsbereiches initialisiert [49].

Das Modul ist mit Tests des Frameworks *Jasmine* versehen, welche von *Karma* verwaltet und ausgeführt werden. Der *Test Runner* ist fehlerhaft konfiguriert, sodass die Tests nicht ausführbar sind. Eigene Korrekturen und ein anschließender Testlauf ergaben, dass 98,86 Prozent der Implementierung mit Tests abgedeckt sind; lediglich eine Anweisung wurde nicht getestet.

Bewertung

Zusammengefasst erfüllt das Modul *angular-authz* die meisten Untersuchungskriterien. Es sind insbesondere die geringen Abhängigkeiten, das vollständige Entfernen aus dem DOM, die umfangreiche Einrichtungsanleitung sowie die hohe Testabdeckung hervorzuheben. Dem gegenüber steht die fehlende Dokumentation des Quellcodes und die inkorrekte Test-Konfiguration. Hinzu kommt, dass nur jeweils eine Rolle beziehungsweise *Wildcard* angegeben werden kann. Aufgrund dessen stellt das Modul einen sehr guten Ansatz dar, jedoch wird das Problem nicht vollständig gelöst.

Tabelle 4.3: maybenulls *angular-authz* – Untersuchungsergebnisse

Untersuchungskriterium	**Ausprägung**
Funktionalität	
Rollenbasiertes Handling des UIs	DOM
Unterstützung von Animationen	Nein
Dynamische Modifikation	Nein
Frühzeitige Intervention	Nein
Mehrere Autorisierungsmodi	Ja
Architektur	
Grad der Abhängigkeit	Gering
Trennung von Zuständigkeiten	Hoch
Einfache Systemintegration	Ja
Fakultative Konfiguration	Ja
Konzentrierter Funktionsumfang	Ja
Qualität	
Strukturierter Code	Ja
Ausführliche Dokumentation	Nein
Best Practices und Designprinzipien	Teils
Verständliche Einrichtungsanleitung	Ja
Testabdeckung	98,86%

4.3.3 Lösungsansätze aus Weblogs

Nachfolgend werden Ideen und Ansätze aus Weblog-Beiträgen untersucht, die das Thema der rollenbasierten Handhabung von UI-Elementen behandeln. Die spezifizierten Untersuchungskriterien werden an dieser Stelle nicht angewandt, da die Beiträge lediglich eine Idee skizzieren.

Jon Samwell

In Jon Samwells Weblog-Beitrag [50] werden UI-Elemente, für welche der Nutzer nicht autorisiert ist, mit der CSS-Klasse `hidden` ausgeblendet. Für den Zugriff steht lediglich eine Direktive zur Verfügung, in welcher geprüft wird, ob dem Nutzer mindestens eine der übergebenen Rollen zugewiesen ist. Die CSS-Klasse ist unter anderem Bestandteil des CSS-Frameworks *Bootstrap* [51]. Zudem bedarf es dem AngularJS-Modul *ngRoute*. Diese Abhängigkeit impliziert einen erweiterten Funktionsumfang, der *Route Security* sowie *Interception* umfasst. Aufgrund der genannten Aspekte stellt Jon Samwells Idee keinen anwendbaren Lösungsansatz für das gegebene Problem dar.

Nadeem Khedr

Der Web-Entwickler Nadeem Khedr beschreibt in seinem Weblog-Beitrag [52] einen Ansatz, bei dem auf die JavaScript-Bibliothek *jQuery* [53] zurückgegriffen wird. Die entsprechende Direktive blendet die UI-Elemente mittels der *jQuery*-Funktion `hide()` aus und akzeptiert zwei Typen von Argumenten in Form einer Liste: normale und negierte Rollen. Damit können einzelne Rollen explizit ausgeschlossen werden. Die Direktive selbst prüft intern, ob die Autorisierung gegen mindestens eine der übergebenen Rollen erfolgreich verläuft. Als einzige Abhängigkeiten sind *jQuery* und *ngRoute* zu nennen, wobei das AngularJS-Modul für das rollenbasierte Handling der Routen benötigt wird. Aus genannten Gründen erfüllt Nadeem Khedrs Idee nicht die wesentlichen Aspekte der gesuchten Lösung.

Gert Hengeveld

Im Weblog-Beitrag des Softwareentwicklers Gert Hengeveld [54] wird auf Direktiven zurückgegriffen, welche bereits in AngularJS enthalten sind: `ng-switch` und `ng-if`. Diese entfernen UI-Elemente, abhängig von der übergebenen Bedingung, aus dem DOM. Da es sich bei diesen Bedingungen auch um Rückgabe-

werte einer Funktion handeln kann, kann die Prüfung flexibel gestaltet werden. Für das Handling von Routen kann wahlweise das Modul *ngRoute* oder *ui-router* eingesetzt werden. Die Flexibilität der Bedingungen geht mit dem Verzicht auf eine semantische Nomenklatur einher, sodass der Bezug zur rollenbasierten Autorisierung und eine klare Trennung der Zuständigkeiten nicht klar zu erkennen ist; der Lösungsansatz ist für das gesuchte Problem unvorteilhaft.

4.4 Fazit der Untersuchung

Die Recherche nach vorhanden Lösungsansätzen hat ergeben, dass kaum Lösungen existieren, welche versuchen das gegebene Problem zu bewältigen. Während sich einige mit dem Thema in Form von Weblog-Beiträgen befassen, sind lediglich zwei als vollständige Implementierung öffentlich verfügbar.

Tabelle 4.4: Gegenüberstellung von *ngAA* und *angular-authz*

Untersuchungskriterium	**ngAA**	**angular-authz**
Funktionalität		
Rollenbasiertes Handling des UIs	CSS	DOM
Unterstützung von Animationen	Ja	Nein
Dynamische Modifikation	Nein	Nein
Frühzeitige Intervention	Nein	Nein
Mehrere Autorisierungsmodi	Ja	Ja
Architektur		
Grad der Abhängigkeit	Hoch	Gering
Trennung von Zuständigkeiten	Hoch	Hoch
Einfache Systemintegration	Ja	Ja
Fakultative Konfiguration	Ja	Ja
Konzentrierter Funktionsumfang	Nein	Ja
Qualität		
Strukturierter Code	Ja	Ja
Ausführliche Dokumentation	Nein	Nein
Best Practices und Designprinzipien	Teils	Teils
Verständliche Einrichtungsanleitung	Ja	Ja
Testabdeckung	86,76%	98,86%

Die Tabelle 4.4 stellt zunächst die beiden AngularJS-Module *ngAA* und *angular-authz* gegenüber. Hierdurch wird ersichtlich, dass die Unterschiede größtenteils bei der Funktionalität zu erkennen sind, da jeweils unterschiedliche Strategien verfolgt werden. In Bezug auf die Qualität differieren die beiden Module kaum. Der strukturierte Code, die verständliche Einrichtungsanleitung und die hohe Testabdeckung sind in beiden Fällen hervorzuheben.

Alle vorgestellten Lösungen haben den Einsatz von Direktiven zur Bewältigung des grundlegenden Problems gemeinsam. Darüber hinaus werden in den meisten Implementierungen und Ideen weiterhin die Routen und der Zugriff auf diese behandelt. Einzig *angular-authz* sticht bei der Analyse deutlich hervor. Dies beruht auf dem Entfernen aus dem DOM, dem fokussierten Funktionsumfang, den geringen Abhängigkeiten und der sehr hohen Testabdeckung.

Obwohl *angular-authz* viele der in Kapitel 3.2 definierten Anforderungen erfüllt, findet es keinen Einsatz in den Projekten der Agentur. Dies begründet sich zum einen durch Defizite hinsichtlich der Universalität, welche auf die fehlende Unterstützung von Animationen und die nur einseitig einsetzbaren Autorisierungsmodi zurückzuführen sind. Zum anderen mindert der nicht dokumentierte Quellcode und der Verstoß gegen *Best Practices* die Qualität und Wartbarkeit der Implementierung.

Die Analyse zeigt, dass die Implementierung einer generischen Lösung notwendig und gerechtfertigt ist, da keine der beschriebenen Lösungen alle Anforderungen abdeckt. Weiterhin wurde ersichtlich, dass die definierten Anforderungen nicht durch Features aus bestehenden Ansätzen ergänzt werden müssen und jene somit als vollständig zu betrachten sind.

4.5 Zusammenfassung

In diesem Kapitel wurden die beiden AngularJS-Module *ngAA* und *angular-authz* sowie diverse Weblog-Beiträge bezüglich des gegebenen Problems analysiert und bewertet. Dafür wurden zunächst Untersuchungsaufbau und -kriterien definiert, welche die Basis der jeweiligen Untersuchungen bilden. Abschließend wurden die Implementierungen gegenübergestellt, deren Gemeinsamkeiten und Unterschiede herausgearbeitet sowie die Notwendigkeit einer generischen Autorisierungsebene aufgezeigt.

Kapitel 5

Konzeption eines Prototyps

Die technische Konzeption des Prototyps dient dazu exakt zu definieren, wie bei der Umsetzung der spezifizierten Anforderungen vorzugehen ist. Das nachfolgende Konzept beinhaltet eine Betrachtung möglicher Lösungsansätze, den Aufbau der Entwicklungsumgebung, die Spezifizierung der benötigten Funktionen und Datenstrukturen, die Architektur der Implementierung sowie Aspekte der Integration und Qualitätssicherung.

5.1 Lösungsansätze

Für die Lösung des gegebenen Problems bestehen unterschiedliche Ansätze, die teilweise grundlegend differieren. Zunächst werden die einzelnen Phasen von AngularJS' Lebenszyklus erläutert, deren Verständnis für die nachfolgende Untersuchung benötigt wird. Daraufhin werden server- und clientseitige Verfahren betrachtet und deren prägnantesten Eigenschaften kurz gegenübergestellt. Abschließend werden die Verfahren bewertet und ein für den Prototyp geeigneter Lösungsansatz ausgewählt.

5.1.1 AngularJS' Lebenszyklus

Der Lebenszyklus einer AngularJS-Anwendung im Browser umfasst drei Phasen: das Initialisieren, das Kompilieren und die Programmlaufzeit. Diesen liegt die Funktionsweise von Direktiven zugrunde, welche die Geschäftslogik mit den dazugehörigen UI-Elementen verknüpft [55]. Die Bewertung der diversen Lösungsansätze setzt ein gewisses Verständnis für die einzelnen Phasen voraus, weshalb diese nachfolgend erläutert werden (vgl. Abbildung 5.1).

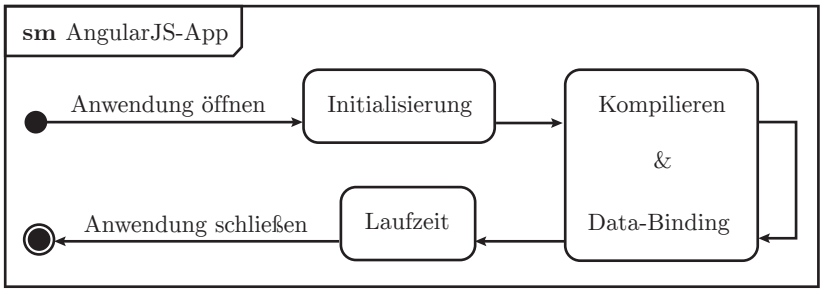

Abbildung 5.1: Zustandsdiagramm des AngularJS-Lebenszyklus

Phase des Initialisierens

Nachdem die Anwendung im Browser geladen wurde, befindet sie sich zunächst in der Initialisierungsphase, in welcher alle Assets inklusive der AngularJS-Datei heruntergeladen werden. Das Framework initialisiert die eigenen Komponenten, lädt das entsprechende Modul und injiziert alle Abhängigkeiten. Zu diesem Zeitpunkt sind alle Funktionen verfügbar, jedoch besteht die Seite vorerst aus statischem HTML [55]. Der Quellcode 5.1 zeigt einen Ausschnitt des statischen HTMLs; alle Direktiven und *Expressions* sind bis dahin ungenutzt.

Quellcode 5.1: Statisches HTML nach dem Initialisieren

```
1 <ul>
2   <li ng-repeat="task in tasks">
3     <span ng-bind="task.title"></span>
4   </li>
5 </ul>
```

Phase des Kompilierens

Im Anschluss an das Initialisieren findet das Kompilieren der Templates statt, bei welchem statisches durch dynamisches HTML ersetzt wird. Dafür wird zuerst das statische HTML durchsucht, alle Direktiven gesammelt und mit der jeweiligen Funktionalität verknüpft [55]. Die Direktiven bestehen im Wesentlichen aus den drei Funktionen `compile`, `pre-link` und `post-link` [56], die nacheinander ausgeführt werden. Für das Verständnis des Lebenszyklus' wird die `pre-link` Funktion nicht weiter benötigt.

Als erste der drei Funktionen wird `compile` aufgerufen, welcher noch keine Daten zur Verfügung stehen. Die Funktion wird verwendet, um Templates zu modifizieren, bevor einzelne Instanzen auf Grundlage dieses Templates generiert werden. Ein Template entspricht im Quellcode 5.1 dem `li`-Element samt seiner Kind-Elemente. Der `compile` Vorgang wird nur einmalig pro Template ausgeführt und hat den Vorteil, dass Templates ohne großen Aufwand modifiziert werden können. Die Template-Modifikationen werden durch das Klonen zum Erzeugen der einzelnen Instanzen automatisch übernommen. Dies erhöht – insbesondere bei der Ausgabe von Tabellen oder Listen – die Performanz des Kompilierungprozesses. Abschließend wird der kompilierte DOM-Teilbaum zurückgegeben, welcher von der `link` Funktion weiterverarbeitet wird [56].

Die `link` Funktion verknüpft die vollständig kompilierten Templates mit den Datenbeständen der Anwendung. Durch das Verknüpfen mit den Daten (vgl. Quellcode 5.2) entsteht letzten Endes dynamisches HTML, welches im Browser ausgegeben wird [56]; der Quellcode 5.3 stellt das Resultat dar.

Quellcode 5.2: Exemplarischer Datenbestand

```
[
  {
    title: "First Task",
    roles: "admin"
  },
  {
    title: "Second Task",
    roles: "user"
  }
]
```

Quellcode 5.3: Resultat des Kompilierungsprozesses

```
<ul>
  <!-- ngRepeat: task in tasks -->
  <li ng-repeat="task in tasks" class="ng-scope">
    <span ng-bind="task.title" class="ng-binding">First Task</span>
  </li>
  <!-- end ngRepeat: task in tasks -->
  <li ng-repeat="task in tasks" class="ng-scope">
    <span ng-bind="task.title" class="ng-binding">Second Task</span>
  </li>
  <!-- end ngRepeat: task in tasks -->
</ul>
```

Phase der Programmlaufzeit

Während der Programmlaufzeit bleiben die Daten mittels bidirektionalem Data-Binding an die UI-Elemente gebunden; hierdurch werden Änderungen zwischen *ViewModel* und *Model* synchronisiert. Die Phase bleibt bestehen bis die Anwendung durch Aktualisieren oder Verlassen der Seite beendet wird [55].

5.1.2 Untersuchung der Lösungsansätze

Clientseitige Verfahren

Da alle Projekte das Framework AngularJS gemeinsam haben, bietet sich eine clientseitige Umsetzung des Prototyps auf dessen Basis an. AngularJS stellt für die Manipulation des DOMs Direktiven zur Verfügung, welche es erlauben, wie in Kapitel 5.1.1 beschrieben, während oder nach dem Kompilieren in die Templates einzugreifen [56]. Die beiden Möglichkeiten stellen die wesentlichen Ansätze auf Client-Seite dar, bei welchen eine hundertprozentige Sicherheit und Integrität nicht gewährleistet werden kann, da sich Daten im Client verhältnismäßig leicht manipulieren lassen. Jedoch kann auf serverseitige Technologien verzichtet, die Abhängigkeiten reduziert und letzten Endes die Wiederverwendbarkeit der Lösung erhöht werden.

a) Die Manipulation während des Kompilierens der Templates wird nur einmalig pro Definition ausgeführt, weswegen die Autorisierung lediglich für das gesamte Template und nicht für einzelne Instanzen erfolgen kann. Zudem sind zu diesem Zeitpunkt noch keine Daten verfügbar, sodass eine instanzspezifische Autorisierung nicht möglich ist. Der Quellcode 5.4 verdeutlicht den Einsatz der Direktive `my-drv` zur Manipulation während dem Kompilieren. Dabei kann lediglich eine definierte Zeichenkette übergeben werden. Die mit der Direktive verknüpften Bedingungsprüfung wird vor dem finalen Kompilieren durch `ng-repeat` ausgeführt, wodurch sich diese auf jedes `li`-Element der Liste gleich auswirkt.

Quellcode 5.4: Einsatz bei der Manipulation während dem Kompilieren

```
1 <ul>
2   <li ng-repeat="task in tasks" my-drv="admin">
3     <span ng-bind="task.title"></span>
4   </li>
5 </ul>
```

Quellcode 5.5: Einsatz bei der Manipulation nach dem Kompilieren

```
<ul>
  <li ng-repeat="task in tasks" my-drv="{{ task.roles }}">
    <span ng-bind="task.title"></span>
  </li>
</ul>
```

b) In Quellcode 5.5 dagegen kann auch eine *Expression* an die Direktive übergeben werden, welche auf nun vorhandene Daten zugreift. Die Bedingungsprüfung wird nach der Kompilierung durch `ng-repeat` für jedes `li`-Element separat ausgeführt. Der Vergleich macht deutlich, dass die Manipulation nach dem Kompilieren der Templates wesentlich flexibler und vielseitiger gestaltet werden kann.

Da viele der eingesetzten CMS über eine RESTful API verfügen, kann diese in beiden Verfahren integriert werden. So können – nach der Übergabe eines entsprechenden URI – die erteilten Berechtigungen selbstständig angefordert, extrahiert und weiterverwendet werden. Alternativ können die Berechtigungen dem clientseitigen Modul über eine eigene Schnittstelle manuell zugewiesen werden. Die erste Möglichkeit setzt eine Konfiguration voraus, da der URI projektspezifisch angepasst werden muss. Für einen vielseitigen Einsatz müssen die Berechtigungen in beiden Fällen jederzeit zuweisbar sein, sprich während und nach dem Initialisieren der AngularJS-Anwendung.

Serverseitige Verfahren

Um die UI-Elemente rollenbasiert zu handhaben, ist der Einsatz einer serverseitigen Implementierung denkbar. Durch Bereitstellen einer Schnittstelle, können die Templates und eine eindeutige Session-ID vom Client entgegengenommen werden. Wie bei der Betrachtung der clientseitigen Ansätze deutlich wurde, müssen die Templates bereits fertig kompiliert und mit Daten versehen sein. Mit Hilfe der Session-ID ist es möglich, die dem Nutzer zugewiesenen Berechtigungen direkt vom zugrunde liegenden CMS zu beziehen. Die UI-Elemente, für die der Nutzer nicht autorisiert ist, können anschließend mit einem HTML/XML-Parser aus dem Template entfernt werden, bevor dieses dem Client übermittelt wird. Die Abbildung 5.2 verdeutlicht das Vorgehen.

Die Verwendung einer serverseitigen Implementierung kommt der Sicherheit und Integrität der Lösung zugute. Grund dafür ist, dass sich eine unbefugte, clientseitige Manipulation der Rollen nicht direkt auf die Logik auswirkt. Die

Client-Server- und Server-Server-Kommunikation, welche bei jedem zu ladenden Template notwendig werden, beeinflusst die Performanz nachteilig. Um auf die Server-Server-Kommunikation zu verzichten, können die Berechtigungen direkt vom Client übertragen werden, jedoch wären dadurch die Vorzüge hinsichtlich Sicherheit und Integrität hinfällig (vgl. Abbildung 5.3).

Abgesehen von den bereits genannten Gründen gegen den Einsatz einer serverseitigen Implementierung, muss der entsprechende Server unter Umständen bisher nicht benötigte Technologien unterstützen. Dadurch werden gegebenenfalls die Abhängigkeiten erhöht und infolge dessen wird der generische Gedanke nicht konsequent umgesetzt. Darüber hinaus wird, entgegen AngularJS' Grundgedanke, ein Teil der Geschäftslogik zurück auf den Server ausgelagert.

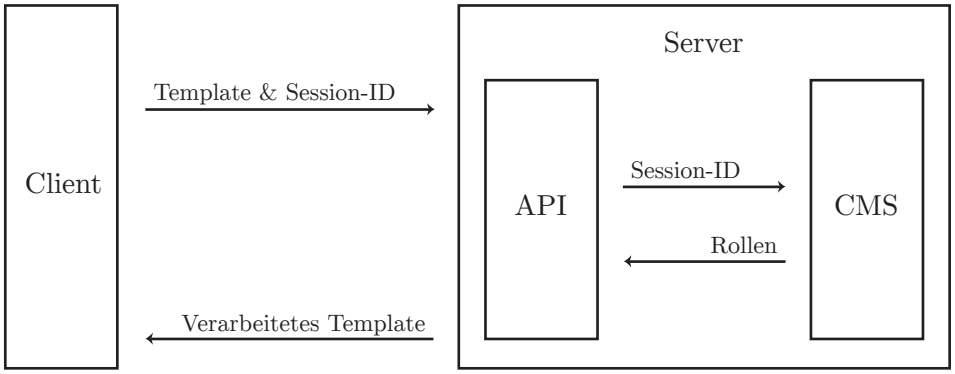

Abbildung 5.2: Serverseitiger Ansatz mit Session-ID

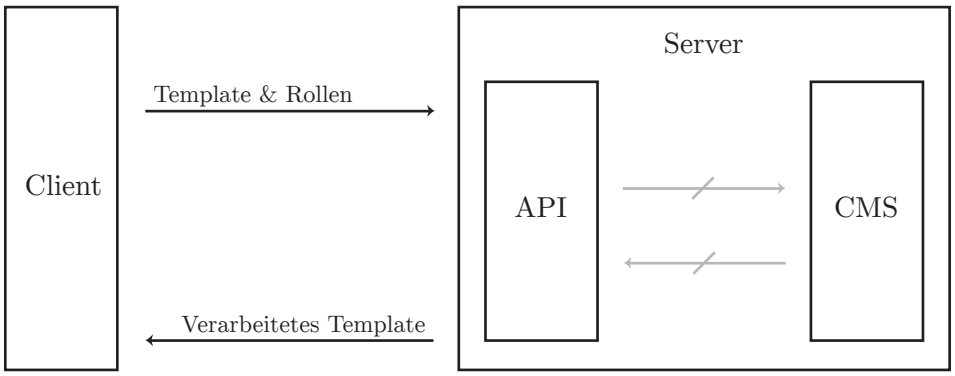

Abbildung 5.3: Serverseitiger Ansatz ohne Session-ID

5.1.3 Bewertung und Auswahl

Für die Bewertung der drei erläuterten Lösungsansätze werden die in Kapitel 3.1 definierten Prämissen hinzugezogen. Aufgrund der Prämissen ist davon auszugehen, dass der Server lediglich sichere Informationen liefert, weshalb die Aspekte der Sicherheit und Integrität bereits als gegeben angesehen werden. Da die beiden Aspekte die wesentlichen Vorzüge der serverseitigen Implementierung darstellen, kommt diese als Lösungsansatz nicht in Betracht.

Durch den Vergleich der beiden Ansätze in Kapitel 5.1.2, wurden die jeweiligen Vor- und Nachteile verdeutlicht. Während die Modifikation während des Kompilierungsprozesses mit einer erhöhten Performanz punktet, sticht jene nach dem Kompilieren durch die flexiblen und vielseitigen Einsatzmöglichkeiten hervor. Da bei einer generischen Lösung – neben der Allgemeingültigkeit – eine hohe Flexibilität von Bedeutung ist, ist der clientseitige Eingriff nach dem Kompilieren der Templates der geeignetste Ansatz für die Lösung des gegebenen Problems.

Die automatische Extraktion der Berechtigungen auf Basis einer RESTful API kommt für die generische Lösung nicht in Betracht, da jene nicht von allen CMS bereitgestellt wird. Dennoch wird für das clientseitige Modul eine Schnittstelle implementiert, welcher die Berechtigungen manuell übergeben werden können. Die Zuweisung kann während und nach dem Initialisieren erfolgen.

5.2 Komponenten der Entwicklungsumgebung

Eine Entwicklungsumgebung ist ein System aus einzelnen Komponenten, welche für die Entwicklung einer Software-Lösung verwendet werden, um jene zu unterstützen und zu vereinfachen. Nachfolgend wird die Entwicklungsumgebung für die Realisierung des Prototyps definiert, indem die einzelnen Komponenten des Systems näher betrachtet werden.

5.2.1 Node.js und Node Package Manager

Die Laufzeitumgebung *Node.js* und der dazugehörige *Node Package Manager* (npm) stellen gewisse Funktionen zur Verfügung, welche die Entwicklung von clientseitigen JavaScript-Modulen simplifizieren können (vgl. Kapitel 2.3). Zu diesen Funktionen zählen unter anderem die Verwaltung von Modulen für die server- und clientseitige Entwicklung sowie das automatisierte Ausführen bestimmter Aufgaben. Nachfolgend wird der jeweilige Einsatz von *Node.js* und

npm im Kontext des zu implementierenden Prototyps begründet und die anfallenden Aufgaben beschrieben.

5.2.2 Paketverwaltung

Wie in Kapitel 2.3.1 beschrieben, lassen sich mit npm verschiedene Module und Pakete verwalten; sämtliche Module, die über npm verfügbar sind, sind darüber zu installieren. Auf weitere Lösungen zur Paketverwaltung, wie *Bower*, ist zu verzichten. Die Gründe hierfür sind vielseitig: Zwei Dienste zur Paketverwaltung sorgen für Redundanz und erhöhen die Abhängigkeiten, die meisten *Bower*-Pakete sind auch per npm verfügbar, Abhängigkeiten werden übersichtlich an zentraler Stelle verwaltet und *Bower* ist stellenweise fehlerhaft [57]. Durch den alleinigen Einsatz von npm können Dritte alle Abhängigkeiten in Form eines Konsolenbefehls installieren: `npm install`.

5.2.3 Task Runner

Für die Automatisierung und das Verwalten von Aufgaben werden in der Regel spezialisierte *Task Runner* wie *Gulp* oder *Grunt* eingesetzt. Da jedoch Hilfsprogramme und Präkompilierer zunehmend über ein CLI verfügen, lassen sich diese ebenso über npm ausführen. Der auf JavaScript spezialisierte Consultant Keith Cirkel beschreibt in zwei seiner Weblog-Beiträge ausführlich, warum man klassische *Task Runner* meiden sollte [58] und wie sich diese durch npm ersetzen lassen [59]. Die Gründe gegen den Einsatz von *Gulp* und *Grunt* lassen sich wie folgt zusammenfassen:

- es wird für jede Aufgabe ein separates Plug-In benötigt
- Plug-Ins bedürfen einer verhältnismäßig ausführlichen Konfiguration
- Plug-Ins sind meist nur Wrapper für Binärdateien und somit redundant
- Plug-Ins bringen oft eine Vielzahl an weiteren Abhängigkeiten mit sich
- *Streaming* ist fehlerhaft oder wird nicht vollständig unterstützt
- unsachgemäßer Umgang mit Fehlermeldungen

Die aufgeführten Nachteile in Kombination mit der Tatsache, dass npm bereits im Projekt verwendet wird, bildet die Grundlage für die Auswahl des *Task Runners*. Durch den erneuten Einsatz von npm werden die Abhängigkeiten reduziert; es resultiert eine einfache Einrichtung der Entwicklungsumgebung.

5.2.4 Benötigte Tasks

Um den Entwicklungsprozess zu vereinfachen, wird das Ausführen bestimmter Aufgaben automatisiert. Die Automatisierung erfolgt mittels sogenannter *File Watcher*. Dabei handelt es sich um Skripte, welche infolge einer Datei-Änderung eine zuvor definierte Aufgabe ausführen; auch mehrere *Tasks* sind zuweisbar. Die *File Watcher* werden im Grunde für zwei Aufgabengebiete benötigt: das Erstellen einer gepackten Version für die Distribution und das Testen. Beide *Tasks* basieren auf den JavaScript-Dateien der eigentlichen Implementierung, weswegen deren Änderungen erfasst werden.

Der Übersichtlichkeit wegen, werden alle Funktionen des Moduls ihren Zuständigkeiten entsprechend in separate Dateien getrennt. Um bei der Integration in Projekte die Aufwände zu reduzieren, müssen alle Dateien für den Produktiveinsatz in der korrekten Reihenfolge gebündelt werden. Das Bündel dient wiederum als Grundlage für eine komprimierte Version. Beide Dateien werden in genannter Reihenfolge erzeugt und in einem eigenen Verzeichnis abgelegt.

Tabelle 5.1: Übersicht über die benötigten *Tasks*

Aufgabenbeschreibung	**Initiator**
Starten der *File Watcher* für die Distribution	Konsolenbefehl
Starten der serverseitigen Testumgebung	Konsolenbefehl
Starten der *File Watcher* für alle Tests	Konsolenbefehl
Bündeln der JavaScript-Dateien	*File Watcher*
Komprimieren der gebündelten JavaScript-Dateien	*File Watcher*
Ausführen der Modultests	*File Watcher*
Anzeigen der Testabdeckung	Konsolenbefehl
Ausführen der End-to-End-Tests	*File Watcher*
Anzeigen des *Minimal Working Examples*	Konsolenbefehl

Die Aufgaben hinsichtlich des Testens gliedern sich in die Modul- und End-to-End-Tests – auch Oberflächentests genannt. Beide Testverfahren werden in Kapitel 5.7.3 ausführlich behandelt. Die Implementierung soll bei jeder Änderung vollständig getestet werden, weswegen die *File Watcher* auf die eigentliche Implementierung angesetzt sind. Mit den Modultests geht die Prüfung der Testabdeckung einher; diese soll dem Entwickler mit einem gesonderten *Task* angezeigt werden. Wie im Kapitel zum Thema Testing beschrieben ist, bedarf es für End-to-End-Tests einer serverseitigen Testumgebung. Diese Testumgebung

soll – bevor die *File Watcher* aktiviert werden – ebenfalls mit einem eigenen Befehl gestartet werden können.

Die Tabelle 5.1 fasst die benötigten *Tasks* aus Gründen der Übersichtlichkeit zusammen. Neben der Aufgabenbeschreibung wird zudem der jeweils dazugehörige Initiator aufgeführt. Alle Aufgaben, welche mit einer eventuell späteren Demo zusammenhängen, werden nicht berücksichtigt; die Übersicht und Beschreibung bezieht sich lediglich auf jene *Tasks*, welche im Zuge der eigentlichen Implementierung benötigt werden.

5.3 Modulname und Namensraum

Für die Entwicklung und Publikation eines AngularJS-Moduls wird ein Name benötigt, welcher diverse Zwecke erfüllt. In erster Linie dient er der eindeutigen Identifikation der Implementierung im Web. Da der Prototyp auf verschiedenen Plattformen veröffentlicht wird, muss dieser über den Namen identifizierbar und auffindbar sein; so wird dieser auch für die Verwaltung der Paketversion, wie sie von npm vorgenommen wird, benötigt. Des Weiteren muss ein Modul gemäß *Best Practices* über einen eigenen Namensraum verfügen, damit Direktiven einem Modul nachvollziehbar zugewiesen werden können.

Die Wahl des Modulnamens fällt auf das Portmanteauwort *uXess*, welches die beiden wesentlichen Charakteristika UX und *access* (engl. für Zugang) durch eine Wortverschmelzung vereint. Alternativ kann der Name auch als eine Wortverschmelzung der Phrase „you access“ interpretiert werden, welche übersetzt „du betrittst“ bedeutet. Indem die Überlappung als Versalie dargestellt wird, lassen sich beide Worte optisch gut voneinander trennen. Da für die Implementierung und den Namensraum hauptsächlich kurze Präfixe benötigt werden, wird hierfür *uxs* verwendet. Die englische Aussprache der einzelnen Buchstaben entspricht der des ganzen Namens.

5.4 Systemfunktionalität

Die Funktionalität zum Ein- und Ausblenden der UI-Elemente auf Basis einer Direktive lässt sich in fünf Kategorien gliedern: die Handhabung der Autorisierungsmodi sowie die der Rollen, die Durchführung der Autorisierung, eine Verknüpfung zwischen den Modi und deren Funktionen zur Autorisierung sowie die Modifikation der Templates. Diese Aspekte werden im Anschluss detailliert betrachtet, deren Charakteristika beschrieben, die Anforderungen definiert sowie deren Rolle in der Implementierung aufgezeigt.

5.4.1 Handhabung der Autorisierungsmodi

Für den korrekten Umgang mit den verschiedenen Autorisierungsmodi, welche in Kapitel 3.2 im Rahmen der Anforderungsspezifikation definiert wurden, bedarf es einiger unterstützender Funktionen.

Der benötigte Autorisierungsmodus kann für jedes UI-Element separat mittels eines HTML-Attributes angegeben werden. Indem auf ein solches Attribut zurückgegriffen wird, ist eine einzige Direktive für die Modifikation der Templates ausreichend. Anderenfalls müsste für jeden Autorisierungsmodus eine entsprechende Direktive implementiert werden, welche sich lediglich hinsichtlich eines Funktionsaufrufs unterscheiden; das Prinzip *Don't Repeat Yourself* würde missachtet werden.

Sollte beim Einsatz des Moduls kein Modus explizit angegeben werden, ist ein vorgegebener Wert notwendig, auf welchen gegebenenfalls zurückgegriffen wird. Um die Integrität dieses Wertes zu gewährleisten, ist dieser nicht öffentlich zugänglich und lediglich über entsprechende Akzessoren zugreifbar. Eine öffentliche Funktion erlaubt den aktuellen Defaultwert abzufragen und eine weitere diesen neu zu definieren. Der sogenannte *Getter* gibt den definierten Wert als Zeichenkette zurück. Der *Setter* zum Ändern des vorgegebenen Wertes nimmt eine Zeichenkette entgegen, welche zunächst verifiziert wird. Zudem muss diese Funktion bereits während dem Initialisieren der Anwendung verfügbar sein, um den Wert frühzeitig ändern zu können. Ist die Verifikation, wird der übergebene Wert geparst und der entsprechenden privaten Variable zugewiesen. Die Prozesse werden nachfolgend detailliert beschrieben.

Das Parsen des Autorisierungsmodi dient ebenfalls dazu die Integrität sicherzustellen; die Funktion ist öffentlich zugänglich. Ihr wird eine Zeichenkette übergeben, welche – sollte es sich tatsächlich um eine Zeichenkette handeln – in Kleinbuchstaben transformiert und von voranstehenden sowie abschließenden *Whitespaces* befreit wird, bevor diese zurückgegeben wird. Sollte keine Zeichenkette übergeben werden, entspricht der Rückgabewert dem vorgegebenen Wert. Die ebenfalls öffentliche Funktion zur Verifikation des Autorisierungsmodi prüft, ob der Modus – welcher in Form einer Zeichenkette übergeben wird – tatsächlich definiert ist. Dazu wird der Parameter geparst und anschließend dessen Existenz geprüft. Der Rückgabewert bildet ein boolescher Wert.

5.4.2 Handhabung der Rollen

Die Handhabung der Rollen, welche dem Nutzer zugewiesen und in den einzelnen Direktiven definiert werden können, erfolgt ähnlich wie jene der Auto-

risierungsmodi. Dabei müssen beide Rollenangaben in ein einheitliches Format überführt werden, um diese bei Bedarf abgleichen zu können.

Die aktuellen Berechtigungen des Nutzers werden in einer privaten Liste hinterlegt, welche zunächst keine Rollen enthält. Für den Zugriff werden erneut öffentliche Akzessoren eingesetzt. Während der *Getter* lediglich die Rollen in Form einer Liste zurückgibt, parst der *Setter* die übergebenen Berechtigungen und weist diese der privaten Variable zu. Darüber hinaus wird nach dem Zuweisen ein Ereignis ausgelöst, das die Anwendung über das Ändern der Rollen informiert. Der *Setter* ist bereits während dem Initialisieren verfügbar, wodurch die Rollen definiert werden können, bevor die Anwendung zugänglich ist.

Das Parsen der Rollen respektive Berechtigungen findet an dieser Stelle in zwei Ebenen statt. Die vorangestellte Funktion nimmt wahlweise eine Zeichenkette oder eine Liste von Zeichenketten entgegen. Im Falle des erst genannten wird die Zeichenkette anhand der enthaltenen Kommata getrennt, sodass eine Liste entsteht. Die Liste wird im Anschluss auf der zweiten Ebene weiterverarbeitet, indem alle Listenelemente in Kleinbuchstaben transformiert sowie voranstehende und abschließende *Whitespaces* entfernt werden. Sollte es sich bei einem Element nicht um eine Zeichenkette handeln, wird dieses – falls es nicht in eine solche umgewandelt werden kann – durch eine leere Zeichenkette ersetzt. Das Ergebnis der zweiten Ebene wird letzten Endes zurückgegeben; es handelt sich um eine Liste mit Zeichenketten. Da die Verarbeitung der Liste die umgebende Funktion voraussetzt, ist das Parsen der Liste nicht öffentlich zugänglich.

5.4.3 Durchführung der Autorisierung

Die Autorisierung, welche dem Ein-/Ausblenden der UI-Elemente zugrunde liegt, beruht auf einer zentralen Funktion. Diese ist öffentlich zugänglich und vereint die Aspekte der Rollen und Autorisierungsmodi, welche als Parameter übergeben werden. Der Modus wird zunächst verifiziert und geparst; anhand des Resultats wird der weitere Verlauf der Autorisierung koordiniert.

Um die Funktionen der Autorisierung mit den entsprechenden Modi zu verknüpfen, müssen die Namen jener aufeinander abgebildet werden. Die Abbildung wird als Konstante definiert, da sich diese während der Programmlaufzeit der Anwendung nicht ändern darf. Für das *Mapping* bieten sich assoziative Listen oder deren jeweilige Entsprechung an.

Die Funktionen für die einzelnen Autorisierungsmodi sind ebenfalls öffentlich, um gegebenenfalls auf die Übergabe des Modus verzichten und die Funktion direkt aufrufen zu können. Für die Prüfung ist eine Iteration über die Liste

der benötigten und zugewiesenen Rollen notwendig. Der erste Modus prüft, ob dem Nutzer alle benötigten Rollen zugewiesen sind. Der nächste Modus prüft, ob mindestens eine der Rolle zugewiesen ist; da es sich hierbei um die meist benötigte Methodik handelt, wird diese als *Default* definiert. Sobald eine gemeinsame Rolle gefunden wird, soll die Suche respektive der Abgleich aus Gründen der Performanz abgebrochen werden. Der letzte Modus erzielt sein Ergebnis, indem auf den zweiten, standardmäßigen Modus zurückgegriffen und dessen Resultat negiert wird; dies beruht auf der mathematischen Regel, dass $X < 1$ die Negation von $X \geq 1$ bildet.

5.4.4 Modifikation der Templates

Wie zu Beginn der Konzeption erläutert, werden Direktiven für die Modifikation der Templates eingesetzt; genauer gesagt findet die Modifikation innerhalb der `link` Funktion nach dem Kompilieren der Templates statt. Im Folgenden wird der Ablauf und die Funktionsweise der Direktive näher geschildert.

Eingangs werden die benötigten Rollen und der entsprechende Autorisierungsmodus aus den Attributen der jeweiligen UI-Elemente extrahiert. Da die Werte der beiden Attribute unter anderem auch *Expressions* enthalten können, müssen diese aufgelöst werden. Die aufgelösten Attributwerte werden anschließend zur Autorisierung des Nutzers verwendet. Ist die Autorisierung erfolgreich, wird das UI-Element eingeblendet; anderenfalls wird es ausgeblendet. Für beide Vorgänge werden die Funktionen `enter` und `leave` des Moduls *ngAnimate* verwendet, welche jene mit Animationen versehen. Die vollumfängliche Funktionsweise, von der Extraktion der Attribute bis zum Ein-/Ausblenden, muss gekapselt werden. Die Kapselung dient dazu, dass der Prozess auch während der Programmlaufzeit erneut angestoßen werden kann. Dafür wird ein Ereignis gesetzt, welches – wie in Kapitel 5.4.2 erwähnt – bei jeder Definition von aktuellen Benutzerrollen ausgelöst wird. Dadurch wird die dynamische Modifikation realisiert.

5.5 Systemarchitektur

Nachdem die Verwendung der beiden JavaScript-Technologien AngularJS und *Node.js* definiert und die dazugehörigen Funktionen beschrieben wurden, lassen sich diese nach ihren Zuständigkeiten aufteilen und geeigneten AngularJS Komponenten zuweisen. Anschließend wird aufgezeigt, wie die Komponenten der Systemarchitektur zueinander in Beziehung stehen. Zudem wird, auf Basis der Anforderungen und der einzelnen Komponenten, eine grobe Verzeichnisstruktur erstellt und beschrieben.

5.5.1 Zuständigkeiten und Architektur

Die in Kapitel 5.4 definierten Funktionen werden anhand der jeweiligen Zuständigkeiten einzelnen Komponenten zugewiesen; dafür wird die bereits vorgenommene Kategorisierung übernommen. Bei den verfügbaren, wesentlichen Komponenten handelt es sich um Module, Controller, Direktiven und Services; letztere lassen sich über unterschiedliche Wege konstruieren.

Als externe Abhängigkeiten werden AngularJS selbst sowie das erweiternde Modul *ngAnimate* benötigt. Der Prototyp bedarf keine weiteren Module. Die Abhängigkeiten stellen Schnittstellen zur Verfügung, welche von der eigentlichen Implementierung genutzt werden. Die Implementierung wird in einem eigenen Modul namens *uXess* zusammengefasst, um die einzelnen Bestandteile zu bündeln und die Zusammengehörigkeit zu definieren. Dieses Modul stellt ebenfalls APIs zur Verfügung, welche hier jedoch zu vernachlässigen sind.

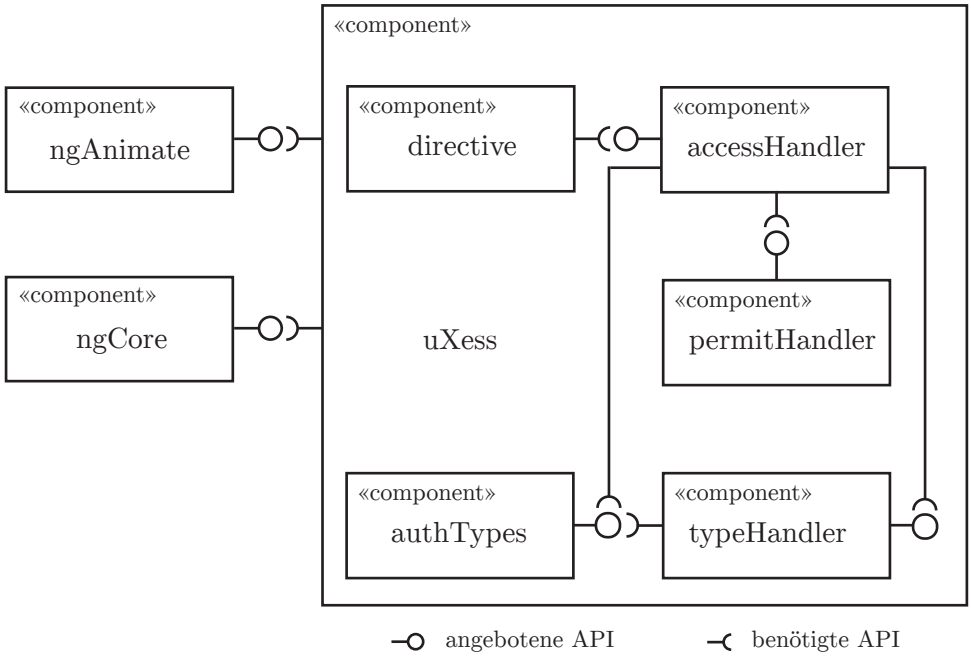

Abbildung 5.4: *uXess* – Komponentendiagramm

Die Abbildung 5.4 verdeutlicht die Zusammenhänge zwischen den einzelnen Komponenten des Prototyps. So werden die Beziehungen zwischen den übergeordneten Modulen und diejenigen der internen Komponenten aufgezeigt. Auf eine detaillierte Definition aller Schnittstellen wird aus Gründen der Übersichtlichkeit verzichtet.

Im Rahmen des Prototyps werden keine Controller eingesetzt, da das bidirektionale Data-Binding an dieser Stelle nicht benötigt wird. Obwohl gemäß des MVC Architekturmusters die Geschäftslogik im Controller untergebracht ist, wird diese – typisch für AngularJS – in diverse Services aufgeteilt. Die ersten beiden Services behandeln die Autorisierungsmodi und Rollen. In den Schaubildern werden sie als *typeHandler* respektive *permitHandler* bezeichnet. Da diese Komponenten unter anderem Funktionen beinhalten, die für die Konfiguration bereits während dem Initialisieren der Single-Page-Anwendung zur Verfügung stehen müssen, wird das sogenannte *Provider Pattern* eingesetzt. Dieses ermöglicht, dass zusätzlich zum eigentlichen Service weitere Funktionen implementiert werden können, auf die bei Bedarf in der Konfigurationsphase zugegriffen werden kann.

Die Autorisierung auf Basis des Autorisierungsmodus und der Berechtigungen wird von einem gewöhnlichen *Service Pattern* namens *accessHandler* übernommen; hierbei sind keine Besonderheiten zu beachten. Der letzte Service bildet die Autorisierungsmodi auf die Funktionen des *accessHandlers* in Form einer Konstante ab; die Konstante trägt den Namen *authTypes*.

Der *accessHandler* stellt die Verknüpfung zwischen der Direktive und den restlichen Komponenten dar, indem er der Direktive eine Schnittstelle zur Verfügung stellt und auf eine solche der jeweils anderen Komponenten zurückgreift. Trotz diverser Autorisierungsmodi, wird lediglich eine Direktive implementiert und somit das Designprinzip *Don't Repeat Yourself* eingehalten.

Das dazugehörige Sequenzdiagramm, welches von den Abbildungen 5.5 und 5.6 dargestellt wird, zeigt die Interaktionen der einzelnen Komponenten in zeitlicher Abfolge auf; es wird das Zusammenspiel der Komponenten auf Basis der definierten Funktionen verdeutlicht. Auslöser des Vorgangs ist entweder das Beenden der Kompilierungsphase oder das Eintreten des definierten Ereignisses. Es wird ersichtlich, dass die Direktive frühzeitig auf den *accessHandler* zugreift und dieser einen Großteil koordiniert. Der Prozess läuft weitestgehend jeweils gleich ab, lediglich gegen Ende sind alternative Wege möglich. Dies beruht zum einen auf der unterschiedlichen Autorisierungsmodi und zum anderen auf dem Erfolg respektive Misserfolg der Autorisierung. Die abschließende Definition des Ereignisses findet ausschließlich dann statt, sollte der Prozess durch Beenden der Kompilierungsphase angestoßen werden.

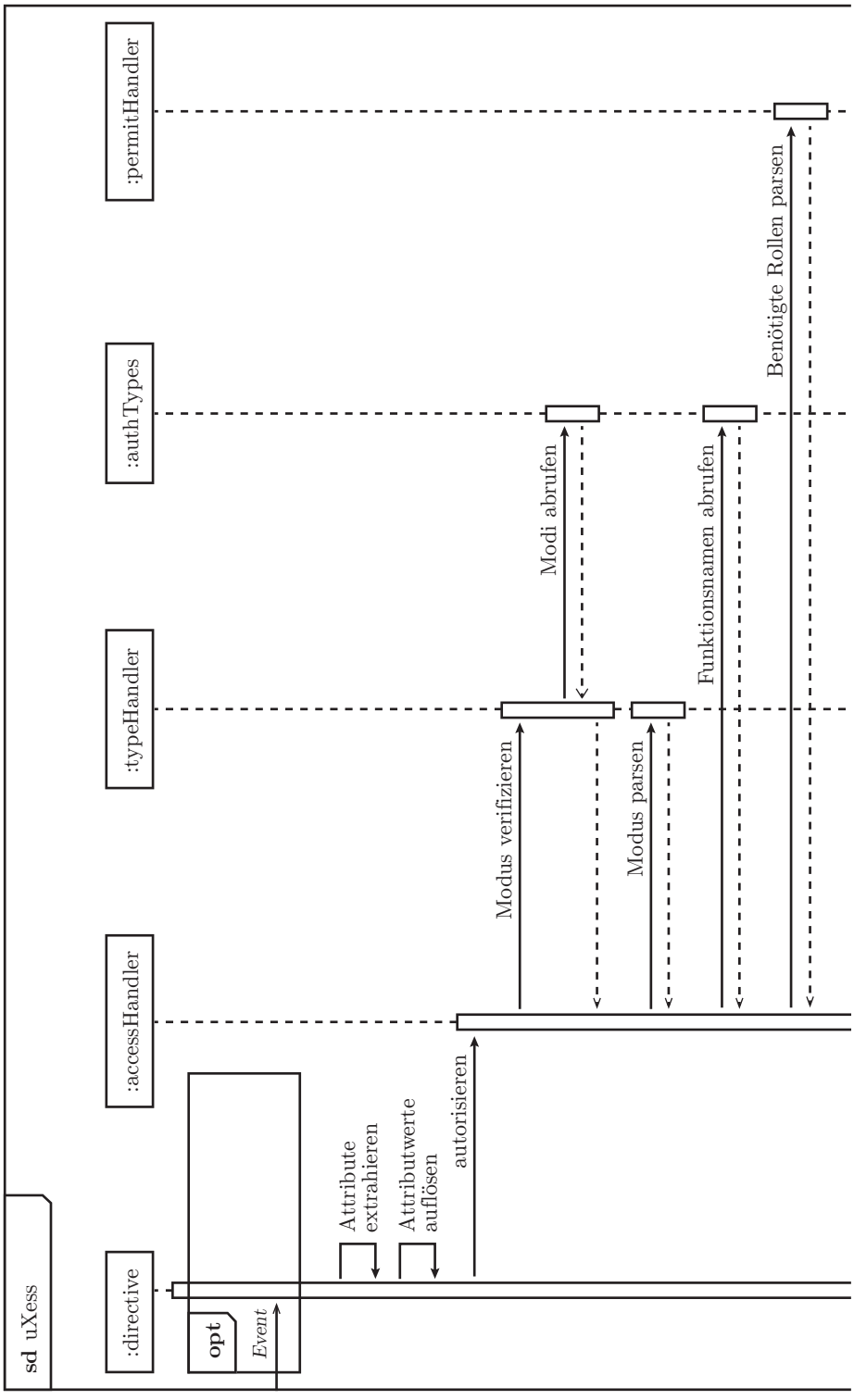

Abbildung 5.5: *uXess* – Sequenzdiagramm I

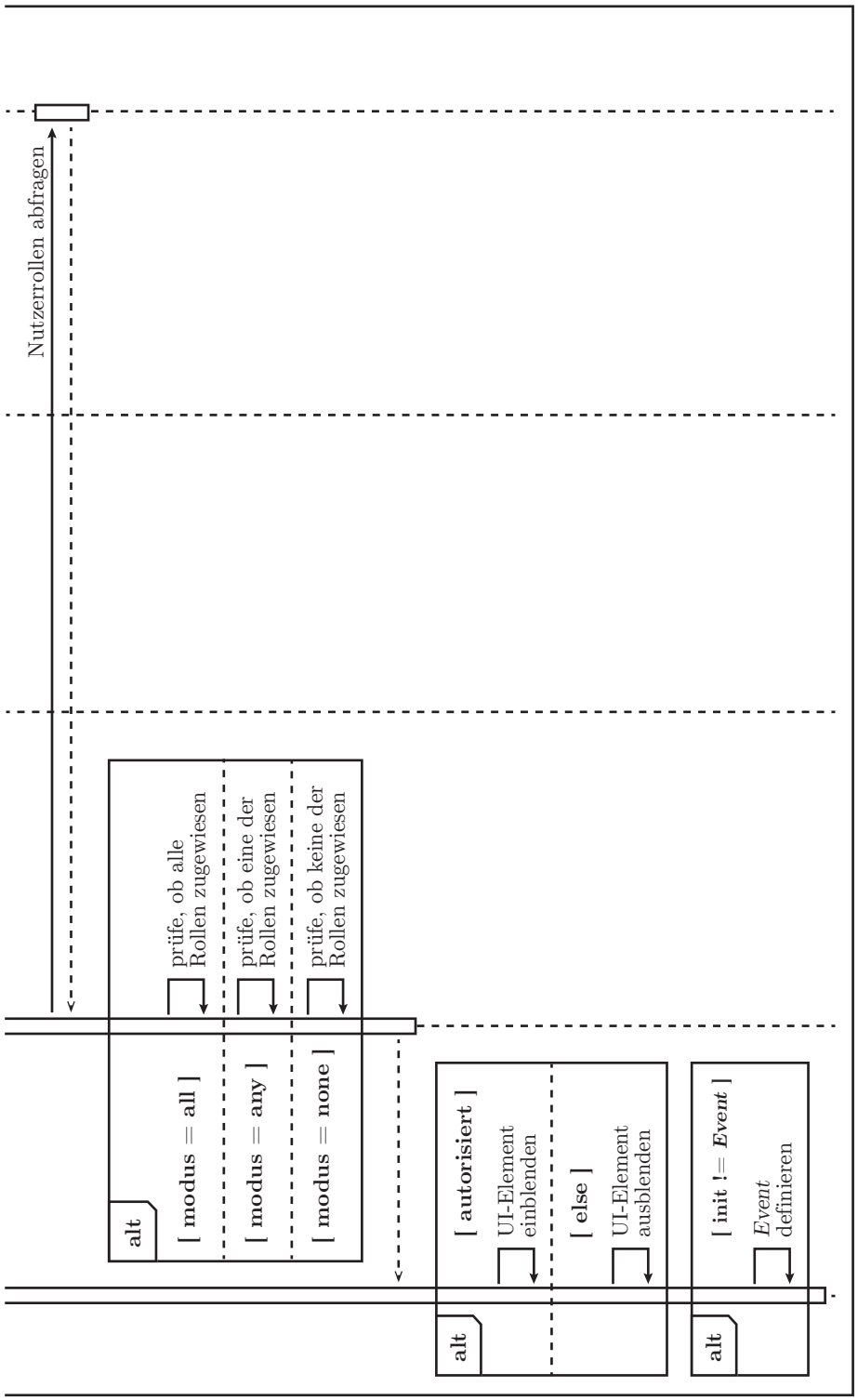

Abbildung 5.6: *uXess* – Sequenzdiagramm II

5.5.2 Verzeichnisstruktur des Prototyps

Die in Abbildung 5.5.2 dargestellte Verzeichnisstruktur beinhaltet lediglich die Verzeichnisse ohne die jeweils enthaltenen Dateien; das Wurzelverzeichnis `/` stellt die Ausnahme dar. Nachfolgend werden die einzelnen Dateien und Verzeichnisse unter Anbetracht der definierten Entwicklungsumgebung und der Systemarchitektur erläutert.

Bei den im Wurzelverzeichnis enthaltenen Dateien `.gitignore`, `package.json`, `LICENSE.md` und `README.md` handelt es sich zum einen um Dateien zum Zwecke der Konfiguration von Versionsverwaltung und npm. Zum anderen dienen sie der Distribution, indem Informationen zur Lizenzierung respektive dem Modul bereitgestellt werden.

Das Verzeichnis `config/` enthält – abgesehen von `package.json` – alle Konfigurationsdateien, welche für die Entwicklung und den Produktiveinsatz des Moduls benötigt werden. Um die einzelnen Dateien eindeutig der Konfiguration zuordnen zu können, werden diese mit dem Suffix `*.conf.*` versehen; das vollständige Suffix lautet in der Regel `*.conf.js`.

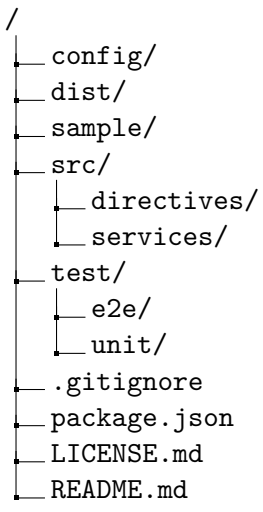

Abbildung 5.7: *uXess* – Grobe Verzeichnisstruktur

Die gebündelten Dateien für den Produktiveinsatz, welche durch zuvor durch die entsprechenden *Tasks* erzeugt wurden werden im Verzeichnis `dist/` abgelegt. Die enthaltenen Dateien identifizieren sich und das Modul über ihren Namen. Das Suffix `*.min.*` deutet auf eine komprimierte Version hin.

Im Verzeichnis `sample/` sind alle Dateien untergebracht, welche für das *Minimal Working Example* benötigt werden. Hierzu zählt die HTML-Datei selbst, die verwendeten Assets und Dateien, welche einen potentiellen Server emulieren. Eventuelle Konfigurationsdateien werden im entsprechenden Verzeichnis `config/` abgelegt. Da die Oberflächentests auf die Demo zurückgreifen, wird das `sample/` Verzeichnis unter anderem auch von diesen verwendet.

Der Quellcode der eigentlichen Implementierung liegt im Verzeichnis `src/`; die Abbreviation steht für „Source“. Auf dieser Ebene befinden sich, neben der zu initialisierenden Hauptdatei `index.js`, weitere Verzeichnisse. Die Verzeichnisse dienen zur Strukturierung der einzelnen Komponenten hinsichtlich ihres Typs. So werden im Unterverzeichnis `services/` alle Dateien vom gleichnamigen Typ abgelegt; dazu zählen zudem *Provider*, *Factories* und *Constants*. Komponenten dieser Art werden mit dem Suffix `*.srv.js` versehen. Die Direktive befindet sich im namensgleichen Verzeichnis und endet mit dem Suffix `*.drv.js`.

Alle Tests, welche für die Implementierung geschrieben werden, sind im Verzeichnis `test/` untergebracht. Dieses untergliedert sich in die Unterverzeichnisse `unit/` und `e2e/`, welche Modul- und Oberflächentests beherbergen. Der Verzeichnisstruktur der Modultests ist analog zu der des Verzeichnisses `src/`. Die Oberflächentests werden in einzelne Dateien unterteilt, die jeweils eine Funktion oder ein UI-Segment testen. Alle Tests haben gemeinsam, dass die Dateinamen zwischen dem Bezeichner und dem Datei-Suffix eine `*.spec.*` Silbe beinhalten. Der Grund hierfür ist, dass die Tests die Anwendung spezifizieren und durch die Silbe eindeutig gekennzeichnet werden.

5.6 Integration in Anwendungen

Die Integration des Moduls in eine AngularJS-Anwendung erfolgt in folgenden Schritten: Einbetten der JavaScript-Dateien, Definition des Moduls als Abhängigkeit, Konfiguration des Moduls und Handhaben der UI-Elemente mittels Direktiven. Die Betrachtung der einzelnen Schritte erfolgt anhand eines kurzes Quellcode-Beispiels, um die jeweiligen Vorgänge zu verdeutlichen.

5.6.1 Einbetten der JavaScript-Dateien

Zunächst müssen das Modul und all seine Abhängigkeiten in der HTML-Datei der Anwendung eingebettet werden. Da *uXess* lediglich von AngularJS selbst und *ngAnimate* abhängt, werden nur diese beiden Dateien zusätzlich benötigt

(vgl. Quellcode 5.6). Alternativ lassen sich alle Dateien zu einer einzigen bündeln, wodurch die Zahl der HTTP-Requests reduziert wird; dies ist für die Funktionsweise des Moduls jedoch irrelevant.

Quellcode 5.6: *uXess* – Einbetten des Moduls und aller Abhängigkeiten

```
1 <script src="angular.min.js"></script>
2 <script src="angular-animate.min.js"></script>
3 <script src="uxess.min.js"></script>
```

5.6.2 Definition der Abhängigkeiten

Bei der Deklaration des für die Anwendung zuständigen Hauptmoduls muss das Modul *uXess* als Abhängigkeit definiert werden. Dazu wird der Modulname in Form einer Zeichenkette an AngularJS übergeben (vgl. Quellcode 5.7). Die Definition von *ngAnimate* ist nicht zwingend notwendig, da dieses Modul bereits als Abhängigkeit von *uXess* aufgeführt wurde. Weitere Abhängigkeiten, die projektspezifisch benötigt werden, sind der Liste beizufügen.

Quellcode 5.7: *uXess* – Definition der Abhängigkeiten

```
1 angular.module('sampleApp', [
2   'uxs'
3 ]);
```

5.6.3 Konfiguration des Moduls

Für die Konfiguration des Moduls stellt AngularJS die beiden Methoden `config` und `run` bereit, welche während dem Initialisieren der Anwendung beziehungsweise direkt im Anschluss ausgeführt werden. Innerhalb der `config` Methode lassen sich lediglich die *Provider* konfigurieren, sprich es lassen sich nur die Rollen und der standardmäßige Autorisierungsmodus definieren. Nach dem Initialisieren, also in der *run* Funktion, sind alle Services des Moduls verfügbar. So lassen sich beispielsweise die aktuellen Berechtigungen vom Server unter Verwendung des AngularJS-internen Services `$http` abfragen und der Anwendung über die öffentlichen Schnittstellen übergeben (vgl. Quellcode 5.8).

Quellcode 5.8: *uXess* – Konfiguration des Moduls

```
1  angular.module('sampleApp')
2    // during the initialisation
3    .config(function(uxsAuthTypeHandlerProvider) {
4      uxsAuthTypeHandlerProvider.setDefaultAuthType('all');
5    })
6    // immediately after the initialisation
7    .run(function($http, uxsPermitHandler) {
8      $http.get('/api/v1/user').then(function(res) {
9        uxsPermitHandler.setPermits(res.permits);
10     });
11   });
```

Wie in Quellcode 5.8 zu sehen ist, kann der Defaultwert des Autorisierungsmodi über `uxsAuthTypeHandlerProvider` definiert werden, indem eine entsprechende Zeichenkette übergeben wird. Während der `run` Methode können dann unmittelbar nach dem Initialisieren die aktuellen Nutzerberechtigungen abgefragt werden. Hierbei bietet sich eine RESTful API oder zumindest die Ausgabe im JSON-Format an. Für die Zuweisung über `uxsPermitHandler` werden die Berechtigungen als Liste oder Zeichenfolge benötigt. Sollte das notwendige Format nicht vorliegen, müssen die Berechtigungen vorweg in ein solches überführt werden.

Der nachfolgende Quellcode 5.9 zeigt eine exemplarische Antwort, die eine REST-konforme Schnittstelle zurückliefert (vgl. Quellcode 5.8, Zeile 8). Die komplette Antwort ist über die Variable `res` verfügbar, sodass `res.permits` auf die Berechtigungen des Nutzers verweist. Da es sich hierbei um eine Liste handelt, kann diese ohne weitere Bearbeitung übergeben werden.

Quellcode 5.9: Exemplarische Nutzerinformationen

```
1  {
2    user: {
3      prename: "John",
4      lastname: "Doe",
5      sex: "male"
6    },
7    permits: [
8      "admin"
9    ]
10 }
```

5.6.4 Intervention mittels Direktiven

Abschließend können den UI-Elementen sowohl die entsprechenden Rollen als auch der notwendige Autorisierungsmodi zugewiesen werden, indem auf die Direktiven `uxs-if` und `uxs-type` zurückgegriffen wird (vgl. Quellcode 5.10). Dabei können sowohl dynamische *Expressions* als auch statische Werte übergeben werden; die Direktiven wandeln die entsprechenden Werte mit Hilfe der Parser-Funktionen in ein einheitliches Format um.

Quellcode 5.10: *uXess* – Intervention mittels Direktiven

```
1 <button uxs-if="user, editor, admin" uxs-type="any">Button</button>
2 <button uxs-if="admin" uxs-type="none">Button</button>
3 <button uxs-if="['user', 'admin']" uxs-type="all"></button>
```

5.7 Qualitätssicherung

Um den spezifizierten, qualitativen Anforderungen gerecht zu werden, werden im Folgenden Maßnahmen zur Qualitätssicherung definiert. Dazu zählen die Programmierrichtlinien, *Best Practices* und Designprinzipien; die Schnittstellen-Dokumentation sowie das Testen in Form von Modul- und Oberflächentests. Neben der allgemeinen Beschreibung der einzelnen Maßnahmen, werden zudem sowohl Besonderheiten aufgezeigt, als auch wie die jeweiligen Maßnahmen im Speziellen umgesetzt werden.

5.7.1 Programmierrichtlinien und Best Practices

Bei der Implementierung werden allgemeine JavaScript-Programmierrichtlinien [49] und solche speziell für AngularJS [45] berücksichtigt. Hinzu kommen weitere *Best Practices* sowie Designprinzipien [32]. Bei der Betrachtung dieser wird nur auf die wesentlichen Aspekte eingegangen, welche zur maßgeblichen Steigerung der Qualität beitragen.

Grundsätzlich werden alle Komponenten durch *Immediately-Invoked Function Expression* (IIFE) gekapselt. Dabei handelt es sich um anonyme Funktionen, die direkt nach ihrer Definition selbst ausgeführt werden (vgl. Quellcode 5.11). In dieser Funktion deklarierte Variablen werden somit in einem eigenen Gültigkeitsbereich definiert, sodass diese nicht mit gleichnamigen Variablen in ande-

ren Komponenten in einen Konflikt geraten [8]. Das voranstehende Semikolon beugt Fehlern bei der Bündelung mehrerer Module vor. Die Anweisung `use strict` aktiviert eine Untermenge der Programmiersprache, die das Verhalten des Quellcodes und die Fehlerbehandlung beeinflusst. Dadurch wird der Quellcode generell weniger anfällig für Fehler [8].

Quellcode 5.11: Konstrukt der Immediately-Invoked Function Expression

```
1 ;(function() {
2
3   'use strict';
4
5   // ...
6
7 })();
```

Insbesondere die Nomenklatur der Komponenten wird von den Programmierrichtlinien abgedeckt. Durch das konsistente Einhalten dieser Richtlinien lassen sich die Komponenten und ihre Eigenschaften schneller identifizieren. Generell werden alle Komponenten mit einem modulspezifischen Präfix versehen, damit sich jene diesem nachvollziehbar zuordnen lassen. Grundsätzlich sollte auf $ als Präfix verzichtet werden, da diese für interne Variablen und Komponenten reserviert ist. Konstanten werden typischerweise in Versalien benannt, um zu diese von Variablen abzuheben. Bei allen Services handelt es sich im Grunde um Konstruktoren, dessen Namen in der Informatik mit einem Versal beginnen; jedoch werden diese ebenfalls mit einem modulspezifischen Präfix versehen, das aus Minuskeln besteht. Die Nomenklatur der Variablen und Funktionen weist abgesehen von zwei Eigenschaften keine Besonderheiten auf. So werden anonyme Funktionen vermieden und wenn möglich mit einem Namen versehen. Dies vereinfacht die Fehlersuche anhand des *Stacktrace*. Zudem wird dem Namen privater Variablen ein Unterstrich vorangestellt.

Im Zuge der Implementierung werden drei Designprinzipien umgesetzt, welche bereits in Kapitel 3.2.2 erwähnt werden: *You Ain´t Gonna Need It*, *Keep It Simple Stupid* und *Don't Repeat Yourself*. Das erste Prinzip beschreibt den Fokus auf die gegebene Problematik, ohne dass eventuell zukünftige Anforderungen berücksichtigt werden; dadurch bleibt in Implementierung schlanker und leichter zu warten. Das Prinzip KISS besagt, dass der Quellcode für andere Entwickler nachvollziehbar und verständlich gestaltet werden soll. So können auch gegebenenfalls geringe Einbußen hinsichtlich der Performanz in Kauf genommen werden, wenn dies die Lesbarkeit signifikant zugutekommt. Das letztgenannte

Designprinzip DRY sagt aus, dass wiederkehrende Quellcode-Fragmente ausgelagert werden sollen. Hierdurch lassen sich Änderungen zentral an einer Stelle vornehmen; die Wartbarkeit wird erhöht.

Jede Komponente wird in einer eigenen Datei untergebracht und mit einem komponentenspezifischen Suffix versehen. Dies erleichtert die Navigation innerhalb des Projektes und ermöglicht eine exakte Benennung der Dateien. Zudem werden die Dateien entsprechend ihres Typs einem Verzeichnis zugeteilt; die Funktion und Art einer Komponente lässt sich somit anhand des Dateinamens respektive des Verzeichnisses erkennen. Flache Verzeichnisstrukturen erlauben einen besseren Überblick. Diese *Best Practices* werden bereits beim Abbilden der groben Verzeichnisstruktur (vgl. Abbildung 5.5.2) berücksichtigt.

5.7.2 Schnittstellen-Dokumentation

Für die Schnittstellen-Dokumentation wird in Kapitel 3.2.3 der Einsatz der Auszeichnungssprache *JSDoc* [34] festgelegt. Da hierdurch eine hohe Anzahl an *Tags* zur Verfügung gestellt wird, die für den Prototyp nicht von Belangen sind, werden nun die relevanten Eigenschaften definiert und vorgestellt.

Bei allen bedeutenden Funktionen und Variablen wird deren Typ angegeben. Im Allgemeinen wird dafür `@type {typeName}` verwendet, jedoch existieren für Konstruktoren und Funktionen zudem die Aliase `@constructor` und `@function`. Sollte das Objekt `this` auf ein nicht naheliegendes Objekt referenzieren, so wird zudem von `@this <namePath>` Gebrauch gemacht. Um zu verdeutlichen, ob ein Objekt öffentlich zugänglich ist oder nicht, werden `@public` und `@private` verwendet. Der Typ und die Zugriffsart sind in jedem Fall anzugeben.

Für eine kurze Beschreibung des Objektes wird der *Tag* `@description <text>` eingesetzt. Dieses Attribut sollte wenn möglich jedes Objekt beschreiben, um weiteren Entwicklern – abgesehen von sprechenden Bezeichnern – einen Überblick zu verschaffen, ohne dass sich diese mit dem Quellcode detailliert auseinandersetzen müssen.

Bei Funktionen sind insbesondere die zu übergebenden Parameter sowie ein eventueller Rückgabewert von Bedeutung. Für die Beschreibung der Parameter wird auf die Syntax `@param {typeName} <paramName> <paramDescription>` zurückgegriffen. Sollte ein Rückgabewert existent sein, so wird dieser mit dem *Tag* `@returns {typeName} <returnsDescription>` gekennzeichnet.

Zu guter Letzt muss das Erstellen von Ereignissen beziehungsweise das Auslösen dieser berücksichtigt werden. Für diese beiden Zwecke bestehen die *Tags*

`@listens <eventName>` und `@fires <eventName>`, welche in der Dokumentation der entsprechenden Funktionen zu verwenden sind.

5.7.3 Testing

Das Testen der Implementierung findet in zwei Schritten statt: zunächst werden alle Funktionen gemäß ihrer Anforderungen und Spezifikation mit Hilfe von Modultests geprüft. Anschließend wird der Produktiveinsatz der vollständigen Implementierung in einem *Minimal Working Example* getestet, wozu eine grafische Oberfläche benötigt wird; diese Tests werden auch Oberflächentests genannt. Da der Prototyp zu einem späteren Zeitpunkt im Browser eingesetzt wird, wird jene auch in den neusten Versionen des *Mozilla Firefox* [60] und *Google Chrome* [61] getestet.

Umgesetzt werden alle Tests auf Basis des *Test Runners Karma* [41] sowie des *Test Frameworks Jasmine* [48]. Der *Runner* führt alle mit *Jasmine* definierte Tests aus und verwaltet diese. Das *Framework* stellt Funktionen bereit, die das Testen vereinfachen. Zudem können die einzelnen Spezifikationen in sogenannten *Test Suites* gruppiert werden. Der Quellcode 5.12 zeigt den exemplarischen Einsatz von *Jasmine*: `describe` definiert *Test Suites* und `it` die Spezifikationen. Die eigentliche Testanweisung wird durch `expect` eingeleitet.

Quellcode 5.12: Exemplarische *Test Suite* in *Jasmine*

```
1 describe("A suite", function() {
2   it("contains spec with an expectation", function() {
3     expect(true).toBe(true);
4   });
5 });
```

Modultests

Die in Kapitel 5.4 spezifizierten Anforderungen an die einzelnen Funktionen der Implementierung werden durch Modultests abgedeckt. Dabei wird versucht alle Funktionen mit Tests zu versehen, auch wenn direkt nur die öffentlich zugänglichen Funktionen testbar sind. Die privaten Funktionen müssen somit über entsprechende Aufrufe der dazugehörigen öffentlichen Funktionen getestet werden. Sollte eine Funktion verschiedene Arten von Parametern akzeptieren, so ist für jede ein einzelner Test zu erstellen, sodass alle möglichen Kombination

abgedeckt werden. Neben dem Prüfen der Ein- und Ausgabe sowie der Rückgabe, müssen auch unterschiedliche Verzweigungen innerhalb der Funktionen abgedeckt werden, um eine hohe Testabdeckung zu erreichen.

Jede mit *Jasmine* deklarierte Spezifikation enthält lediglich eine Prüfung, sodass bei fehlgeschlagenen Tests die Ursache schnell identifiziert werden kann. Verschiedene `it` Anweisungen für eine Funktion werden dabei in einer eigenen *Test Suite* gruppiert; dies dient der Übersichtlichkeit. Zudem entspricht die Verzeichnisstruktur der Modultests derjenigen der Implementierung.

Neben *Karma* und *Jasmine* wird für die Modultests noch das Modul *ngMock* [62] benötigt. Dieses Modul ermöglicht es Abhängigkeiten in die einzelnen Tests zu injizieren und diverse AngularJS Services zu emulieren. Darüber hinaus wird das Modul *Istanbul* [63] über eine Konfigurationsdatei in *Karma* integriert; dieses stellt die Testabdeckung der Modultests grafisch dar.

End-to-End-Tests

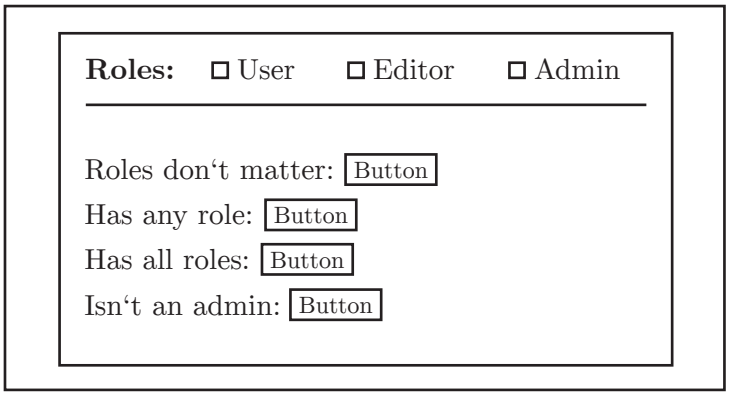

Abbildung 5.8: *uXess* – Wireframe des *Minimal Working Example*

Die Oberflächentests werden benötigt, um die Kernfunktionalität der Implementierung unter Zuhilfenahme einer Demo zu testen. Hierfür wird zunächst ein *Minimal Working Example* entwickelt und im Verzeichnis `sample/` abgelegt. Die Abbildung 5.8 zeigt das Wireframe der zu implementierenden Demo. Diese besteht aus drei Eingabefeldern vom Typ *Checkbox*, über welche sich die aktuellen Rollen setzen lassen. Der Hauptbereich beinhaltet vier Zeilen Text, die jeweils kurz die Bedingungen zusammenfassen. Dahinter befindet sich in jeder Zeile ein Button, der beim Erfüllen der Bedingungen dem Nutzer ange-

zeigt werden soll. Für das Zuweisen der Rollen wird eine Komponente vom Typ Controller implementiert.

Für die folgenden Tests werden jeweils eigene Dateien erstellt: Setzen von Rollen und Autorisierungsmodus während dem Initialisieren, das Wechseln von Rollen während der Laufzeit sowie die drei verfügbaren Autorisierungsmodi.

Die Umsetzung der Oberflächentests erfolgt mit Hilfe des *Frameworks Protractor* [64], das speziell auf End-to-End-Tests im AngularJS-Umfeld ausgerichtet ist. Zudem wird die Testumgebung *Selenium* [65] eingesetzt. Die Kombination dieser Module erlaubt das automatisierte Ausführen und Testen vordefinierter Interaktionen mit einer Weboberfläche.

5.8 Zusammenfassung

In diesem Kapitel wurde der zu implementierende Prototyp konzipiert, indem zunächst mögliche Lösungsansätze – sowohl auf client- als auch auf serverseitiger Basis – betrachtet und bewertet wurden. Für die weitere Konzeption wurde die Entwicklungsumgebung inklusive aller benötigten *Tools* wie npm definiert sowie der zu verwendende Modulname deklariert. Im Anschluss daran wurde die Funktionalität des Prototyps spezifiziert. Im Zuge dessen wurden die Anforderungen an einzelne, benötigte Funktionen beschrieben und diese hinsichtlich ihrer Zuständigkeiten gegliedert. Die Trennung der Zuständigkeiten bildete die Grundlage für die komponentenbasierte Architektur des Prototyps. Des Weiteren wurde der allgemeine Ablauf unter Verwendung der spezifizierten Funktionen beschrieben und eine grobe Verzeichnisstruktur konstatiert. Um eine hohe Qualität der Implementierung sicher zu stellen, wurden obendrein Richtlinien für den Quellcode und dessen Dokumentation abgesteckt. Zu guter Letzt wurde auf Wege zur Integration in Anwendung und auf die Verfahren der Modul- und Oberflächentest eingegangen. Das Resultat ist die fertige Konzeption der prototypischen Implementierung.

Kapitel 6

Realisierung eines Prototyps

Im Rahmen der Umsetzung des Prototyps traten einige Fälle auf, die eine ausgefallene Herangehensweise erforderten. In diesem Kapitel werden die Besonderheiten hinsichtlich der Lösungswege hervorgehoben und erläutert, indem auf Funktionen und Quellcode-Ausschnitte zurückgegriffen wird. Dabei soll nicht der Quellcode an sich, sondern vielmehr die jeweils zugrunde liegende Strategie oder Idee anschaulich erörtert werden.

6.1 Handhabung der Rollen

Das Parsen der Rollen findet in zwei Schritten statt: während zunächst eine eventuelle Zeichenkette in eine Liste transformiert wird, wird im zweiten Schritt die entstehende Liste von Rollen geparst. Hierbei muss über diese Liste iteriert werden, um jedes einzelne Element dieser Liste einzeln zu behandeln. Für die Iteration wird in JavaScript für gewöhnlich eine `for`-Schleife verwendet, welche jedoch dem imperativen Programmierstil entspricht und weniger der funktionalen Programmierung. Da bei der Anwendung des funktionalen Programmierstils schneller deutlicher wird, welche Funktion und Zweck die Iteration erfüllt [8], wird auf die Funktion `map` aufgebaut (vgl. Quellcode 6.1).

Diese Funktion erlaubt die Definition einer *Callback*-Funktion, welche für die einzelnen Elemente jeweils aufgerufen wird. Innerhalb dieser *Callback*-Funktion bedarf es der Berücksichtigung von eventuell beim Parsen auftretenden Fehlern, die die Anwendung vorläufig und unvorhergesehen beenden könnten. Für das Abfangen der *Exceptions* wird auf `try` und `catch` zurückgegriffen.

Quellcode 6.1: *uXess* – Funktion zum Parsen einer Liste von Rollen

```
function _parsePermitList(permits) {
  return permits.map(function (permit) {
    var trimmedPermits;
    var parsedPermits;

    try {
      trimmedPermits = permit.toString().trim();
      parsedPermits = angular.lowercase(trimmedPermits);
    } catch(error) {
      parsedPermits = '';
    }

    return parsedPermits;
  });
}
```

Wie in Kapitel 5.4.2 beschrieben, wird bei jedem Ändern der Nutzerberechtigungen ein zuvor definiertes Ereignis ausgelöst, welches die Direktiven erneut durchlaufen lässt. Dies erfolgt mittels der Funktion `$broadcast` über den globalen Gültigkeitsbereich (vgl. Quellcode 6.2).

Quellcode 6.2: *uXess* – Auslösen des spezifischen Ereignisses

```
function setPermits(permits) {
  _data.permits = this.parsePermits(permits);
  $rootScope.$broadcast('uxsPermitsChanged');
};
```

6.2 Durchführung der Autorisierung

Die Autorisierung des Nutzers basiert im Grunde auf drei Funktionen, die überprüfen, ob dem Nutzer keine, eine oder alle der übergebenen Rollen zugewiesen sind. Dadurch beruhen alle Funktionen auf dem Vergleich zweier Listen; würde man den Vergleich in allen drei Funktionen separat implementieren, würde das DRY-Prinzip verletzt werden. In JavaScript stehen dem Listen-Objekt unter anderem die beiden Funktionen `some` und `every` zur Verfügung, welche lediglich

überprüfen, ob die in einer *Callback-Funktion* definierten Bedingung für mindestens eines oder alle Listenelemente gilt. Wie in Kapitel 5.4.3 erläutert, ist für den dritten Autorisierungsmodus eine Negation des `some`-Ergebnisses ausreichend. Aus diesem Grund genügt die in Quellcode 6.3 beschriebene Funktion als gemeinsame *Callback*-Funktion. Hierbei wird geprüft, ob das übergebene Listenelement in der anderen Liste enthalten ist.

Quellcode 6.3: *uXess* – Funktion zum Überprüfen der Rollen

```
function _comparePermits(element) {
  return uxsPermitHandler.getPermits().indexOf(element) !== -1;
}
```

Der Quellcode 6.4 verdeutlicht den gemeinsamen Einsatz der in Quellcode 6.3 definierten *Callback*-Funktion `_comparePermits`. Die Funktion `every` weist den Vorteil auf, dass diese die Iteration unterbricht, sollte einer der Bedingungen fehlschlagen; ähnlich verhält es sich bei der Funktion `some`, die abbricht, sobald die Bedingung erfüllt wurde. Damit wird unnötigen Iterationen entgegengewirkt und eine Anwendung mit vielen Rollen performanter gehalten.

Quellcode 6.4: *uXess* – Funktionen zum Prüfen der Autorisierung

```
function hasPermits(permits) {
  var parsedPermits = uxsPermitHandler.parsePermits(permits);

  return parsedPermits.every(_inspectPermits);
};

function hasAnyPermits(permits) {
  var parsedPermits = uxsPermitHandler.parsePermits(permits);

  return parsedPermits.some(_inspectPermits);
};

function hasNonePermits(permits) {
  return !this.hasAnyPermits(permits);
};
```

Nachteilig an der Implementierung der Funktionen `some` und `every` ist, dass diese erst ab dem Internet Explorer 9 unterstützt werden. Da AngularJS je-

doch auch nur inklusive dieser Version testet und die korrekte Funktionsweise der Frameworks gewährleistet [66], können die Kompatibilitätsbedenken vernachlässigt werden.

6.3 Modifikation der Templates

Die Kernkomponente des Prototyps, die Direktive für die Modifikation der Templates, verfügt ebenfalls über besondere Problemstellungen. Die Direktive wird ausführlich erläutert, da sie im Allgemeinen interessante Ansätze verfolgt und sie die Basis der Implementierung bildet.

Die Direktive orientiert sich an der AngularJS-Direktive `ng-if`, welche unter anderem das korrekte Entfernen aus dem DOM behandelt. Diese Implementierung wurde in Bezug auf die spezifizierten Anforderungen erweitert und hinsichtlich ihrer Performanz optimiert. Die Optimierungen basieren in erster Linie auf der Art des Triggers; so verwendet `ng-if` die `$watch`-Funktion, die Änderungen im Datenbestand als Auslöser verwendet. Im Falle des Prototyps wird auf einfache Ereignisse zurückgegriffen, um die Direktive nur auszulösen, sollte diese wirklich benötigt werden (vgl. Quellcode 6.2 und 6.5).

Quellcode 6.5: *uXess* – Definition des Ereignisses

```
1 scope.$on('uxsPermitsChanged', checkVisibility);
```

Das Entfernen und Wiedereinfügen der UI-Elemente findet in beiden Implementierungen identisch statt, indem auf die AngularJS-Funktion `transclude` zurückgegriffen wird. Diese erlaubt es, Elemente aus dem DOM zu entfernen und die Referenz zu einem Klon dieses Elementes zu speichern. So lässt sich die korrekte Position im DOM für das Wiedereinfügen festhalten.

Quellcode 6.6: *uXess* – Funktion zum Prüfen des Attributwert-Typs

```
1 function isExpression(attr) {
2   return attr && attr.search(/{{2}.*}{2}/) !== -1;
3 }
```

Für den zu übergebenden Attributwert stehen drei mögliche Arten zur Verfügung: statische Werte, Variablennamen oder *Expressions*. Um alle Arten abzudecken und den korrekten Wert weiterzuverarbeiten, muss der Attributwert über die Funktion `getAttributeValue` (vgl. Quellcode 6.7) extrahiert werden. Diese beruht auf der ergänzenden Funktion `isExpression`, die in Quellcode 6.6 dargestellt wird. Dabei wird zunächst geprüft, ob es sich um eine Variable oder eine *Expression* handelt, und dem Ergebnis entsprechend eine passende Funktion zum Parsen beziehungsweise Interpolieren gewählt. Sollte das Parsen fehlschlagen, wird der eigentlichen Attributwert als Ergebnis betrachtet.

Quellcode 6.7: *uXess* – Funktion zum Extrahieren der Attributwerte

```
1  function getAttributeValue(scope, attr) {
2    var hasExpression = isExpression(attr);
3    var parseFn = $parse;
4    var extractedAttr;
5
6    if(hasExpression) {
7      parseFn = $interpolate;
8    }
9
10   try {
11     extractedAttr = parseFn(attr)(scope);
12   } catch(error) {
13     extractedAttr = attr;
14   }
15
16   return extractedAttr || attr;
17 }
```

6.4 Zusammenfassung

In diesem Kapitel wurden diverse Problemstellungen der einzelnen Prototyp-Komponenten vorgestellt und die dazugehörigen Lösungsansätze erläutert. Neben der besonderen Behandlung von Fehlermeldungen oder Designprinzipien wurde vor allem die Kernkomponente der Implementierung näher betrachtet. Bei der Modifikation der Templates standen das Extrahieren von Attributwerten, das Prüfen dessen Typs sowie das entsprechende Parsen respektive Interpolieren im Fokus. Darüber hinaus wurde zudem die Handhabung des prototypspezifischen Ereignisses verdeutlicht.

Kapitel 7

Einsatz und Evaluation des Prototyps

Um die prototypische Implementierung im Vergleich zu bestehenden Ansätzen evaluieren zu können, wird diese zunächst in eine exemplarische Anwendung integriert. Anhand dieser Kombination wird untersucht, inwiefern die spezifizierten Anforderungen im Rahmen des Prototyps umgesetzt wurden. Im Anschluss werden für nicht realisierte Anforderungen und weitere identifizierte Schwachstellen jeweils Lösungsansätze konzipiert und mitunter implementiert.

7.1 Einsatz des Prototyps

Die exemplarische Anwendung, welche der nachfolgenden Evaluation zugrunde liegt, orientiert sich architektonisch, technisch und optisch an einem realen Projekt der Agentur. Jene wurde auf das Wesentliche reduziert, um den Funktionsumfang der Anwendung übersichtlich zu halten. Die elementaren Elemente, insbesondere in Bezug auf den Prototyp, bleiben jedoch erhalten. Dadurch lässt sich die Implementierung in einer nahezu realen Umgebung untersuchen und bewerten.

Die Anwendung stellt ein Video-Portal dar, das themenspezifische Videos basierend auf den Nutzerberechtigungen auflistet. Die Aufstellung der einzelnen Beiträge lässt sich über eine Filterfunktion clientseitig beeinflussen (vgl. Abbildung 7.1). Das Modul *uXess* ist in diesem Zusammenhang für das Ein- und Ausblenden der einzelnen Filter sowie des „Logout“-Buttons in der Menüleiste verantwortlich. Da die einzelnen Filter beim Server angefragt werden, muss

den Daten jeweils die notwendigen Berechtigungen beigefügt werden. Diese jeweiligen Berechtigungen werden über die AngularJS-interne *Template Engine* ausgegeben. Der „Logout“-Button soll dagegen angezeigt werden, sobald dem Nutzer mindestens eine Rolle zugewiesen ist. Die Videos werden aus Sicherheitsgründen bereits vom Server gemäß der Nutzerberechtigungen gefiltert.

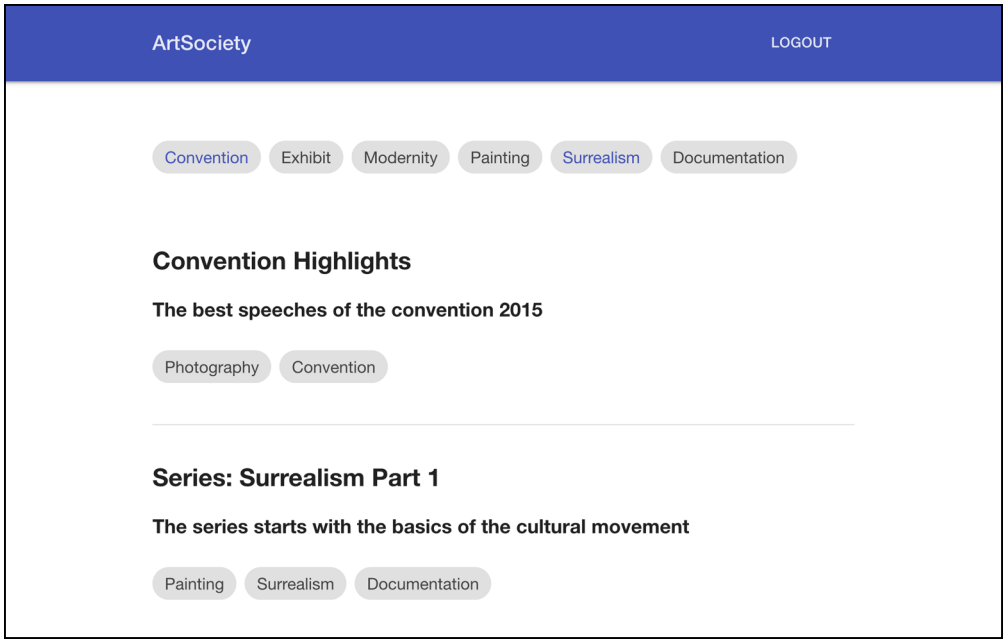

Abbildung 7.1: Screenshot der Evaluation zugrunde liegenden Anwendung

7.2 Prüfen der Anforderungen

Im Zuge der Evaluation werden die spezifizierten Anforderungen im Bezug auf die prototypische Implementierung überprüft. Dabei wird der Prototyp hinsichtlich seiner Funktionalität, Charakteristika, Architektur und Qualität näher betrachtet und anschließend bewertet.

7.2.1 Funktionale Anforderungen

Das rollenbasierte Ein- und Ausblenden von UI-Elementen erfolgt wie vorhergesehen, indem diese UI-Elemente vollständig aus dem DOM des HTML-Dokumentes entfernt werden. Auch das Wiedereinblenden der entsprechenden Elemente an korrekter Position verläuft reibungslos. Für den Prozess des Ein-

und Ausblendens wird zudem das Modul *ngAnimate* vollständig unterstützt, sodass die Anpassungen im UI individuell animiert werden können.

Entsprechende UI-Elemente werden mit der Direktive `uxs-if` versehen, die Auswahl von einem der drei zur Verfügung stehenden Autorisierungsmodi erfolgt über das HTML-Attribut `uxs-type`. Bei den Autorisierungsmodi handelt es sich um folgende Bedingungsprüfungen:

- Ist dem Nutzer eine der übergebenen Rollen zugewiesen?
- Ist dem Nutzer keine der übergebenen Rollen zugewiesen?
- Sind dem Nutzer alle übergebenen Rollen zugewiesen?

Durch den Einsatz von Ereignissen und *Event Listenern* wirkt sich eine Änderung der Nutzerberechtigungen unverzüglich auf die Darstellung der Elemente aus, sodass weder das Aktualisieren der Seite noch das Laden von *Template Partials* notwendig ist. Somit werden entsprechende Inhalte dynamisch ausgeblendet, sollte der Nutzer – beispielsweise aufgrund eines *Session Timeouts* – Berechtigungen verlieren.

Das frühzeitige Eingreifen des Prototyps in den Kompilierungsprozess wurde nicht realisiert. Dies bedeutet, dass ohne weitere Maßnahmen die UI-Elemente beim Laden der Seite kurz aufflackern, bevor diese aus dem DOM entfernt werden; die UX wird beeinträchtigt. Diese Anforderung wurde bewusst nicht umgesetzt. Die Gründe hierfür sind zum einen, dass der damit verbundene Aufwand nicht in Relation zum daraus resultierenden Nutzen steht. Zum anderen würde die Realisierung der Anforderung die Implementierung unverhältnismäßig verkomplizieren. Zudem stellt AngularJS die eigene Direktive `ng-cloak` zur Verfügung, welche speziell für diesen Zweck entwickelt wurde. In Kapitel 7.3.1 wird erläutert, wie unter Zuhilfenahme der Direktive `ng-cloak` die Einbußen hinsichtlich der UX reduziert werden können.

7.2.2 Nicht-funktionale und architektonische Anforderungen

Das Modul *uXess* besitzt als einzige Abhängigkeit *ngAnimate* zum Animieren von UI-Elementen (vgl. Kapitel 3.2.1). Abgesehen davon werden für den Produktiveinsatz keine weiteren Module benötigt. Weitere Abhängigkeiten sind als *Development Dependencies* einzustufen und für die Bewertung des Prototyps unmaßgeblich.

Die Trennung der Zuständigkeiten erfolgt akkurat, sodass diese in jeweils eigene Komponenten und Dateien aufgeteilt werden. Als Zuständigkeiten sind die Ma-

nipulation des UIs sowie die Handhabung von Rollen, Autorisierungsmodi und der eigenen Autorisierung zu nennen. Die Komponenten für die Handhabung der Modi und der Rollen lassen sich projektspezifisch konfigurieren.

Der Funktionsumfang des Moduls berücksichtigt keine weiteren Aspekte, sodass jener eindeutig definiert und kompakt gehalten wurde. Daraus resultiert ein guter Überblick über die bereitgestellten Schnittstellen und Funktionen, sodass sich der Prototyp schnell und intuitiv sowohl in bestehende als auch neue Anwendungen integrieren lässt. Bedingt durch die einfache Integration in Anwendung und die geringe Anzahl an Abhängigkeiten lässt sich die Implementierung als universell respektive allgemeingültig bezeichnen.

7.2.3 Qualitative Anforderungen

Eine verständliche Einrichtungsanleitung im *Markdown*-Format, welche diverse Aspekte wie Installation, Integration, Konfiguration, Schnittstellen und unterschiedliche Anwendungsmöglichkeiten abdeckt, erleichtert die Einrichtung und Weiterentwicklung. Zusätzlich wird eine Demo bereitgestellt, die ausgewählte Funktionen inklusive der dynamischen Modifikation veranschaulicht.

Die Funktionen und Schnittstellen des Prototyps sind mit einer umfangreichen Dokumentation versehen, welche die in der Konzeption definierten Bestandteile berücksichtigt. Durch die einheitliche Dokumentation soll es Dritten erleichtert werden, die Implementierung nachzuvollziehen, ohne sich mit den einzelnen Vorgängen innerhalb der Funktionen beschäftigen zu müssen. Der Quellcode selbst ist gemäß der Zuständigkeiten und in flachen Verzeichnisstrukturen gegliedert.

Im Zuge der Implementierung werden *Best Practices* und Designprinzipien eingehalten, indem insbesondere sich wiederholender Quellcode vermieden und die in der Konzeption definierte Nomenklatur angewandt wird. So werden zum einen IIFEs eingesetzt, um Gültigkeitsbereiche zu erstellen. Zum anderen wird die Implementierung durch Befolgen der in der Konzeption beschriebenen Prinzipien DRY, KISS und YAGNI schlank und übersichtlich gehalten.

Der Prototyp ist zudem mit Modul- und End-to-End-Tests der Frameworks *Jasmine* und *Protractor* versehen, welche vom *Test Runner Karma* verwaltet werden. Die Modultests decken dabei 99,4 Prozent der Implementierung ab, lediglich ein Quellcode-Zeile wird nicht getestet. Bei dieser Zeile handelt es sich um die Behandlung eines eventuellen Fehlers, der durch keine Testfall hervorgerufen werden kann. Die Oberflächentests helfen zusätzlich bei der Sicherstellung einer fehlerfreien Implementierung.

7.2.4 Identifizierte Defizite des Prototyps

Neben den in Kapitel 7.2.1 erläuterten Einbußen der UX, welche aus dem zu späten Eingriff in den Kompilierungsprozess resultieren, wird die Implementierung auf weitere Defizite untersucht. Dafür wird auf die zuvor erstellte Anwendung zurückgegriffen, um den realen Einsatz bestmöglich nachvollziehen zu können. Die identifizierten Defizite werden im Anschluss ausführlich erläutert.

Insbesondere beim „Logout“-Button der Anwendung, in welcher der Prototyp zum Einsatz kommt, wurden die nachfolgend beschriebenen Mängel ersichtlich. Dieses UI-Element soll wieder eingeblendet werden, sobald der Nutzer autorisiert und ihm mindestens eine Rolle zugewiesen ist. Dabei ist irrelevant welche konkreten Berechtigungen er besitzt. Demnach müssen der Direktive alle verfügbaren Rollen übergeben werden; in der exemplarischen Anwendung mit drei möglichen Rollen, ist der damit verbundene Aufwand überschaubar. Sollte ein System jedoch eine Vielzahl von Rollen verwenden, kann dies sowohl zu unübersichtlichen Templates als auch erhöhtem Aufwand führen. Hinzu kommt, dass entsprechende Stellen im *Template* gepflegt werden müssen, sollten weitere Rollen hinzukommen oder aus dem Repertoire entfernt werden.

Bei der Evaluation der prototypischen Implementierung wurden darüber hinaus keine weiteren Mängel identifiziert, sodass das zu späte Intervenieren und das umständliche Prüfen einer allgemeinen Autorisierung die bisher einzig bekannten Defizite darstellen.

7.2.5 Bewertung der prototypischen Implementierung

Das Modul deckt – wie die Tabelle 7.1 auf der nachfolgenden Seite verdeutlicht – nahezu alle spezifizierten Anforderungen ab und wird der Konzeption gerecht. Insbesondere ist dabei hervorzuheben, dass alle nicht-funktionalen und qualitativen Anforderungen umgesetzt wurden. Lediglich eine funktionale Spezifikation erfüllt der Prototyp nicht. Auch bei der Integration in eine reale Anwendung beweist sich das Modul und funktioniert zuverlässig.

Den positiven Aspekten stehen die beiden ermittelten Schwachpunkte, die nicht umgesetzte Anforderung sowie Mängel bei der Handhabung, gegenüber. Dies reduzieren die UX sowohl für Entwickler als auch Endanwender. Da es sich bei der Steigerung der UX um eines der Hauptziele der Implementierung handelt, sind die Schwachstellen zu beseitigen. Trotz vereinzelter Mängel erfüllt das Modul die in der Anforderungsspezifikation und Konzeption definierten Aspekte zufriedenstellend. Aus diesem Grund stellt die Implementierung eine passende Lösung für das gegebene Problem dar.

Tabelle 7.1: *uXess* – Realisierte Systemanforderungen

Priorität	**Anforderung**	**Realisiert**
	Rollenbasiertes Handling des UIs	
Obligatorisch	Ein-/Ausblenden der Elemente	✓
Optional	Manipulation des DOMs	✓
Optional	Unterstützung von Animationen	✓
Optional	Dynamische Modifikation	✓
Optional	Frühzeitige Intervention	
	Diverse Autorisierungsmodi	
Obligatorisch	Ist dem Nutzer eine der übergebenen Rollen zugewiesen?	✓
Optional	Ist dem Nutzer keine der übergebenen Rollen zugewiesen?	✓
Optional	Sind dem Nutzer alle übergebenen Rollen zugewiesen?	✓
	Allgemeingültigkeit & Modularität	
Obligatorisch	Hohe Universalität der Lösung	✓
Obligatorisch	Geringe Abhängigkeiten	✓
Obligatorisch	Trennung der Zuständigkeiten	✓
Obligatorisch	Einfache Integration in bestehende Systeme	✓
Optional	Fakultative Konfiguration	✓
Optional	Konzentrierter Funktionsumfang	✓
	Einfache Wart- und Testbarkeit	
Obligatorisch	Ausführliche Dokumentation	✓
Obligatorisch	Strukturierter Code	✓
Obligatorisch	Einhaltung von *Best Practices* und Designprinzipien	✓
Optional	Verständliche Einrichtungsanleitung	✓
Optional	Exemplarische Demo	✓
Optional	Hohe Testabdeckung	✓

7.3 Konzeption und Realisierung von Lösungen

Für die beiden identifizierten Schwachstellen des Prototyps, werden im Folgenden Lösungswege konzipiert und teilweise realisiert. Das Ziel dahinter ist es, die UX von Anwendungen weiterhin zu erhöhen, damit die Gedanke hinter der Implementierung bestmöglich erfüllt wird.

7.3.1 Meiden des Aufflackerns von UI-Elementen

Um das Aufflackern von UI-Elementen zu vermeiden, bevor alle entsprechenden *Expressions* und Direktiven durchlaufen sind, stellt AngularJS die Direktive `ng-cloak` zur Verfügung. Diese basiert auf CSS, indem dadurch alle damit versehenen UI-Elemente rein visuell ausgeblendet werden. Nach Beenden der Kompilierungsphase, entfernt sich die Direktive selbst vom Element, sodass die CSS-Regeln nicht weiter greifen.

Für die Verwendung von `ng-cloak` stehen diverse Wege zur Wahl. Zum einen lässt sich die Direktive spezifisch zu den einzelnen UI-Elementen hinzufügen, wodurch diese nach Durchlaufen der Kompilierungsphase spezifisch behandelt werden. Zum anderen lässt sich die Direktive global definieren, indem beispielsweise das Wurzelelement `body` mit ihr versehen wird. Dies hat zur Folge, dass die vollständige Anwendung erst sichtbar wird, sobald alle Direktiven ausgeführt und beendet wurden. Im Produktiveinsatz ist meist eine Kombination aus beiden Optionen notwendig, um sowohl das initiale Laden der Anwendung als auch neu geladene *Template Partials* abzudecken.

7.3.2 Komponente zur Handhabung einer Wildcard

Um die Überprüfung, ob dem Nutzer mindestens eine Rolle zugewiesen ist, zu erleichtern, wird die Implementierung um eine weitere Komponente erweitert. Diese Komponente soll die Handhabung einer *Wildcard* gewähren und wird durch einen Service vom Typ *Provider* realisiert, da die Möglichkeit zur Konfiguration bestehen soll.

Die Komponente besteht unter anderem aus einem *Getter* und *Setter*. Beim *Setter* wird der entgegengenommene Wert vor dem Speichern zunächst geparst. Hierfür wird eine weitere Funktion benötigt, die versucht, den Wert in eine Zeichenkette zu transformieren; sollte dies nicht möglich sein, wird stattdessen auf den definierten Defaultwert `*` zurückgegriffen. Alle Funktionen sind dabei öffentlich zugänglich.

Auf die bereitgestellten Schnittstellen wird von derjenigen Komponente zugegriffen, welche für die Autorisierung verantwortlich ist (vgl. Abbildung 7.2). Sollte der Direktive ein Wert übergeben werden, welche der definierten *Wildcard* entspricht, wird lediglich geprüft, ob dem Nutzer mindestens eine Rolle zugewiesen ist; der Autorisierungsmodus ist in diesem Fall zu vernachlässigen.

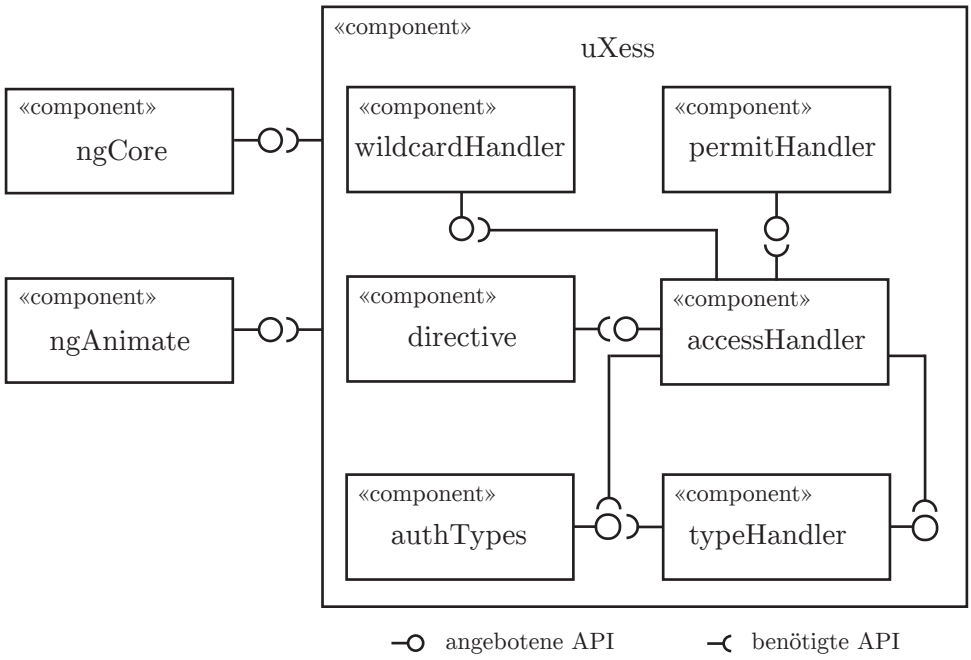

Abbildung 7.2: *uXess* – Erweitertes Komponentendiagramm

7.4 Zusammenfassung

In diesem Kapitel wurde zunächst die Anwendung beschrieben, in welche der Prototyp zur Evaluation integriert wird, und welche spezielle Rolle der Prototyp in dieser Anwendung einnimmt. Anschließend wurde die prototypische Implementierung unter Beachtung der spezifizierten Anforderungen untersucht, weitergehende Mängel ausfindig gemacht sowie die Implementierung bewertet. In einer abschließenden Konzeption wurden mögliche Lösungswege für die beiden identifizierten Defizite aufgezeigt und beschrieben. Auf Basis dessen erfolgte die Realisierung, wodurch eine weitere Komponente hinzugefügt wurde.

Kapitel 8

Reflexion und Ausblick

Die Inhalte dieser Arbeit werden im einleitenden Abstract zusammengefasst. In diesem Kapitel wird die Arbeit reflektiert und ein perspektivischer Ausblick hinsichtlich der weiteren Entwicklung des Moduls gewährt.

Im Zuge der Recherche nach Implementierungen, welche das gegebene Problem behandeln, wurden lediglich zwei AngularJS-Module identifiziert. Bei der Untersuchung dieser wurde ersichtlich, dass keines der beiden den spezifizierten Anforderungen gerecht wird. Erst im Verlauf der Konzeption und Implementierung des Prototyps wurden die Herausforderungen deutlich: das Entfernen der entsprechenden UI-Elemente aus dem DOM bedurfte einer korrespondierenden Referenz, um das Element an der korrekten Stelle wieder in den DOM integrieren zu können. Hinzu kam ein *Event Handling* zum gezielten Auslösen der Berechtigungsprüfung. Dabei durfte besonders der generische und modulare Charakter nicht vernachlässigt werden. Der Einsatz der Implementierung in einer realen Anwendung zeigte neben der korrekten und zuverlässigen Funktionsweise, dass der Prototyp Schwächen in Bezug auf die frühzeitige Intervention und die Handhabung der Rollen aufweist. Hierfür konnten jedoch unmittelbar im Anschluss Lösungsansätze konzipiert und realisiert werden. Die Behebung der Schwächen, das Erfüllen nahezu aller Anforderungen und die korrekte Funktionsweise sorgen dafür, dass die prototypische Implementierung schon jetzt in reale Projekte integriert werden kann.

Es bleibt die weitere Entwicklung des Moduls *uXess*, sowohl im Unternehmen als auch in der Open Source Community, abzuwarten. Insbesondere die große Community hinter AngularJS birgt großes Potential. Für eine hohe Verbreitung wird die Implementierung in der npm-Paketdatenbank sowie auf *GitHub* (`https://github.com/felixheck/uXess`) und *ngModules* [12] veröffentlicht.

Hinsichtlich der Weiterentwicklung sind neben Anpassungen auch Erweiterungen denkbar. Dabei sind Anpassungen der Implementierung insbesondere in Bezug auf Performanz, Qualität und Architektur möglich; eine Modifikation der Funktionalität hängt von den künftigen Erfahrungen der Entwickler im Umgang mit dem Prototyp ab. Als funktionale Erweiterungen sind alternative Möglichkeiten in der Handhabung von Rollen sowie eine REST-Komponente denkbar. Letztere könnte die Interaktion des Moduls mit einer RESTful API abstrahieren und somit vereinfachen.

Literaturverzeichnis

[1] C. J. Ihrig und A. Bretz, *Full Stack JavaScript Development with MEAN*, 1. Aufl. Melbourne, Australia: SitePoint, 2014.

[2] F. Monteiro, *Learning Single-page Web Application Development*, 1. Aufl. Birmingham, UK: Packt Publishing, 2014.

[3] J. Dickey, *Write Modern Web Apps with the MEAN Stack: Mongo, Express, AngularJS, and NodeJS*, 1. Aufl. San Francisco, USA: Peachpit Press, 2015.

[4] J. Cravens und T. Q. Brady, *Building Web Apps with Ember.js*, 1. Aufl. Sebastopol, USA: O'Reilly Media, 2014.

[5] S. Tilkov *et al.*, *REST und HTTP: Entwicklung und Integration nach dem Architekturstil des Web*, 3. Aufl. Heidelberg, Deutschland: dpunkt.verlag, 2015.

[6] K. Nagayama. (2015, 24. Oktober) *Deprecating our AJAX crawling scheme*. Google Webmaster Central Blog. [Online]. Verfügbar: http://googlewebmastercentral.blogspot.de/2015/10/deprecating-our-ajax-crawling-scheme.html (Zugriff: 19. Oktober 2015)

[7] P. Far. (2014, 27. Oktober) *Updating our technical Webmaster Guidelines*. Google Webmaster Central Blog. [Online]. Verfügbar: http://googlewebmastercentral.blogspot.de/2014/10/updating-our-technical-webmaster.html (Zugriff: 19. Oktober 2015)

[8] P. Ackermann, *Professionell entwickeln mit JavaScript: Design, Pattern, Praxistipps*, 1. Aufl. Bonn, Deutschland: Rheinwerk Verlag, 2015.

[9] I. Hickson und Google, „Web Storage (Second Edition)," W3C, Candidate Recommendation, 09. Juni 2015. [Online]. Verfügbar: http://www.w3.org/TR/2015/CR-webstorage-20150609/ (Zugriff: 21. Oktober 2015)

[10] S. Panda, *AngularJS: From Novice to Ninja*, 1. Aufl. Melbourne, Australia: SitePoint, 2014.

[11] Google. *AngularJS*. [Online]. Verfügbar: https://github.com/angular/angular.js (Zugriff: 06. Oktober 2015)

[12] J. Hoskins. *Angular Modules*. [Online]. Verfügbar: http://ngmodules.org (Zugriff: 06. Oktober 2015)

[13] R. Kiran. (2015, 27. Januar) *Writing AngularJS Apps Using ES6*. [Online]. Verfügbar: http://www.sitepoint.com/writing-angularjs-apps-using-es6/ (Zugriff: 06. Oktober 2015)

[14] P. Panda. (2015, 02. März) *What's New in AngularJS 2.0*. [Online]. Verfügbar: http://www.sitepoint.com/whats-new-in-angularjs-2/ (Zugriff: 07. Oktober 2015)

[15] Node.js Foundation. *Node.js*. [Online]. Verfügbar: https://nodejs.org/ (Zugriff: 19. November 2015)

[16] E. Brown, *Web Developmeht with Node and Express*, 1. Aufl. Sebastopol, USA: O'Reilly Media, 2014.

[17] A. Q. Haviv, *MEAN Web Development*, 1. Aufl. Birmingham, UK: Packt Publishing, 2014.

[18] npm. *Node Package Manager*. [Online]. Verfügbar: https://www.npmjs.com/ (Zugriff: 04. November 2015)

[19] A. Beletsky. (2015, 02. April) *NPM for Everything*. [Online]. Verfügbar: http://beletsky.net/2015/04/npm-for-everything.html (Zugriff: 20. November 2015)

[20] G. Elke. (2014, 19. Juni) *Ist Node.js ein Superheld?* [Online]. Verfügbar: https://blog.codecentric.de/2014/06/ist-node-js-ein-superheld/ (Zugriff: 21. November 2015)

[21] R. Fielding, „Architectural Styles and the Design of Network-based Software Architectures," Ph.D. Dissertation, UC, Irvine, USA, 2000.

[22] L. Richardson und M. Amundsen, *RESTful Web APIs*, 1. Aufl. Sebastopol, USA: O'Reilly Media, 2013.

[23] L. Richardson und S. Ruby, *RESTful Web Services*, 1. Aufl. Sebastopol, USA: O'Reilly Media, 2007.

[24] M. Massé, *REST API Design Rulebook*, 1. Aufl. Sebastopol, USA: O'Reilly Media, 2011.

[25] T. Berners-Lee *et al.*, „Uniform Resource Identifier (URI): Generic Syntax," IETF, RFC 3986, Januar 2005. [Online]. Verfügbar: https://www.ietf.org/rfc/rfc3986.txt (Zugriff: 13. Oktober 2015)

[26] R. Fielding *et al.*, „Hypertext Transfer Protocol – HTTP/1.1," IETF, RFC 2616, Juni 1999. [Online]. Verfügbar: https://www.ietf.org/rfc/rfc2616.txt (Zugriff: 13. Oktober 2015)

[27] L. Dusseault *et al.*, „PATCH Method for HTTP," IETF, RFC 5789, März 2010. [Online]. Verfügbar: https://www.ietf.org/rfc/rfc5789.txt (Zugriff: 13. Oktober 2015)

[28] S. Purewal, *Learning Web App Development*, 1. Aufl. Sebastopol, USA: O'Reilly Media, 2014.

[29] L. Bassett, *Introduction to JavaScript Object Notation*, 1. Aufl. Sebastopol, USA: O'Reilly Media, 2015.

[30] T. Bray und Google, „The JavaScript Object Notation (JSON) Data Interchange Format," IETF, RFC 7159, März 2014. [Online]. Verfügbar: https://www.ietf.org/rfc/rfc7159.txt (Zugriff: 15. Oktober 2015)

[31] Google. *ngAnimate*. [Online]. Verfügbar: https://docs.angularjs.org/api/ngAnimate (Zugriff: 03. November 2015)

[32] S. Springer, „AngularJS done right: Best practices im Umgang mit AngularJS," *PHP Magazin*, Vol. 6, S. 65–70, Oktober/November 2015.

[33] E. Elliott, *Programming JavaScript Applications*, 1. Aufl. Sebastopol, USA: O'Reilly Media, 2014.

[34] M. Mathews. *@use JSDoc*. [Online]. Verfügbar: http://usejsdoc.org/ (Zugriff: 05. November 2015)

[35] lykmapipo. *ngAA*. [Online]. Verfügbar: https://github.com/lykmapipo/ngAA (Zugriff: 26. Oktober 2015)

[36] Auth0. *JWT: JSON Web Tokens*. [Online]. Verfügbar: http://jwt.io/ (Zugriff: 03. November 2015)

[37] AngularUI. *AngularUI Router*. [Online]. Verfügbar: https://angular-ui.github.io/ (Zugriff: 03. November 2015)

[38] G. K. Lee. *ngStorage*. [Online]. Verfügbar: https://github.com/gsklee/ngStorage (Zugriff: 03. November 2015)

[39] Twitter. *Bower: A package manager for the web*. [Online]. Verfügbar: http://bower.io/ (Zugriff: 03. November 2015)

[40] B. Alman. *Grunt: The JavaScript Task Runner*. [Online]. Verfügbar: http://gruntjs.com/ (Zugriff: 03. November 2015)

[41] F. Ziegelmayer. *Karma*. [Online]. Verfügbar: http://karma-runner.github.io/ (Zugriff: 03. November 2015)

[42] T. Holowaychuk. *Mocha: simple, flexibel, fun*. [Online]. Verfügbar: https://mochajs.org/ (Zugriff: 03. November 2015)

[43] Jake Luer. *Chai Assertion Library*. [Online]. Verfügbar: http://chaijs.com/ (Zugriff: 03. November 2015)

[44] Google. *Best Practices*. [Online]. Verfügbar: https://github.com/angular/angular.js/wiki/Best-Practices (Zugriff: 03. November 2015)

[45] T. Motto. (2014, 23. Juli) *Opinionated AngularJS styleguide for teams*. [Online]. Verfügbar: http://toddmotto.com/opinionated-angular-js-styleguide-for-teams/ (Zugriff: 03. November 2015)

[46] Apache Software Foundation. *Apache Shiro*. [Online]. Verfügbar: http://shiro.apache.org/ (Zugriff: 04. November 2015)

[47] maybenull. *angular-authz: Authrorization for AngularJS applications*. [Online]. Verfügbar: http://maybenull.github.io/angular-authz/ (Zugriff: 04. November 2015)

[48] Pivotal Labs. *Jasmine: A JavaScript Testing Framework*. [Online]. Verfügbar: https://github.com/jasmine/jasmine (Zugriff: 04. November 2015)

[49] Airbnb. *Airbnb JavaScript Style Guide*. [Online]. Verfügbar: https://github.com/airbnb/javascript/tree/master/es5 (Zugriff: 04. November 2015)

[50] J. Samwell. (2014, 04. Juli) *URL Route Authorization and Security in Angular*. [Online]. Verfügbar: http://jonsamwell.com/url-route-authorization-and-security-in-angular/ (Zugriff: 26. Oktober 2015)

[51] Twitter. *Bootstrap*. [Online]. Verfügbar: http://getbootstrap.com/ (Zugriff: 12. November 2015)

[52] N. Khedr. (2013, 25. November) *How to do Authorization and Role based permissions in AngularJs*. [Online]. Verfüg-

bar: http://nadeemkhedr.com/how-to-do-authorization-and-role-based-permissions-in-angularjs/ (Zugriff: 26. Oktober 2015)

[53] The jQuery Foundation. *jQuery: write less, do more.* [Online]. Verfügbar: https://jquery.com/ (Zugriff: 12. November 2015)

[54] G. Hengeveld. (2014, 12. März) *Techniques for authentication in AngularJS applications.* [Online]. Verfügbar: https://medium.com/opinionated-angularjs/techniques-for-authentication-in-angularjs-applications-7bbf0346acec (Zugriff: 02. November 2015)

[55] B. Dayley, *Learning AngularJS*, 1. Aufl. Boston, USA: Addison-Wesley, 2014.

[56] A. Lerner, *ng-book: The Complete Book on AngularJS*, 1. Aufl. San Francisco, USA: Fullstack.io, 2013.

[57] N. Heiner. (2015, 13. Feburar) *Why my team uses npm instead of bower.* [Online]. Verfügbar: https://medium.com/@nickheiner/why-my-team-uses-npm-instead-of-bower-eecfe1b9afcb (Zugriff: 19. November 2015)

[58] K. Cirkel. (2014, 30. Oktober) *Why we should stop using Grunt and Gulp.* [Online]. Verfügbar: http://blog.keithcirkel.co.uk/why-we-should-stop-using-grunt/ (Zugriff: 20. November 2015)

[59] K. Cirkel. (2014, 09. Dezember) *How to Use npm as a Build Tool.* [Online]. Verfügbar: http://blog.keithcirkel.co.uk/how-to-use-npm-as-a-build-tool/ (Zugriff: 20. November 2015)

[60] Mozilla. *Firefox.* [Online]. Verfügbar: https://www.mozilla.org/de/firefox/new/ (Zugriff: 12. Dezember 2015)

[61] Google. *Chrome.* [Online]. Verfügbar: https://www.google.de/chrome/browser/desktop/ (Zugriff: 12. Dezember 2015)

[62] Google. *ngMock.* [Online]. Verfügbar: https://docs.angularjs.org/api/ngMock (Zugriff: 12. Dezember 2015)

[63] Yahoo. *Istanbul.* [Online]. Verfügbar: https://github.com/gotwarlost/istanbul (Zugriff: 12. Dezember 2015)

[64] Google. *Protractor: end to end testing for AngularJS.* [Online]. Verfügbar: http://www.protractortest.org/ (Zugriff: 12. Dezember 2015)

[65] Selenium. *SeleniumHQ: Browser Automation*. [Online]. Verfügbar: http://www.seleniumhq.org/ (Zugriff: 12. Dezember 2015)

[66] Google. *Internet Explorer Compatibility*. [Online]. Verfügbar: https://docs.angularjs.org/guide/ie (Zugriff: 11. Januar 2016)

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und dabei nur die angegebenen Quellen und Hilfsmittel verwendet, sowie wörtliche Zitate und Paraphrasen als solche gekennzeichnet habe.

Die Arbeit wurde bisher weder einem anderen Prüfungsamt in gleicher oder vergleichbarer Form vorgelegt, noch veröffentlicht. Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Konsequenzen zur Folge haben wird.

_____

Felix Heck Offenburg, 15. Februar 2016