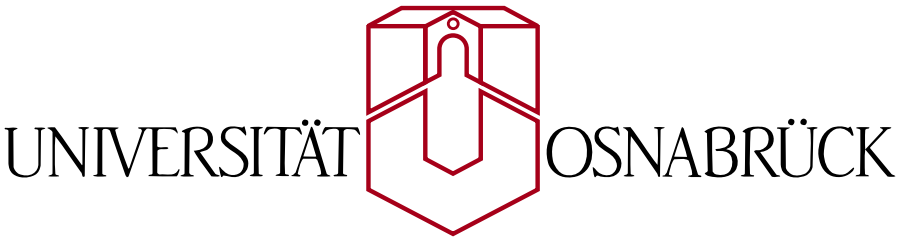
UNIVERSITY OF OSNABRÜCK



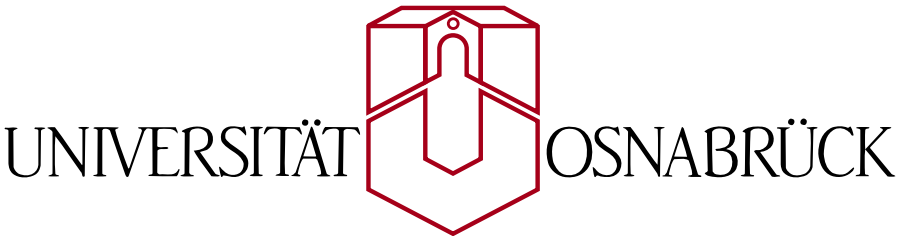
Department of Cognitive Science
Department of Computer Science

BACHELOR'S THESIS

CONCEPTUALIZATION AND IMPLEMENTATION OF A WEB-APPLICATION FOR INTERACTIVE NAVIGATION OF GERMAN FEDERAL LAW

ALEXANDER HÖRETH

12. September 2016
Cognitive Science, Bachelor of Science
Primary supervisor: Prof. Dr. Oliver Vornberger
Secondary supervisor: Dr. phil. Tobias Thelen



Institut für Kognititionswissenschaft
Institut für Informatik

BACHELORARBEIT

KONZEPTUALISIERUNG UND ENTWICKLUNG EINER WEB-APPLIKATION ZUR INTERAKTIVEN NAVIGATION DEUTSCHER BUNDESGESETZE

ALEXANDER HÖRETH

12. September 2016
Cognitive Science, Bachelor of Science
Erstprüfer: Prof. Dr. Oliver Vornberger
Zweitprüfer: Dr. phil. Tobias Thelen

The logo on the preceding pages is meant for the reader to identify the organization University of Osnabrück, a subject of public interest. This document is no official document issued by the University of Osnabrück.

ABSTRACT

The German legal system is in its core very conservative and only slowly advancing. Although there have been some modernization attempts, they have not proven too successful. Most of the work in courts, law offices and legal studies still rely primarily on analogue approaches. Still – there have been some promising signals pointing to an upcoming paradigm shift. This thesis investigates the upcoming opportunities for digital solutions in the juridical field. Specifically a software service set out to rival traditional collections and commentaries of legal documents is being conceptualized and a minimal viable product in form of a web application developed.

ZUSAMMENFASSUNG

Das Rechtswesen ist seit jeher ein eher konservativer Bereich und tut sich schwer aus alten Mustern auszubrechen. Trotz mehrerer zögerlicher Vorstöße ins digitale Zeitalter findet ein großer Teil der Arbeit in Kanzleien, an Gerichten und im Jurastudium weiterhin analog statt. Allerdings zeigen sich auch Perspektiven auf einen Paradigmenwechsel. Diese Arbeit beschäftigt sich mit der sich ergebenden Chance für digitale Lösungen im juristischen Bereich. Speziell wird eine Softwaredienstleistung als Konkurrenzprodukt zu kurzfristig traditionellen Gesetzessammlungen und langfristig auch Gesetzeskommentaren konzeptioniert und eine initiale Version einer Webapplikation zum Testen des Interesses im Markt entwickelt.

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

LISTINGS

1

EINLEITUNG

Die vorliegende Arbeit beschäftigt sich mit der Konzeptualisierung und Entwicklung einer Webapplikation zur interaktiven Navigation deutscher Bundesgesetze.

Zu Beginn wird hierfür in Abschnitt 2 der als Zielgruppe bestimmte Markt und die Relevanz einer durch eine solche Applikation angebotenen Dienstleistung analysiert. Hierbei werden besondere Anforderungen und nötige Alleinstellungsmerkmale im Vergleich zu bereits vorhandenen Konkurrenzprodukten hervorgehoben. Dabei wird auch die diese Arbeit übersteigende größere Zielsetzung dargestellt: die hier entwickelte Applikation dient zum Ausloten des am Markt erwarteten Interesses und als Grundlage für eine umfangreichere Softwaredienstleistung.

Im darauf folgenden Abschnitt 3 werden bei der Implementierung der Applikation verfolgte Paradigmen und verwendete Technologien diskutiert. Hierbei werden Entscheidungen wiederum auch mit Blick auf die langfristige Ausrichtung der Dienstleistung bezogen. Die entwickelte Webapplikation soll so nur die Grundlage einer ersten Version des Produktes sein und gute Erweiterungsmöglichkeiten in Bezug auf Umfang als auch hin zu nativen Mobilapplikationen bieten. Teil davon ist zum Beispiel die Trennung der Applikation in einen Client, die Webseite mit der der Nutzer interagiert, und einen API-Server, die Schnittstelle von welcher der Client Daten bezieht. In diesem und dem folgenden Abschnitt wird grundlegendes Wissen über Softwarearchitektur und -entwicklung sowie über Webtechnologien vorausgesetzt. Es werden primär Besonderheiten der verwendeten Paradigmen und Technologien und insbesondere deren Unterscheidungsmerkmale zu oft vorherrschenden Alternativen diskutiert.

Abschnitt 4 beschreibt daraufhin die konkrete Implementation der Applikation. Hier wird neben einer generellen Übersicht über jeweils die Struktur des API-Servers und der Clientapplikation beispielhaft eine einzelne Ansicht der Applikation und das Ineinandergreifen der für sie benötigten Bestandteile beschrieben. Außerdem wird beispielhaft auf eine sich bei der Entwicklung gezeigte Herausforderung und ihre Bewältigung eingegangen.

Abschließend wird in Abschnitt 5 ein Rück- sowie Ausblick geboten. Auf der Softwareseite werden so beispielsweise die zuvor getroffenen

Entscheidungen bezüglich der eingesetzten Werkzeuge auf Grundlage der neu gesammelten Erfahrungen diskutiert. Neben im Rahmen eines solchen Projektes auftretenden auch kritischen Überlegungen bezüglich der Relevanz des Gesamtproduktes wird zum Ende auch eine positive Perspektive zum weiteren Verlauf dargelegt.

Hinweis zu gendergerechter Sprache: Bei allen Bezeichnungen von Personengruppen meint, wenn nicht explizit anders angemerkt, die gewählte Formulierung beide Geschlechter.

2

KONZEPTUALISIERUNG

Im folgenden Abschnitt wird der Zustand des Marktes im Allgemeinen für juristische Softwaredienstleistungen und im Speziellen für ein digitales Konkurrenzprodukt zu klassischen Gesetzbüchern analysiert. Abschnitt 2.1 geht dabei auf den sich im Rechtswesen generell anbahnenden digitalen Umschwung und die sich dadurch ergebende Chance und Notwendigkeit für moderne Softwarelösungen ein. Anschließend beschäftigt sich Abschnitt 2.2 mit dem konkreten traditionellen Markt für Gesetzestexte und -kommentare, bevor in Abschnitt 2.4 der aktuelle Zustand des digitalen Marktes für solche und die in ihm bereits etablierten und aktuell aufkommenden Konkurrenzprodukte betrachtet werden. Zum Abschluss dieses Kapitels wird in Abschnitt 2.5 die Zielsetzung für eine neue digitale Dienstleistung in diesem Bereich dargelegt und der Rahmen des als Teil dieser Arbeit umgesetzten Softwareprojektes konkretisiert.

2.1 DIE DIGITALE REVOLUTION DES RECHTSWESENS

Für den sich anbahnenden Paradigmenwechsel hin zu einer verstärkt digitalen Arbeitsweise im Rechtswesen gibt es viele Anhaltspunkte. Im Folgenden werden exemplarisch Vorstöße von drei unterschiedlichen Akteuren in Richtung einer elektronischen rechtlichen Kommunikation aufgeführt: von Seiten des Gesetzgebers, der Bundesrechtsanwaltskammer (BRAK) und, beispielhaft an einer einzelnen, modernen Kanzleien.

Die deutsche Bundesregierung hat im letzten Jahrzehnt verstärkt Interesse gezeigt, dass moderne digitale Werkzeuge im Rechtswesen Einzug halten. Einerseits wird dies durch Gesetzesänderungen deutlich, welche den Einsatz digitaler Hilfsmittel dort legalisieren, wo er zuvor undenkbar war. Andererseits durch den Versuch, die Realisierung von geeigneten Softwarelösungen durch neue Gesetze gewissermaßen zu erzwingen.

Ein Beispiel hierfür ist unter anderem die Kommunikation im Rechtsverkehr zwischen Kanzleien und Gerichten: Bereits seit 2001 besteht das *Zustellungsreformgesetz* (Deutscher Bundestag, 2001) und seit 2005 das *Justizkommunikationsgesetz* (Deutscher Bundestag, 2005). Ersteres legte bereits sehr früh den Grundstein für kleinere Pilotprojekte zum

Einsatz elektronischer Kommunikationsverfahren an einzelnen Gerichten. Das Justizkommunikationsgesetz hingegen sollte diese für die breite Masse an Juristen verfügbar machen. Durch die einerseits hohen Anforderungen an Datenschutz und Privatsphäre sowie die Notwendigkeit der Rechtsverbindlichkeit (z.B. durch digitale Signaturen) und den anderseits fehlenden bundesweiten Standards und die daraus resultierende hohe Einstiegshürde für potenzielle Nutzer hielt sich die Adoptionsrate stark in Grenzen.

Um dem entgegen zu wirken, wurde ab 2004 vom Bundesverwaltungsgericht und dem Bundesamt für Sicherheit in der Informationstechnik das elektronische Gerichts- und Verwaltungspostfach (EGVP) entwickelt. Dieses sollte auf Grundlage der neuen rechtlichen Möglichkeiten erstmals einen Standard schaffen, welcher die digitale Rechtskommunikation einer breiteren Masse zur Verfügung stellt. Auch hier führten allerdings rechtliche Hindernisse wie die Notwendigkeit einer vorhergehenden individuellen Registrierung der Gerichte für das Verfahren und auch die Art der Implementierung zu Problemen. Obwohl das EGVP als eine Standardisierung für rechtliche Kommunikation geplant war, orientiert es sich selbst nicht an technologischen Standards, sondern setzte auf proprietäre Software und zeigte starke Mängel in Bezug auf die Anwenderfreundlichkeit (Justizverwaltungen des Bundes und der Länder & Berufskammern und -verbände der Rechtsanwälte und Notare, 2007).

Erst 2013 folgte mit dem *Gesetz zur Förderung des elektronischen Rechtsverkehrs mit den Gerichten* (Deutscher Bundestag, 2013) der nächste Vorstoß: Der Gesetzgeber verpflichtete hierin die BRAK zur Einrichtung des sogenannten *besonderen elektronischen Anwaltspostfaches* (beA) für jeden in Deutschland zugelassenen Rechtsanwalt. Das beA soll im Großen und Ganzen nichts anderes zur Verfügung stellen als verschlüsselte E-Mail-Kommunikation zwischen Kanzleien und Gerichten. Geplagt von Problemen wurde das gesetzlich festgelegte Ziel die Plattform bis zum 01.01.2016 fertigzustellen wegen „nicht ausreichender Qualität“ der Software zuerst verfehlt (Bundesrechtsanwaltskammer, 2015a) und dann, kurz vor der für 10 Monate später angesetzten Fertigstellung im zweiten Anlauf (Bundesrechtsanwaltskammer, 2016b), durch ein einstweiliges Verfahren erneut gestoppt: Zwei Rechtsanwälte hatten beantragt, dass ihr besonderes elektronisches Anwaltspostfach (beA) nur mit ihrer ausdrücklichen Zustimmung freigeschaltet werde. Laut BRAK ist diese individuelle Freischaltung allerdings technisch nicht möglich, womit sich die Einrichtung des beAs weiter verzögert (Bundesrechtsanwaltskammer, 2016a).

Obwohl der Werdegang des beAs nicht unproblematisch ist, zeigt es zusätzlich eine andere Seite: Das Interesse an der Umsetzung des beAs geht klar auch von der BRAK und damit den Rechtsanwälten aus. Schon vor dem Beginn der Entwicklung des Postfaches bemühte sich die BRAK um eine Stärkung der Akzeptanz des elektronischen Rechtsverkehrs – mit vielversprechenden Hinweisen auf die Zielsetzung des

Postfaches als eine benutzerfreundliche, an bereits etablierten Standards orientierte Plattform (Bundesrechtsanwaltskammer, 2008). Diese Forderungen galten zu Beginn noch einer Weiterentwicklung des EGVP, resultierten endgültig aber in der Entwicklung des beA.

Ein dritter Akteur, welcher eine wichtige Rolle bei einem solchen Paradigmenwechsel hin zu einer verstärkt elektronischen Kommunikation im Rechtswesen spielt, sind die Kanzleien. Obwohl sie ein immenses wirtschaftliches Interesse an einer breiten Verfügbarkeit des elektronischen Rechtsverkehrs haben, sind sie an seiner Realisierung primär durch die BRAK beteiligt. Kanzleiintern hingegen kann die elektronische Kommunikation von ihnen selbstständig umgesetzt werden. Aus persönlichen Unterhaltungen mit Rechtsanwälten der multinational agierenden Kanzlei *Osborne Clarke* hat sich ergeben, dass der Einsatz moderner Hilfsmittel zur Kommunikation bei vielen Kanzleien nicht bei E-Mails stehen geblieben ist. Vielfach wird auch schon, ähnlich zu modernen Technologieunternehmen, auf erst in den letzten Jahren aufgekommene Kommunikationsplattformen wie *Slack* gesetzt. Dies deutet auf einen immensen Innovationswillen hin – solange die Innovation mit einer Effizienzsteigerung einhergeht.

2.2 GESETZBÜCHER UND -KOMMENTARE

Neben der weiterhin fast ausschließlich papiergestützten rechtlichen Kommunikation stechen besonders die *roten Bücher*, offiziell *Schönfelder – Deutsche Gesetze*, des Beck-Verlages ins Auge: Mit einer Masse von fast 2,5 Kilogramm gilt es als das Standardwerk des deutschen Rechtswesens. Da das deutsche Recht kein stehendes Konstrukt ist, sondern laufend Änderungen unterliegt, erscheinen drei bis vier Mal jährlich sogenannte *Ergänzungslieferungen*: Nach dem kostenpflichtigen Erwerb gilt es, diese in das als Loseblattsammlung konzipierte Ursprungswerk in Kleinstarbeit manuell einzusortieren. Dabei ist der Schönfelder nur eines von vielen ähnlich umfassenden und auf aktuellem Stand zu haltenden Werken. Aufgrund der Notwendigkeit dieser regelmäßigen Aktualisierung bewirbt der Beck Verlag die von ihm vertriebenen Gesetzessammlungen primär in Kombination mit Abonnements des hauseigenen *Aktualisierungsservices*.[1] Zu Marketingzwecken empfiehlt der Verlag angehenden Juristen mit dieser und ähnlichen Gesetzessammlungen „einen Bund fürs Leben zu schließen [...] und die Beziehung als Freundschaft mittels Kommunikation und Interesse aufrecht und lebendig zu gestalten“ (Talkner, 2015).

Die Zielgruppe für diesen Kassenschlager ist schlichtweg jeder, der mit dem Rechtswesen zu tun hat: das sind bundesweit ca. 165.000 aktive Rechtsanwälte (Bundesrechtsanwaltskammer, 2015b), über 25.000 Richter und Staatsanwälte im öffentlichen Dienst (Bundesamt

1 Quelle: *Schönfelder* im Beck Shop, beck-shop.de/productview.aspx?product=2205, Stand 01/2016.

für Justiz, 2013) und über 70.000 weitere Juristen, welche anderweitige juristische Tätigkeiten im öffentlichen Dienst und der freien Wirtschaft ausüben (vergleiche Abbildung 1 aus Bundesagentur für Arbeit (2016)). Außerdem gibt es aktuell über 100.000 Jurastudenten (Wissenschaftsrat, 2012). In all diesen Bereichen ist die Tendenz weiterhin steigend (Bundesagentur für Arbeit, 2016).

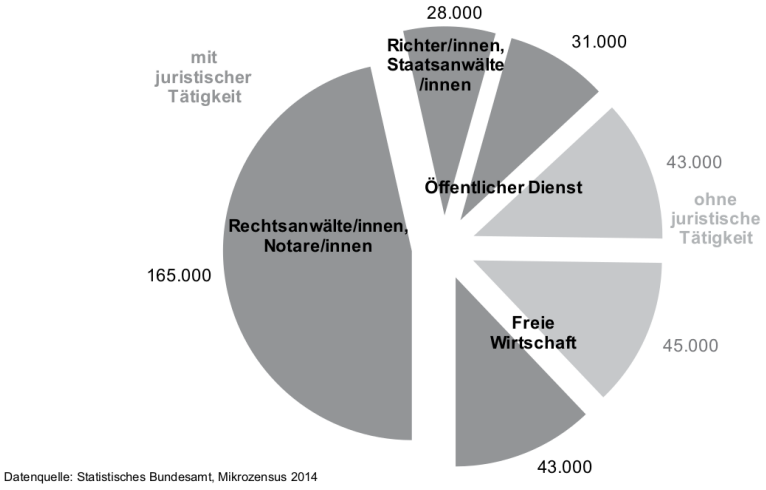


Abbildung 1: Erwerbstätige mit juristischer Ausbildung

Beispielhaft ergeben sich aus diesen Zahlen alleine durch die von Land und Bund beschäftigten Richter und Anwälte jährliche Kosten von über 1 Millionen Euro für den öffentlichen Dienst. Dabei ist wichtig zu bemerken, dass die für diese Schätzung angenommenen Werte mit pro Jurist einer Gesetzessammlung mit je drei Ergänzungslieferungen pro Jahr zu je 15 € niedrig angesetzt sind.[2] Ausübende Juristen sind meist auf drei oder mehr solcher Werke angewiesen.

Da die eigentlichen Normen eines Gesetzes, um möglichst viele Alltagssituationen abzudecken, meist abstrakt gehalten sind, ist ihre Auslegung für Juristen oft schwierig. Um diesem Problem entgegen zu wirken, haben sich bereits früh sogenannte Gesetzeskommentare etabliert und nehmen großen Einfluss auf das Rechtswesen (Henne, 2006). Sie beschäftigen sich mit der Erläuterung und Interpretation der eigentlichen Gesetze in Bezug auf konkrete Gegebenheiten und verwenden gerichtliche Entscheidungen und dem Gesetz vorhergegangene Regierungsentwürfe als Referenzen. In ihrem Umfang übersteigen diese meist die Gesetze, zu welchen sie Stellung beziehen, und sind für kaum einen Juristen vernachlässigbar (Der Spiegel, 1981).

2 Quelle: Verschiedene *Ergänzungslieferungen* gelistet im Beck Shop, z.B. Stand 08/2016, beck-shop.de/productview.aspx?product=16454788, Stand 08/2016.

2.3 GESETZESTEXT – OPEN DATA?

Die rechtliche Grundlage für die freie Verwendung von Gesetzen ist durch das deutsche Urheberrecht gegeben. Die so geartete Rechtslage wurde im Rahmen dieses Projektes auch anwaltlich, speziell in Bezug auf eine eventuell auch kommerzielle Nutzung, bestätigt. Gleiches gilt auch für Rechtsprechungen durch öffentliche Gerichte.

> Gesetze, Verordnungen, amtliche Erlasse und Bekanntmachungen sowie Entscheidungen und amtlich verfaßte Leitsätze zu Entscheidungen genießen keinen urheberrechtlichen Schutz.
>
> — Urheberrechtsgesetz (UrhG), § 5, Satz 1

Wo sich die Gesetzgebung auf den ersten Blick ganz im Sinne von Open Data und Creative Commons erst einmal hervorragend anhört, erweist sie sich auf den zweiten als nur unzureichend umgesetzt: Obwohl Gesetze und Urteile rechtlich frei von Urheberrechten sind, ist es nicht möglich, diese ohne Umweg über private Anbieter mit eigenen wirtschaftlichen Interessen zu beziehen.

Als optimale Quelle für diesen Datensatz würde initial man das vom Bundesministerium der Justiz und für Verbraucherschutz (BMJV) herausgegebene Bundesgesetzblatt (BGBL) annehmen. Es dient der verpflichtenden Verkündung aller Bundesgesetze, welche erst durch eben diese Veröffentlichung in Kraft treten können (Grundgesetz, Artikel 82). Zwar ist das BMJV Herausgeber des BGBL, allerdings wird der Vertrieb durch die 2006 vollständig privatisierte *Bundesanzeiger Verlag GmbH* vertrieben (Der Spiegel, 2006). Durch einen kostenpflichtigen Abonnentenzugang und den Vertrieb der Papierversion verdient der Verlag an dem Blatt. Die freie Weiterverwendung der veröffentlichten Gesetze schränkt der Verlag einmal im kostenlos zugänglichen *Bürgerzugang* durch technische Mittel[3] und im *Abonnentenzugang* mutmaßlich durch seine AGB ein (Bundesanzeiger Verlag GmbH, o. J.). Eine Klärung der Tragweite der AGB und der Rechtmäßigkeit solcher Einschränkungen würde einen größeren, dem Umfang dieser Arbeit unangemessenen, rechtlichen Aufwand mit sich bringen.

Die zweite eng mit dem Bund verknüpfte mutmaßlich freie Quelle für Gesetze ist die Webseite *gesetze-im-internet.de*. Von der Juris GmbH betrieben, an welcher der Bund 50,1% Anteile hält, gibt es hier die Möglichkeit die aktuelle Version der Gesetze einzusehen. Zusätzlich werden alle Gesetze auch im maschinell besser konsumier-

3 Ausschnitt aus *Fragen & Antworten* von bgbl.de, abgerufen 08/2016:

> Wie dürfen die Daten aus BGBl. Online weiterverwendet werden?
>
> Die Version im Bürgerzugang ist gegen Weiterverarbeitung geschützt. Die entgeltliche Version ermöglicht es Ihnen, Textausschnitte zu markieren und mittels „copy & paste“ in andere Programme einzufügen und entsprechend unserer AGB [...] für das Online-Abonnement weiter zu verarbeiten.

baren XML-Dateiformat zur Verfügung gestellt. In Rücksprache mit dem BMJV wurde zu diesen Daten bestätigt, dass sie vollständig für die Weiterverwendung durch Dritte auch für gegebenenfalls kommerzielle Unterfangen freigegeben sind. Auch hierbei ist wiederum als problematisch zu betrachten, dass die Juris GmbH ein wirtschaftliches Unternehmen ist und so zum Beispiel für den Zugriff auf überholte Versionen der Gesetzestexte ein Abonnement (mit ungewissen Einschränkungen) notwendig ist. Allerdings: Für das konkrete Ziel dieser Arbeit sind über *gesetze-im-internet.de* alle notwendigen Daten verfügbar.

Ähnlich ist die Situation bei eigentlich laut Gesetz urheberrechtsfreien Gerichtsentscheidungen. Zwar sind diese nicht für das konkrete Ziel der im Rahmen dieser Arbeit entwickelten Software notwendig, wären aber ein erster Schnittpunkt für eine mögliche Erweiterung. Das BMJV stellt auch hier zumindest „ausgewählte Entscheidungen des Bundesverfassungsgerichts, der obersten Gerichtshöfe des Bundes sowie des Bundespatentgerichts“ (Bundesministerium der Justiz und für Verbraucherschutz, o. J.) in Zusammenarbeit mit der Juris GmbH zur privaten und kommerziellen Nutzung im Internet bereit. Dies ist allerdings ein Novum. Für Rechtsprechungen anderer Gerichte ist es notwendig im Einzelfall zu klären wie das Gericht die Veröffentlichung und die Möglichkeiten der Weiterverwendung zum Beispiel in Form einer Wiederveröffentlichung kommerziell sowie nicht kommerziell handhabt. Viele Gerichte setzen hierbei auf eine Art Freemium-Modell, bei welchem die kostenlose Nutzung für Privatpersonen möglich ist. Für die Weiterverwendung werden jedoch individuelle Absprachen bis hin zu einer Gebühr pro Entscheidung nötig. Vielversprechend hierbei ist allerdings, dass viele Gerichte eine freie umfängliche Nutzung in Aussicht stellen, wenn „Entscheidungen für Zwecke abgerufen werden, deren Verfolgung überwiegend im öffentlichen Interesse“ (Justizministerium des Landes Nordrhein-Westfalen, o. J.) liegt.[4]

Hierbei kritisch, aber in den letzten Jahren rechtlich ins Wanken gekommen, sind exklusive Vereinbarungen von Bund und Gerichten mit privaten Anbietern. So ist nicht nur die Verbindung des Bundes zur Juris GmbH als direkter Teilhaber zu sehen, sondern auch, dass zwischen beiden lange Zeit beispielhafte Verträge über eine exklusive Belieferung mit aufbereiteten Urteilen bestand (Baden-Württemberg, 2013). Diesbezüglich gibt es dem folgenden Zitat wenig hinzuzufügen:

> Es gibt keinen Grund für den Staat sich auf diesem Gebiet wirtschaftlich zu betätigen. Er sollte Rechtsnormen und Urteile, die nicht dem Urheberrecht unterliegen, von einer

4 Die geschilderten Gegebenheiten ergaben sich aus Nachforschungen über das Justizportal des Bundes und der Länder: www.justiz.de/onlinedienste/rechtsprechung/index.php, Stand 08/2016

> gemeinnützigen Organisation digitalisieren lassen und jedermann kostenlos zur Verfügung stellen.
>
> — Markus Reithwiesner, Geschäftsführer Rudolf-Haufe-Verlag (Frankfurter Allgemeine Zeitung, 2009)

2.4 SOFTWARE-AS-A-SERVICE

Der Markt für elektronische Rechtsinformationen wird, ähnlich dem Markt für traditionelle Formate, von wenigen Akteuren dominiert. Zusätzlich liegt eine starke Überschneidung zwischen den großen Anbieter im digitalen und den traditionellen Verlagen im analogen Bereich vor. Dies resultiert daraus, dass die meisten Anbieter auf die Verbindung von Gesetzestexten mit den in ihrem Verlagsrepertoire befindlichen ergänzenden Werken wie Gesetzeskommentare setzen. Neben der Erweiterung des Angebots um verlagseigene und eingekaufte Gesetzeskommentare wird stark auf die direkte Verknüpfung von Gesetzen zu Rechtsprechungen gesetzt.

Als Zielgruppe haben diese Systeme allesamt professionelle Anwender, welche zur Zahlung der teils immensen Gebühren gewillt sind. So kostet beispielsweise der Bezug des gesamten deutschen Bundesrechtes, welches, wie in Abschnitt 2.3 beschrieben, urheberrechtsfrei ist, 40 € pro Anwender monatlich – ohne jegliche ergänzende Literatur (Verlag C. H. Beck, 2010).

Dabei mangelt es den bestehenden Anbietern stark an digitaler Innovation. So bieten sie beispielsweise keine umfassenden Mobillösungen an – außer einigen Nischenprodukten sind die elektronischen Rechtsinformationssysteme allesamt schwergewichtige Webseiten. An diesem Punkt setzt dieses Projekt an: Lawly, so der Name, verfolgt das Ziel einer modernen mobilen und insbesondere auch vollständig offenen Lösung.

2.5 LAWLY – FREIES RECHT

Als initiales Produkt wird eine Plattform zur interaktiven Navigation deutscher Bundesgesetzen entwickelt. In einem zweiten Schritt soll diese um Verknüpfungen mit Rechtsprechungen erweitert werden.

Zentrale Aspekte der Plattform sind Open Data und Open Source. Der verfolgte Ansatz ist hierbei vielschichtig. Es gilt, komplett auf frei verfügbare Daten (Open Data) aufzubauen und diese mit Hilfe von frei verfügbaren Technologien (Open Source) aufzubereiten. Damit geht auch einher, dass die Entwicklung der Plattform selbst offen gestaltet wird sowie auch freie Schnittstellen zu den aufbereiteten Daten zur Verfügung stellt und somit zu der Open Access Bewegung beiträgt. So soll sie zum Start nicht nur als Plattform, sondern auch

im Quelltext offen sein und im Idealfall Dritte in die Entwicklung einbinden. Neben solchen ideellen Ansätzen gilt es zusätzlich, konkrete für den Endnutzer spürbare Alleinstellungsmerkmale gegenüber bestehenden Angeboten zu schaffen.

Hierbei steht im Vordergrund, dass aktuelle digitale Anbieter auf traditionelle Webseiten setzen. Daraus ergeben sich starke Einschränkungen für die Benutzererfahrung: Die Handhabung von Informationsmengen, wie sie bei juristischen Informationen vorliegen, muss interaktiv gestaltet sein. Unterbrechungen durch Ladezeiten, die beispielsweise beim Wechsel zwischen einzelnen Normen, dem Aufruf verknüpfter Rechtsprechungen oder dem Durchsuchen der verfügbaren Daten entstehen, stören den Interaktionsfluss.

Zusätzlich spielt Portabilität heutzutage eine immer größere Rolle. Webseiten bestehender Anbieter sind zur Verwendung an eine bestehende und zuverlässige Internetverbindung gebunden. Genauso wie traditionelle Buchformate, welche aufgrund ihrer Größe und Masse nur schwer portabel sind, kommen diese somit nicht für den mobilen Einsatz infrage.

Es gilt also die Vorteile bestehender digitaler Angebote, also Aktualität und Umfang der Inhalte, mit modernen Ansätzen der Portabilität zu verbinden. Um dies zu erreichen, verfolgt der praktische Aspekt dieser Arbeit zwei Phasen.

Zu Beginn muss die Akkumulation der Bundesgesetze von *gesetze-im-internet.de* in zumindest teil-automatisierter Form umgesetzt werden. Hierbei gilt es, die rohen XML-Daten abzugreifen, in ein einheitliches leichter zu verarbeitendes Format zu übersetzen und in die eigene Datenbank einzuspeisen. Außerhalb dieser Arbeit ist es nötig, einen vollständig automatisierten Prozess zu entwickeln, welcher die bei diesem Vorgang gesammelten Datensätze auf ähnlichem Weg tagesaktuell erneuert.

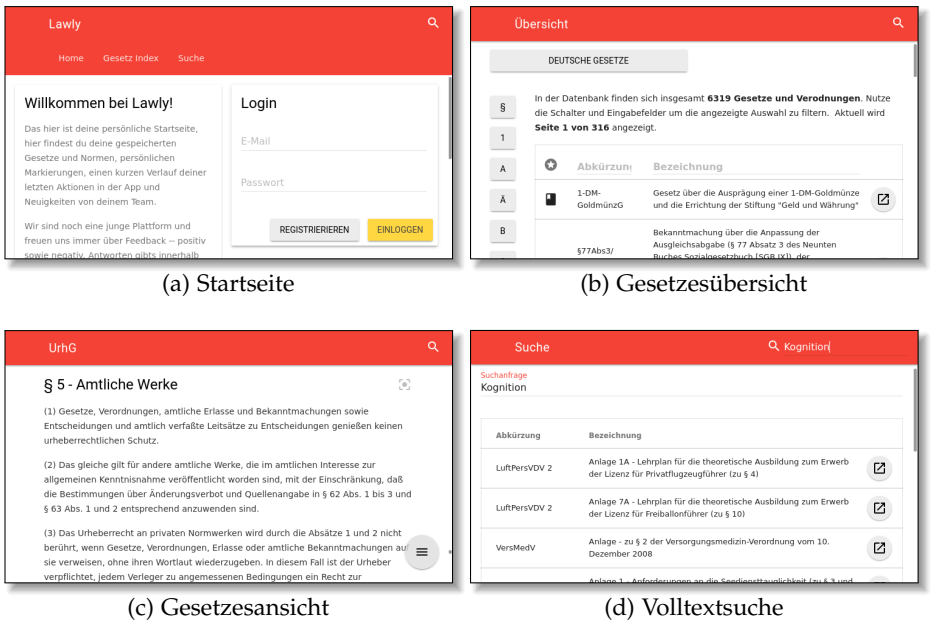
Sobald der Datensatz akkumuliert wurde, wird auf dessen Grundlage eine Applikation entwickelt, welche auf einer möglichst großen Auswahl an Geräten mit gleichermaßen vollem Funktionsumfang einsetzbar ist. Dabei ist es wichtig ein Nutzungserlebnis zu ermöglichen, welches dem eines für eine Plattform nativ entwickelten Programms möglichst gleichkommt. Dies spiegelt sich nicht nur in der allgemeinen Performance der Applikation wieder, sondern auch in der Möglichkeit zentrale Teile der Applikation ohne bestehende oder mit nur unzuverlässiger Internetverbindung nutzen zu können. So sollen zum Beispiel die Gesamtübersicht und insbesondere die vom Nutzer zuvor abgerufenen oder für die weitere Verwendung explizit vorgemerkten Gesetze immer verfügbar sein. Zusätzlich müssen diese Daten gleichermaßen online als auch offline durchsuchbar sein.

Obwohl sich die im Rahmen dieser Arbeit entwickelte Software auf eine Webapplikation zum interaktiven Navigieren deutscher Bundesgesetze beschränkt, wird bei der Implementierung und den ihr vor-

hergehenden technischen Entscheidungen vorausschauend geplant: Neben der Integration von Rechtsprechungen wird auch die Möglichkeit der Erweiterung um native Applikationen für iOS- und Android-Geräte bedacht. In persönlichen Gesprächen hat sich insbesondere für eine native iPad-Anwendung bei professionellen Anwendern großes Interesse gezeigt.

Neben den oben benannten technischen Anforderungen soll die initiale Plattform für den Nutzer die folgenden konkreten Ansichten und Funktionen bieten:

- Eine Gesetzesübersicht, welche nach vordefinierten Sammlung (zum Beispiel von traditionellen Büchern bekannte zusammenstellungen oder Themengebiete), Kürzeln und Bezeichnungen filterbar ist.
- Eine Ansicht, auf welcher individuelle Gesetze in ihrem gesamten Umfang betrachtet und mithilfe einer Inhaltsübersicht navigiert werden können.
- Eine Volltextsuche, die sowohl online als auch offline während der Eingabe in Echtzeit Ergebnisse liefert.
- Die Möglichkeit sich als Nutzer zu registrieren, Gesetze zu speichern und diese auch ohne Internetverbindung zu betrachten und zu durchsuchen.



(a) Startseite (b) Gesetzesübersicht

(c) Gesetzesansicht (d) Volltextsuche

Abbildung 2: Vorschau der zentralen Ansichten der Applikation (Smartphone, horizontal)

3

TECHNOLOGIEN UND PARADIGMEN

Im folgenden Abschnitt wird auf die technologische Grundlage des Projekts, die diesbezüglichen Entscheidungen und die ihnen zu Grunde liegenden Überlegungen eingegangen. Außerdem werden in der Umsetzung des Projekts verfolgte Paradigmen und ihre Abgrenzung zu ihren in der Industrie verbreiteten Alternativen erläutert.

Nach einer allgemeinen Diskussion sicherheitsrelevanter Aspekte (Abschnitt 3.1) folgt eine Einleitung in JavaScript als die zentrale für dieses Projekt eingesetzte Programmiersprache (Abschnitt 3.2). Daraufhin wird die konkrete Applikationsarchitektur analysiert, in Server (Abschnitt 3.3) und Client (Abschnitt 3.4) unterteilt. Die Überlegungen reichen von allgemeinen Aspekten wie Webbrowsern als moderne Softwareplattform und der aufkommenden Möglichkeiten zur Umsetzung von JavaScript-basierten nativen Mobilapplikationen hin zu einer sich langsam etablierenden Alternative zum traditionellen Model-View-Controller (MVC)-Paradigma auf Client-Seite und den sich bewiesenen Representational State Transfer (REST)-Architekturen im Vergleich zu zukunftsträchtigen Socket-Verbindungen auf Server-Seite.

Da bei der Umsetzung der Applikation auf modernste Standards gesetzt wird fällt dieser Abschnitt zu Technologien und Paradigmen im Vergleich zu dem über die konkrete Implementation (Abschnitt 4) verhältnismäßig lang aus. Hierbei wird sehr bewusst behandelt wo die Limitierungen moderner Webapplikationen im Kontrast zu nativen Mobilapplikationen liegen.

Wenn nicht anders angegeben, sind sämtliche folgende Listings valider JavaScript-Quelltext – vorausgesetzt die in diesen explizit importierten Bibliotheken sind verfügbar. Allerdings werden gegebenenfalls nicht direkt im Listing definierte globale Variablen als Platzhalter verwendet.

3.1 SICHERHEIT

Sicherheit wird heutzutage immer größer geschrieben. Themen wie Privatsphäre und Verschlüsselung sind in aller Munde und es vergeht kaum ein Monat in dem nicht eine weitere große Firma ein *Da-*

tenleck zu verbüßen hat.[1] Die Schwierigkeit liegt darin, Sicherheit und Nutzungskomfort in Einklang zu bringen.

Im Folgenden sollen drei zentrale Problembereiche und ihre in der Umsetzung dieser Arbeit eingesetzten Lösungen erörtert werden:

1. Die Kommunikation zwischen Client und Server.
2. Die physikalische sowie softwaremäßige Integrität der Servers.
3. Die vom Endnutzer eingesetzte Hardware und die Integrität der an diese ausgelieferten Applikation.

Außer für die abschließende Erläuterung von JSON Web Tokens (JWTs) wird in diesem Abschnitt umfänglich auf Tilkov, Eigenbrodt, Schreier, & Wolf (2015) als Referenz zurückgriffen.

3.1.1 *HTTPS*

Zur Absicherung des ersten Bereichs und als Grundpfeiler für die Sicherheit einer modernen Applikation dient die umfassende Verschlüsselung der Kommunikation. Durch *Ende-zu-Ende*-Verschlüsselung ist es möglich den Datenverkehr mit heutigen Mitteln auf recht einfache Art und Weise gegen das Abgreifen durch Dritte (*Man-in-the-Middle Attack*) zu schützen. Um dies zu erreichen, wird auf eine in Abbildung 3 dargestellte Kombination aus asymmetrischer und symmetrischer Verschlüsselung gesetzt, das Hypertext Transfer Protocol Secure (HTTPS).

Bevor der Server die Möglichkeit hat eine sichere HTTPS-Verbindung anzubieten, muss er ein asymmetrisches Schlüsselpaar generieren und sich durch eine Certificate Authority (CA), einem vertrauenswürdigen Dritten, ein Zertifikat für dieses ausstellen lassen (*a*). Gleichermaßen muss der Client von vertrauenswürdigen CAs sogenannte Root-Zertifikate einholen (*b*).

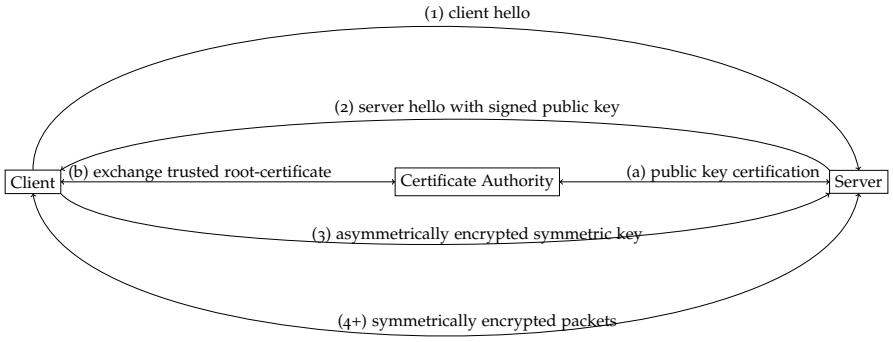


Abbildung 3: HTTPS-Verschlüsselte Kommunikation

1 Für eine mit Referenzen versehene Visualisierung siehe informationisbeautiful.net/visualizations/worlds-biggest-data-breaches-hacks, Stand 08/2016

Zu Beginn einer individuellen Kommunikationssession meldet sich dann ein Client, unverschlüsselt, mit Informationen zu unterstützten Verschlüsselungsverfahren beim Server (*1*). Dieser antwortet daraufhin zusammen mit dem von der CA ausgestellten Zertifikat und mit auf Grundlage der zuvor erhaltenen Informationen bestimmten konkreten Rahmenbedingung für die weitere Kommunikation (*2*).

Bevor der Client auf weitere Kommunikation eingeht, überprüft er mithilfe des Root-Zertifikats der CA ob die Signatur des erhaltenen Zertifikats authentisch ist. Wenn dies gilt, generiert er einen neuen, exklusiv für diese Kommunikation zu verwendenden symmetrischen Schlüssel und überträgt ihn, mit dem zuvor erhaltenen öffentlichen Schlüssel verschlüsselt an den Server (*3*). Diesen Schritt nennt man asymmetrische Verschlüsselung: Der Client kann Pakete zwar auf diese Weise verschlüsseln, aber, da er nur den öffentlichen Teil des asymmetrischen Schlüsselpaares kennt, nicht entschlüsseln. Da der private Teil des Schlüsselpaares den Server nie verlassen hat, können auch dritte auf den so übertragenen symmetrischen Schlüssel nicht zugreifen. Die gesamte zukünftig in dieser Session stattfindende Kommunikation wird nun von beiden Seiten mit diesem symmetrischen Schlüssel geschützt, der daher auch *Session-Key* genannt (*4+*). Dadurch, dass dieser zu Beginn nur asymmetrisch verschlüsselt an den Server übertragen wurde, können auch möglicherweise die Kommunikation belauschende Dritte die Daten nicht entschlüsseln.[2]

3.1.2 *Server-Integrität*

Der zweite problematische Bereich, die Integrität des Servers, ist offensichtlich durch das Vermeiden von Sicherheitslücken zu schützen. Obwohl dies selbstverständlich anzustreben ist, ist es, durch den Einsatz von Softwarebibliotheken dritter, selbst bei einfachen Applikationen nicht hundertprozentig zuverlässig umsetzbar. Um trotz eventueller Sicherheitslücken die Risiken einer Ausnutzung im Rahmen zu halten, gilt es, nur die Informationen vorzuhalten, welche für die Verwendung der Applikation unbedingt notwendig sind. So ist es zum Beispiel eine Sache, ob Angreifer Passwörter aus einer Datenbank abgreifen konnten, aber eine ganz andere, wenn diese Passwörter nur in verschlüsselter Form vorliegen. Da Passwörter für einen Servicebetreiber niemals im Klartext benötigt werden sollten, kann kryptographisches Hashing eingesetzt: Das Passwort wird vor seiner Speicherung mit Hilfe einer sogenannten Einwegfunktion unkenntlich gemacht. Notwendige Eigenschaften einer solchen kryptographisches Hashfunktion sind, dass sie immer das gleiche eindeutige Ergebnis produziert und nicht umkehrbar ist. Dadurch ist es zwar möglich

2 Durch den Einsatz von durch die freie Let's Encrypt CA ausgestellte Zertifikate und der konsequenten Befolgung aktueller Sicherheitsempfehlungen erhalten die Lawly API und die Lawly Webapplikation jeweils Bestnoten von den auf das Testen von Sicherheitskonfigurationen von Webservices spezialisierten Anbietern ssllabs.com und securityheaders.io.

Passwörter mit dem gespeicherten Hash zu vergleichen, aber nicht, diese aus dem Hash wieder herzustellen.

3.1.3 *Client-Integrität*

Ähnliches gilt für Bereich drei, clientseitige Sicherheit. Auf die Integrität des Mobiltelefons oder Rechners des Nutzers haben Servicebetreiber keinerlei Einfluss und müssen sich hier noch mehr als bei ihren eigenen Systemen darauf verlassen, möglichst wenig sensible Informationen auf dem Endgerät zu speichern. Bei einer modernen Webapplikation ist, wie in Abschnitt 3.3 erläutert, allerdings eine sogenannte zustandslose Kommunikation vorzuziehen. Bei einer solchen ist jede Anfrage an den Server für sich atomar, also muss auch individuell authentifiziert werden. Einer der einfachsten Ansätze ist die sogenannte *HTTP Basic Authentication*[3], bei welcher mit jeder Anfrage an den Server Nutzername und Passwort im Klartext übertragen werden.

Der offensichtliche Nachteil hieran ist nicht nur die Übertragungsform (Klartext), sondern insbesondere auch, dass der Nutzer entweder für jede Anfrage manuell seine Nutzerdaten eintragen muss (was aus offensichtlichen Gründen kein gute Nutzungserfahrung bietet) oder die Nutzerdaten nach initialer Eingabe auf dem Endgerät gespeichert werden müssen. Im Fall der Entwendung des Gerätes ist der Nutzer in einem solchen Fall auf das Zurücksetzen seines für den Dienst verwendeten Passwortes angewiesen, was besonders aufgrund der Tendenz für viele Dienstleister das gleiche Passwort zu verwenden problematisch ist. Ein Lösung hierfür finden sich in Autorisierungstokens mit vorgeschrieben Lebensdauern. Nach der initialen Authentifizierung des Nutzers auf Grundlage seiner Nutzerdaten wird vom Server ein signierter Token bereitgestellt, welcher bei zukünftigen Serveranfragen mit übertragen wird. Im Fall des Verlustes des Endgerätes wird so nur der in seiner Lebenszeit begrenzte Token kompromittiert, allerdings nicht die Nutzerdaten. Dem Nutzer kann zusätzlich sehr einfach das Widerrufen von alten Tokens angeboten werden.

3 *Authentication* und *Authorization* werden, obwohl eigentlich zwei unterschiedliche Dinge, in der HTTP-Spezifikation beziehungsweise ihrer Umsetzung oft ungenau verwendet. Ein Beispiel hierfür sind die HTTP Statuscodes 401 und 403:

- *401 Unauthorized*: „Request lacks valid authentication credentials.“ (tools.ietf.org/html/rfc7235) – wörtlich dann wohl eher ein *unauthenticated* Zugriff (fehlende Authentifizierung).
- *403 Forbidden*: „The provided authentication credentials are considered insufficient.“ (tools.ietf.org/html/rfc7231) – gleichermaßen wörtlich also eher ein *unauthorized* oder unerlaubter Zugriff, obwohl die Anfrage eigentlich Authentifiziert ist.

Eine ausführlichere Diskussion hierzu findet sich zum Beispiel unter stackoverflow.com/q/3297048.

Die Lawly Applikation setzt hierfür auf den offen Standard von JSON Web Tokens (JWTs). Ein JWT setzt sich aus drei Teilen zusammen: Im ersten Teil, dem Header, finden sich Informationen über den Token selbst, also zum Beispiel die Bezeichnung des Verschlüsselungsalgorithmus, welcher bei der Signierung eingesetzt wurde. Der zweite Teil beinhaltet die *claims*, sogenannte Ansprüche die der Token an seine Zugriffsrechte in der angefragten Ressource stellt. Dies beinhaltet zum Beispiel die E-Mailadresse des Nutzers, da diese für die Verbindung des Nutzers zu seinen vorgenommenen Vermerken herstellt. Der letzte Teil ist die Signatur. Sie ist ein kryptographischer Hash, welcher aus den beiden anderen Teilen und einem geheimen Schlüssel, welcher nur dem Server bekannt ist, generiert wurde. Dadurch, dass dieser Hash eindeutig aus diesen drei Teilen erstellt wurde, kann keiner dieser drei Teile verändert werden, ohne das der Hash seine Gültigkeit verliert. (Jones, Bradley, & Sakimura, 2015)

Der Token kann somit sicher auf Nutzerseite gespeichert werden, ohne dass eine Kompromittierung des Gerätes die Aufgabe des Passworts bedeuten würde. Durch ein relativ kurzes Ablaufdatum, im *claim* als *expiration* gespeichert, kann zusätzlich automatisch für ein Verfallen des Tokens gesorgt werden. Bei aufeinanderfolgenden Anfragen wird dabei jedes Mal überprüft ob das Verfallsdatum naheliegt und wenn dem so ist gegebenenfalls ein neuer Token für folgende Anfragen bereitgestellt.

3.2 JAVASCRIPT

Als initiales Minimum Viable Product (MVP), also erste grundlegende Version des Produktes, wird im Rahmen dieser Arbeit eine Webapplikation mit dazugehörigem API-Server entwickelt. Beide setzen auf JavaScript in der aktuellsten Spezifikation, ECMAscript (ES) 7. JavaScript spaltet seit jeher die Entwicklergemeinde: Oft als Grundlage für Kritik hervorgehoben wird die ursprüngliche Entwicklungszeit der Sprache von nur knapp zwei Wochen und die oft stark variierende Umsetzung der offiziellen Spezifikationen in den verschiedenen Browsern.

Im Kontrast zu dieser Kritik stand allerdings immer ein zentrales Alleinstellungsmerkmal: Durch den Browser als primäre Plattform für JavaScript hat keine Programmiersprache eine ähnlich breite Auswahl an ausführenden Geräten. Durch den Trend zu immer mehr und immer leistungsstärkeren mobilen Endgeräten, welche alle über die Möglichkeit verfügen Webapplikationen auszuführen, wurde auch die JavaScript Entwicklergemeinschaft und damit die Entwicklung der Sprache selbst voran getrieben. Initial noch zum größten Teil zum „aufhübschen“ von Websites verwendet, ist JavaScript in den letzten Jahren durch die Entwicklung von Node.js zu einer Full-Stack Sprache gediehen: JavaScript ist nicht mehr nur auf Client-Seite, im Brow-

ser, sondern auch direkt auf Betriebssystemebene und damit auf der Serverseite von Applikationen einsetzbar. (Brehm, 2013)

Das Open Source Projekt Node.js (auch *node* genannt) wurde 2009 vorgestellt. Ryan Dahl begann die Entwicklung aufgrund seiner Frustration mit den zu diesem Zeitpunkt vorherrschenden Webserver-Laufzeitumgebungen. Durch die Notwendigkeit zu immer mehr und komplexeren Ein- und Ausgabeoperationen (I/O), also der Kommunikation eines Programms mit seinem Umfeld, stießen traditionelle Systeme zunehmend an ihre Grenzen. Um dieses Problem anzugehen ist JavaScript dank seiner ereignisbasierten asynchronen Funktionsweise perfekt geeignet. (Dahl, 2009)

Durch die Verwendung der gleichen Sprache auf Client- und Serverseite wird nicht nur die Wartung erleichtert, sondern auch die Möglichkeit des Einsatzes von universellem, sogenannten isomorphe, Code eröffnet: Server und Client können auf die gleichen Bibliotheken zugreifen und Funktionen und Klassen teilen (Brehm, 2013). Dies erleichtert den Einstieg für angehende Entwickler und im Speziellen auch die Vereinigung von Frontend- und Backend-Rollen in Unternehmensstrukturen. Ein insbesondere bei jungen Unternehmen oder kleineren Unternehmungen großer Vorteil.

Die flexible Einsetzbarkeit, die dadurch entstandene Popularität und die daraus wiederum folgende massive Weiterentwicklung der Sprache und dem zugehörigen Ökosystem hat zu einem weiteren interessanten Trend geführt: Die Entwicklung nativer Applikationen. Native Applikationen sind solche, welche auf einem Endgerät unabhängig vom Browser und seinen Limitierungen ausgeführt werden. Zudem setzen sie für die Darstellung anstatt auf das dem Bowser eigene Document Object Model (DOM) auf die vom jeweiligen Betriebssystemanbieter zur Verfügung gestellte Bibliotheken in nativer Maschinensprache. Heutzutage ist es möglich für die Applikationslogik JavaScript einzusetzen und nur die von dieser Logik abhängende Darstellungsebene der Applikation für verschiedene Geräte gesondert nativ zu implementieren.

Als Resultat der rasanten Weiterentwicklung von JavaScript wurde in den letzten Jahren das erste mal seit langem eine starke Neuerungen umsetzende Version standardisiert, ECMAscript (ES) 6 und, vor kurzem, 7. Mit diesen halten endlich in anderen Skriptsprachen schon lange etablierten Funktionalitäten auch in JavaScript Einzug. Dazu gehören zum Beispiel Klassen (als Modernisierung von JavaScripts Prototypen-basierten Design) und Generatoren (zur Vereinfachung des Arbeitens mit den in JavaScript zentralen asynchronen Funktionen). (Ecma International, 2015, 2016)

Bei der Entwicklung von Lawly wird intensiv auf ES 7 gesetzt. Dieser Quelltext wird automatisiert, mit Hilfe der darauf spezialisierten Bibliothek *Babel*, vor dem Produktiveinsatz zu einer von allen verbreiteten Browsern unterstützten älteren Version der Sprache transpiliert.

3.3 SERVER

Im Folgenden wird zu Beginn auf die Entscheidung eingegangen, Server und Client strukturell vollständig voneinander zu trennen. In den Abschnitten 3.3.1 und 3.3.2 wird dann, respektive, die eingesetzte Server- und Datenbank-Architektur diskutiert.

Aufgrund der Anforderung einer in Zukunft flexibel auf weitere Plattformen erweiterbaren Applikation kommt nur eine klare Trennung von Server und Client in Frage. Bei der Entwicklung eines reinen Application Programming Interface (API) Servers ist es wichtig, dass dieser Daten in einer von ihrer für den Endnutzer endgültigen Repräsentation unabhängigen, leicht zu verarbeiteten Form zur Verfügung stellt. Allerdings gilt es dabei gleichermaßen, die bereitgestellten Schnittstellen nicht zu abstrakt zu gestalten, sondern Daten sinnvoll zu bündeln, um vermeidbare Anfragen durch Client-Applikationen vorzubeugen.

Aus dieser Unabhängigkeit ergibt sich die Möglichkeit serverseitige Logik, wie beispielsweise Datenbankanfragen und Nutzer-Authentifizierung, wiederzuverwenden. Im besten Fall wird für eine bestimmte Aufgabe, wie etwa die Registrierung eines neuen Nutzers, die serverseitige Logik nur ein einziges mal implementiert, so dass beliebige darauf angewiesene clientseitige Applikationen in Zukunft nur noch eine einmalig standardisierte Anfrage an den Server stellen müssen. Dadurch ist es nicht Notwendig bei der Erstellung zusätzlicher Apps mit ähnlichem Funktionsumfang wie dem initialen Produkt Änderungen am Server vorzunehmen. Auch bei der Implementierung neuer Funktionalität auf allen Plattformen muss so nur einmalig eine neue API-Schnittstelle hinzugefügt werden.

In Bezug auf eine Webapplikation ergibt sich zusätzlich der Vorteil, dass die API und die eigentliche für den Nutzer zur Verfügung gestellte Applikation auf unabhängigen Servern bereitgestellt werden können. So ist es möglich, flexibel auf Lastspitzen durch Skalierung der jeweils stärker betroffenen Hardware zu reagieren.

3.3.1 *Application Programming Interface*

Grundlegend gibt es aktuell zwei für den Einsatz als APIs interessante Konzepte:

1. Eine Hypertext Transfer Protocol (HTTP)-REST-API, über welche primär statische Daten wie Gesetzesdokumente zur Verfügung gestellt werden, und
2. Websockets, welche für die Synchronisierung von atomaren Interaktionen der Nutzer mit der Applikation zwischen Datenbank und über mehrere Geräte hinweg zuständig sind.

Die zentrale Unterscheidung zwischen den beiden ist das `pull/push` -Prinzip. Bei einer REST-API werden mithilfe von sogenannten HTTP-Verben einzelne Anfragen an den Server gestellt, worauf hin eine Antwort erwartet wird. Diese Anfragen sind im besten Fall für den Server komplett voneinander unabhängig und nicht auf einen gewissen serverseitig vorgehaltenen Zustand angewiesen. Bildlich werden per Anfrage also Daten vom Server *gepulled*. Ein Übertragen von Daten an den Client ist hierbei immer nur als Antwort auf eine von diesem initiierte Anfrage möglich.

Im Gegensatz dazu muss sich bei der Verwendung von WebSockets der Client nur für einen initialen Verbindungsaufbau beim Server anmelden. Ab diesem Zeitpunkt wird eine Verbindung zwischen Server und Client aufrechterhalten, an die beide, wann immer aktualisierte Informationen vorhanden sind, Daten *pushen* und auf erhaltene Daten reagieren können. Hierbei wird nicht mehr von einzelnen Anfragen, sondern von Ereignissen, auf welche die Teilnehmer reagieren, gesprochen.[4]

Für das initiale Produkt wird auf eine reine HTTP-Schnittstellen gesetzt. Im geplanten Funktionsumfang ist, initial, kein Austausch von Informationen in Echtzeit nach dem push Prinzip notwendig. Der Client arbeitet nur mit konkret von ihm angefragten Daten, wie zum Beispiel der Gesetzesübersicht und den Normen eines bestimmten Gesetzes. Zwar ist es interessant, Interaktionen, wie zum Beispiel das Vormerken von Gesetzen, mithilfe von einer durch WebScokets realisierten Echtzeitverbindung zwischen mehreren aktiven Geräten des gleichen Nutzers per push zu synchronisieren, allerdings mit einem großen Mehraufwand verbunden, so dass es erst zu einem späteren Zeitpunkt umgesetzt werden soll.

3.3.1.1 *HTTP & REST*

Ein wichtiger Punkt, welcher beim Einsatz von WebSockets für einen stark erhöhten Implementierungsaufwand sorgen würde, ist, dass ein WebSocket-Server niemals zustandslos sein kann. Er muss nach einer initialen Anfrage eines Clients die Verbindung zu diesem aufrecht erhalten um im weiteren Verlauf mit ihm kommunizieren zu können. Im Gegensatz dazu kann bei einer HTTP-API der Server so implementiert werden, dass er keinerlei Informationen über vorhergehende Anfragen vorhält, wodurch die Infrastruktur, wenn notwendig, horizontal skaliert werden kann.

4 Für eine bessere Kompatibilität mit älteren Browsern ist es auch möglich statt Web Sockets sogenanntes *long-polling* einzusetzen. Hierbei erfolgt der Verbindungsaufbau wie bei einer REST-API, allerdings wird die ankommende Anfrage nicht sofort beantwortet. Der Server hält die Verbindung offen und antwortet erst zu einem späteren Zeitpunkt auf die Anfrage. Sobald der Client eine Antwort erhält sendet er unmittelbar eine erneute Anfrage um dem Server wieder die Möglichkeit zu eröffnen ihm gewissermaßen ungefragt Daten zu übermitteln.

Horizontale Skalierung (*horizontal scaling*) beschreibt dabei die Möglichkeit, ein System dynamisch, durch variieren der Anzahl der über einen Load Balancer zusammengeschalteten Server, für die Handhabung unterschiedliche Last zu rüsten. Im Gegensatz dazu wird bei der vertikalen Skalierung das aus einem einzelnen Server bestehende System nur durch anpassen der in dem System vorhandenen Rechenkernen oder Arbeitsspeichermenge beeinflusst. (Vaquero, Rodero-Merino, & Buyya, 2011)

Um die auf Grund ihrer höheren Flexibilität zu bevorzugende horizontale Skalierbarkeit zu ermöglichen, ist es notwendig, dass jede Anfrage alle für ihre Beantwortung notwendigen Informationen mitliefert – dazu gehören beispielsweise auch die jeweils relevanten Authentifizierungsinformationen (siehe hierzu den dritte Teil von Abschnitt 3.1, Sicherheit). Im Fall von WebSockets ist es notwendig, dass Verbindungen entweder zentral, also für alle im Cluster befindlichen Server erreichbar, vorgehalten werden oder aber dafür gesorgt wird, dass ein Client sich immer nur mit dem Server, mit welchem er initial von dem das Cluster verwaltende Load-Balancer verbunden wurde, kommunizieren kann.

Durch die Orientierung an einer *RESTful Resource-Oriented Architecture*, ergibt sich die Entwicklung einer intuitiv verständlichen API. Anfragen werden so anhand ihrer URL an eine bestimmte Ressource gerichtet und mithilfe der HTTP-Methode (*GET*, *POST*, *PUT* etc.) die gewünschte Aktion spezifiziert. Daraus ergibt sich, dass aus der ersten Zeile der HTTP-Anfrage bereits hervorgeht, was der Client zu erreichen gedenkt. Eine solche erste Zeile sieht in der entwickelten Anwendung zum Beispiel wie folgt aus: `GET /laws/BGB HTTP/1.1` – der Client erwartet als Antwort ein Gesetz, welches durch *BGB* eindeutig identifizierbar ist. (Tilkov u. a., 2015)

Um auch geschützte Ressourcen verfügbar zu machen bietet HTTP und das in der entwickelten Applikation eingesetzte, nach initialer Verbindung gleich zu verwendende, HTTPS den sogenannten `Authorization`-Header. Über diesen wird der alle für die Autorisation notwendigen Informationen enthaltende Autorisierungstokens (siehe Abschnitt 3.1) bei jeder Anfrage nach der initialen Anmeldung eines Nutzers übertragen.

3.3.1.2 *Express*

Eine zentrale und gleichermaßen für viele Neulinge verwirrende Eigenschaft von JavaScript ist der Event Loop. Bei beispielsweise in PHP geschriebenen Servern wird bei jeder eintreffenden Netzwerkanfrage eine Instanz gestartet in welcher der komplette Quelltext ausgeführt wird. Im Falle eines Node.js-Servers hingegen wird die Applikation initial einmalig gestartet und lauscht von nun an auf auftretende Ereignisse. Ein solches Event ist im Fall eines REST-API-Servers zum Beispiel das Eintreffen einer Netzwerkanfrage. Wie in Listing 1

dargestellt, lauscht der Server auf ein solches `request`-Event und behandelt es mithilfe einer vordefinierten Funktion, eines sogenannten Handlers. Solche Handler sind in JavaScript *Callbacks* für asynchrone Funktionsaufrufe. Sie werden also bei Fertigstellung der asynchron ausgeführten Operation mit dem Ergebnis oder eventuellen Fehlern aufgerufen.

```
import http from 'http';
const server = http.createServer();
server.on('request', (request, response) => {
  response.end(`Received request at ${request.url}!`);
});
server.listen(8080, () => {
  console.log('Listening on port 8080');
});
```

Listing 1: Einfacher Node.js Server

Um leichter dem REST-Prinzip folgen zu können wird bei der Applikation auf das Express-Framework gesetzt. Wie in Listing 2 sichtbar, filtert es die eintreffenden HTTP-Anfragen und verteilt sie auf klar definierte Handler für einzelne HTTP-Verben und Pfade.

```
import express from 'express';
const app = express();
app.get('/obj', (request, response) => {
  response.end(`GET at 2!`);
});
app.put('/obj/:id', (request, response) => {
  response.end(`PUT at /obj/${request.params.id}!`);
});
app.listen(8080, () => {
  console.log('Listening on port 8080');
});
```

Listing 2: Node.js Server mit Express

3.3.2 *Datenbank*

Im folgenden wird knapp auf die zentrale Unterscheidung zwischen normalisierten SQL- und denomalisierten dokumentorientierten Datenbanken eingegangen – breiteres Vorwissen wird hier allerdings vorausgesetzt. Primär wird der durch neuere Versionen der Open Source Datenbank PostgreSQL ermöglichte Hybrid-Ansatz erläutert.

Als Datenbank kommen grundlegend zwei verschiedene Ansätze in Frage: Traditionellere SQL- und in den letzten Jahren aufgekommene NoSQL- bzw. Dokument-Datenbanken. Besonders für schnell wachsende und im großen Stil Daten anhäufende Anwendungen haben sich die NoSQL-Datenbanken bewiesen, müssen sich aber in ihrer Ausdauer noch im Vergleich zu den jahrzehntelang gehärteten SQL-Systemen beweisen. Sie basieren auf der Grundidee einzelne, nicht direkt voneinander abhängige Dokumente zu speichern. Ähnlich wie

bei dem verfolgten Ansatz eines zustandslosen API-Servers, beschrieben in Abschnitt 3.3, ist es so möglich die Datenbank horizontal durch hinzufügen neuer Instanzen zu skalieren.

Ein zentraler Aspekt von SQL-Datenbanken ist im Kontrast dazu eine *normalisierte* Struktur: Bei strenger Einhaltung werden dabei Informationen niemals redundant abgespeichert. In dem Fall, dass ein Nutzer ein bestimmtes Gesetz vorgemerkt hat, ergeben sich so zwei gegensätzliche Herangehensweisen: In einer dokumentorientierten Datenbank werden der Titel dieses Gesetzes direkt in dem einem Nutzer zugeordneten Dokument abgespeichert, so dass bei Zugriff auf dieses direkt alle vom jeweiligen Nutzer vorgemerkten Gesetze verfügbar sind. In einer normalisierten SQL-Datenbank hingegen würde stattdessen ein neuer Eintrag erstellt werden, welcher auf die eindeutigen Identifikationsnummern des Nutzers und des Gesetzes verweist. Beim Abrufen der Nutzerinformation würde dann der Nutzer über solche *one-to-many* Beziehungen mit den vorgemerkten Gesetzen in Verbindung gebracht werden.

Ein Nachteil starker Normalisierung ist allerdings eine steigende Komplexität des Systems und der hohe Ressourcenanspruch. Bei komplexen Systemen sind mehrschichtige `Joins`, also das auflösen von normalisierten Beziehungen, rechenintensiv. Die Lösung hiervon besteht meist in intensivem Caching der Ergebnisse, was allerdings nur bei oft wiederkehrenden Anfragen hilfreich ist. Wird so zum Beispiel der aktuell Anfragende Nutzer in den Query mit einbezogen, kann nicht auf den Cache eines vorhergehenden Queries eines anderen Nutzers zurückgegriffen werden.[5]

Gleichermaßen führt die bei dokumentorientierten Systemen notwendige Denormalisierung der Daten zu einer starken Redundanz der gespeicherten Informationen und dadurch zu der Gefahr eines Integritätsverlust. In einer normalisierten Datenbank wird die Integrität der gespeicherten Daten durch die fehlende Redundanz garantiert. Um die Integrität in einem denormalisierten System aufrecht zu erhalten, ist es Notwendig die von einer geänderten Information abhängigen Dokumente alle möglichst gleichzeitig zu aktualisieren. Bei großen strukturell geteilten Systemen geschieht dies oft erst zeitversetzt, was teilweise auch für den Nutzer merklich ist. So kann es in großen sozialen Netzwerken oftmals vorkommen, dass an manchen stellen noch

5 Die geschilderte Erfahrung brachte die Entwicklung der Kurs-Suchfunktion für die *Study Planning Machine* für den Cognitive Science Studiengang an der Universität Osnabrück. Dabei galt es, eine in Echtzeit Anfragen beantwortende Lösung zu entwickeln, welche innerhalb der Suchergebnisse bereits die Verbindung des suchenden Nutzers zu dem gelieferten Kurs darstellt – also zum Beispiel ob der anfragende Nutzer diesen Kurs bereits in seine Sammlung aufgenommen hat. Die in diesem Fall verwendete Datenbankstruktur ist streng normalisiert: *Studienordnungen* ↔ *Module* ↔ *Kurse* ↔ *Lehrende* ↔ *Studenten*. Um verstärkt von Caching Gebrauch machen zu können, wurde die Verbindung zum anfragenden Studenten endgültig Clientseitig gelöst. Applikation auf cogsci.uos.de/~SPAM, Quelltext unter github.com/ahoereth/spam.

ein veraltetes Profilbild eines Nutzers angezeigt wird, wenn dieser es vor kurzer Zeit erneuert hat.

Um die Güte beider Systeme zu vereinen wird für Lawly auf das SQL-Datenbanksystem PostgreSQL gesetzt. Obwohl PostgreSQL zur Grundlage eine traditionelle an Tabellen orientierte Struktur nutzt, bringt es in neueren Versionen aus dokumentorientierten Systemen bekannte JSON-Datentypen mit. Diese eröffnen die Möglichkeit in einzelnen Spalten nicht nur mehr als einen einzelnen Wert zu speichern, sondern auch komplexere Dokumentstrukturen abzulegen. Dabei ist es weiterhin möglich diese Dokumente oder in ihnen verschachtelte individuelle Einträge in SQL-Anfragen und -Indices zu nutzen. Dadurch kann für sich oft ändernde Informationen eine normalisierte Struktur, allerdings für langfristig statische aber oft abgefragte Beziehungen, eine denormalisierte Struktur eingesetzt werden.

3.4 CLIENT

Zentral orientiert sich die angestrebte Nutzererfahrung der Applikation an dem 2015 von Alex Russel beschrieben Gesamtkonzept einer Progressive Web Application (PWA): Eine Single Page Application (SPA) welche die besten Seiten von nativen mobilen Applikationen und Webapplikationen vereint (Russell, 2015). Hierbei handelt es sich nicht um ein völlig neues Konzept, sondern vielmehr um eine clevere Kombination von bereits zuvor etablierten Einzelansätzen. Im Mittelpunkt dieser steht eine SPA (Abschnitt 3.4.1), mit offline- (Abschnitt 3.4.2) sowie mobile-first (Abschnitt 3.4.3) Grundsätzen entwickelt. Als Ergänzung zu diesen Grundsätzen wird außerdem von den von modernen Mobilsystemen zur Verfügung gestellten Möglichkeiten zur Beeinflussung der Applikationsumgebung über HTML-Metatags oder einem Manifest (vergleichbar mit dem für native Applikationen) Gebrauch gemacht.

Als die zentrale architektonische Grundlagen für die Applikation wird eine Komponenten-Hierarchie (Abschnitt 3.4.4.1) mit einem uni-direktionalem Datenfluss (Abschnitt 3.4.4.2) von unveränderbaren Datenstrukturen (Abschnitt 3.4.4.3) vorgestellt und eingesetzt. Diese wird dabei im starkem Kontrast zum traditionell verbreiteten MVC-Konzept diskutiert.

3.4.1 *Single Page Applications*

Traditionell wird bei dem Aufruf einer Webseite ein einzelnes HTML-Dokument vom Server an den Client übertragen, welches alle Informationen in einer bereits für die Darstellung strukturierten Form beinhaltet. Jede Interaktion mit einer solchen Webseite führt zu einer

neuen Anfrage an den Server welcher daraufhin eine neue Seite gegebenenfalls individuell für den anfragenden Client generiert und zur Verfügung gestellt. Dies stellt gewissermaßen einen sich wiederholenden Kreislauf von Anfrage an den Server, Darstellung der Webseite und Nutzerinteraktion mit dieser dar. Dieser Kreislauf wird als *vollständigen Rundtrip* bezeichnet, da jede Abarbeitung dieses Kreislaufes gewissermaßen einem Neuanfang gleichkommt.

In Abbildung 4 wird dieser Kreislauf beispielhaft durchnummeriert dargestellt: Auf die vom Browser gestellte initiale Anfrage an den Server (1) antwortet dieser mit einer Webseite (2) mit welcher der Nutzer interagiert (3). Diese Interaktion löst eine Anfrage an den Server aus (4) welcher eine neue Webseite bereitstellt (5). Diese neue Webseite kann sich zwar inhaltlich mit der alten überschneiden, muss aber vom Client komplett neu interpretiert werden. An dieser Stelle beginnt der Kreislauf erneut: Der Nutzer interagiert mit der Seite (6), was in einer Anfrage an den Server resultiert (7) und so weiter.

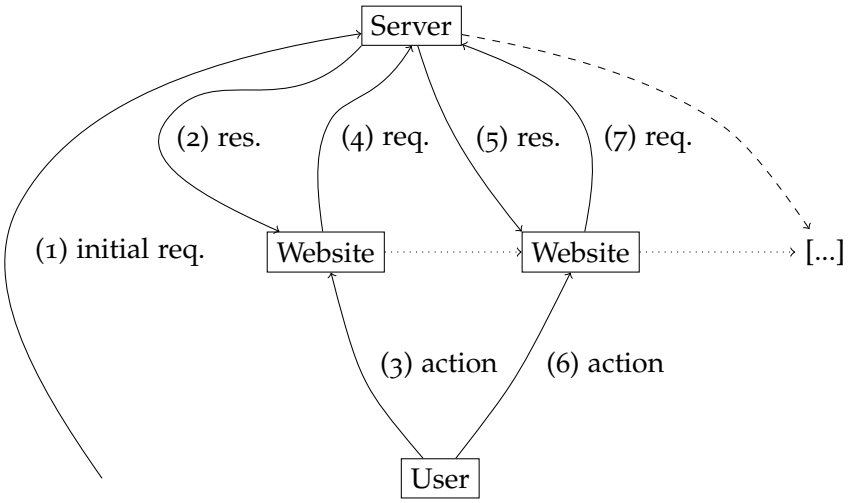


Abbildung 4: Synchrone Client-/Server-Kommunikation

Nachdem dieser Ansatz zu Beginn der breiteren Verbreitung von asynchronous JavaScript (AJAX) durch einzelne interaktiven Elementen erweitert wurde, wird bei einer SPA der Rundtrip vollständig abgelöst. Dieses Modell wird in Abbildung 5 dargestellt. Am Anfang steht dabei wieder eine initiale Anfrage an den Server (1) woraufhin dieser die Webseite zur Verfügung stellt (2). Im Kontrast zur normalen Webseite schaltet sich nun der bei dieser Anfrage übertragene JavaScript-Code zwischen Nutzerinteraktionen und Server (gestrichelte Kanten). Nutzerinteraktionen werden nun direkt von der Applikation selbst verarbeitet, ohne unmittelbar in einer Anfrage an den Server zu resultieren (3+). Das Skript kümmert sich dabei anschließend nicht nur um einzelne interaktive Inhalte, sondern auch um die Navigation innerhalb der Seite. Auch für das Absetzen eventueller Serveranfragen (4+) und die Aktualisierung der dargestellten Ansicht entsprechend ankommender Daten (5+) ist nun der JavaScript-Code verantwortlich. Serveranfragen in Folge von Nutzeraktionen sind von

nun an allerdings oftmals optional da, je nach Implementation, benötigte Daten möglicherweise schon bei vorhergehenden Anfragen geladen wurden.

Die Abbildung stellt den Content- und API-Server bereits als zwei strukturell unabhängige Instanzen dar. Obwohl diese Trennung optional ist, ist sie oft erstrebenswert. Siehe hierzu den vorhergehenden Abschnitt 3.3.

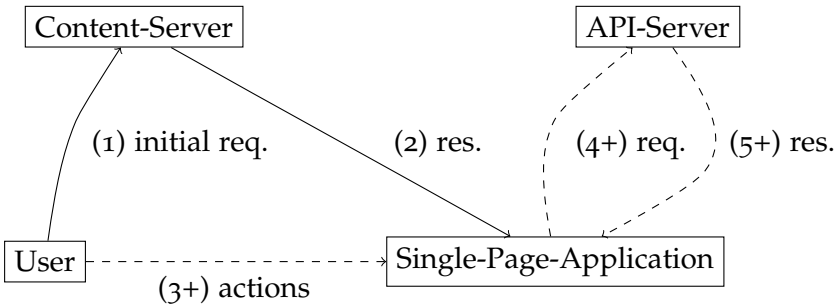


Abbildung 5: Asynchrone Client-/Server-Kommunikation

Durch die zentrale Anforderung an die im Rahmen dieser Arbeit entwickelte Applikation sich ähnlich nativer Software zu verhalten ist die Entwicklung einer SPA unabdingbar. Durch den Einsatz einer SPA ist es möglich Interaktionen sehr viel flüssiger zu handhaben: Ladezeiten können durch das Vorladen von Daten im Hintergrund und die, durch die fehlende Notwendigkeit strukturelle Informationen zu übertragen, kleinere Größe der benötigten Daten verringert werden. Außerdem ist es möglich durch visuelle Anpassungen wie Animationen ein flüssigeres Nutzungserlebnis zu simulieren, wo bei traditionellen Webseiten eine von Stillstand geprägte Wartezeit zu finden war.

Hierbei ist es wichtig zu bemerken, dass durch den Einsatz einer SPA nicht unbedingt die Vorteile einer Webseite verloren gehen. Der direkt Einstieg in eindeutige Pfaden und auch das teilen solcher bleibt erhalten – oft ein Vorteil gegenüber nativen Applikationen.

Ein Problem daran den Großteil der Arbeit erst auf Clientseite zu verrichten ist, dass es im Vergleich zu einer traditionellen Webseite zu einem verlängerten Seitenaufbau kommen kann. Um dies zu vermeiden kann der für JavaScript einzigartige Vorteil es auf Client und Server ausführen zu können genutzt werden. So ermöglicht das React-Framework (mehr zu diesem in Abschnitt 3.4.4.1) sogenanntes Server Side Rendering (SSR): hierbei wird die SPA universell auch auf dem Server zum generieren der angefragten Ansichten eingesetzt, so dass ähnlich zu traditionellen Webseiten die vorbereitete Ansicht an den Client übertragen werden kann. Dadurch ergibt sich das beste aus beiden Ansätzen: der initiale Seitenaufbau wird nicht durch die Ausführung des JavaScript-Codes verlangsamt wodurch der Inhalt ist auch für Suchmaschinen leichter auffindbar ist woraufhin alle

weiteren Änderungen allerdings wieder dynamisch lokal ausgeführt werden. Problematisch hierbei ist die zusätzliche Belastung des Servers, besonders bei für jeden Nutzer individuell zusammengestellten Inhalten. Obwohl geplant ist dies langfristig umzusetzen, war es im Rahmen dieser Arbeit nicht möglich das SSR effizient genug für den Produktiveinsatz zu gestalten.

Stattdessen wird allerdings auf einen Zwischenweg gesetzt, die sogenannten *App-Shells*. Diese werden durch den gleichen auch für SSR eingesetzten Code erstellt, dann allerdings als statische HTML-Seiten abgespeichert und ausgeliefert. Dabei wird an Stelle von dynamischen Inhalten auf Platzhalter gesetzt, welche dem Nutzer einen Eindruck über die zu erwartenden Daten gewähren und dadurch die gefühlte Performance verbessern. Zusätzlich kann in diese App-Shells auch ein Grundbestand der für einen Großteil der initial aufgerufenen Ansichten notwendigen Datensatzes eingebunden werden. Im Fall von Lawly sind dies beispielsweise die für die Darstellung der jeweils ersten Seite der nach Anfangsbuchstabe der Gesetzeskürzel unterteilten Gesetzesübersicht. Diese können daraufhin unmittelbar dargestellt werden während im Hintergrund der restliche Datensatz vorgeladen wird.

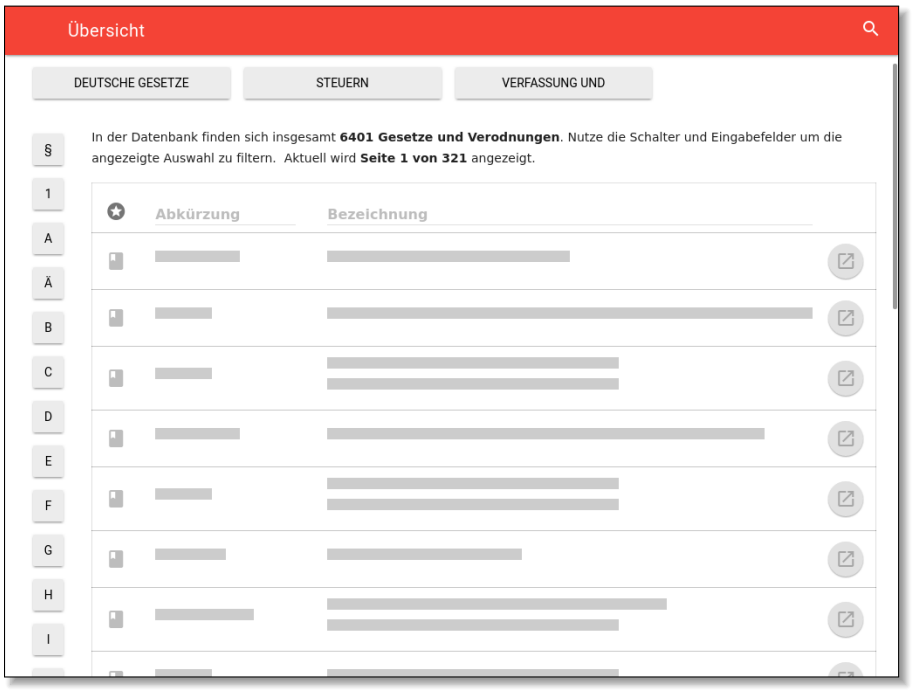


Abbildung 6: Gesetzesübersicht App-Shell (Tablet, horizontal)

3.4.2 *Offline-First*

Ein zentraler Teil der Nutzererfahrung von nativen Mobil- und Desktopanwendungen ist, dass sie, zumindest zu einem gewissen Maße, auch ohne aktive Internetverbindung nutzbar sind. Der zuvor beschrieben Ansatz einer SPA birgt bereits die Grundlage um solche

Funktionalität auch für eine Webapplikation zu realisieren: Nach dem initialen Laden der Seite werden alle Interaktionen des Nutzers von dem geladenen JavaScript gehandhabt. Interaktionen, für welche die Applikation keine weiteren Anfragen an den Server stellen muss, sind so also schon ohne eine Internetverbindung durchführbar. Bei einem erneuten Aufruf der Seite und bei der Verwendung von Teilen der Applikation, welche auf Serveranfragen angewiesen sind, ist allerdings ohne weitere Maßnahmen eine bestehende Internetverbindung notwendig. (Sauble, 2015)

Initial gilt es, den wiederholte Seitenaufrufe auch ohne Internetverbindung zu ermöglichen. Hierfür gibt es zwei standardisierte Technologien: Den *Application Cache* und *Service Worker*. Obwohl Service Worker die mächtigere Technologie sind, wird für Lawly, aufgrund der mangelnden Verfügbarkeit von Service Workern auf Apple Geräten, auf den Application Cache gesetzt.[6] Der Application Cache erlaubt es mithilfe eines *Cache Manifest* zuverlässig Dateien zu definieren, welche auch ohne Internetverbindung verfügbar sein sollen. Das ausführende System kümmert sich dann darum diese im Hintergrund zu laden und zu aktualisieren. Zukünftige Aufrufe der Applikation werden aus dem Cache ohne jegliche Netzwerklatenzen beantwortet. Dies hat den zusätzlichen Vorteil den Server und auch das eventuell limitierte Datenvolumen des Clients zu schonen: die im Manifest aufgeführten Dateien werden nur erneut abgerufen falls eine Änderung am Manifest vorgenommen wurde. Ansonsten wird bei jedem Seitenaufruf nur das Manifest vom Server abgefragt.

Um nicht nur statische Dateien bei fehlender Internetverbindung zur Verfügung stellen zu können gilt es, Anfragen innerhalb der Applikation so zu gestalten, dass sie nicht nur durch Zugriff auf die API, sondern auch aus einem auf dem Gerät zur Verfügung stehenden lokalen Speicher beantwortet werden können. Um dies zu ermöglichen wird auf eine zentrale Klasse gesetzt, welche sämtliche API-Anfragen verwaltet. Um auch hier Latenzen zu vermeiden bevorzugt diese Klasse beim Beantworten von Anfragen den lokalen Speicher gegenüber der externen API. Falls also die gesuchten Daten im lokalen Speicher vorhanden sind, werden diese unmittelbar verarbeitet. Gleichzeitig hierzu wird auch eine Anfrage an die externe Schnittstelle gestellt, mit deren eventueller Antwort gegebenenfalls die zuvor bereits durchgeführte Verarbeitung erneut ausgeführt sowie die lokal zwischengespeicherten Daten aktualisiert werden.

Zu guter Letzt ist es nicht nur notwendig, relevante Daten ohne Internetverbindung verfügbar zu machen, sondern auch Interaktionen lokal zu simulieren und bei Wiedererlangen einer Verbindung mit dem Server abzugleichen. Dies nennt sich optimistisches Nutzeransicht, da Aktionen gewissermaßen optimistisch lokal durchgeführt

6 Safari, welcher unter iOS auch als Grundlage für die Browser Dritter dient, implementiert noch keine Service Worker. Quelle: caniuse.com/#feat=serviceworkers, Stand 08/2016.

und erst im Nachhinein mit dem Server abgeglichen und eventuell durch den von diesem vorgegebenen faktischen Zustand überschrieben werden (Stubailo, 2015). Auch hierbei ist wieder die im letzten Absatz beschrieben zentrale Klasse zum Verwalten von Anfragen hilfreich. Falls diese feststellt, dass eine manipulierende Anfrage an den Server (also die HTTP-Methoden PUT, POST oder DELETE) fehlgeschlagen ist, werden die notwendigen Informationen gespeichert und die Anfrage bei wiedererlangen einer Verbindung zur API erneut ausgeführt.

Die Folge der bei einem solchen optimistischen Nutzerinterface ausgeführten Operationen wird abstrahiert in Abbildung 7 dargestellt. Gestrichelte Kanten sind in der Darstellung solche, die auf eine Internetverbindung angewiesen und von Verbindungslatenzen betroffen sind. Die leitende Zahl an jeder Kante spezifiziert auch hier wieder die Ausführungsordnung.

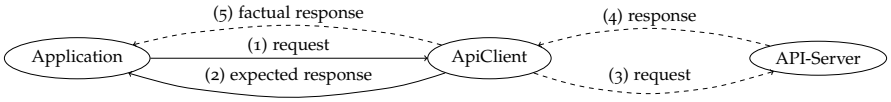


Abbildung 7: Optimistische Nutzerinterfaces

Da die Umsetzung solcher lokaler Speicher in den verschiedenen Browser stark variieren, wird auf die *localForage* Bibliothek der Mozilla Foundation gesetzt. Zwar bieten manche moderne Browser bereits sehr mächtige lokale Datenbanken, allerdings kommt für dieses Projekt eine Einschränkung auf diese wenigen Plattformen nicht in Frage. *localForage* ermöglicht es in allen verbreiteten Browsern zuverlässig einen implementationsunabhängigen Key/Value-Store einzusetzen.[7]

3.4.3 *Mobile-First*

Ein weiterer zentraler Aspekt einer PWA ist, dass sie auf sämtlichen Endgeräten gleichermaßen komfortabel verwendbar ist. Dabei gilt es dem *mobile-first* Ansatz zu folgen: das gesamte Nutzerinterface wird demnach initial für einen kleinen Smartphone-Bildschirm, gewissermaßen das schwächsten Glied der Kette, entwickelt. Dabei wird auf flexible Komponenten gesetzt, welche sich bei Verfügbarkeit von mehr Platz, also zum Beispiel auf Tablet- oder Desktopbildschirmen, fließend umverteilen und ausbreiten. (Gasston, 2014)

7 Die einzigen durch *localForage* nicht unterstützten Browser sind Versionen <8 des Internet Explorers, welche auf einen Marktanteil von unter 1% kommen. Quelle: https://www.netmarketshare.com/browser-market-share.aspx; Abgerufen 08/2016.

3.4.4 *Modularisierung*

Software Projekte aller Art leiden ab einer gewissen Größe unter ihrem eigenen Funktionsumfang, da der zugrunde liegende Code Gefahr läuft mit wachsendem Umfang unübersichtlich und dadurch schwer wartbar zu werden. Um dies zu vermeiden ist eine klare Codestruktur notwendig. Für Endnutzerapplikationen hat sich diesbezüglich das MVC Prinzip etabliert, bei welchem jede einzelne Ansicht einer Applikation durch drei ineinander verankerte Einzelteile beschrieben werden. Knapp zusammenfassend steht dabei der *Controller* als Verantwortlicher für das Durchführen von Aktionen und verarbeiten von Daten im Mittelpunkt. Dabei stellt er dem *View*, welcher für die Visualisierung zuständig ist, Informationen und Aktionen bereit und kommuniziert Änderungen und erhält Rohdaten vom gewissermaßen im Hintergrund befindlichen *Model*.

Mit der fortschreitenden Entwicklung zu dynamischeren und stärker in ihren Ansichten ineinander verzahnten Applikationen hat sich dieser Ansatz allerdings als problematisch erwiesen. Es ist nicht länger der Fall, dass eine einzelne Funktionalität nur in einer bestimmten Ansicht dargestellt wird. Bestandteile einer Applikation interagieren immer stärker miteinander, wobei es notwendig ist alle vom gleichen Datensatz abhängige Modelle synchron zu halten.

Um dieses Problem anzugehen hat sich speziell in der Webentwicklung in den letzten Jahren ein neuer Trend zu einer sehr viel stärkeren Modularisierung von Applikationen hervor getan. Die Vielzahl an Models, Views und Controllers werden dabei respektive durch einen globalen Zustand, eine Komponenten-Hierarchie und zentral definierte Aktionen abgelöst. In den folgenden Abschnitten wird auf diese Ansätze und die für ihre Anwendung verwendeten Technologien eingegangen.

3.4.4.1 *Komponenten-Hierarchien*

Die Komponenten einer nach diesem Prinzip aufgebauten Applikation ergeben dabei durch flexible Kombination die dynamische Nutzeroberfläche. Eine einzelne Komponente ist dabei ein möglichst in sich geschlossenes System, welches sein Umfeld nicht beeinflusst und nur von einem klar definierten, von außen übergebenen, Zustand abhängt. Also eine direkte Abbildung von einem übergebenen Zustand zu einer Darstellung, in JavaScript wie folgt darstellbar: `state => view`. Durch hierarchische Vererbung an andere Komponenten können so in einer sich ergebenden Baumstruktur komplexe Interfaces umgesetzt werden, in welcher der Zustand von oben nach unten vererbt wird. Einzelne Komponenten bleiben so in sich selbst leicht verständlich und gut testbar.

Die verbreitetste Plattform, welche diesen Ansatz in der Webentwicklung gewissermaßen massentauglich machte, ist das JavaScript-Framework *React*. Zusätzlich entstand kurze Zeit später *React Native*, welches die Prinzipien von React auf die Entwicklung von nativen Android- und iOS-Applikationen überträgt. Bei der Verwendung von React Native wird zwar weiterhin JavaScript-Code geschrieben, allerdings für die Darstellungsebene komplett auf native Schnittstellen gesetzt. Während der Kompilierung einer solchen Applikation wird die *Geschäftslogik* von der *Darstellungslogik* getrennt und auf dem Endgerät, anders als im Browser, in zwei sich gegenseitig nicht blockierenden Threads ausgeführt. Da diese Möglichkeit, native Applikationen in JavaScript ohne erlernen weiterer Paradigmen zu entwickeln, ein vollständiges Alleinstellungsmerkmal des React-Ökosystems ist, wurde es als zentrale Bibliothek für die Entwicklung von Lawly gewählt.

Da Änderungen des DOMs rechenintensiv sind und alle Operationen einer Webseite nur in einem einzelnen Thread ausgeführt werden, kümmert sich React darum, solche Änderungen zu bündeln. Dafür hält React eine virtuelle Representation des DOMs vor, sammelt in dieser entstehende Änderungen und wendet sie gebündelt auf die eigentliche Darstellung im Browser an. Die beiden folgenden verfolgten Ansätze, uni-direktionaler Datenfluss und unveränderbare Datenstrukturen, helfen, diesen Vorgang noch effizienter zu gestalten.

3.4.4.2 *Uni-direktionaler Datenfluss*

Um Komponenten leicht testbar und auch bei starker Verzahnung möglichst unabhängig voneinander zu gestalten, ist es notwendig, den Zustand auszulagern. Hierbei verfolgen wir das Prinzip eines uni-direktionalen Datenflusses. Dabei verfügt die Applikation über einen globalen Zustand (Der vom **Store** verwaltete **State**), welcher über die Komponenten-Hierarchie in zugeschnittenen Teilmengen vererbt wird.

Anstatt, dass wie beim MVC-Prinzip jede Komponente nun den ihr zugeordneten Teil des Zustandes manipuliert, werden solche Veränderungen global ausgeführt; der lokale Zustand darf nicht direkt verändert werden. Über die gleiche Vererbungsstruktur werden hierfür nicht nur die Daten selbst, sondern auch Funktionen vererbt (sogenannte **Action Creators**), über welche exklusiv Einfluss auf den globalen Zustand genommen werden kann.

Eine von einer solchen Funktion erstellte Aktion (**Action**) ist dabei ein serialisierbares Objekt mit klar spezifiziertem Typ und Struktur. Wird so eine Aktion auf den Store angewendet, transformiert dieser auf ihrer Grundlage den Zustand und übergibt den neuen Zustand wiederum an die Komponenten-Hierarchie. Durch diese zentrale Aktualisierung und Vererbung wird die Integrität der zur Darstellung einer Komponente dienenden Daten zu jedem Zeitpunkt garantiert.

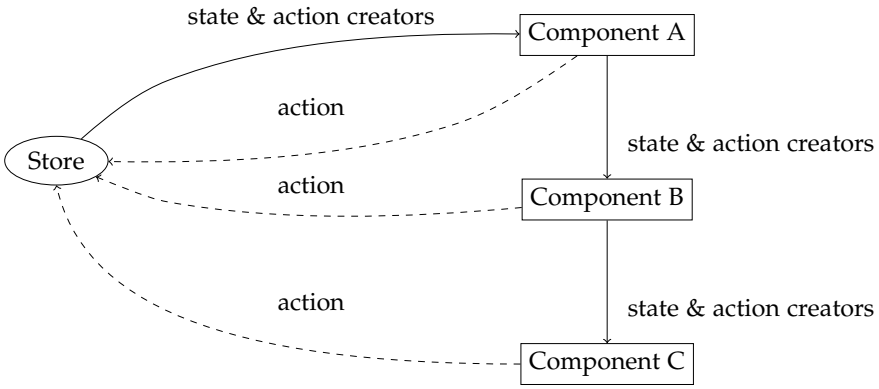


Abbildung 8: Uni-direktionaler Datenfluss

Die Zentralisierung von Zustand und Zustand-manipulierenden Aktionen führt zusätzlich zu einer leichter durchschaubaren und insbesondere vorhersehbaren Applikation. Durch Veränderung des globalen Zustandes können zentral deterministisch Ansichtsänderungen vorgenommen werden. Dadurch ergeben sich auch die Möglichkeiten den Zustand persistent für zukünftige Aufrufe der Applikation zu speichern, einzelne Aktionen zu simulieren und, da der Zustand niemals direkt verändert sondern nur in einen neuen transformiert wird, gegebenenfalls rückgängig zu machen.

Initial wurde dieser Ansatz 2014 von Facebook unter dem Namen *flux* präsentiert (Chen, 2014). Seitdem wurde das Konzept von der JavaScript-Gemeinschaft intensiv aufgegriffen und hat sich in der hier vorgestellten und eingesetzten Form durch die *Redux*-Bibliothek[8] stark verbreitet.

3.4.4.3 *Unveränderbare Daten*

Zur effizienten und konsequenten Umsetzung der beiden zuvor beschrieben Prinzipien wird auf sogenannte unveränderbare Datenstrukturen gesetzt. Dies ergibt Sinn, da der globale Zustand wie zuvor erläutert niemals direkt verändert werden darf. Als Referenz für die folgenden Erläuterungen und das Beispiel dient zentral Byron (2015).

Der verbreitetere und aus der objektorientierten Programmierung bekannte Ansatz ist es, mit Instanzen von Objekten zu arbeiten, welche während der Laufzeit eines Programms manipuliert werden. Hierbei stellen Objekte meist Methoden zur Verfügung, über welche ihr Zustand manipuliert werden kann. Das Problem dabei ist, dass solche Entwicklungsmuster in nicht vorhersehbaren Zuständen resultieren können. In Listing 3 ist Beispielhaft die Funktion `take` definiert, welche den Wert eines Attributes eines ihr übergebenen Objektes zurückgibt. Nachträglich verändert sie allerdings den eigentlichen Wert

8 redux.js.org

innerhalb des Objektes.[9] Falls diese Funktion nicht vom sie einsetzenden Programmierer geschrieben wurde oder auch nur grade nicht direkt vorliegt könnte dies zu unvorhersehbaren Resultaten im weiteren Programmablauf führen.

```
function take(ref, key) { return ref[key]--; }
const obj = { foo: 42 };
const foo = take(obj, 'foo');
// obj: { foo: 41 }; foo: 42
```

Listing 3: Pass-by-reference und mutierbare Objekte

Unveränderbare Datenstrukturen lösen dieses Problem, indem jede auf ihnen ausgeführte Operation ein neues Objekt zurückgibt, anstatt das alte direkt zu verändern (siehe Listing 4).

```
import { List } from 'immutable';
const arr = [1, 2, 3];
arr.push(4);
// arr: [1, 2, 3, 4]
const list1 = List([1, 2, 3]);
const list2 = list1.push(4);
// list1: [1, 2, 3]; list2: [1, 2, 3, 4]
```

Listing 4: Mutable Arrays und immutable Listen

Für ein besseres Verständnis kann man *immutables* statt als Objekte als individuelle Werte betrachten. Werte sind in den meisten Programmiersprachen beispielsweise Integer oder Strings. Werden diese verändert, wird immer ein neuer Wert zurück gegeben, niemals der eigentliche mutiert. Objekte hingegen sind eher eine Art Sammlung von Referenzen und Werten, welche innerhalb des Objektes verändert werden können – dadurch bleibt das Objekt allerdings das selbe. Werden statt solchen mutierbaren Objekten allerdings unveränderbare Datenstrukturen eingesetzt, können auch diese als Werte behandelt werden: Veränderungen erzeugen einen neuen Wert, ohne den alten direkt zu mutieren.

Um nun nicht bei jeder Mutation eine veränderte tiefe Kopie des originalen Objektes zu erstellen und damit unnötig Speicher zu belegen wird bei der verwendeten Bibliothek, *immutablejs*, für die interne Repräsentation unveränderbarer Objekte auf sogenannte gerichtete azyklische Graphen mit gemeinschaftlicher Nutzung (eng.: *directed azyclic graphs with structural sharing*) gesetzt. In Abbildung 9 wird dies visualisiert: Im originalen Graph mit Wurzel a soll der verschachtelte Kind-Knoten g manipuliert werden. Um dies mit möglichst wenig Aufwand zu erreichen und den ursprünglichen Graphen nicht

9 Interessant hierbei ist auch, dass die Variable `obj` als `const`, also Konstante, definiert wurde: Konstant bedeutet hier allerdings nicht, dass der Wert des Objektes unverändert bleibt, sondern die in der Variable gespeicherte Referenz zu einem bestimmten Objekt. Das referenzierte Objekt ist allerdings veränderbar. Obwohl dieses Verhalten bei angehenden Programmierern oft für Verwirrung sorgt, ist es ein auch aus anderen Sprachen wie C++ (`const`) oder Java (`final`) bekanntes Verhalten. Unveränderbare Datenstrukturen lösen dies.

zu verändern, werden die hier gestrichelt dargestellten Knoten *a*, *c* und *g* in Form der als gepunktete Nachbarn dargestellten Knoten, respektive *a2*, *c2* und *g2*, kopiert. Die Mutation findet auf dem neu erstellten Knoten *g2* statt und der neue Graph mit Wurzel *a2* setzt sich so speichersparend aus, im dargestellten Beispiel zu über 50%, bereits existierenden unveränderten Knoten zusammen (durchgezogene Umrandungen). Falls im System keine Referenz mehr auf den Ursprungsgraphen *a* besteht, können die gestrichelten Knoten automatisch von der Speicherbereinigung endgültig gelöscht werden.

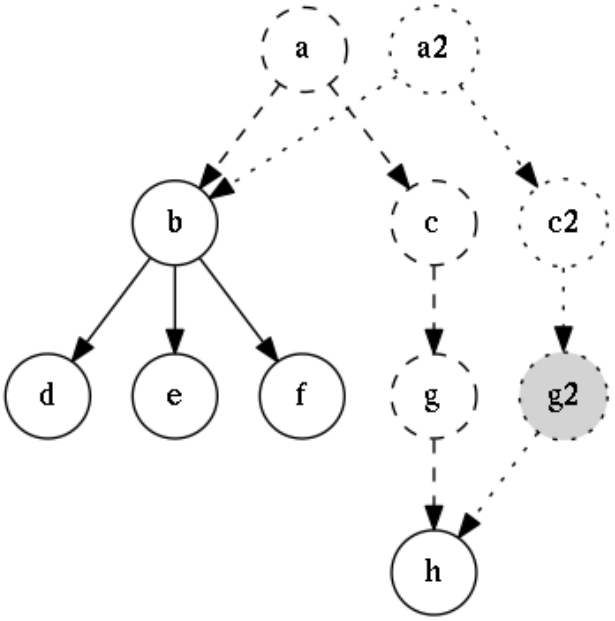


Abbildung 9: Mutation eines gerichteten azyklischen Graphen mit gemeinschaftlicher Nutzung

Insgesamt ergeben sich aus der Verwendung von unveränderbaren Datenstrukturen so gleich mehrere Vorteile: Einerseits ist es, wie bereits erläutert, für Entwickler leichter über Veränderungen von Datenstrukturen zu urteilen. Außerdem ermöglichen unveränderbare Datenstrukturen es, effizient zu prüfen ob die Aktualisierung einer Komponente bei Veränderung des globalen Zustands überhaupt notwendig ist – wenn also der neue Zustand in seiner Identität der selbe wie der vorherige ist, hat er sich nicht verändert. Bei traditionellen Objekten wäre es hier notwendig einen tiefen Vergleich jedes Attributes durchzuführen.

4

IMPLEMENTATION

Im Folgenden wird auf die konkrete Implementation der zu Beginn angekündigten beiden Phasen, Datenaggregation (Abschnitt 4.1) und Applikationsimplementation (wieder unterteilt in Server, Abschnitt 4.2, und Client, Abschnitt 4.3) eingegangen. Besonders im Client-Abschnitt erfolgt dies nicht umfänglich sondern für einzelne Bereiche beispielhaft, da dies sonst den Rahmen diese Dokuments übersteigen würde. Zum Abschluss wird in Abschnitt 4.4 auf die Veröffentlichung der beiden Applikationsbestandteile für den Produktiveinsatz eingegangen.

Für die Versionskontrolle des Projekts wird auf drei `git`-Repositories gesetzt, welche über den Anbieter GitHub[1] verwaltet werden: Einmal für den Server-Quelltext (*lawly_api*[2]), den Client-Quelltext (*lawly_web*[3]) und das in Markdown verfasste vorliegende Dokument (*bsc*[4]). Um die Verbindung zwischen den folgenden Beschreibungen und dem eigentlichen Quelltext zu erleichtern werden besprochene Dateien in den Fußnoten anhand ihres Repositories und Pfades mit Hyperlink direkt zu GitHub referenziert.

4.1 DATEN-AGGREGATION

Wie in Abschnitt 2.5 bereits angesprochen, gilt es vor Beginn der eigentlichen Implementierung die Daten, also die Gesetzestexte, zu aggregieren. Dabei wird auf die von *gesetze-im-internet.de* zur Verfügung gestellten XML-Dateien zurückgegriffen.

Ohne viel Magie wird dabei ein zweistufiger Prozess eingesetzt:

1. Download der Rohdaten in ein temporäres Verzeichnis.
2. Normalisieren der Daten und eintragen in die Datenbank.

Für den ersten Schritt gilt es zuerst das zur Verfügung gestellte XML-Inhaltsverzeichnis zu verarbeiten.[5] Um mit XML-Dokumenten arbeiten zu können wird `libxml` als Parser mit XPath als Query-Sprache

1 github.com
2 github.com/ahoereth/lawly_api
3 github.com/ahoereth/lawly_web
4 github.com/ahoereth/bsc
5 lawly_api: scripts/fetchGiiXmls.js

eingesetzt. Aus dem Inhaltsverzeichnis werden so die Links zu einzelnen Gesetzen extrahiert. Diese, zum Zeitpunkt des Verfassens, 6456 Links verweisen auf individuelle ZIP-Dateien welche ihrerseits wieder XML-Dateien beinhalten die die gemeinschaftlich ein Gesetz ergebenden Normen auflisten.

Um den *gesetze-im-internet.de*-Server nicht zu überlasten oder dazu zu verleiten Anfragen des Skriptes zu blockieren, wird maximal eine Datei pro 50 Millisekunden angefordert – dies wurde durch über Versuche an mehreren Tagen als ein akzeptabler Wert bestimmt. Sequentiell werden geladene ZIP-Dateien in einen Buffer geladen, im Speicher entpackt und, für die weitere Verarbeitung, das enthaltene XML-Dokument in einen temporären Order auf dem System geschrieben.

Sobald alle XML-Dokumente in ihrer aktuellen Form auf dem System zwischengespeichert sind werden diese weiter verarbeitet.[6] Eine Datei wird zuerst in den Speicher eingelesen und ihr Inhalt mithilfe der `GiiParser`-Klasse[7] verarbeitet. Diese traversiert den XML-Baum wiederum mit XPath und gleicht beispielsweise Unregelmäßigkeiten in Knoten-Bezeichnungen aus um eine über sämtliche Gesetze hinweg gleichmäßige Datenstruktur zu erhalten. Die eigentlichen in den Normen enthaltenen Fließtexte werden zusätzlich von der verwendeten Auszeichnungssprache, einer Mischung aus HTML und XML, zu Markdown übersetzt – auch hierbei steht wieder eine Normalisierung über verschiedene zum Einsatz kommende Strukturen im Vordergrund.

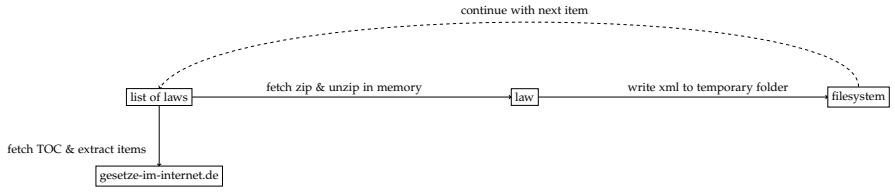


Abbildung 10: Sequentielles aggregieren der Rohdaten

Hierbei fiel die Entscheidung auf den Einsatz von Markdown, da es ein auch ohne jegliche Nachbearbeitung leicht les- und auch verfassbare Auszeichnungssprache ist. Da sie zum Beispiel keine XML-Tags beinhaltet ist es möglich auf den rohen Texten zu suchen. Außerdem ist es geplant Markdown im weiteren Entwicklungsprozess der Applikation möglichst weitflächig einzusetzen. Durch seine natürliche Struktur ist Markdown denkbar einfach für neue Anwender erlernbar und auch Vergleiche (sogenannte *diffs*) verschiedener Textversionen gut visualisierbar.

Aufgrund der schieren Menge an Normen, aktuell über 112.000, traten bei der Implementierung dieses Prozesses gleich mehrere limitie-

6 lawly_api: scripts/parseGiiXmls.js
7 lawly_api: scripts/GiiParser.js

rende Faktoren auf. Nicht nur kamen die für die Übersetzung eingesetzten Maschinen an ihre Grenzen, sondern auch auf Datenbank-Seite führten die rasanten Eintragungen tausender Datensätze zu einem massiven Rückstau. Um dies zu vermeiden verfolgt die aktuelle Implementierung einen streng sequentiellen Ansatz bei dem erst eine weitere Datei bearbeitet wird sobald das Ergebnis der vorhergehenden Operation erfolgreich in die Datenbank eingetragen wurde. Aktuell benötigen eine bei Amazon Web Services (AWS) angemietete `t2.micro` Elastic Cloud Computing (EC2) Instanz in Verbindung mit einer gleichwertigen Relational Database Service (RDS) PostgreSQL Instanz für den gesamten Prozess unter voller Auslastung knapp eine dreiviertel Stunde.

Um diesen Prozess in Zukunft vollständig zu automatisieren ist eine zentrale Optimierung notwendig: Es muss schon vor dem eigentlichen Zugriff auf die eigene Datenbank festgestellt werden können, ob sich Gesetze geändert haben. Entweder muss dies durch die Metadaten der durch die Quelle zur Verfügung gestellten ZIP-Dateien oder am besten noch zuvor beispielsweise durch Analyse der Webseite *gesetze-im-internet.de* oder direkt des Bundesgesetzblattes geschehen. Regelmäßig alle ZIP-Dateien herunterzuladen und zu analysieren kommt nicht in Frage. Im Rahmen dieser Arbeit wurde hierfür noch keine effiziente Lösung gefunden oder gar implementiert.

4.2 SERVER

Durch die Entscheidung den Server als reine API umzusetzen, fällt dieser denkbar einfach aus. Im Fokus der folgenden Beschreibung liegen daher die definierten REST-Schnittstellen und das standardisierte Antwortformat. Zusätzlich muss der Server effizient mit Authentifizierung von Anfragen umgehen können und durch starke Modularisierung für eine zukünftige Erweiterung gerüstet sein. Um eine einfachere Modularisierung zu ermöglichen wird, wie zuvor erläutert, das Express-Framework eingesetzt.

4.2.1 *Authentifizierung*

Die Authentifizierung von Anfragen findet mithilfe einer *Middleware* statt.[8] Middlewares sind Funktionen welche von Express zwischen das Eingehen einer Anfrage und ihrer Bearbeitung durch einen bestimmten Handler geschaltet werden und die enthaltenen Request- und Response-Objekte erweitern können. Die implementierte Authentifizierungs-Middleware überprüft, ob der `Authorization`-Header gesetzt ist. Wenn dies gilt, wird aus diesem der JWT extrahiert und, mithilfe der Open Source Bibliothek `node-jsonwebtokens` auf Validität überprüft. Zusätzlich wird

8 lawly_api: /server/config/authentication.js#L144ff

überprüft ob der Token nur noch weniger als 24 Stunden gültig ist und gegebenenfalls ein neuer ausgestellt. Die im JWT enthaltenen Nutzerdaten und der eventuelle neue Token werden dem Request-Objekt hinzugefügt und dieses an nachfolgende Middlewares bzw. den Route-Handler weitergereicht. Vergleiche in Bezug hierauf auch Abschnitt 3.1.

4.2.2 *HTTP-Endpunkte*

Um große Applikationen mit vielen HTTP-Endpunkten besser Unterteilen zu können bietet Express die Möglichkeit, bestimmte Uniform Resource Locator (URL)-Pfade in *Routern* zu bündeln. Dabei wird ein solcher Router angesprochen, sobald auf dem ihm zugeordneten beziehungsweise einem diesem untergeordneten Pfad eine Anfrage eintrifft. Der Router kümmert sich dann um die weitere Verteilung an ihm zugeordnete Handler.

Obwohl die aktuelle API nur einen sehr überschaubaren Umfang hat, wird bereits vorausschauend auf die Unterteilung in zwei Router gesetzt: Respektive sind diese jeweils für Anfragen an die Pfade `/laws` und `/users` zuständig.[9] Wie in Abschnitt 3.4 angekündigt, ergeben sich die Aufgaben dieser Router bereits selbsterklärend aus den ihnen zugeordneten HTTP-Pfaden.

Unter `/laws` finden sich lediglich zwei GET Schnittstellen, da kein Nutzer der Applikation zum Eintragen oder ändern von Gesetzen bemächtigt ist und zu diesem Zeitpunkt kein die API nutzendes administratives Interface existiert.[10] Direkte GET-Anfragen an den Wurzel-Pfad werden mit der Übersicht über alle verfügbaren Gesetze beantwortet. Um das zur Verfügung gestellte Paket dabei möglichst klein zu halten, ist in der Übersicht jedes Gesetz nur mit seinem eindeutigen Kürzel (z.B. *BGB*) und seinem Titel (z.B. *Bürgerliches Gesetzbuch*) vertreten. Mithilfe des Query-Parameters `search` ist es hier zusätzlich möglich das Ergebnis durch eine Volltextsuche zu filtern.

Für detailliertere Informationen über ein Gesetz steht der Endpunkt `/laws/:groupkey` zur Verfügung, dabei wird als Wert für den `groupkey` Parameter das Kürzel eines Gesetzes bezeichnet – die Bezeichnung *groupkey* resultiert daraus, dass das Kürzel eines Gesetzes in der Datenbank den eindeutigen Schlüssel für eine Sammlung von Normen beschreibt.

Unter `/users` hingegen werden nicht nur lesende GET-, sondern auch schreibende Anfragen (also PUT, POST oder DELETE) bereitgestellt.[11] So dient POST `/users` zum Beispiel der Erstellung eines neu-

9 lawly_api: /server/routes/index.js
10 lawly_api: /server/routes/laws.js
11 lawly_api: /server/routes/users.js

en oder Authentifizierung eines bestehenden Benutzeraccounts.[12] Die zweite zentrale aktuell bereitgestellte Route ist etwas verschachtelter: `PUT /:email/laws/:groupkey/:enu?`. Hierüber können Nutzer per PUT Anfrage, also einer Anfrage um einen bestehenden Datenbestand zu verändern, Gesetze und Normen in ihre Sammlung aufnehmen. Der `:email` Parameter spezifiziert dabei, wessen Sammlung verändert werden soll – aktuell gilt es, dass Nutzer nur zur Veränderung ihrer eigenen Sammlung autorisiert sind (die Adresse wird also mit dem JWT abgeglichen), langfristig ist es aber denkbar, dass auch Gruppen gemeinsame Sammlungen anlegen und bearbeiten können. `:groupkey` spezifiziert das Kürzel des Gesetzes und `:enu` die eindeutige Enumeration einer Norm innerhalb des Gesetzes – falls `:enu` nicht angegeben ist, wird die Wurzel-Norm zugegriffen. Innerhalb des Anfragen-Körpers wird hierbei nun ein JSON-Objekt mit dem Feld `starred` erwartet, dessen Boole'scher Wert angibt, ob die spezifizierte Norm gemerkt oder vergessen werden soll.

4.2.3 *Antworten*

Um Antworten gleichmäßig zu gestalten wurde ein Klasse implementiert, welche eine Response-Objekt erwartet (welches jedem Route-Handler übergeben wird) und Methoden anbietet dieses einheitlich zu verarbeiten.[13] Grundlegend gilt hierbei, dass jede Methode der Klasse für einen bestimmten HTTP-Statuscode zuständig ist und es damit vereinfacht, diesen korrekt zu setzen. Zusätzlich werden die an eine Antwort angehängten Daten dabei in ein einheitliches JSON-Objekt verpackt, welches zusätzliche Informationen wie den Erfolg oder Misserfolg der Operation und einen eventuell durch die zuvor beschriebe Middleware generierten neuen Autorisierungstoken beinhaltet. Außerdem werden ein paar Unregelmäßigkeiten in der Handhabung von Antworten ausgeglichen. Ein Beispiel hierfür ist der Statuscode `204 No Content`, bei welchem laut Standard generell keine Daten enthalten sein dürfen (Fielding & Reschke, 2014) und dessen Header von älteren Versionen des Internet Explorer komplett ignoriert wird – um dies zu vermeiden und auch bei `204 No Content` einen erneuerten Token senden zu können, werden solche Antworten auf Status `200 OK` umgeschrieben.

4.3 CLIENT ARCHITEKTUR

Ähnlich dem Abschnitt 4.2 setzt auch die Client Applikation im Sinne einer besseren Übersichtlichkeit und Testbarkeit auf stark modu-

12 Hierbei handelt es sich um den einzigen Endpunkt, an dem das Nutzerpasswort erwartet wird. Alle anderen Endpunkte, wenn in ihrem Zugriff beschränkt, benötigen für die Autorisierung einen hierüber ausgestellten oder von der in Abschnitt 4.2.1 beschrieben Middleware erneuerten JWT.

13 lawly_api: /server/helpers/reply.js

laren Code. Da eine komplette Analyse des Applikationsquelltextes bei rund 6000 Zeilen Code den Rahmen dieser Arbeit sprengen würde, wird im folgenden beispielhaft anhand der Gesetzesübersicht die konkrete Implementation erläutert (Abschnitt 4.3.1). Außerdem werden aufgrund ihrer für die Zielsetzung besonderen Relevanz die Umsetzung der Offline-Funktionalität (Abschnitt 4.3.2) und der lokalen Suchfunktion (Abschnitt 4.3.3) besprochen. Da das Ziel der Implementation ein nicht nur theoretisch, sondern auch praktisch an Browser auslieferbares Paket ist, wird zum Abschluss auf den umgesetzten Buildprozess eingegangen (Abschnitt 4.4).

4.3.1 *Gesetzesübersicht*

Im Folgenden wird, beispielhaft für die Gesamtarchitektur, detailliert die Implementation der Gesetzesübersicht, siehe Abbildung 11[14], behandelt. Diese listet die Gesetze auf, bietet die Möglichkeit über die Buch-Icons links in der Tabelle Gesetze zu speichern und über die Action-Buttons rechts die Individualansichten aufzurufen. Zusätzlich können verschiedene Filter angewendet werden: oben kann aus einer (noch zu erweiternden) Liste von vordefinierten Sammlungen gewählt und über die Schalter links nur Gesetze mit einem bestimmten Kürzel-Anfangsbuchstaben angezeigt werden. Mit den drei Tabellenkopfspalten können außerdem respektive von links nach rechts nur markierte Gesetze oder nur solche mit einem bestimmten Bestandteil in Kürzel oder Bezeichnung angezeigt werden. Damit der Nutzer trotz der vielen Optionen den Durchblick behält werden direkt über der Tabelle die angewendeten Filter und deren Ergebnismenge knapp in natürlicher Sprache zusammengefasst.

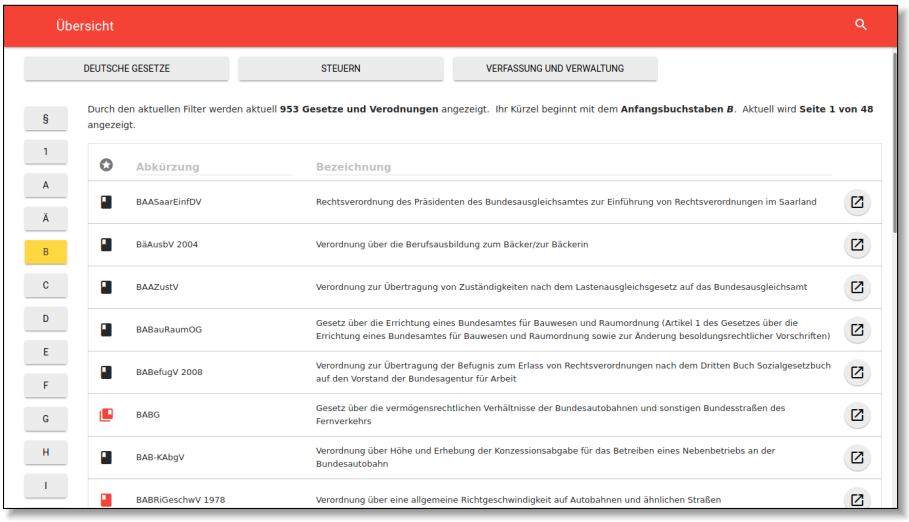


Abbildung 11: Gesetzesübersicht (Desktop)

14 web.lawly.de/gesetze

Die für diese Darstellung entwickelten Redux-Module und React-Komponenten werden als Vererbungshierarchie in Abbildung 12 dargestellt – zusätzlich verwendete Elemente wie das übergeordnete Layout oder Komponenten aus anderen Bibliotheken wurden dabei ausgelassen.

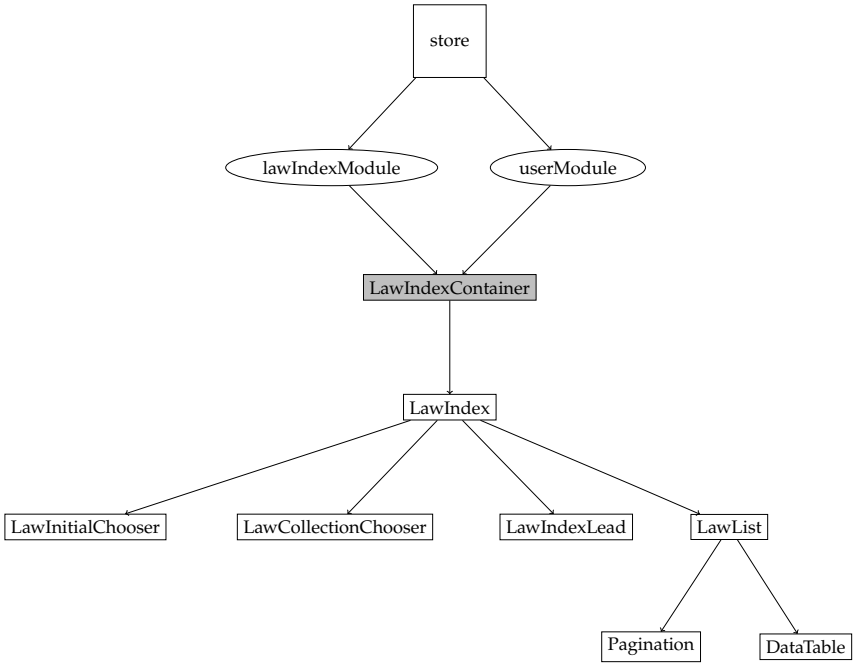


Abbildung 12: Modulhierarchie Gesetzesübersicht

Die Gesetzesübersicht benötigt die Daten aus zwei *Redux-Modulen*: dem `lawIndexModule`[15] und dem `userModule`[16]. Redux-Module sind dabei eine Zusammenfassung von für das Arbeiten mit einem bestimmten Teil des Zustandes nötigen Funktionalitäten. Dies sind einerseits Selektoren, zum strukturierten Lesen, und anderseits Action Creators, zum Eintragen und Manipulieren von Daten (siehe Abschnitt 3.4.4.2). Durch die Zentralisierung dieser Bestandteile wird eine einheitliche Interaktion mit dem Zustand garantiert.

Im Mittelpunkt zwischen Redux-Modulen und React-Komponenten steht der `LawIndexContainer`[17], in der Abbildung ausgefüllt dargestellt. Obwohl in der Implementation eigentlich selbst eine Komponente, dient er als Schnittstelle zur Abstraktion der Verbindung. Die ihm untergeordneten Komponenten können so vollständig ohne Wissen über die Herkunft ihrer Daten verwendet und dementsprechend einfach getestet werden.

Die Gesetze erhält der Container durch die `lawIndexModule`-Selektoren. Da der Zustand aber zentralisiert ist, filtert der Container diese nicht selbst. Stattdessen werden über die Action Creators die

15 lawly_web: /src/modules/law_index.js#L96ff
16 lawly_web: /src/modules/user.js
17 lawly_web: /src/containers/LawIndexContainer.js

vom Nutzer angewendeten Filter in den Zustand geschrieben, welcher dann durch die Hierarchie propagiert wird. Dabei werden auch die Selektoren von der Neuerung benachrichtigt und filtern den Index neu.

Die Selektoren gestalten komplexe Transformationen durch die Verkettung von reinen (nebeneffektfreien) Funktionen einfach und mithilfe von Memorisierung der in der Kette entstehenden Zwischenergebnissen effizient. Dies wird in Listing 5 dargestellt: Bei der Auswahl des Anfangsbuchstaben *B* (Abbildung 11) muss so das Ergebnis des `getLawsByCollection`-Selektors[18] nicht neu berechnet werden, da seine Eingabewerte sich nicht verändert haben. In der Kette weiter hinten angesetzte Selektoren, wie beispielsweise die für die aktuell angezeigte Seite (Seitenschalter nicht im Bild), werden, da sich ihre Eingabe geändert hat, automatisch neu berechnet. Der Container erhält so nur das bereits endgültig gefilterte Ergebnis und reicht es an die ihm untergeordneten Komponenten weiter. Die dargestellte Selektorenkette ist dabei nur ein Ausschnitt der eigentlich in der Applikation eingesetzten.[19]

```
import { createSelector } from 'reselect';
const getLawIndex = state => state.get('law_index');
const getCollection = state => state.get('collection');
const getInitial = state => state.get('initial');
const getLawsByCollection = createSelector(
  [getLawIndex, getCollection],
  (laws, collection) => laws.filter(/*[...]*/)
);
const getLawsByInitial = createSelector(
  [getLawsByCollection, getInitial],
  (laws, initial) => laws.filter(law =>
    law.get('groupkey')[0].toLowerCase() === initial
  )
);
export const getLawsByPage = createSelector(/*[...]*/);
```

Listing 5: Ausschnitt der Gesetzesübersicht-Selektoren

Die direkt unter dem `LawIndexContainer` angeordnete Komponente ist `LawIndex`.[20] Diese ist zentral nur für die Weiterverteilung der ihr übergebenen Attribute und die visuelle Aufteilung der Ansicht zuständig.

Bei einem Blick auf Listing 6 fällt auf, das für die Umsetzung der Komponentenhierarchie kein reines JavaScript, sondern *JSX* eingesetzt wird. JSX orientiert sich an der von HTML bekannten Struktur, welche durch ihre hierarchische Natur für die Darstellung dieses Konzeptes sehr gut geeignet ist. Obwohl es auch möglich ist, React ohne diese rein visuelle Vereinfachung zu nutzen, wird so viel unnötiger Code gespart und Übersichtlichkeit gewonnen. Außerdem wird in

18 lawly_web: /src/modules/law_index.js#L135-L142
19 lawly_web: /src/modules/law_index.js#L96ff
20 lawly_web: /src/components/laws/LawIndex.js

diesem Listing die Einfachheit des angewendeten funktionellen Ansatzes klar: Reine Komponenten sind nur eine direkte Abbildung ihrer Eingabe zu einer Darstellung. React kümmert sich hierbei wieder um die Effizienz. Ähnlich wie bei den zuvor beschrieben Selektoren wird eine solche Abbildung nur erneut berechnet, wenn sich ihre Eingabe verändert hat – durch den Einsatz von in Abschnitt 3.4.4.3 beschriebenen unveränderbaren Datenstrukturen ist dieser Vergleich besonders effizient umsetzbar.

```
import { Grid, Cell } from 'react-mdl';
export const LawIndex = ({
  initials, selectInitial, selectedInitial, /* [...] */
}) => (
  <Grid>
    <Cell>
      <LawCollectionChooser {/* [...] */} />
    </Cell>
    <Cell>
      <LawInitialChooser
        initials={initials}
        selected={selectedInitial}
        onSelect={selectInitial}
      />
    </Cell>
    <Cell>
      <LawIndexLead {/* [...] */} />
      <LawList {/* [...] */} />
    </Cell>
  </Grid>
);
```

Listing 6: Vereinfachte LawIndex Komponente

Erst in der ihr untergeordneten Ebene wird mit den Daten im eigentlichen Sinne gearbeitet. Wieder beispielhaft zeigt dafür Listing 7 die Komponente zur Darstellung der Initialen-Auswahl: diese bildet, wiederum als reine Funktion, die übergebene Liste von Anfangsbuchstaben auf die einzelnen Schalter ab. Per `onClick`-Handler wird das Initial eines Schalters über die auch durch die Hierarchie vererbte `onSelect`-Funktion an den Store gesendet und durch die Propagierung des aktualisierten Zustandes der Schalter dessen `initial` mit dem `selected` Wert übereinstimmt farbig dargestellt.

```
import { Grid, Cell, Button } from 'react-mdl';
export const LawInitialChooser = ({
  initials, selected, onSelect
}) => (
  <Grid>
    {initials.map(initial => (
      <Cell>
        <Button
          colored={initial === selected}
          onClick={() => onSelect(initial)}
        >
          {initial}
```

```
13          </Button>
14        </Cell>
15      ))}
16    </Grid>
17  );
```

Listing 7: Vereinfachte LawInitialChooser Komponente

Obwohl die `LawInitialChooser`-Komponente ein Blattknoten des Graphen aus Abbildung 12 ist, vererbt sie selbst weiter an importierte Komponenten aus der `react-mdl` Bibliothek. Diese Bibliothek ist eine Implementierung von Googles Material Design Guidelines auf Grundlage von HTML-Elementen wie `div` oder `button`. An dieser Stelle ist es allerdings auch denkbar, dass statt HTML-Elementen native Android- oder iOS-Elemente eingesetzt werden – der hier implementierte Code ist von der in der nächsten Ebene der Hierarchie eingesetzten Architektur unabhängig. Einem späteren Austausch der `react-mdl`-Komponenten durch native Komponenten für eine native Umsetzung einer Applikation mit ähnlicher Funktionalität steht also nichts im Wege.

4.3.2 *Offline-Funktionalität*

Wie zuvor dargelegt, ist die Funktionalität der Applikation auch ohne Internetverbindung zu gewährleisten. Um dies zu erreichen werden mehrere Aspekte umgesetzt. Vergleiche hierzu auch den Abschnitt 3.4.2.

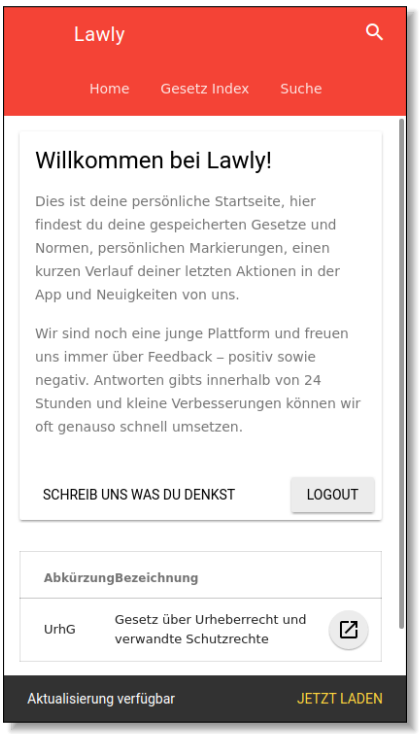


Abbildung 13: Aktualisierung (Smartphone, vertikal)

Einerseits wird beim Bundling der Applikation (siehe Abschnitt 4.3.4) ein Cache Manifest erstellt, welches Informationen über die vom *Application Cache* (siehe Abschnitt 3.4.2) vorzuhaltenden Dateien beinhaltet. Sobald diese einmal geladen wurden, werden alle zukünftigen Anfragen aus dem Cache beantwortet und der Server nur nach einem geänderten Manifest befragt. Ändert sich das Manifest, wird der Cache im Hintergrund und die Webseite entweder durch Neuladen (beispielsweise als Reaktion auf die in Abbildung 13 dargestellte Benachrichtigung) oder beim nächsten Aufruf aktualisiert.

Anderseits gilt es, Anfragen an den API-Server zumindest teilweise optional zu gestalten. Um dies zu erreichen wurde die `ApiClient`-Klasse[21] entwickelt, welche für alle Anfragen an den Server und auch die lokale Suche (mehr dazu in Abschnitt 4.3.3) zuständig ist. Innerhalb der Applikationen werden zur einfacheren Handhabung API-Anfragen als serialisierbare Objekte dargestellt (siehe Listing 8). Diese Objekte werden von der `ApiClient`-Klasse je nach beinhalteter Attribute unterschiedlich gehandhabt.

Ist eine Anfrage zum Beispiel `cachable`, so wird zuerst der lokale Key/Value-Store angefragt – in diesem dienen diese Request-Objekte als Schlüssel. Liefert der lokale Store ein Ergebnis, wird mit diesem unmittelbar, falls spezifiziert, die Redux-Aktion (`action`-Attribut) ausgeführt. Daraufhin wird getestet ob für diesen Cache-Wert ein Ablaufdatum gesetzt ist und ob dieses überschritten wurde – wenn letzteres gilt oder überhaupt kein Ablaufdatum vorhanden ist, wird die Anfrage an den Server gestellt, der Cache bei erhalten einer Antwort aktualisiert und gegebenenfalls erneut die Redux-Aktion mit den aktualisierten Werten aufgerufen. Durch die Verwendung eines mit Ablaufsdatum versehenen Caches wird nicht nur die Interaktion des Nutzers mit der Webseite beschleunigt, sondern auch die Belastung des Servers durch weniger Anfragen reduziert.

```
1 const urhgRequest = {
2   method: 'get', cachable: true,
3   name: 'laws', groupkey: 'urhg',
4   action: FETCH_SINGLE
5 };
```

Listing 8: Serialisierbares API-Request Objekt

Um auch bei nicht zwischenspeicherbaren Anfragen wie beispielsweise dem Vormerken von Gesetzen eine fließende Nutzererfahrung zu bieten, können Anfragen simuliert werden. Dafür wird das gewissermaßen vorhergesagte Ergebnis als *payload*-Attribut an das lokale Request-Objekt angehangen womit daraufhin die Redux-Aktion auch ohne vorhandenen Cache unmittelbar ausgelöst werden kann.

4.3.3 *Lokale Volltextsuche*

Als ein Teil der zentralen Funktionalität der Applikation gilt es auch die Volltextsuche ohne Internetverbindung zur Verfügung zu stellen. Das Problem hierbei: Volltextsuche ist auf nicht dafür spezialisierten Systemen sehr rechenintensiv.

Für die Gesetzesübersicht, beschrieben im vorhergehenden Abschnitt, wird das Filtern durch die Anwendung von regulären Ausdrücken in einer Schleife über die Gesetze realisiert. Da die Zahl der Gesetze nur bei ungefähr 7.000 liegt und nur nach Kürzel und Bezeichnung

21 lawly_web: /src/helpers/ApiClient.js

gefiltert wird, ist dies noch durch die zuvor beschriebene Memorisierung performant umsetzbar. Bei der Volltextsuche gilt es allerdings auch die individuellen Normen und insbesondere deren Textkörper zu durchsuchen. Hierbei wächst die Datenmenge mit der Nutzung der Applikation durch den individuellen Nutzer – speichert er mehr Gesetze für die offline Verwendung ab, gilt es auch mehr Gesetze zu durchsuchen.

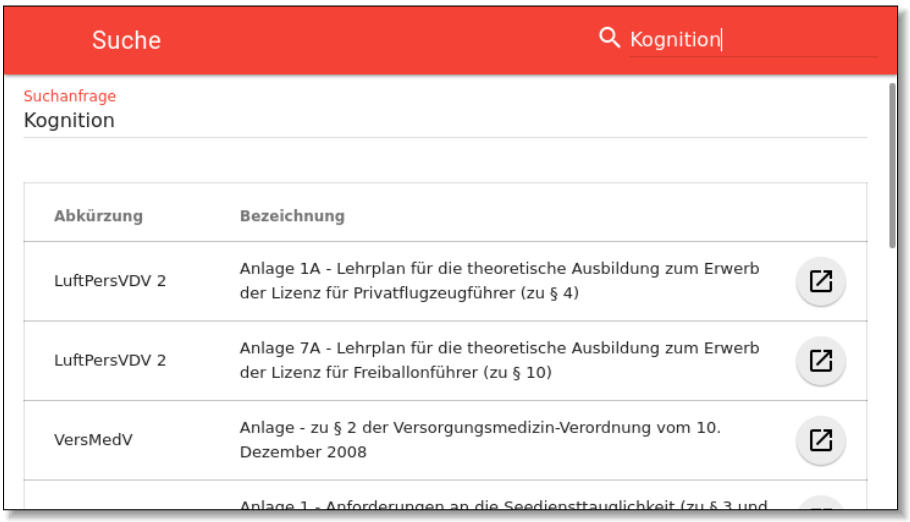


Abbildung 14: Volltextsuche (Smartphone, horizontal)

Anders als bei anderen Teilen der Applikation wird hierbei standardmäßig nicht auf die lokale Implementation gesetzt, solange eine Internetverbindung verfügbar ist. Stattdessen wird, um umfassende Ergebnisse über den gesamten Corpus von ca. 113.000 Normen zu liefern, die Suche durch die serverseitige PostgreSQL Datenbank bevorzugt.

Falls der API-Server nicht verfügbar ist, wird lokal gesucht. Hierbei hat sich als sehr problematisch erwiesen, dass Webseiten nur in einem einzelnen Prozess ausgeführt wird. Blockiert also das JavaScript durch sequentielle Operationen diesen Prozess, ist auch die dargestellte Webseite nicht reaktionsfähig. So wäre es zum Beispiel während einer Suche nicht möglich die Suchanfrage im Eingabefeld zu verfeinern. Zusätzlich ist nicht nur die eigentliche Suche sondern auch das erstellen des notwendigen Suchindexes sehr rechenintensiv und würde, wenn im gleichen Prozess ausgeführt, das Starten der Applikation massiv verlangsamen. Um dies zu umgehen wird ein *Web Worker* eingesetzt.

Web Worker sind eine in neueren Browsern[22] zur Verfügung gestellte Funktionalität zum Auslagern von JavaScript-Operationen in einen gesonderten Prozess. Dabei wird zwischen dem Hauptprozess und dem Web Worker ähnlich wie auf Serverseite beim Eintreffen von Anfragen über Ereignisse kommuniziert. Um dies auf Clientseite zu abstrahieren wurden zwei Klassen entwickelt: `LocalSearch`[23]

22 Quelle: caniuse.com/#search=webworker, Stand 08/2016
23 lawly_web: /src/helpers/LocalSearch.js

und `LocalSearchWorker`[24]. Erstere wird von der zentralen API-Abstraktion (siehe Abschnitt 3.4.2) auf ähnliche Weise wie der API-Server angesprochen und Antworten asynchron verarbeitet. Die `LocalSearch`-Klasse überträgt Anfragen zusammen mit einem eindeutigen Hash an die in ihrem individuellen Prozess ausgeführte `LocalSearchWorker`-Klasse und lauscht auf das durch den Hash identifizierbare Ergebnis auf diese konkrete Anfrage.

4.3.4 *Transpilierung & Bündlung*

Wie in Abschnitt 3.2 beschrieben werden bei der Entwicklung der Applikation moderne noch nicht in allen gängigen Browsern verfügbare JavaScript-Funktionalitäten eingesetzt. Um trotzdem eine möglichst große Bandbreite an Browsern zu unterstützen wird der Code zu einer älteren Version der Sprache übersetzt. Dieser Vorgang, moderne Sprachfeatures durch ältere, breiter etablierte, auszudrücken, nennt sich Transpilierung. Zusätzlich ist es von Interesse den benötigten Quelltext in einer Einzeldatei zu bündeln, um beim initialen Seitenaufruf möglichst wenige HTTP-Anfragen durchführen zu müssen. Bei jeder HTTP-Anfrage kommt es zu Wartezeiten zwischen Anfrage und Beginn des Antworterhalts, so dass das Übertragen von vielen Einzeldateien mehr Zeit in Anspruch nimmt als die Übertragung eines einzelnen größeren Pakets. Um die Ladezeit weiter zu verringern ist es wichtig, nur Code in das endgültige Paket zu übernehmen, welcher auch aktiv Verwendung findet. Dies ist besonders beim einsetzen externer Bibliotheken nicht trivial, da diese oft unübersichtlich verschachtelt sind. Durch den Einsatz von JavaScript-Modulen ist es möglich von einem Einstiegspunkt aus den Modulbaum zu traversieren und durch sogenanntes *tree-shaking* nur wirklich genutzten Code aus jedem Modul in das endgültige Paket zu übernehmen (Harris, 2015).

Um diese Schritte im Entwicklungsprozess zu automatisieren wird *Webpack* eingesetzt: Webpack traversiert von einem Einstiegspunkt aus den Modulbaum, lädt die einzelnen Module mit Hilfe von verschiedenen Erweiterungen und kümmert sich um das Abtrennen von ungenutztem Code. Eine der Erweiterungen ist beispielsweise *Babel*, welches für die Transpilierung zuständig ist. Andere solche Erweiterungen sind für das Laden von in bestimmten Komponenten eingebundene Cascading Style Sheets (CSS)-Styles, Schriftarten oder Bilder zuständig.

4.4 DEPLOYMENT

Der Begriff Deployment beschreibt die Veröffentlichung einer Applikation auf das Produktivsystem und damit an den Endnutzer. An die

24 lawly_web: /src/helpers/LocalSearchWorker.js

Ausführung der Applikation in dieser Phase sind sehr viel höhere Anforderungen gestellt als während der Entwicklung. So gilt es beispielsweise auf mögliche Lastspitzen, welche während der Entwicklung wenn überhaupt nur in kontrollierten Tests auftreten, schnell und möglichst sogar automatisiert reagieren zu können. Außerdem ist es Notwendig eine möglichst hohe Verfügbarkeit zu garantieren – also, dass die Applikation rund um die Uhr an jedem Tag des Jahres erreichbar ist und auch gegen eventuelle Stromausfälle oder andere unerwartete Ereignisse abgesichert ist.

Für Lawly wird dafür aktuell auf zwei unabhängige Anbieter gesetzt. Für Server und Datenbank werden jeweils AWS Instanzen verwendet. Die API läuft dabei auf einer durch *ElasticBeanstalk* verwalteten EC2 Instanz und interagiert mit einer auf Amazons Relational Database Service (RDS) betriebenen PostgreSQL-Datenbank. Die Client-Applikation tritt im Gegensatz dazu sehr viel kleiner: Sie wird von einem Server bei Uberspace[25], einem relativ kleinen deutschen Hoster aus Frankfurt, ausgeliefert.

Aus der breiten Front der Cloud-Anbieter fiel hierbei die Wahl auf Amazon, da nur dieser PostgreSQL in seiner neusten Version als verwaltetes Datenbanksystem anbietet. Die neuste Version ist aufgrund des geplanten Einsatzes von nur in dieser Version verfügbaren JSON-Funktionalitäten erforderlich. Zwar ist es bei allen Anbietern möglich, selbst verwaltete Instanzen zu betreiben, allerdings meist mit höheren Kosten und insbesondere einem höheren Aufwand als bei einem verwalteten System verbunden. Zusätzlich kommt ElasticBeanstalk als eine angenehme weitere Abstraktionsebene ohne zusätzliche Mehrkosten gegenüber einer selbstverwalteten EC2-Instanz mit komfortablen Commandline-Tools und einer engen Verknüpfung zur eingesetzten `git`-Versionsverwaltung daher. So kann ElasticBeanstalk sich zum Beispiel mit wenigen Klicks um die automatische horizontale Skalierung der API und der damit zusammenhängenden Konfiguration des Load-Balancers kümmern. Außerdem übernimmt der Service selbstständig eigentlich komplexe Operationen wie das fließende Aktualisieren der Applikation ohne zwischenzeitliche Downtime.

Für den bei Uberspace betriebenen Server wird hingegen nur eine sehr niedrige Last erwartet, da er nur mit der Auslieferung der statischen Dateien betraut ist und diese, wie zuvor beschrieben, durch intensives Caching auf Clientseite nur selten erneut angefragt werden müssen. Anfragen werden hierbei mithilfe einfacher Apache Umschreibungen an die richtigen Dateien dirigiert – es wird an dieser Stelle keine Node.js-Instanz betrieben. Dies ist allerdings nur eine vorläufige Lösung. Langfristig soll auch an diese Stelle ein Node.js-Server treten der sich um Server Side Rendering (SSR) kümmert. Auf den vorläufigen Wegfall von SSR wurde bereits in Abschnitt 3.4.1 eingegangen. Voraussichtlich wird dann auch dieser Teil der Applikation auf eine eigene ElasticBeanstalk-Instanz umgestellt.

25 uberspace.de

5

DISKUSSION

Zum Zeitpunkt des Abschlusses der vorliegend Arbeit sind beide Endpunkte, Client und Server, noch zugriffsbeschränkt. Der API-Server erlaubt so aktuell nur Anfragen von der auf der offiziellen Domain betriebenen Webapplikation und die Webapplikation benötigt eine zusätzliche Authentifizierung. Dies soll möglichst bald, soweit das weitere Vorgehen geklärt ist und insbesondere eine Landingpage als Einführung für neue Nutzer erstellt wurde, geändert werden. Gleichermaßen ist es für diesen Zeitpunkt auch geplant den Code vollständig unter einer Open Source Lizenz zu veröffentlichen – die Details dies bezüglich müssen noch entschieden werden.

5.1 RÜCKBLICK

Rückblickend wurden die ursprünglich für diese Arbeit gesetzten Ziele erreicht. Die geplanten Ansichten und Funktionalitäten wurden umgesetzt, die Webapplikation zeigt einer nativen Applikation ähnliche Performance und ist für den mobilen offline Einsatz gerüstet. Außerdem ist sie für den Produktiveinsatz vorbereitet und kann, außer einiger letzter Nachbesserungen, aus Knopfdruck freigeschaltet werden.

Der Weg dahin war allerdings von einigen Komplikationen geprägt, insbesondere, da einige sehr vielversprechende Technologien wie beispielsweise Service Worker bisher nicht von allen Browserherstellern umgesetzt wurden. Zusätzlich hat sich auch die Wahl des zuvor noch nicht verwendeten React-Ecosystem als sehr arbeitsintensiv erwiesen: anders als bei beispielsweise Angular.js (mit welchem Vorerfahrung besteht), ist React nicht „*batteries-included*“. Das bedeutet, dass es nicht von Hause aus den Großteil der für die Entwicklung und den Produktiveinsatz benötigten Werkzeuge mitbringt. Alleine die Konfiguration des für das produktive Arbeiten notwendige Webpacks hätte mehrere Seiten dieser Arbeit füllen können.

Ähnliches gilt für die neuartige clientseitige Applikationsarchitektur. Die objektorientierte Programmierung und den Einsatz des MVC-Konzeptes gewöhnt, ist der Einstieg in die eingesetzten Konzepte, wie beispielsweise der uni-direktionale Datenfluss, aufwendig.

Allerdings hat sich die Entscheidung für den gewählten Entwicklungsstack als richtig erwiesen: die in Abschnitt 3.4 aufgeführten Punkte wie zum Beispiel die sehr viel bessere Verständlichkeit der Architektur und die Vorhersehbarkeit von Zustandsänderungen haben den Entwicklungsprozess nach einer zu Beginn steilen Lernkurve sehr angenehm gestaltet.

In Bezug auf das Gesamtprojekt ist zu bemerken, dass in den letzten Wochen vor der Fertigstellung dieser Arbeit ein österreichischer Anbieter aufgetreten ist, welcher sehr vieles richtig macht. *openlaws.com* setzt ähnlich wie Lawly auf Open Data im juristischen Bereich aufzubereiten und zu verknüpfen. Mit einem scheinbar über zweijährigen Entwicklungsvorsprung konnte die Plattform außerdem bereits Fördergelder der Europäischen Union verbuchen. Obwohl der Start dieses direkten Konkurrenten anfänglich frustrierend war, bestätigt er auch die Notwendigkeit einer solchen Dienstleistung. Zusätzlich sind bei *openlaws.com* Open Data und Open Source zwar für Marketingzwecke beliebte Schlagwörter, allerdings keine von der Plattform selbst umgesetzten Prinzipien. Weder ist ihr Quelltext frei verfügbar, noch stellt sie eine Schnittstelle für die von ihr vermarkteten aufbereiteten Daten bereit. Beides Grundsätze, welche für Lawly zum Konzept gehören.

5.2 AUSBLICK

Obwohl der Rückblick etwas dämpft, ist der Ausblick positiv. So ermöglichen die technologischen Entscheidungen durch das Heranreifen von React Native langfristige Entwicklungschancen – um den Markt zu erreichen ist so beispielsweise die Entwicklung einer nativen iPad-Applikation mittelfristig denkbar.

In Bezug auf die realisierte Webapplikation gilt es vorerst die Performance weiter zu optimieren. Dazu gehört insbesondere auch die *gefühlte Performance*, welche die geschickte Verwendung von Animationen erfordert – eine Thematik welche im bisherigen Entwicklungsverlauf noch nicht angegangen wurde. Außerdem gilt es, Server Side Rendering (SSR) auch in Produktion einzusetzen und die Seite damit für Suchmaschinen zu optimieren. Dazu gehört auch der Gedanke mithilfe des gleichen SSR-Codes durch die Generierung von Accelerated Mobile Pages (AMP) einen Rankingvorteil bei Google zu erreichen. Google bevorzugt offiziell Seiten, welche diesen von ihnen offen entwickelten Standard einsetzen.[1]

Außerdem erfordert auch die Ansicht für die eigentlichen Gesetzestexte noch weitere Aufmerksamkeit, da dies die Ansicht ist, auf der Nutzer voraussichtlich die meiste Zeit verbringen werden. Von ver-

1 googleblog.blogspot.de/2016/02/amping-up-in-mobile-search.html, abgerufen 08/2016

breiteten Anbietern für Lesedienste wie beispielsweise Pocket[2] sind so zum Beispiel die Möglichkeiten zur Anpassung von Schriftart und -größe oder auch der Einsatz von mehrspaltigen Layouts bekannt.

Letztendlich ist auch die Finanzierung des Projektes nicht zu vernachlässigen. Kurzfristig gilt es, zumindest die Kosten für den Betrieb zu decken und langfristig auch die Weiterentwicklung zu finanzieren. Vorläufig soll, auf Grundlage der in Abschnitt 2.5 vorgestellten Ideale, eine nicht kommerzielle Finanzierung im Vordergrund stehen. Dies wäre zum Beispiel durch eine Plattform wie Patreon[3] denkbar, bei welcher regelmäßige kleine Spenden zur langfristigen Finanzierung von Projekten im Vordergrund stehen. Außerdem steht der Prototypefund[4], eine staatlich geförderte Initiative zur Förderung deutscher Open Source Projekte von öffentlichem Interesse, aus. Die Bewerbung um eine Förderung durch letzteren ist wohl der erste anstehende Schritt.

Abbildung 15: Aktuelles *lawly.org* Logo

2 getpocket.com
3 patreon.com
4 prototypefund.de

ABKÜRZUNGSVERZEICHNIS

AJAX asynchronous JavaScript

AMP Accelerated Mobile Pages

API Application Programming Interface

AWS Amazon Web Services

beA besonderes elektronisches Anwaltspostfach

BGBL Bundesgesetzblatt

BMJV Bundesministerium der Justiz und für Verbraucherschutz

BRAK Bundesrechtsanwaltskammer

CA Certificate Authority

CSS Cascading Style Sheets

DOM Document Object Model

EC2 Elastic Cloud Computing

EGVP Elektronisches Gerichts- und Verwaltungspostfach

ES ECMAscript

HTML HyperText Markup Language

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

JWT JSON Web Token

MVC Model-View-Controller

MVP Minimum Viable Product

URL Uniform Resource Locator

PWA Progressive Web Application

RDS Relational Database Service

REST Representational State Transfer

SPA Single Page Application

SSR Server Side Rendering

REFERENZEN

Baden-Württemberg, V. (2013). AZ 10 S 281/12.

Brehm, S. (2013). The future of web apps is — ready? — isomorphic JavaScript. *Venture Beat*, Stand August 2016. Abgerufen von http://venturebeat.com/2013/11/08/the-future-of-web-apps-is-ready-isomorphic-javascript/

Bundesagentur für Arbeit. (2016). Gute Bildung - gute Chancen. Der Arbeitsmarkt für Akademikerinnen und Akademiker in Deutschland, 75–78.

Bundesamt für Justiz. (2013). Zahl der Richter, Richterinnen, Staatsanwälte, Staatsanwältinnen und Vertreter, Vertreterinnen des öffentlichen Interesses in der Rechtspflege der Bundesrepublik Deutschland.

Bundesanzeiger Verlag GmbH. (o. J.). Bundesgesetzblat Online AGB, Stand August 2016. Abgerufen von http://www1.bgbl.de/fileadmin/Betrifft-Recht/Dokumente/BGBl/bgbl_agb_online.pdf

Bundesministerium der Justiz und für Verbraucherschutz. (o. J.). Rechtsprechung im Internet, Stand August 2016. Abgerufen von https://www.rechtsprechung-im-internet.de

Bundesrechtsanwaltskammer. (2008). Vorschläge zur Verbesserung der Akzeptanz des elektronischen Rechtsverkehrs. *Presseerklärung*, *1*. doi:10.1017/CBO9781107415324.004

Bundesrechtsanwaltskammer. (2015a). beA kommt später. *Presseerklärung*, *20*.

Bundesrechtsanwaltskammer. (2015b). Große Mitgliederstatistik zum 01.01.2015.

Bundesrechtsanwaltskammer. (2016a). AGH-Verfahren zum besonderen elektronischen Anwaltspostfach. *Presseerklärung*, (7), 2016.

Bundesrechtsanwaltskammer. (2016b). Elektronisches Anwaltspostfach geht an den Start. *Presseerklärung*, (3), 2016.

Byron, L. (2015). Immutable Data and React, Stand August 2016. Abgerufen von https://youtu.be/I7IdS-PbEgI

Chen, J. (2014). Rethinking Web App Development at Facebook, Stand August 2016. Abgerufen von https://youtu.be/nYkdrAPrdcw?t=10m21s

Dahl, R. (2009). Original Node.js Presentation, Stand August 2016. Abgerufen von https://youtu.be/ztspvPYybIY

Der Spiegel. (1981). Herrschende Meinung, *8*.

Der Spiegel. (2006). M. DuMont Schauberg Verlag schluckt Bundesanzeiger, Stand August 2016. Abgerufen von http://spiegel.de/kultur/gesellschaft/a-448095.html

Deutscher Bundestag. (1949). Grundgesetz für die Bundesrepublik Deutschland. *Bundesgesetzblatt*, *1*(1).

Deutscher Bundestag. (1965). Gesetz über Urheberrecht und verwandte Schutzrechte; Urheberrechtsgesetz (UrhG). *Bundesgesetzblatt*, *1*(51), 1273–1293.

Deutscher Bundestag. (2001). Gesetz zur Reform des Verfahrens bei Zustellungen im gerichtlichen Verfahren (Zustellungsreformgesetz – ZustRG). *Bundesgesetzblatt*, *1*(29), 1206–1214.

Deutscher Bundestag. (2005). Das Gesetz über die Verwendung elektronischer Kommunikationsformen in der Justiz. *Bundesgesetzblatt*, *2005*(18), 837ff.

Deutscher Bundestag. (2013). Gesetzes zur Förderung des elektronischen Rechtsverkehrs mit den Gerichten. *Bundesgesetzblatt*, *1*(62), 3786–3798.

Ecma International. (2015). ECMAScript 2015 Language Specification.

Ecma International. (2016). ECMAScript 2016 Language Specification.

Fielding, R. T., & Reschke, J. F. (2014). Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. *Internet Engineering Task Force (IETF)*, 1–101. doi:10.1017/CBO9781107415324.004

Frankfurter Allgemeine Zeitung. (2009, Mai). Streit unter den Paragraphen-Sammlern.

Gasston, P. (2014). *Moderne Webentwicklung*. Heidelberg: dpunkt.verlag GmbH.

Harris, R. (2015). Tree-shaking versus dead code elimination, Stand August 2016. Abgerufen von https://medium.com//@Rich_Harris/tree-shaking-versus-dead-code-elimination-d3765df85c80

Henne, T. (2006). Die Prägung des Juristen durch die Kommentarliteratur.

Jones, M., Bradley, J., & Sakimura, N. (2015). JSON Web Token, RFC 7519. *Internet Engineering Task Force*, *1*.

Justizministerium des Landes Nordrhein-Westfalen. (o. J.). Abruf zur Veröffentlichung sowie gewerbliche Nutzung, Stand August 2016. Abgerufen von https://www.justiz.nrw.de/BS/nrwe2/gewerbl_nutzer/index.php

Justizverwaltungen des Bundes und der Länder, & Berufskammern und -verbände der Rechtsanwälte und Notare. (2007). Zehn-Punkte-Plan der zur "Förderung des elektronischen Rechtsverkehrs".

Russell, A. (2015). Progressive Web Apps: Escaping Tabs Without

Losing Our Soul, Stand August 2016. Abgerufen von https://hinfrequently.org/2015/06/progressive-apps-escaping-tabs-without-losing-our-soul

Sauble, D. (2015). *Offline First Web Development*. Birmingham, UK: Packt Publishing.

Stubailo, S. (2015). Optimistic User Interfaces with Meteor, Stand August 2016. Abgerufen von http://info.meteor.com/blog/optimistic-ui-with-meteor-latency-compensation

Talkner, R. (2015). Das Leben mit Loseblattwerken – eine erfüllte Liebesbeziehung, Stand August 2016. Abgerufen von http://blog.beck-shop.de/blog/2015/11/12/leben-mit-loseblattwerken/

Tilkov, S., Eigenbrodt, M., Schreier, S., & Wolf, O. (2015). *REST und HTTP* (3. Aufl.). Heidelberg: dpunkt.verlag GmbH.

Vaquero, L. M., Rodero-Merino, L., & Buyya, R. (2011). Dynamically scaling applications in the cloud. *ACM SIGCOMM Computer Communication Review*, *41*(1), 45–52. doi:10.1145/1925861.1925869

Verlag C. H. Beck. (2010). Recht. Steuern. Wirtschaft.

Wissenschaftsrat. (2012). Perspektiven der Rechtswissenschaft in Deutschland, *21*, 1–111. doi:10.1515/9783110826371.888

SELBSTSTÄNDIGKEITSERKLÄRUNG

Hiermit erkläre ich, dass ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die aus fremden Quellen direkt oder indirekt übernommenen Gedanken als solche kenntlich gemacht habe. Die Arbeit habe ich bisher keinem anderen Prüfungsamt in gleicher oder vergleichbarer Form vorgelegt. Sie wurde bisher nicht veröffentlicht.

Osnabrück, 12. September 2016

Ort, Datum	Alexander Höreth