

Software Architektur Elemente des JavaScript Clients in einer Single Page Webapplikation

am Beispiel einer Fotoverwaltungssoftware

Bachelor-Thesis
zur Erlangung des akademischen Grades B.Sc.

Anastasia Stieb

Matrikel-Nr. 2023959



Hochschule für Angewandte Wissenschaften Hamburg
Fakultät Design, Medien und Information
Department Medientechnik

Erstprüfer: Prof. Dr. Andreas Plaß

Zweitprüfer: Prof. Dr. Torsten Edeler

Hamburg, 16. 7. 2021

Thema der Arbeit

Software Architektur Elemente des JavaScript Clients in einer Single Page Webapplikation am Beispiel einer Fotoverwaltungssoftware

Stichworte

SPA, Client-Side Architecture, Rich Client, MVC, Angular, Responsive Webdesign, Angular, REST, JWT

Zusammenfassung

Diese Bachelorarbeit befasst sich mit der Realisierung eines potentiell an Benutzerinteraktionen reichen, nativ ähnlichen Clients einer Webanwendung am Beispiel der Implementierung einer Fotoverwaltungssoftware.

Title of Paper

Software architecture elements of a JavaScript client in a single page webapplication exemplified by a photo editing software

Keywords

SPA, Clientseitige Architektur, MVC, Rich Client, Responsive Webdesign, Angular, REST, JWT

Abstract

This bachelor thesis focuses on the realisation of a potentially user interaction rich, similar to native clients in a webapplication by implementing a photo management software as example.

Inhaltsverzeichnis

1 Einleitung	6
1.1 Motivation	6
1.2 Zielsetzung	8
2 Analyse	9
2.1 Einleitung	9
2.2 User Experience	9
2.2.1 Schnelles Feedback	9
2.2.2 Flaches Design	9
2.2.3 Responsive Design	10
2.2.4 Fotozentrierung	10
2.3 User Stories	10
2.3.1 Authentifizierung	10
2.3.2 Navigation Menü	10
2.3.3 Foto Galerie	10
2.3.4 Paginierung/Nachladen der Fotos	11
2.3.5 Foto Freitext Suche	11
2.3.6 Fotos nach Erstellungsmonat gruppieren	11
2.3.7 Foto Details	11
2.3.8 Foto Großansicht	11
2.3.9 Foto Slider	11
3 Grundlagen	12
3.1 Client-Server-Modell	12
3.2 HTTP	13
3.3 Serverseitige Webanwendungen	15
3.3.1 Grundprinzip	15
3.3.2 Architektur	17
3.4 Clientseitige Webanwendungen	18
3.4.1 Grundprinzip	18

Inhaltsverzeichnis

3.4.2 Datenübergabe	19
3.4.3 Architektur	20
3.5 Responsive Webdesign	21
4 Design	26
4.1 Farb Palette	26
4.2 Authentifizierung	26
4.3 Navigation Menü	27
4.4 Foto Galerie	28
4.5 Paginierung	31
4.6 Foto Details	31
5 Implementierung	36
5.1 Einleitung	36
5.2 Layout	36
5.3 Grund Setup	38
5.4 Routing	40
5.5 Projekt Struktur	41
5.5.1 Verzeichnis Struktur	42
5.5.2 Modul Struktur	43
5.5.3 Komponenten Struktur	44
5.6 Session Handling	45
5.6.1 Grundverfahren	45
5.6.2 Authentifizierung	46
5.6.3 Autorisierung	49
5.7 Menüführung	51
5.8 Foto Galerie	53
5.8.1 Laden	53
5.8.2 Gruppieren	55
5.8.3 Paginieren	57
5.8.4 Rendering	59
5.9 Foto Suche	59
5.10 Foto Details	60
5.11 Foto Slider	61
6 Fazit und Ausblick	63
6.1 Fazit	63
6.2 Ausblick	63

Inhaltsverzeichnis

Abbildungsverzeichnis	67
Quellcode Verzeichnis	68
Literaturverzeichnis	69

1 Einleitung

1.1 Motivation

„There is no cloud it's just someone else's Computer“ - eigentlich ein ganz triviales Statement, doch es wurde zu einem Internetphänomen, auch „Meme“ genannt, weil die Internetindustrie es geschafft hat, für einen Benutzer transparent werden zu lassen, dass hinter so manchem Dienst sich in der Realität ein ganzes Rechenzentrum befindet.

Die große Rechenleistung, die von jedem Ort, jeder Zeit verfügbar ist, machte auch eine Breite verschiedener portabler Anzeigegeräte ubiquitär. Die Werbemarketingsspezialisten sprechen von einem „Second Screen“, aber in Wirklichkeit ist jedes andere internetfähige Gerät gemeint, welches parallel zum laufenden Fernsehprogramm genutzt wird. Und bei der Auswahl aus Netbook, Tablet, Phablet, Smartphone, Smartwatch ist bei manchem Anwender die Zahl dieser Geräte längst über zwei. Viele kleine Applikationen sollen diese portablen Geräte zu intelligenten persönlichen Assistenten machen. TechCrunch spricht sogar von einem neuen Software Goldrausch, der in den letzten 7 Jahren stattgefunden hat.

„Like all gold rushes, they must come to an end. It is clear that everyone close to technology is suffering from what the market is calling “app fatigue.” If it wasn't already hard enough to differentiate your app from the millions of others in the app store, it's now becoming even harder. From a consumer perspective, there are just too many apps. New apps, by and large, are not providing nearly enough value for consumers to come back, and most simply replicate existing experiences with a story of a better design. Apps are not an order of magnitude better than their predecessor; thus, adoption drops off as quickly as it started.“ [Schippers \(2016\)](#)

Es stellt sich daher eine Situation dar, in der die Anwender zwar die schnelle Reaktionsfähigkeit nativer Applikationen zu schätzen wissen, jedoch nicht sofort bereit sind, weitere Software auf ihre Geräte zu installieren. Und das macht eine bestimmte Applikation wieder populärer den jeden Webbrowser.

1 Einleitung

Auch im Zeitalter der Smartphone hat sich an den Grundprinzipien und Protokollen, die das World Wide Web seit 1991 zu Nutze macht, nicht viel geändert. Eine sog. Single Page Webapplikation wird vom Browser auf die gleiche Art und Weise geladen, wie die allererste HTML Webseite. Lediglich eine grundlegende Neuerung kam im Jahr 1995 hinzu. Netscape ermöglichte es, den Entwicklern mehr Interaktivität durch Auslieferung vom Script Code in die statischen Webseiten einzubauen, und leitete den Aufstieg von JavaScript - gegenwärtig einer der populärsten Programmiersprachen - ein. Allerdings ist diese Berühmtheit ganz und gar nicht dadurch entstanden, dass die Sprache eine besonders elegante Erfindung war. Es war einfach die einzige von Bürgern von Haus aus unterstützte Option. Ganz im Gegenteil, JavaScript basiert auf einigen schlechten Entscheidungen. Ein populäres Fachbuch nennt sich auch daher nicht umsonst - „JavaScript - the good parts“.

„JavaScript is a language with more than its share of bad parts. It went from nonexistence to global adoption in an alarmingly short period of time. It never had an interval in the lab when it could be tried out and polished. It went straight into Netscape Navigator 2 just as it was, and it was very rough. When Java™ applets failed, JavaScript became the “Language of the Web” by default. JavaScript’s popularity is almost completely independent of its qualities as a programming language.

Fortunately, JavaScript has some extraordinarily good parts. In JavaScript, there is a beautiful, elegant, highly expressive language that is buried under a steaming pile of good intentions and blunders. The best nature of JavaScript is so effectively hidden that for many years the prevailing opinion of JavaScript was that it was an unsightly, incompetent toy.“([Crockford 2008](#): S. 2)

Bei vielen Webanwendungen beschränkt sich daher heutzutage die Hauptinteraktion immer noch darauf, beim Betätigen eines Knopfes neuen Markup vom Server zu laden. Alles, was an Benutzerinteraktion darüber hinaus geht, ist ein Nebengedanke. Und so kommt es vor, dass der serverseitige Teil, der zuständig für das Rendern der Hauptinhalte ist, mit allen bekannten Prinzipien des guten Software Designs realisiert ist, und der clientseitige Teil, für die weitergehende Interaktivität zuständiger JavaScript-Teil, aber eine bloße Ansammlung loser Skripte darstellt. Bei kleinem Anteil solchen clientseitigen Programmcodes wird diese Praxis aus Kostengründen toleriert, ist jedoch bei jeder mittleren Komplexität nicht mehr hinnehmbar. Diese Arbeit betrachtet die Implementierung eines solchen komplexen Webanwendungsclients in einer Single Page Webapplikation.

1.2 Zielsetzung

Diese Arbeit befasst sich mit der Realisierung eines potentiell an Benutzerinteraktionen reichen, nativ ähnlichen Clients einer Webanwendung am Beispiel der Implementierung einer Fotoverwaltungssoftware.

Dabei wird auf folgende Schwerpunkte eingegangen:

- Adaptierung an verschiedene Endgeräte
- Auslagerung der Darstellungslogik an den Client
- Architektonische Trennung von Verantwortlichkeiten
- Kommunikation mit dem Server und Batching der Daten
- Authentifizierung der Benutzersitzung
- Menüführung und Navigation zwischen Unterbereichen
- Verwaltung des lokalen Anwendungszustands über mehrere Komponenten

2 Analyse

2.1 Einleitung

In dem Kapitel [Motivation](#) wurde geschildert, dass eine Web-Fotoverwaltungssoftware als Beispiel für die Zielsetzungen dieser Arbeit dienen wird. Nun soll analysiert werden, welche Realanforderungen notwendig sind, um den zuvor definierten [Zielsetzungen](#) gerecht zu werden.

2.2 User Experience

Folgend werden Anforderungen beschrieben, die sich mit dem Nutzungserlebnis und der visuellen Gestaltung der Applikation befassen.

2.2.1 Schnelles Feedback

Die Hauptanforderung an die Webanwendung besteht darin, dem Benutzer das an eine native Applikation angelehnte Nutzungserlebnis zu gewährleisten. Die bei einer klassischen Webanwendung entstehenden Ladezeiten, welche nach jeder Benutzerinteraktion durch das erneute Laden und Darstellen des gesamten Inhaltes auftreten, sollen vermieden werden.

2.2.2 Flaches Design

Das Benutzerinterface der Anwendung soll unter Anwendung der Paradigmen vom flachen Design minimalistisch, jedoch mit klar erkennbaren Aktionsaufrufen, gestaltet werden.

2.2.3 Responsive Design

Die Webanwendung soll sich auf potentiell unterschiedliche Displaygrößen anpassen. Das Benutzerinterface soll dabei nicht komplett für jede mögliche Abstufung der Displaygröße neu gestaltet werden. Für das Layout sollen Regeln verwendet werden, die es dem gleichen Interface erlauben, seine Elemente bei schrumpfender bzw. wachsender Größe neu zu positionieren.

2.2.4 Fotozentrierung

Ein besonderes Unterproblem der Adaptierung an verschiedene Gerätedisplays ist die Fotobetrachtung. Hier sind sowohl die Auflösung des Fotos als auch des Displays für die Software nicht zu Implementierungszeit, sondern erst zur Laufzeit bekannt. Um das Foto in der Gesamtheit auf einem beliebigen Display zu betrachten, soll es zur Laufzeit in der Detailansicht zentriert werden. Der Zentrierung soll sich an eine Änderung der Auflösung anpassen.

2.3 User Stories

In diesem Unterkapitel werden die einzelnen Features der Beispiel-Applikation definiert.

2.3.1 Authentifizierung

Die Fotosoftware soll den Benutzern nur anhand einer Benutzerkennung und eines Passwortes den Zugang gewähren.

2.3.2 Navigation Menü

Der Benutzer soll im Stande sein, zwischen den Hauptfunktionen der Anwendung aus jedem beliebigen Unterbereich zu navigieren.

2.3.3 Foto Galerie

Dem Benutzer soll eine Auflistung seiner gespeicherten Fotos dargestellt werden.

2.3.4 Paginierung/Nachladen der Fotos

Falls sich sehr viele Fotos in der [Fotogalerie](#) befinden, sollen diese nicht im selben Augenblick geladen werden, damit die Anwendung nicht überlastet wird. Stattdessen soll zuerst eine bestimmte Anzahl von Fotos dargestellt werden. Und anschließend soll es dem Benutzer ermöglicht werden, weitere Bilder stapelweise nachzuladen.

2.3.5 Foto Freitext Suche

Der Benutzer soll in Beschreibungen und Namen nach seinen Fotos durch Eingabe von Freitext suchen können. Das Resultat der Suche soll ebenfalls wie die Fotogalerie paginiert werden.

2.3.6 Fotos nach Erstellungsmonat gruppieren

Dem Benutzer soll ermöglicht werden, die Fotos nach Erstellungsmonat gruppiert zu betrachten. Die Darstellung der Monatsgruppen erfolgt in absteigender Reihenfolge.

2.3.7 Foto Details

Der Benutzer soll in der Lage sein, Fotos mit Metadaten wie Name und Beschreibung zu annotieren. Bei der Auswahl eines einzelnen Fotos in der Galerie sollen diese annotierten Fotoinformationen veranschaulicht werden.

2.3.8 Foto Großansicht

Dem Benutzer soll ermöglicht werden, ein bestimmtes Foto in der vollständiger Größe zu betrachten.

2.3.9 Foto Slider

Wenn der Benutzer die Detailansicht eines Fotos aus der Galerie auswählt, soll es ihm ferner möglich sein, aus der Detailansicht zum nächsten bzw. vorherigen Foto zu navigieren.

3 Grundlagen

3.1 Client-Server-Modell

Webanwendungen sind eine erweiterte Form von normalen Webseiten und funktionieren nach den selben Prinzipien des World Wide Webs. Diesen liegt wiederum das Client-Server-Modell zu Grunde.

Der Client ist ein Programm des Benutzers und ist dafür zuständig, den Inhalt der Applikation oder Webseite auf dem Bildschirm in benutzerfreundlicher Art und Weise zu verarbeiten. Ein solcher typischer Client ist der Web Browser.

Der Inhalt selbst befindet sich auf einem entfernten Rechner, genannt der Server. Server verarbeiten eingehende Anfragen der Clients nach Inhalten und liefern eine Kopie dieser Inhalte aus. Der heruntergeladene Inhalt kann schließlich vom Client angezeigt werden.

Die Fachbezeichnung für den Remote-Inhalt ist Ressource. Ressourcen können aus Bildern, Videos, Webseiten und anderen Dateien bestehen. Aber, wie am Anfang angedeutet, sind Ressourcen nicht nur auf Dateien und Webseiten beschränkt. Sie können auch in Form von Software vorkommen, welche beispielsweise es erlauben mit Aktien zu handeln oder Videospiele zu spielen. Ressourcen werden dabei durch einen eindeutigen Bezeichner - die URL - identifiziert (Siehe Abb. 3.1).

Historisch ergeben, nutzen Client und Server das Kommunikationsprotokoll HTTP für die Kommunikation untereinander. Diese Übertragung ist Zustandslos. Diese Eigenschaft wurde absichtlich konzipiert, um die Protokoll-Implementierung einfach zu halten und um Serverressourcen zu sparen. Der Server muss dabei keine Benutzerinformation zwischen den Anfragen merken. Im Fehlerfall muss ebenfalls nichts aufgeräumt werden. Die beiden Gründe machen HTTP zu einem sehr belastbaren Protokoll, aber auch gleichzeitig zu einem schwierigen Protokoll, um Zustandsbehaftete Webanwendungen zu implementieren.

(Parikh u. a. 2015: Background) beschreibt das Problem wie folgt:

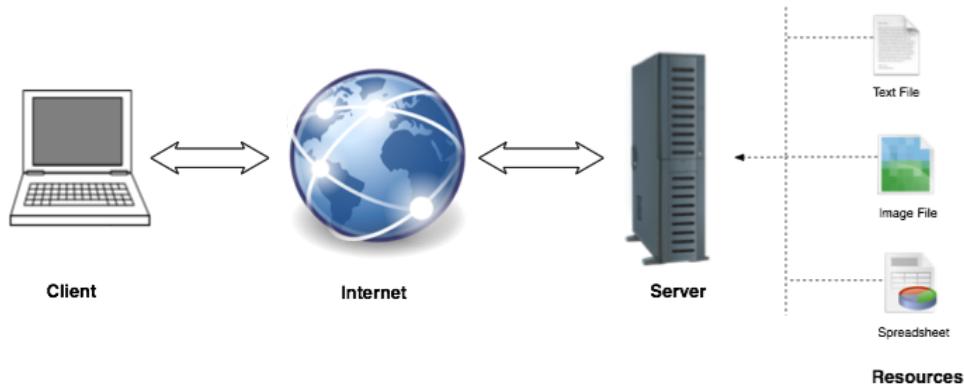


Abbildung 3.1: Client-Server-ModellSingSing. Quelle: [Parikh u. a. \(2015\)](#), Kap. Background

„When you go to Facebook, for example, and log in, you expect to see the internal Facebook page. That was one complete request/response cycle. You then click on the picture – another request/response cycle – but you do not expect to be logged out after that action. If HTTP is stateless, how did the application maintain state and remember that you already input your username and password? In fact, if HTTP is stateless, how does Facebook even know this request came from you, and how does it differentiate data from you vs. any other user? There are tricks web developers and frameworks employ to make it seem like the application is stateful...“

Es gibt daher eine Reihe von verschiedenen Techniken, welche auf Anwendungsebene realisiert werden müssen, um Zustandhaftigkeit in einem zustandslosen Protokoll zu gewährleisten.

Vgl. ([Parikh u. a. 2015](#): Background), [Federico u. a. \(2021\)](#)

3.2 HTTP

In [3.1](#) wurde erwähnt, dass im World Wide Web das Kommunikationsprotokoll HTTP verwendet wird. Ein Protokoll zeichnet sich zunächst durch drei grundlegende Eigenschaften aus:

- Syntax - Datenformat und Kodierung
- Semantik - Steuerungsinformation und Fehlerbehandlung
- Zeitablauf - Geschwindigkeitsanpassung und Reihenfolge

[Dubost \(2012\)](#) zeigt, dass Kommunikationsprotokolle nicht nur ein künstliches Konstrukt sind, sondern auch aus der realen Welt stammen:

3 Grundlagen

„When two people meet, they engage using a communication protocol: for example, in Japan, a person will make a specific gesture with the body. One such gesture is a bow, which is the syntax used for the interaction. In Japanese customs, the gesture of the bow (among others) is associated with the semantics of greeting someone. Finally, when one person bows to another person, a sequence of events has been established between the two in a specific timing.“

Weiterhin beschreibt Dubost (2012), dass in einem Online-Kommunikationsprotokoll die gleichen Elemente vorkommen: die Syntax - die Abfolge von Zeichen, etwa Bezeichnern, die für das Schreiben des Protokolls verwendet werden; die Semantik - die Bedeutung, die mit diesen Bezeichnern assoziiert wird; und schließlich der Zeitablauf - eine vorgegebene Reihenfolge, in der Client und Server diese Bezeichner austauschen.

HTTP ist dabei ein sog. Application Level Protocol, d.h, dass es in der Abstraktionsebene höher angesiedelt ist. Es setzt wiederum auf eine Reihe weiterer Protokolle auf, welche sich etwa um die Übertragung der eigentlichen Datenpakete oder die physikalische Übertragung der elektrischen Signale kümmern. HTTP selbst beschreibt hingegen die Bedeutung und das Format der gesamten übertragenen Nachricht.

Folgend ist eine HTTP GET Anfrage aufgelistet:

```
GET / HTTP/1.1
Host: www.opera.com
User-Agent: Opera
```

Quellcode 1: HTTP GET Request

Diese Nachricht spezifiziert, dass der Client eine Ressource erhalten möchte. Die Ressource befindet sich im root-Verzeichnis. Die Übertragung soll mittels HTTP-Version 1.1 stattfinden. Der Client versucht eine spezifische Webseite zu erreichen, die sich unter der URL `www.opera.com` befindet. Ferner teilt der Client über einen sog. HTTP Header, namens *User-Agent*, Informationen über das Programm, welches für die Kommunikation verwendet wurde, mit.

Eine Antwort vom Server könnte dabei wie folgt aussehen:

```
HTTP/1.1 200 OK
Date: Wed, 23 Nov 2011 19:41:37 GMT
Server: Apache
Content-Type: text/html; charset=utf-8
Set-Cookie: language=none; path=/; domain=www.opera.com;
    expires=Thu, 25-Aug-2030 19:41:38 GMT
Set-Cookie: language=en; path=/; domain=.opera.com;
    expires=Sat, 20-Nov-2030 19:41:38 GMT
Vary: Accept-Encoding
Transfer-Encoding: chunked

<!DOCTYPE html>
<html lang="en">
...
```

Quellcode 2: HTTP GET Response

Der Server antwortet, dass er das Protokoll HTTP Version 1.1 versteht. Die Anfrage war erfolgreich und wurde daher mit dem Response Code 200 sowie einer verständlichen Annotation *OK* versehen. Anschließend wird eine Reihe weiterer HTTP Header gesendet, welche beschreiben, wie die Nachricht verstanden werden soll. Und letztendlich wird der Inhalt der Ressource - hier ein HTML Dokument - in den Rumpf (body) der Nachricht eingefügt.

Weitere HTTP-Methoden sind: OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE, CONNECT. Jede von denen hat eine unterschiedliche Rolle. Siehe ([Fielding und Reschke 2014: Kap. 4](#)).

Vgl. [Dubost \(2012\)](#)

3.3 Serverseitige Webanwendungen

3.3.1 Grundprinzip

Als das World Wide Web geboren wurde, existierte nur ein Webserver und ein Webclient. Dieser Webserver namens httpd war nur in der Lage, statische Ressourcen wie Bilder und Dokumente auszuliefern. Schon bald jedoch machte der Überfluss an Online-Ressourcen Suchmaschinen notwendig. Das bedeutete, dass Benutzer in der Lage sein mussten, Daten, wie den Suchbegriff, an den Server abzuschicken und der Server seinerseits im Stande sein musste, diese Daten zu verarbeiten und dynamisch entsprechende Inhalte zu liefern.

Hierfür wurde das Common Gateway Interface (CGI) spezifiziert. Es entwickelte sich zum Standard, um externe Applikationen mit Webservern zu verbinden und um dynamische Information

zu generieren. Ein CGI Programm kann beinahe in jeder Programmiersprache implementiert werden. Es muss nur die Fähigkeit besitzen, Parameter Eingaben über Standardeingabe *STDIN* zu lesen und das Resultat auf die Standardausgabe *STDOUT* zu schreiben.

Folgend ist ein exemplarisches CGI „Hello World“ Programm dargestellt. Ein Benutzer namens Doug gibt seinen Namen ein, welcher vom Webserver ausgegeben werden soll. Dabei generiert sein Webclient einen HTTP GET Request an folgende URL:

<http://example.com/cgi-bin/hello.pl?username=Doug>

Wenn der Webserver diese Anfrage bekommt, weiß er, wie er die URL in zwei Teile trennt: den Pfad zu dem CGI Perl Programm *hello.pl* und den Teil mit der Benutzereingabe (username=Doug, genannt QUERY_STRING). Er leitet also diese Anfrage über Sitzungs-Identifikationsnummer an das *hello.pl* Script weiter. Die Aufgabe des Scriptes ist nun, den QUERY_STRING nach dem Schlüssel *username* zu parsen und dessen Wert über SDTOUT auszugeben. Der Webserver wird wiederum diese Ausgabe an den Client weiterleiten. Das Beispiel „Hello user“ Programm ist in Quellcode 3 abgebildet.

```
#!/usr/bin/perl

use CGI qw(:standard);
my $username = param('username') || "unknown";

print "Content-type: text/plain\n\n";
print "Hello $username!\n";
```

Quellcode 3: "Hello user"CGI script

Ein solches serverseitiges Programm generiert für gewöhnlich dynamische Inhalte, indem es die Information dafür aus einer Datenbank bezieht. Heutige serverseitige Webanwendungen nutzen weiterhin entweder eine Weiterentwicklung der CGI Schnittstelle oder ein ähnliches Prinzip.

Vgl. ([Bekman und Cholet 2003](#): Kap. 1.1)

Im Kapitel 3.1 wurde auf die Diskrepanz hingewiesen, dass HTTP ein zustandsloses Protokoll ist, eine Webanwendung jedoch Zustandshaftigkeit benötigt, um Benutzersitzungen, etwa Besuch und Einkauf in einem Webshop, auseinanderzuhalten. Eine solche Session kann von dem Client und Server Programm künstlich aufrecht erhalten werden.

Dabei generiert das serverseitige Programm eine Sitzungs-Identifikationsnummer (Session-ID) beim ersten Besuch des Benutzers und sendet diese an den Client. Der Client wiederum sendet

diese ID bei jeder weiteren Anfrage an den Server mit. Ein Mechanismus für die Benutzersessions im Web ist das Setzen von HTTP-Cookies, welche dann automatisch bei den nachfolgenden Anfragen angehängt werden. Weitere manuelle Möglichkeiten sind die Übertragung mittels custom HTTP Header oder das Umwandeln der URLs durch das Anhängen des zusätzlichen *session_id* Parameters.

Siehe ([Parikh u. a. 2015: Stateful Web Applications](#))

3.3.2 Architektur

In objektorientierter Software ist für die Erstellung von graphischen Anwendungen das Model-View-Controller-Muster vorherrschend.

[Lahres und Rayman \(2009\)](#) schreibt: „Mit Model-View-Controller (MVC) wird ein Interaktionsmuster in der Präsentationsschicht von Software beschrieben. MVC ist wohl einer der schillerndsten Begriffe im Bereich der objektorientierten Programmierung. Viele Varianten haben sich herausgebildet, teilweise einfach aufgrund eines falschen Verständnisses des ursprünglichen MVC-Musters, teilweise als Weiterentwicklung oder Anpassung an neue Anwendungsfälle.“

Unabhängig von der jeweiligen Abwandlung des MVC-Musters gilt, dass der Controller für die Benutzereingaben, das Model für den Zustand und der View für das Darstellen dieses Zustands verantwortlich sind. Vgl. [Lahres und Rayman \(2009\)](#) 8.2.3

Serverseitige Webanwendungen interpretieren MVC folgendermaßen, indem die Benutzerinteraktionen weitgehend zu Anfragen einer komplett neuen Ressource führen, e.g.:

```
GET http://example.com/articles
GET http://example.com/articles/1
GET http://example.com/articles/1/comments
```

Jeder ankommende HTTP Request wird von einem bestimmten Controller verarbeitet. Dieser liest HTTP Header sowie Request Parameter aus und verwendet Model Objekte, um die notwendige Daten für eine Benutzeranfrage zu liefern. Models laden diese Daten üblicherweise aus einer Datenbank. Und schließlich generiert der Controller die gesamte Seite neu, welche sich nur um den neuen Inhalt von der vorherigen unterscheidet, dessen Layout, Menüs, Kopfzeile der Webseite etc. aber gleich bleiben. Hierfür wird eine HTML-Template-Engine und ein passendes Template - das View - verwendet. Dieses beinhaltet im Grunde HTML Code mit Platzhaltern für dynamische Daten, welche vom Controller durch die Model-Daten ersetzt werden. Siehe [3.2](#)

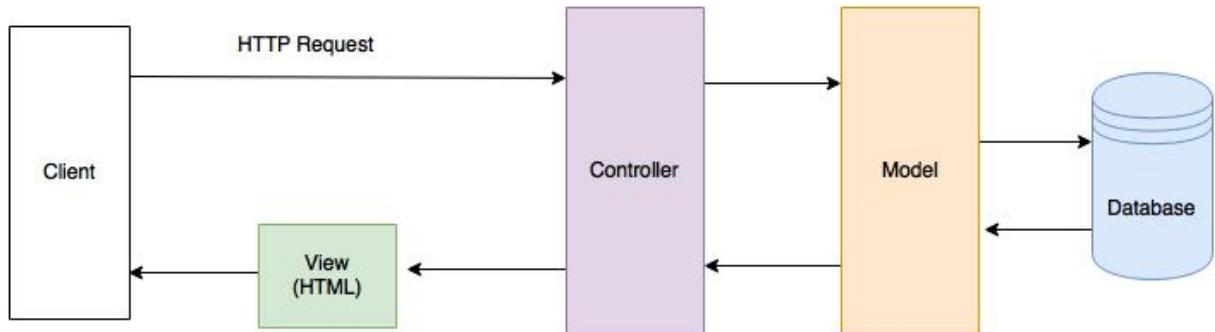


Abbildung 3.2: Serverseitiges MVC

3.4 Clientseitige Webanwendungen

3.4.1 Grundprinzip

In dem Aufsatz, der den Namen *Ajax* geprägt hat, schrieb [Garrett \(2005\)](#): „The classic web application model works like this: Most user actions in the interface trigger an HTTP request back to a web server. The server does some processing — retrieving data, crunching numbers, talking to various legacy systems — and then returns an HTML page to the client. It's a model adapted from the Web's original use as a hypertext medium, but as fans of The Elements of User Experience know, what makes the Web good for hypertext doesn't necessarily make it good for software applications.“

Schon in den 90er Jahren implementierten Browserhersteller Skripting Möglichkeiten für Webseiten. Es entstand die Möglichkeit, Programmcode an den Client Rechner zusammen mit dem Markup auszuliefern, um interaktive Animationen auf der Webseite auszulösen. Die Programmiersprache Javascript war geboren. Sie wurde später unter dem Begriff ECMAScript standardisiert.

Im weiteren Verlauf ermöglichte die Implementierung der XMLHttpRequest Schnittstelle, HTTP Anfragen aus Javascript unabhängig von dem Browser Client auszuführen. Dies legte den Grundstein für die von Garret unter dem Begriff *Ajax* zusammengefasste Sammlung von Techniken und somit die Entstehung echter clientseitiger Webanwendungen.

So schreibt [Garrett \(2005\)](#) weiter: „Ajax isn't a technology. It's really several technologies, each flourishing in its own right, coming together in powerful new ways.“ Und er definiert die Komponenten, welche Ajax ausmachen.

- Standards-basierte Darstellung, unter Verwendung von XHTML und CSS

- dynamische Darstellung und Interaktion, unter Verwendung von Document Object Model
- Datenaustausch und Manipulation mit Hilfe von XML und XSLT
- Asynchrone Datenabfragen, unter Verwendung des XMLHttpRequests
- Javascript, welches das Ganze zusammenbindet

Anstatt eine Webseite am Anfang der Benutzersitzung zu laden, lädt der Browser eine Ajax Engine - implementiert in Javascript. Diese Engine ist sowohl verantwortlich für das Rendern des Benutzerinterfaces als auch für die Kommunikation zwischen dem Server seitens des Benutzers.

Die von Garret formulierten Ajax Bestandteile gelten noch heute. Allerdings spielt das Datenformat XML keine primäre Rolle. Es ist kein bestimmtes Datenaustausch Format vorgeschrieben, wobei überwiegend das kompakte JSON Format zur Übertragung benutzt wird. Der Schnittstellename - XMLHttpRequest blieb aus Kompatibilitätsgründen erhalten. Ab ECMAScript6 existiert eine neue HTTP Api, namens *fetch*.

Vgl. [Garrett \(2005\)](#)

3.4.2 Datenübertragung

Die meist konventionelle Art und Weise, in der Ajax Applikationen mit dem Server kommunizieren, ist eine sog. REST Api. REST steht für representational state transfer.

- *representational* bezieht sich darauf, wie eine Representation einer Ressource übertragen wird.
- *state transfer* bezieht sich darauf, dass HTTP ein zustandsloses Protokoll ist, und dass alles, was der Server zur Verarbeitung einer Anfrage braucht, sich in der Anfrage selbst befindet.

Die Grundideen hinter REST wurden auf den Beobachtungen dessen basiert, wie das Web bereits funktionierte. Das Laden von Webseiten, das Absenden von Formularen und das Benutzen von Links, um verwandte Inhalte zu finden, sind Faktoren, welche definieren, was REST ist und wie es auf das Web und das Schnittstellendesign zutrifft.

Alle Aktionen innerhalb REST konzentrieren sich also auf Ressourcen und somit stellen das Erzeugen (Create), Lesen (Read), Aktualisieren (Update) und Löschen (Delete) die einzigen Aktionen dar, welche auf diese Ressourcen angewendet werden. Das Akronym, welches diese vier Aktionen beschreibt, ist demnach *CRUD*. Beispielsweise würde innerhalb des REST Paradigma,

um einen Benutzer einzuloggen, nicht etwa eine Remote Procedure *User.login(username, password)* aufgerufen, sondern eine Ressource *UserSession* mit den Attributen *username, password* auf dem Server erstellt.

Eine anschauliche Art, REST Schnittstellen zu implementieren, ist, alle Aktionen auf zwei Kriterien zu brechen:

- *Was* - Auf welche Ressource wird eingewirkt ?
- *Wie* - Was passiert mit der Ressource ?

Folgend ist tabelarisch die Akzentuierung des „Was“ und des „Wie“ auf den Design einer REST API für den Benutzerlogin dargestellt.

Zielsetzung	Wie		Was	
	Operation	HTTP Methode	Ressource	Pfad
Information über die Session bekommen	Read	GET	Session	/sessions/:id
Eine neue Session erzeugen (Benutzer einloggen)	Create	POST	Sessions Liste	/sessions
Neue Daten zur Session hinzufügen	Update	PUT	Session	/sessions/:id
Session löschen (Benutzer ausloggen)	Delete	DELETE	Session	/sessions/:id

Vgl. ([LaunchSchool 2016](#): Kap. REST and CRUD)

3.4.3 Architektur

Clientseitige Webanwendungen verzichten durchgehend auf komplettes Laden der Webseite bei Benutzerinteraktionen. Sie reagieren auf Eingaben, indem sie das Document Object Model (DOM) der Webseite im Speicher direkt verändern und somit ein Neurendern der betroffenen Teilbereiche auslösen. Anstelle des kompletten Markups werden lediglich die notwendigen neuen Daten in einem serialisierten Format von einem Webservice über eine definierte API abgefragt.

Auch hier wird bei einer objektorientierten Umsetzung das MVC Pattern bzw. eine Abwandlung davon verwendet. Dabei werden Benutzereingaben direkt von einem dafür zuständigen Controller verarbeitet. Der Controller leitet die Anfrage an das Model weiter, welches mit einem Webservice interagieren kann. Anschließend benachrichtigt der Controller das View über die Änderung

des Zustands der Applikation. Schließlich sorgt das View für die direkte DOM Manipulation eines bestimmten Teilbereiches der Seite. Siehe 3.3.

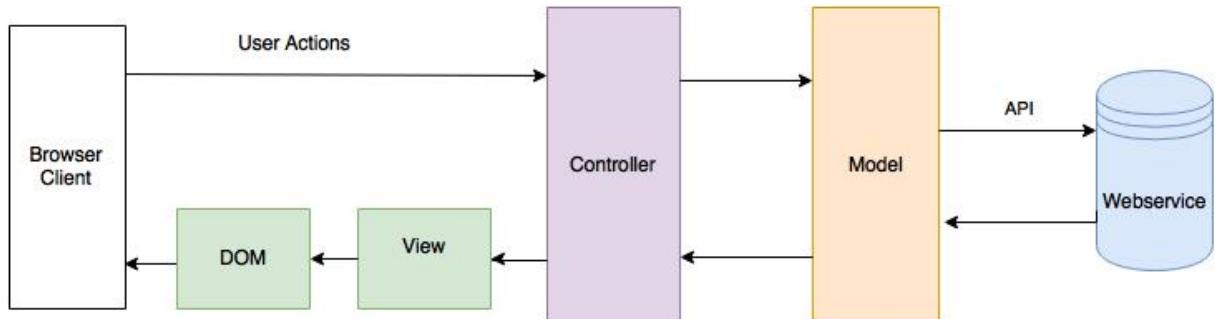


Abbildung 3.3: Clientseitiges MVC

Die meisten clientseitigen Frameworks führen eine Technik namens [Databinding](#) ein. Dabei wird einst aus einem Markup Template erstelltes View an bestimmte Datenfelder in einem Model angebunden. Das View aktualisiert sich mit der Zustandsänderung des Models automatisch. Somit bleibt kaum noch Eventverarbeitungsaufwand für einen typischen Controller notwendig, weswegen dieser eher als Presenter (siehe [MSDN \(2010\)](#)) oder sog. ViewModel (siehe [MSDN \(2009\)](#)) verstanden werden kann.

3.5 Responsive Webdesign

([Marcotte 2011](#): S.8) führt den Begriff Responsive Webdesign mit folgenden Worten ein:

„But web designers, facing a changing landscape of new devices and contexts, are now forced to overcome the constraints we've imposed on the web's innate exibility. We need to let go. Rather than creating disconnected designs, each tailored to a particular device or browser, we should instead treat them as facets of the same experience. In other words, we can craft sites that are not only more exible, but that can adapt to the media that renders them. In short, we need to practice responsive web design“

Hierfür legte Marcotte drei Grundelemente fest:

- Flexibles, rasterbasiertes Layout
- Flexible Bilder und Medien
- Media Queries, ein Modul aus der CSS3 Spezifikation

Üblicherweise platzieren Webdesigner die Webseitenelemente auf ein imaginäres Raster. Somit sind bestimmte Komponenten aneinander anhand dieser virtuellen Linien ausgerichtet. In Abb. 3.4 sieht man ein solches Raster. Das Design besteht hier aus zwei Textspalten *main* - 566px breit und *other* - 331 px breit. Diese befinden sich wiederum in einem umschließenden *blog* 900 px Container, welcher wiederum in den *page* Container eingebunden ist. Die Elemente definieren entsprechende Abstände (*margin*), so dass aus Breite der Elemente und der Abstände sich eine Textpositionierung anhand der virtuellen Grid Linien ergibt, wie in der Abbildung 3.4 dargestellt.

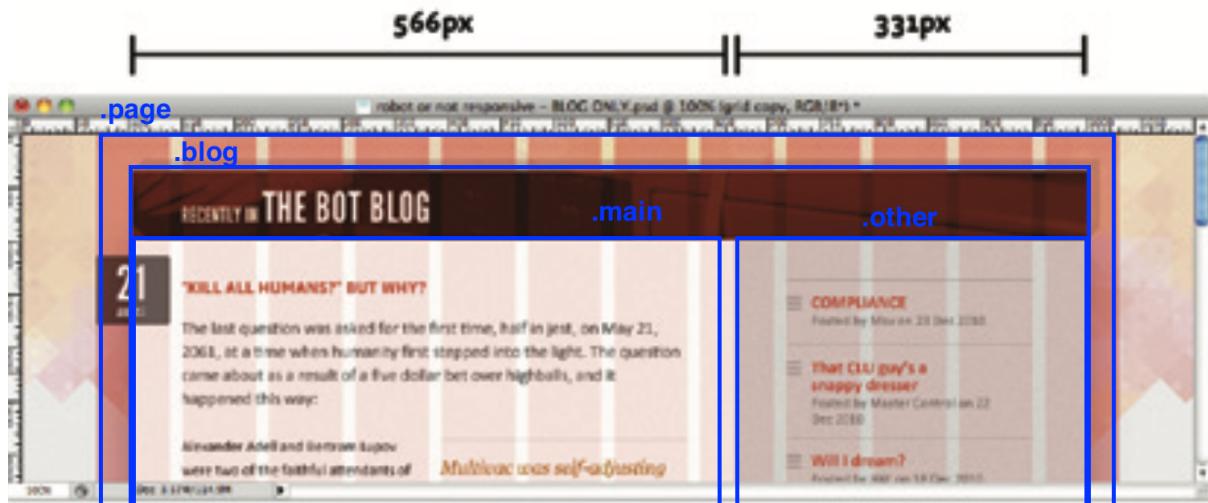


Abbildung 3.4: Webdesign Raster. Quelle: Marcotte (2011), S.27

Das Layout für die Abbildung 3.4 kann durch den CSS Code in Quellcode 4 erzeugt werden.

```
#page {
    margin: 36px auto;
    width: 960px;
}
.blog {
    margin: 0 auto 53px;
    width: 900px;
}
.blog .main {
    float: left;
    width: 566px;
}
.blog .other {
    float: right;
    width: 331px;
}
```

Quellcode 4: Raster CSS

3 Grundlagen

Um nun von einem starren Raster auf ein flexibles Raster zu kommen, welches sich an Bildschirmgrößen anpasst, werden feste Pixel Angaben durch relative Prozentangaben ersetzt. Hierfür wird eine einfach Umrechnungsformel verwendet - $target/context = result$

Wenn also sich das Zielelement *main* mit 566px Breite bei einem festen Raster innerhalb des Kontextelementes *blog* mit 900px Breite befindet, ergibt sich dafür laut $566/900$ eine 62.8888889% Breite:

```
.blog .main {  
    float: left;  
    width: 62.8888889%;  
}
```

Quellcode 5: Flexibles Raster CSS

Im Prinzip wird die obige Umrechnung auf alle festen Breiten-, Abstand- und Fontgrößenangaben angewendet. Daraus ergibt sich ein anpassbares Raster(siehe Abb. 3.5).



Abbildung 3.5: Flexibles Webdesign Raster. Quelle: Marcotte (2011), S.33

Bezüglich flexibler Bilder und Medien verhält es sich ähnlich, wie mit dem Grundansatz des flexiblen Rasters. Werden Bilder in Markup Container eingebettet, so sollte dieser Container seine Abstände sowie Breite wiederum relativ zu seinem Contexcontainer gesetzt haben. Siehe Quellcode.

```
<div class="figure">  
    <p>  
          
        <b class="figcaption">Lo, the robot walks</b>  
    </p>  
</div>
```

Quellcode 6: Container mit Bild Markup

```
.figure {
    float: right;
    margin-bottom: 0.5em;
    margin-left: 2.53164557;
    width: 48.7341772%;
}
```

Quellcode 7: Container mit Bild CSS

Diese Einstellung allein wird aber nicht ausreichen. Sollte das Bild in Wirklichkeit größer sein, als sich Platz dafür resultierend aus der dynamischen Breitenangabe und der jeweiligen Gerätauflösung ergibt, wird es standardmäßig mit Scrollbalken innerhalb seines Containers in original Größe gerendert.

Die Lösung dieser Problematik hängt von dem gewünschten Effekt ab. In vielen Fällen wird einfach das gleiche Skalierverhalten für das Bild gewünscht, wie für den Rest des Inhalts. Hierfür sorgt eine *max-width: 100%* Einstellung auf das Bildelement selbst.

```
.figure {
    float: right;
    margin-bottom: 0.5em;
    margin-left: 2.53164557;
    width: 48.7341772%;
}

.figure img {
    max-width: 100%;
}
```

Quellcode 8: Bildskalierung

Es kann aber auch ausreichend sein, dass lediglich ein bestimmter Bildausschnitt gerendert wird, so im Falle von Hintergrundbildern. Dieses wird mit Hilfe einer *overflow: hidden*% CSS Regel auf dem Contextcontainer des Bildes erreicht.

```
.figure {
    overflow: hidden;
}
.feature img {
    display: block;
    max-width: auto;
}
```

Quellcode 9: Bildausschnitt

3 Grundlagen

Denkbar wäre auch die Absicht ein komplett anderes, eventuell ähnliches, aber unterschiedlich arrangiertes Bild für kleinere Auflösungen zu laden, im Falle von Grafiken beispielsweise. Dieses erfordert jedoch eine zusätzliche Implementierung auf der Serverseite.

In dieser Arbeit wird eine weitere Technik für den Fotoslider geschildert, wobei das Foto nicht nur mit dem Inhalt skaliert, sondern zusätzlich zentriert sein muss.

4 Design

Das folgende Kapitel beschreibt das User Experience Design für die in Kapitel 2 definierten Anforderungen.

4.1 Farb Palette

Die Anwendung benutzt die Farbpalette *blue-grey* als Hauptpalette und *blue-grey* als Akzentpalette gemäß Material Design Terminologie. Das Resultat ist in Abb. 4.1 dargestellt.

#455A64	#CFD8DC	#607D8B	#FFFFFF
DARK PRIMARY COLOR	LIGHT PRIMARY COLOR	PRIMARY COLOR	TEXT / ICONS
#00BCD4	#212121	#757575	#bdbdbd
ACCENT COLOR	PRIMARY TEXT	SECONDARY TEXT	DIVIDER COLOR

Abbildung 4.1: Farbpalette

4.2 Authentifizierung

Das Visual Design für die Authentifizierung (2.3.1) ist in der Abb. 4.2 dargestellt.

Der Benutzer gibt hier seine Login-Daten ein. Bei positiver Eingabe gelangt er in die Anwendung. Als Standardansicht wird die Fotogalerie angezeigt.

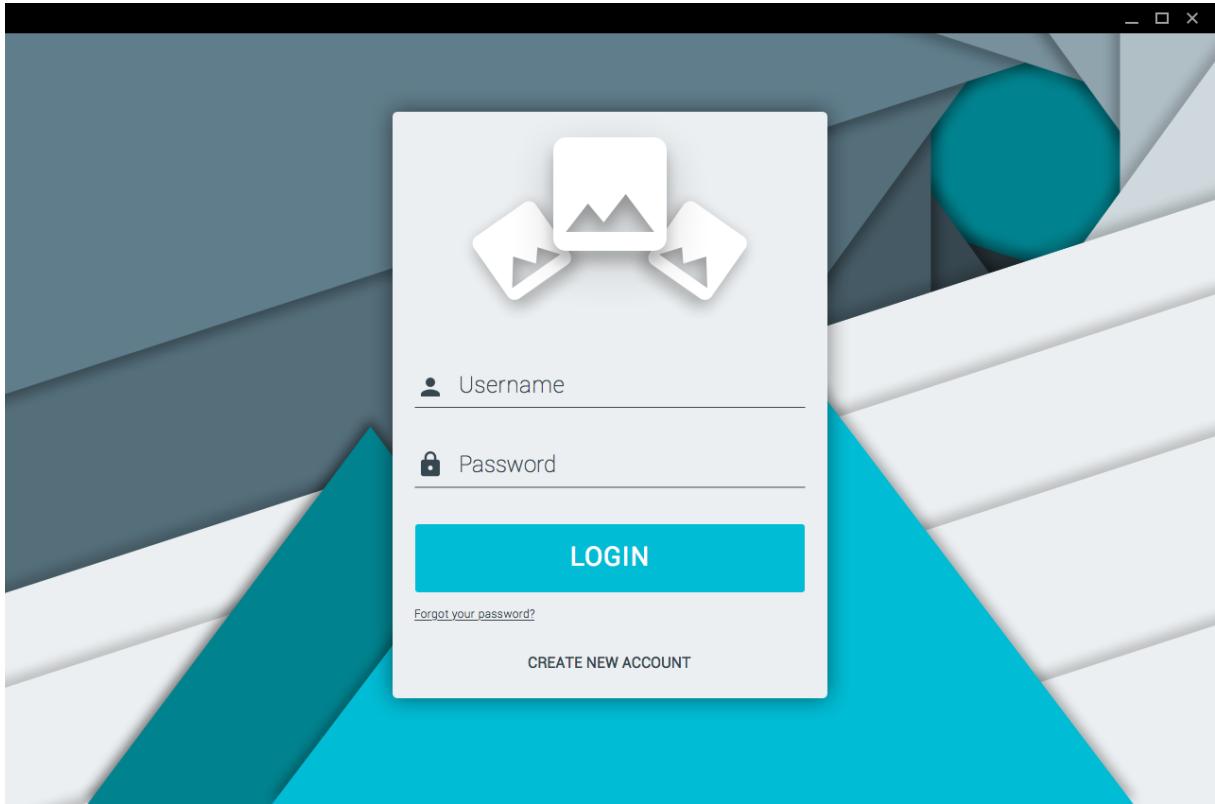


Abbildung 4.2: Authentifizierung

4.3 Navigation Menü

Die Navigation innerhalb der Anwendung (Anforderung 2.3.2) geschieht durch ein von links ausklappbares Seitenmenü (Abb. 4.3). Dieses legt sich über den Inhalt im Hauptbereich der Anwendung. Der Button zum Ausklappen befindet sich in jedem Unterbereich der Anwendung oben links in der Tab-Leiste des Headers (siehe Abb 4.4). Das Menü nimmt auf kleinen Geräten die gesamte Bildschirmgröße ein.

Zusätzlich bekommt jeder Unterbereich der Anwendung eine spezifische Werkzeugpalette für die Aufgaben des jeweiligen Kontextes. Die spezifischen Werkzeuge haften am rechten oberen Rand der Tab-Leiste. Das sind unter anderem der Such-Icon für die Sucheingabe, das Plus-Icon zum Hinzufügen weiterer Bilder in die Galerie oder in der Detailansicht des Fotos sind es die Icons zum Löschen des jeweiligen Bildes, zum Heranzoomen und der Info-Icon für die Detailinformationen zum Bild wie der Name, die Beschreibung oder auch die Exif-Daten.

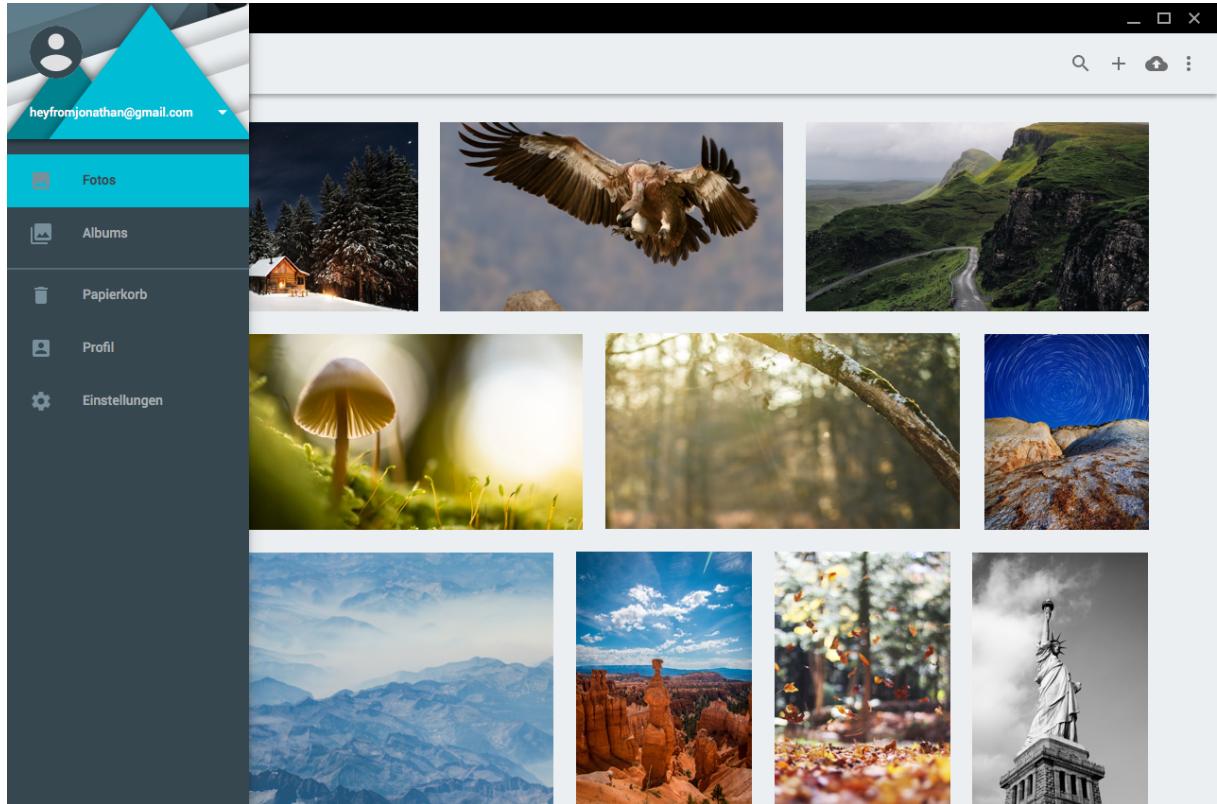


Abbildung 4.3: Foto Galerie Large

4.4 Foto Galerie

Das Design der Galerie aus Anforderung 2.3.3 ist für die Bildschirmgrößen - Large, Normal, Small in den Abb.4.3, 4.4, 4.5 entsprechend visualisiert. Die Anwendung passt sich flexibel an wachsende Bildschirmgrößen an. Generell findet ein Umfluss der Bilder von mehr auf weniger Spalten statt. Die Größe Large umfasst sechs Spalten, in der Medium- bzw. Normalgröße sind es vier Spalten, bis letztendlich eine einzelne Spalte mit Bildern übrig bleibt.

Die beiden Entwürfe in den Abb. 4.4, 4.5 zeigen ebenfalls die Gruppierung der Fotos nach dem Erstellungsmonat (Anforderung 2.3.6) und ein Suchfeld in der oberen Werkzeugeiste (Anforderung 2.3.5). Die Eingabe eines Suchbegriffs führt instantan eine Suche durch, sobald mehr als zwei Zeichen eingegeben wurden. Das Ergebnis der Suche ist ebenfalls nach Erstellungsmonat gruppiert.

4 Design

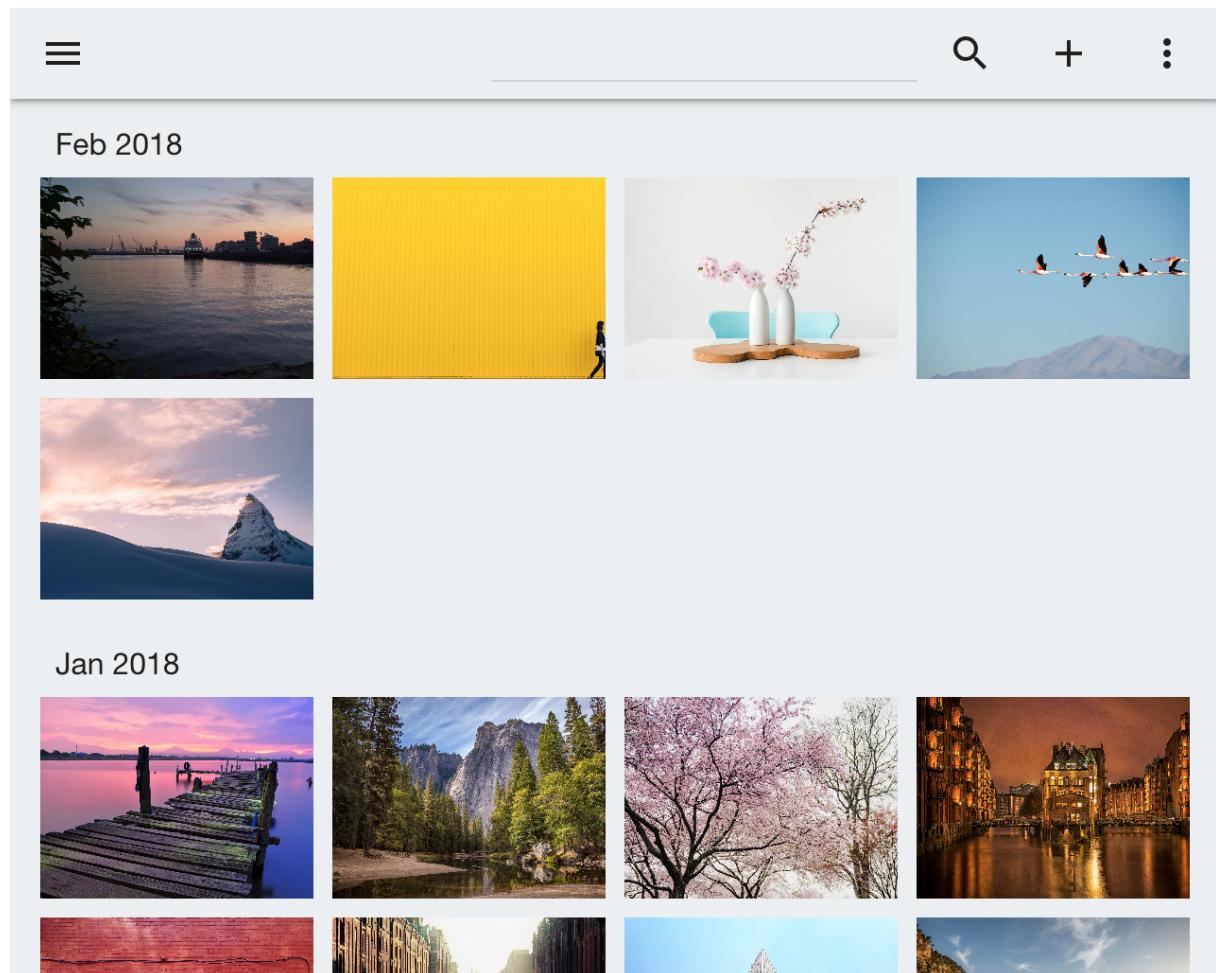


Abbildung 4.4: Foto Galerie Normal

4 Design

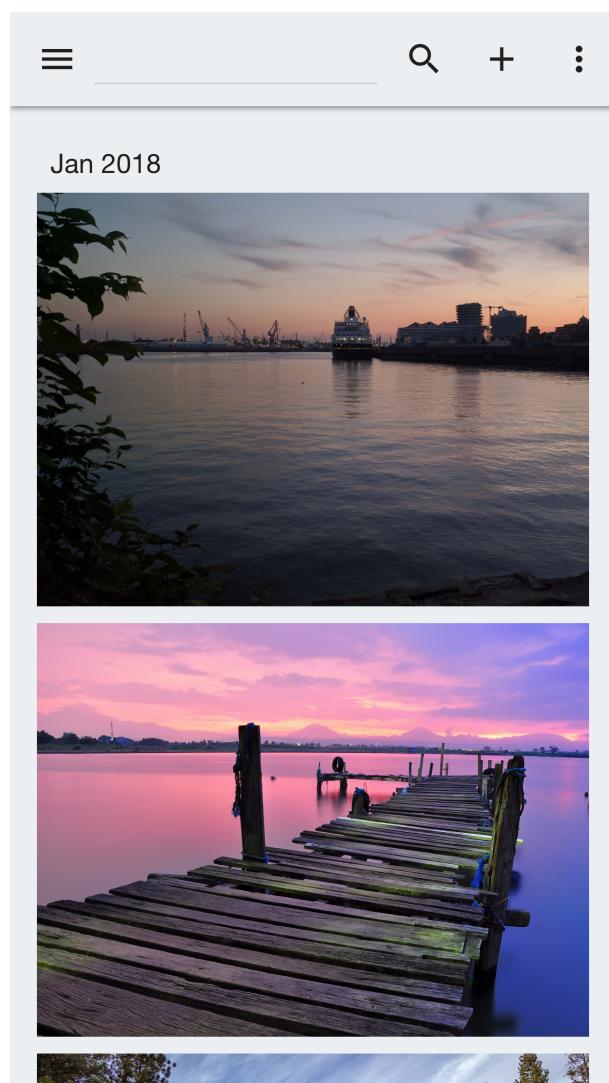


Abbildung 4.5: Foto Galerie Small

4.5 Paginierung

Für die Anforderung der Paginierung (2.3.4) bietet die Anwendung dem Benutzer die verlinkten Seitenzahl-Buttons für die jeweils zwei vorherige und zwei kommende Fotostapel. Die Paginierungsbuttons befinden sich am unteren Ende des jeweiligen Fotostapels. Die aktuelle Seite ist farbig hervorgehoben. (siehe Abb. 4.6).

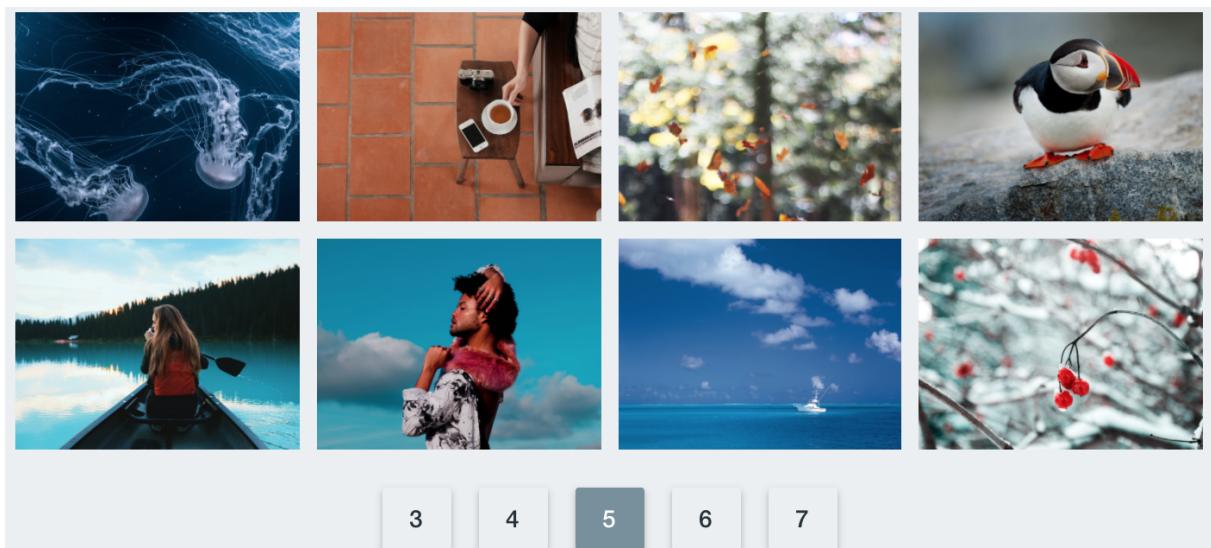


Abbildung 4.6: Paginierung

4.6 Foto Details

Die Anforderung der Großansicht des Bildes und der Metadaten-Betrachtung und -Bearbeitung (2.3.7) ist in dem Visual Design 4.7 für die Bildschirmgröße Large abgebildet. Der Benutzer gelangt in die Detailansicht des Fotos mit dem Klick auf ein entsprechendes Foto in der Galerie.

Die Metadatenansicht ist im Standardzustand erstmal nicht sichtbar und wird mit dem Klick auf das Info-Icon von rechts ausgeklappt. Das zentrierte Foto verschiebt sich entsprechend und skaliert im restlichen Bereich.

Auf kleineren Geräten nimmt entweder das Foto (Abb. 4.8) oder das Metadatenformular (Abb. 4.9) den gesamten Platz ein.

Die EXIF-Kameradaten werden nach Erstellungsdatum, Kameraeinstellungen und Fotodimensionen gesondert kategorisiert mit einem entsprechenden Icon pro Kategorie angezeigt.

4 Design

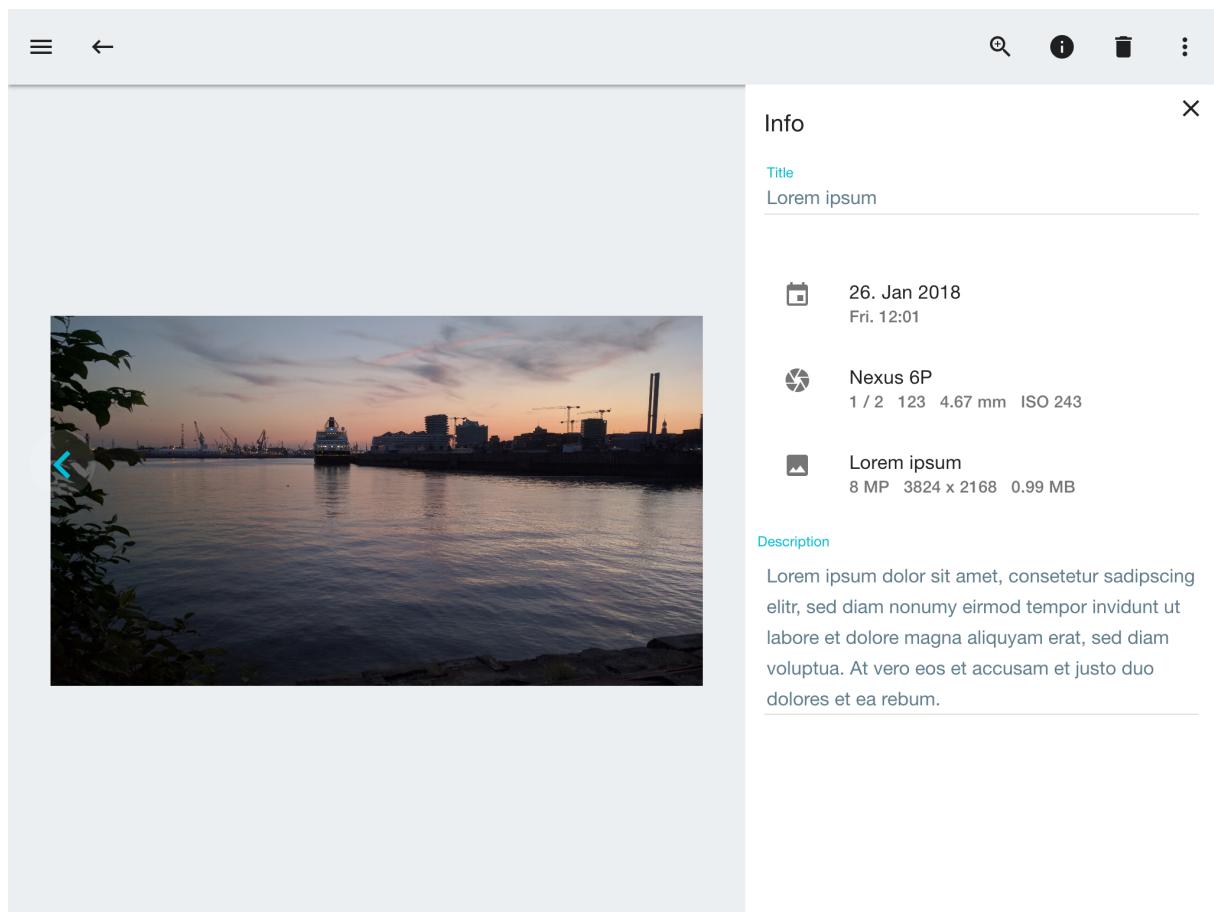


Abbildung 4.7: Foto Details Large

4 Design

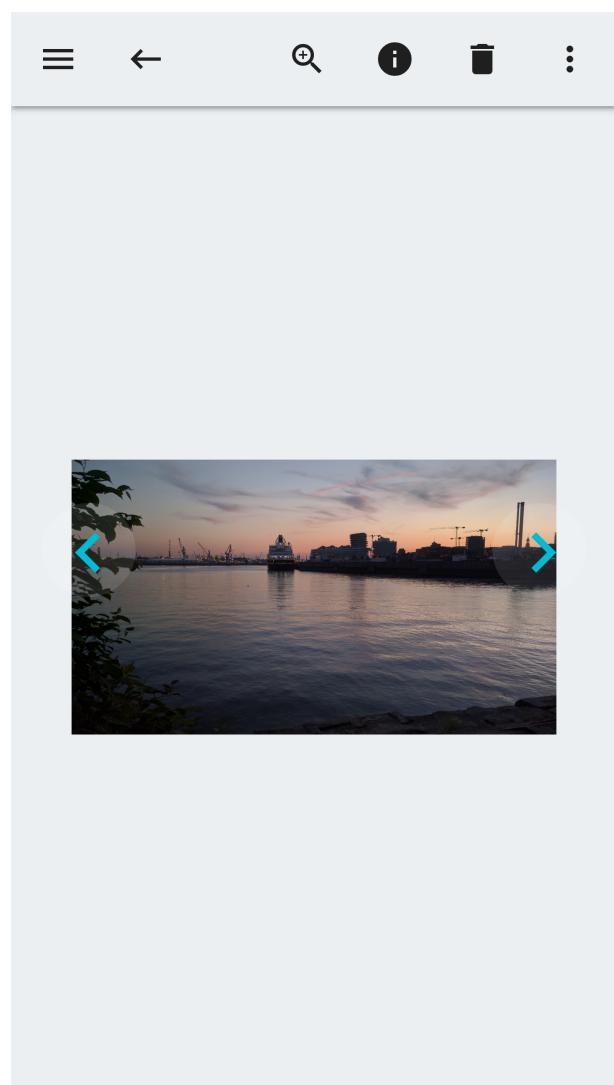


Abbildung 4.8: Foto Details Bild Small

4 Design

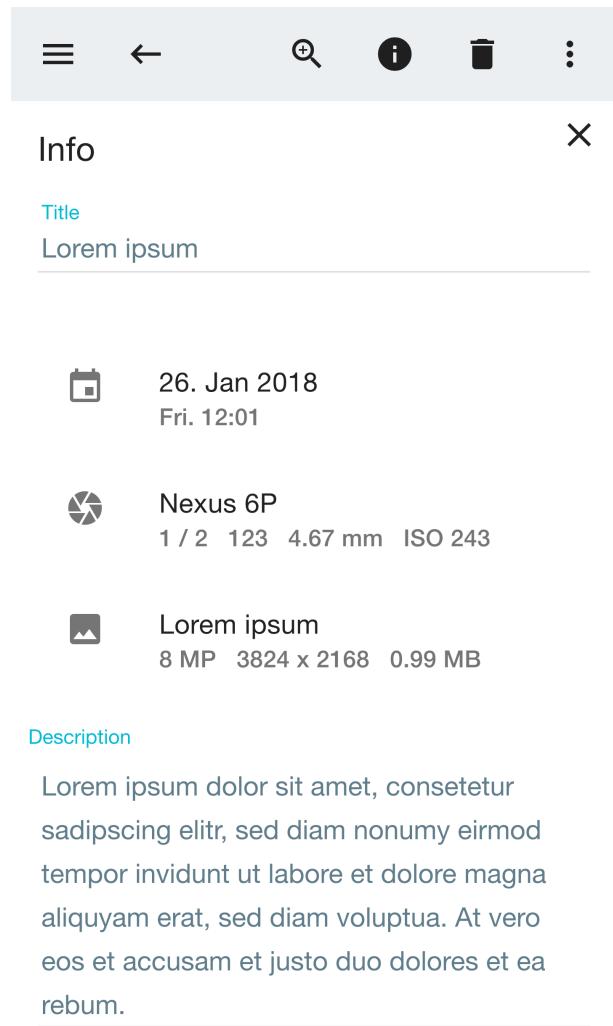


Abbildung 4.9: Foto Details Metadaten Small

4 Design

Die Abbildungen zeigen ebenfalls, dass das betrachtete Foto bei abnehmender Displaygröße proportional herunterskaliert und immer auf dem Bildschirm zentriert wird (Anforderung [2.2.4](#)).

Des Weiteren sind über der Fotoansicht mittig blaue Pfeile für das Sliden zur nächsten bzw. vorherigen Detailansicht eines Fotos in den Entwürfen [4.7](#), [4.8](#) dargestellt. (Anforderung [2.3.9](#))

Die Buttons mit den Pfeilen werden im inaktiven Zustand, wenn man die Maus über dem Foto ruhen lässt, leicht transparent dargestellt, um die Ansicht des Fotos nicht zu stören.

5 Implementierung

5.1 Einleitung

In Kapitel 1.2 wurde als Zielsetzung bestimmt, dass diese Arbeit grundlegende Problemstellungen bei der Implementierung einer komplexen clientseitigen Webanwendung betrachtet. Ferner hat 3.4 gezeigt, dass hierfür eine komplette *Ajax-Engine* an den Client über HTTP ausgeliefert werden muss.

Für die Umsetzung dieser clientseitigen Architektur existiert eine Reihe von JavaScript Frameworks. Für die tatsächliche Implementierung der exemplarischen Software zur Fotoverwaltung verwendet diese Arbeit das AngularJS 1.x Framework. Dieses hält sich stark an die Konzepte objektorientierter Programmierung, liefert Bibliotheken zur Umsetzung des MVC-Patterns aus Kapitel 3.4.3 und eine Reihe weiterer für die Webbrowserumgebung relevanten Werkzeuge wie URL-Routing und DOM-Rendering.

Für die Umsetzung des Benutzerinterfaces wird das mit AngularJS kompatible Angular Material Framework verwendet. Dieses bringt Material Farbenpaletten, graphische Elemente und Layout Abstraktion als [AngularJS Direktive](#) auf der Grundlage des CSS Flexbox Layouts mit vordefinierten CSS Media Queries für die Responsive Design Umgebung.

5.2 Layout

Die Haupt- und Akzentpalette lassen sich bei dem Einbinden von Angular Material einmalig konfigurieren. Die UI Komponenten setzen das Theming oft automatisch um. Dort, wo es nicht passiert bzw. weitere Farbzustände für das Feedback bei Benutzerinteraktion benötigt werden, lassen sich über vorgegebene Parameter z.B. *md-primary*, *md-raised* die Zustände setzen bzw. per Databinding umschalten.

Für das Navigationsmenü (Abb. 4.3, links) wird die [AngularJS Direktive](#) *md-sidenav* verwendet. Die Komponente bietet Schnittstellen zum Ein- und Ausklappen. Die Seitenavigation lässt sich

5 Implementierung

parametrisiert entweder über den Inhalt legen oder kann den Inhalt wegschieben. Im Fall des Navigationsmenüs wird die erste Option verwendet.

Das Kontextmenü in der Tab-Leiste (Abb. 4.3, oben) wird mit Hilfe des *md-toolbar* Elements umgesetzt. Werkzeuge, welche nicht auf die Leiste passen, lassen sich mit *md-menu-content* in ein Kontextmenü verstecken, welches erst beim Klicken erscheint.

Angular Material bringt ebenfalls Container Abstraktionen für das Anordnen der Elemente. Es existieren Zeilen (*layout="row"*), Spalten (*layout="column"*) und Gridcontainer (*md-grid*), welche Elemente horizontal bzw. vertikal anordnen. Dieses wird intern mit Hilfe von entsprechenden Flexbox-Eigenschaften umgesetzt. Die Container sorgen automatisch dafür, dass die Größe der inneren Elemente für verschiedene Auflösungen skaliert wird. Alle Größenangaben werden prozentual zu der Größe des Elternelements gemacht, damit die responsive Eigenschaft gewahrt bleibt (siehe Abb. 5.1).

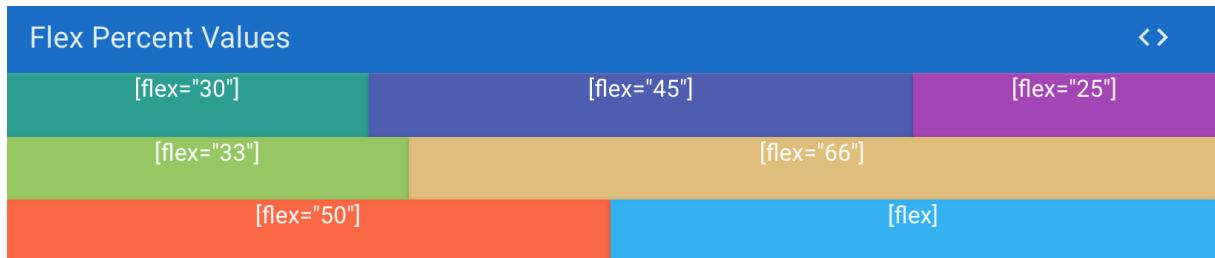


Abbildung 5.1: Layout Container. Quelle: material.angularjs.org

Die Fotogalerie (Abb. 4.3, mittig) besteht aus jeweils einem Zeilencontainer pro Gruppierungsmonat der Aufnahmen. Die Zeilencontainer enthalten wiederum zwei weitere Container - die Monatsüberschrift und einen Gridcontainer *md-grid-list* für die Fotokacheln.

Foto-Details-Metadaten (Abb. 4.7, rechts) befinden sich in einem von rechts ausklappbaren *md-sidenav* Element. Es schiebt den Inhalt mit der Großansicht des Fotos weg und verdrängt es komplett auf kleinen Geräten. Die *md-input* Formularfelder selbst lassen sich in einer Spalte als Zeilen anordnen.

Das Foto (Abb. 4.7, links) wird mit Hilfe der Flexbox *justify-content* Eigenschaft zentriert. In Angular Material wird dies über den Parameter *layout-align="center center"* auf dem Wrapper Container erreicht. Damit das Foto aber auch korrekt proportional zu seinen eigentlichen Dimensionen skaliert, müssen alle weiteren Elterncontainer des Imageelements 100% ihrer verfügbaren Breite und Höhe einnehmen. Hierfür sorgt die Eigenschaft *layout-fill*.

5.3 Grund Setup

Um die *Ajax-Engine* zu bündeln, kommt [Webpack](#) zum Einsatz. Dieses erlaubt das Zusammenbauen mehrerer öffentlicher (vendor) und eigener JavaScript-Module zu wenigen Buildfiles, welche bequem im Hauptmarkup eingebunden werden. Zudem wird dadurch ermöglicht, moderne JavaScript Syntax (e.g [ES2016](#)) durch Code Preprocessing durch sog. Transpiler zu verwenden. Ein solcher Transpiler wandelt z.B. einen modernen JavaScript Standard in einen ursprünglichen Standard, welcher von den Zielbrowsern der Anwendung bereits implementiert wurde, um.

Das Resultat eines solchen Builds ist eine einzige HTML-Seite *index.html*, die der Benutzer am Ende ausgeliefert bekommt. Die Index Seitebettet wiederum nur zwei zusammengebaute Scripte aus allen bestehenden ein - *index.bundle.css* und *index.bundle.js*. Diese sind entsprechend für das gesamte Layout und die gesamte Business-Logik der Anwendung verantwortlich (siehe Quellcode 10).

```
<!doctype html>

<head>
  <link rel="stylesheet" type="text/css" href="build/index.bundle.css">
  <title>Pika</title>
</head>
<body ng-app="pika">
  ...

<script src="build/vendor.bundle.js"></script>
<script src="build/index.bundle.js"></script>
</body>
</html>
```

Quellcode 10: index.html

[Webpack](#) setzt auf das in [ES2015](#) eingeführte *Modules*Feature auf. Hierbei wird ein Entry-Script in der [Webpack](#)-Konfigurationsdatei deklariert (ähnlich dem Main File eines herkömmlichen Programms). [Webpack](#) durchläuft beim Build jedes darin importierte File rekursiv und lässt es anhand weiterer deklarerter Regeln durch entsprechende Präprozessoren verarbeiten. Sowohl Stylesheets als auch Skripte werden so verarbeitet. Der Output dieser Präprozessoren wird am Ende, wie in Quellcode 10 abgebildet, zu einzelnen Files gebündelt.

In Quellcode 11 ist ein Auszug aus der *Pika* Build Konfiguration zum Transpilieren von [ES2016](#) JavaScript Standard zu [ES5](#) Standard mit Hilfe des *babel-loader*, sowie von [SASS](#) zu CSS mit Hilfe des *sass-loader* dargestellt.

```
{
  entry: {
    index: ['./index.js']
  },
  module: {
    rules: [
      {test: /\.scss$/, use: ['sass-loader']},
      {test: /\.js$/, use: ['babel-loader']}
    ]
  }
}
```

Quellcode 11: webpack.config.js

Für diese Applikation werden insgesamt folgende Regeln deklariert:

- Herkömmliche CSS-Bündelung aus anderen Paketen
- Statische Assets-Einbindung und Fingerprinting für besseres Browsecaching
- Präprozessing von [ES2016](#) zu [ES5](#) für moderne JavaScript Features, z.B. Klassen, Modularisierung, Arrow Functions, Parameter Destructuring, Promises
- Präprozessing von [SASS](#) zu CSS für erweitertes Stylesheet Tooling, wie Schachtelung, Vererbung, Variablen-Verwendung
- Präprozessing durch [Pug](#) (ehemal. *Jade*) Templates zu HTML für bessere Markup Lesbarkeit
- Extrahierung des separaten CSS-Bundles
- Trennung von Vendor und Applikation Codes in einzelne Bundles

Wie bereits oben erwähnt, bekommt der Benutzer eine einzelne Seite mit den gebündelten Skripten ausgeliefert. Der tatsächliche Inhalt der Applikation wird also erst durch das Bundle Skript auf dem Client erzeugt. Hierfür muss das AngularJS Framework wissen, welcher Platzhalter im DOM das Ziel des clientseitigen Renderings darstellen soll.

Quellcode 10 zeigt, dass der *body* Tag hierfür mit dem entsprechenden Attribut *ng-app="pika"* markiert worden ist. Man spricht hier auch von einer [AngularJS Direktive](#). Damit AngularJS die Kontrolle über den Inhalt des *body* Tags übernehmen kann, ist schlussendlich der folgende Aufruf im *index.js* File notwendig.

```
import angular from 'angular';
angular.module('pika', [
  //list of submodules
])
```

5.4 Routing

Da Pika immer noch eine Webanwendung ist und im Browser ausgeführt wird, erwarten Benutzer, dass das Navigieren zwischen einzelnen Bereichen der Applikation mit einer Veränderung der URL in der Adresszeile des Browsers einhergeht. Ebenfalls soll es möglich sein, zu einem bestimmten Bereich zu gelangen, indem man eine URL direkt in die Adresszeile des Browsers eingibt.

In 3.1 wurde jedoch erläutert, dass mit einer neuen Adresseingabe auch eine separate HTTP Anfrage verbunden ist. AngularJS bietet daher ein Routing Mechanismus an, welcher das Standardverhalten der Browser bei Adresseingaben kapert.

URL-Adresseingaben gehen zunächst durch den AngularJS Router in dem Client Code. Stellt dieser eine Adresse fest, welche in der Routingkonfiguration festgelegt wurde, so wird ein entsprechender Controller aufgerufen und seine Ausgabe in dem Hauptlayoutplatzhalter gerendert.

```
<body ng-app="pika" layout="row" layout-fill>
<div layout="column" layout-fill ng-cloak="">
  <side-menu />
  <ui-view role="main" layout="column" layout-fill></ui-view>
</div>
</body>
```

Quellcode 12: Hauptlayout

Quellcode 12 zeigt den restlichen Inhalt der *index.html*-Datei. Entscheidend ist die Direktive *ng-app="pika"*. Sie sorgt dafür, dass das in *index.js* (Quellcode 16) deklarierte Angular Hauptmodul das Rendering des Inhalts hier übernehmen kann.

Der Inhalt der Anwendung besteht aus den Direktiven für das Seitenmenü (*side-menu*) und dem jeweiligen Inhalt des Routers (*ui-view*).

Letztendlich braucht das AngularJS Hauptmodul eine Konfiguration für das Routing von URLs zu den jeweiligen Hauptkontrollern der Applikation. Quellcode 13 zeigt einen Auszug dieser

5 Implementierung

Konfiguration. In dieser Arbeit wird ein 3rd Party Router, namens *UI-Router* verwendet (Der Standardrouter ist bereits für den exemplarischen Anwendungsfall dieser Arbeit sehr eingeschränkt. Die Notwendigkeit für den UI-Router wird im weiteren Verlauf deutlich).

```
// routes.js
import photosTemplate from './photos/index.pug';
import photoDetailTemplate from './photo-detail/index.pug';

function routesConfig($stateProvider, $urlRouterProvider) {
  $stateProvider
    // ...
    .state('photos', {
      url: '/photos/:page?search',
      template: photosTemplate,
      controller: 'photosController',
      controllerAs: '$ctrl',
      resolve: { authenticate: authenticate }
    })
    .state('photo-detail', {
      url: '/photo-detail/:id',
      template: photoDetailTemplate,
      controller: 'photoDetailController',
      controllerAs: '$ctrl',
      resolve: { authenticate: authenticate }
    });
}
```

Quellcode 13: Routing Konfiguration

Die obige Konfiguration definiert folgendes Verhalten:

Die Adresseingabe von <https://pika.cloud/photos?page=7> führt zum Instanziieren des *PhotosController*. Dieser rendert die Seite 7 der Fotogalerie in den *ui-view*. Ein Klick auf den Link hinter der Miniansicht eines Bildes ruft die URL <https://pika.cloud/photo-details/p13tr3> auf und sorgt entsprechend dafür, dass der *PhotoDetailController* die Großansicht des Fotos mit der ID *p13tr3* rendert.

5.5 Projekt Struktur

Die Analyse der Aufgabestellung für diese Anwendung im Kapitel Design ergab die Hauptdomänenbereiche - Authentifizierung und Session Handling (*session*), Menünavigation (*side-menu*), Fotogalerie und Suche (*photos*), Detail-Fotoansicht und Metadateneingabe (*photo-detail*). Entsprechend dieser Domänenbereiche wird die Modularisierung vorgenommen.

5.5.1 Verzeichnis Struktur

In zahlreichen Einstiegsquellen für Organisation von JavaScript Projekten findet man eine Bündelung nach Funktionsverantwortung der Komponenten im Angular Framework vor. So werden etwa Controller, Direktiven, Factories, Services in einzelnen Unterverzeichnissen organisiert. Dieser Aufbau hat den Nachteil, dass nicht sofort ersichtlich wird, wo sich der Applikationscode einer bestimmten Domain befindet, da er nun auf verschiedenen Orte verteilt ist.

Pika verwendet daher eine Gliederung der Verzeichnisse nach ihrer Domain. Diese Praxis wird unter JavaScript Entwicklern favorisiert¹ und etwa in [Kukic \(2014\)](#) detailliert dargestellt. Es ergibt sich daher die Verzeichnis Struktur aus Quellcode 14.

¹<https://stackoverflow.com/questions/18542353/angularjs-folder-structure>

5 Implementierung

```
# Detail Fotoansicht und Metadateneingabe
|-- photo-detail
|   |-- controller.js
|   |-- form-component.js
|   |-- form.pug
|   |-- index.pug
|   |-- index.js
|   |-- index.scss
|   |-- toolbar-component.js
|   |-- toolbar.pug

# Foto Gallery und Suche
|-- photos
|   |-- client.js
|   |-- controller.js
|   |-- gallery.js
|   |-- index.pug
|   |-- index.js
|   |-- index.scss
|   |-- toolbar-component.js
|   |-- toolbar.pug

# Authentifizierung und Session Handling
|-- session
|   |-- controller.js
|   |-- index.js
|   |-- session.js

# Menünavigation
|-- side-menu
|   |-- index.pug
|   |-- index.js
|   `-- index.scss
|-- http-interceptor.js

# Übergreifender Code
|-- index.js
|-- index.scss
|-- routes.js
|-- theming.js
```

Quellcode 14: Directory Structure

5.5.2 Modul Struktur

Entsprechend der Domain-Aufteilung lässt sich also jeder einzelne Bereich als ein einzelnes Angular Modul abbilden. Hierbei exportiert jedes einzelne Modul seine eigene Angular API

5 Implementierung

Deklaration (siehe Quellcode 15). Diese Exports werden schließlich im dem *index.js* File an das *Pika* Hauptmodul übergeben (siehe Quellcode 16).

```
//photos/index.js
import angular from 'angular';

import client from './client';
import gallery from './gallery';
import controller from './controller';
import toolbarComponent from './toolbar-component.js';

export default angular.module('pika.photos', [
  client.name,
  gallery.name,
  controller.name,
  toolbarComponent.name
]);
```

Quellcode 15: Modul Export

```
//index.js
import session from './session';
import sideMenu from './side-menu';
import photos from './photos';
import photoDetail from './photo-detail';

angular.module('pika', [
  'ui.router',
  'ngMaterial',
  'ngFileUpload',
  sideMenu.name,
  session.name,
  photos.name,
  photoDetail.name
])
```

Quellcode 16: Modul Zusammenbau

5.5.3 Komponenten Struktur

Kapitel 3.4.3 erläutert das Paradigma des clientseitigen Javascript-MVC-Frameworks. Es liegt darin, einzelne Komponenten für Teilbereiche des HTML DOMs zu erstellen. Eine Komponente nimmt alle Benutzerinteraktionen in ihrem Teilbereich entgegen, speichert und rendert dessen Zustand.

5 Implementierung

Sie besteht aus mehreren weiteren Einheiten - einem View, einem Controller und einem bis zu mehreren Models. In dieser Arbeit nimmt der Controller, wie am Ende von Kapitel 3.4.3 geschildert, eine Presenter Rolle ein. D.h. er speichert nicht nur die Rohdaten für die Darstellung, sondern exakt den Zustand des Benutzerinterfaces.

Angewendet auf dem Prozess der Fotogalerie Darstellung bedeutet es, dass der entsprechende *PhotosController* die Daten exakt in der Form speichert, wie diese gerendert werden. Die Darstellung der Fotogalerie ist nach dem Monat der Aufnahme gruppiert. Die Rohdaten werden aber üblicherweise von einer Serverschnittselle als sortierte Liste zurückkommen. Es ist demnach die Aufgabe des Controllers in der Presenter Rolle, die Daten aus einer Liste gruppiert nach Aufnahmemonat abzulegen. Dafür kann er natürlich weitere Model Services hinzuziehen. Das View-Markup bleibt somit sehr schlicht. Dieses Vorgehen ergibt eine maximale Trennung von visueller Darstellung und Applikationslogik.

Komponenten in Angular können entweder durch URL Routing in einem Platzhalter ausgetauscht werden (siehe 5.4) oder direkt durch eine Angular Direktive als eine Art Custom HTML Tag instanziert werden, falls sich ihr Verhalten nicht auf die URL Änderung auswirken muss.

Für diese Arbeit ergibt sich folgende generelle Komponenten Gliederung:

- Die vier Hauptdomainbereiche - *session*, *side-menu*, *photos*, *photo-details* - bilden entsprechende Komponente und bestehen aus einem Hauptcontroller und einem zugehörigen Markup-Template in [Pug Syntax](#) (mit Ausnahme von *session*).
- Photos- und PhotoDetails Controller werden über URL Routing angesprochen
- Die Hauptbereiche besitzen weitere Komponenten Gliederungen, wie PhotoToolbar, PhotoDetails Toolbar, PhotoForm. Sie werden als Direktiven im Markup initialisiert.

5.6 Session Handling

5.6.1 Grundverfahren

Kapitel 2.3.1 definiert die Anforderung zur Benutzerauthentifizierung. Die Grundvorgehensweise dabei schildert sich wie folgt:

- Der Benutzer gibt seine Anmeldedaten beim initialen Aufruf der Anwendung ein.
- Die Anmeldedaten werden auf dem Server validiert.

- Nach positiver Prüfung erhält der Benutzer einen Session Token.
- Der Client speichert den Session Token permanent ab.
- Der Session Token wird mit jeder weiteren Anfrage an den Server gesendet.

Das obige Vorgehen liefert den Grundsatz einer Lösung auf das in Kapitel 3.1 geschilderte Problem der Zustandsbehaftung in dem zustandslosen HTTP Protokoll. Jede HTTP Anfrage erhält durch die obige Technik schließlich einen eindeutigen Session Token, über den der Server den Benutzer authentifizieren kann.

Für die konkrete Implementierung existiert eine Reihe von Verfahren. Die einfachste Variante nutzt den Standard Cookie Mechanismus von HTTP, um den Session Token zu speichern. Wenn ein solcher Cookie vom Server gesetzt wurde, ist kein weiterer Client Code erforderlich, um diesen mit nachfolgenden Anfragen zu senden. Folglich ist es auch die ursprüngliche und einzige Variante, um rein serverseitige Webanwendungen zu authentifizieren.

Die Cookie Authentifizierung bietet allerdings die meiste Angriffsfläche. Aus diesem Grund verwenden eigenimplementierte Webclients in aller Regel einen separaten Mechanismus. Der RFC 7519 Standard JWT taucht oft in Zusammenhang von Authentifizierung von Single Page Webapplikationen auf. Tatsächlich spezifiziert JWT nur den Aufbau- und Verifizierungsmechanismus der Session Token. Vgl. ([Peyrott 2018: Kap. 1](#))

Für den tatsächlichen Authentifizierungsablauf bietet es lediglich Richtlinien an. In einfacher Form benötigt der Client kein Wissen darüber, welche Art von Token Standard auf dem Server verwendet wurde. Vgl. ([Peyrott 2018: Kap. 2.2](#)). Daher zeigt diese Arbeit die Implementierung eines Authentifizierungsverfahrens, welches einer solchen Richtlinie aus dem JWT Standard folgt.

5.6.2 Authentifizierung

Den Aufruf der Root URL bearbeitet die `SessionController#index` Methode und rendert das Login Formular, wenn der Benutzer nicht eingeloggt ist (siehe Quellcode [17](#)).

Die `SessionController#create` Methode nimmt die Anmelddaten aus dem Login Formular entgegen. Daraufhin sendet der Controller `POST '/api/session-token'` mit den Benutzereingaben an den Server. Sollte die Eingabe falsch sein, wird die Fehlermeldung aus der Antwort in einer Instanzvariable des Controllers abgelegt und per [Databinding](#) im Formular gerendert. Die gültige Antwort des Servers enthält ein JSON Objekt mit den Daten des gegenwärtigen Users inklusive des JWT Session Tokens.

5 Implementierung

Diese Daten müssen nun permanent abgelegt werden. Diese Aufgaben übernimmt das *Session* Objekt. Der *SessionController* leitet die Daten des gegenwärtigen Users an die *Session#authorize* Methode weiter. Darauf legt das Session Objekt diese über den [LocalStorage](#) Mechanismus des Browsers ab.

Beim erneuten Instanziieren des Session Objekts liest dessen Konstruktor das Token aus dem LocalStorage. Wird der *SessionController#index* erneut aufgerufen, prüft der *SessionController* über *Session#isAuthenticated*, ob ein *User* Objekt bekannt ist, und übergibt das Rendering an den *PhotosController*. Dies wird über die Router Schnittstelle *this.\$state.go('photos')* erreicht. Der Befehl verändert die Adresszeile des Browsers auf die unter dem *photos* Schlüssel konfigurierte URL. Ebenfalls unter diesem Schlüssel in der Routerkonfiguration wird festgelegt, dass der *PhotosController* für die Navigation auf die obige URL zuständig ist. Angular sorgt dafür, dass der entsprechende Controller instanziert wird und das Rendering übernimmt. Der Benutzer ist somit eingeloggt und sieht die Fotogalerie.

5 Implementierung

```
//session/controller.js
class SessionController {
  /** @ngInject */
  constructor($state, $http, session) {
    //...
  }

  index() {
    if (this.session.isAuthenticated()) {
      return this.$state.go('photos')
    }
  }

  create() {
    const _this = this;
    this.$http.post('/api/session-token', this.user)
      .then((response) => {
        _this.session.authenticate(response.data);
        _this.$state.go('photos');
      });
  }
}

export default angular.module('session.controller', [])
  .controller('sessionController', SessionController)

//session/session.js
class Session {

  constructor() {
    this.user = JSON.parse(localStorage.getItem('user')) || {};
  }

  authenticate(user) {
    this.user = user;
    localStorage.setItem('user', JSON.stringify(user));
  }

  isAuthenticated() {
    return !!this.user.token;
  }
}

export default angular.module('session.session', [])
  .service('session', Session)
```

Quellcode 17: Session Handling

5.6.3 Autorisierung

Nachdem die erfolgreiche Authentifizierung des Users abgeschlossen ist, müssen einzelne Bereiche der Applikation nur für eingeloggte Nutzer autorisiert werden. Hierfür muss das Session Objekt für andere Komponenten im System verfügbar gemacht werden.

Solche Abhängigkeiten werden in AngularJS über das [Dependency Injection](#) Mechanismus aufgelöst. Der DI-Container kümmert sich um die Instanziierung aller registrierten Objekte. Es kann hier festgelegt werden, ob eine Abhängigkeit nur ein Mal pro Modul als [Angular Service](#) verfügbar ist oder jedes Mal neu mit Hilfe einer [Angular Factory](#) instanziert wird, wenn sie gebraucht wird. Die [Session Klasse](#) wird als Service registriert, da dessen Instanz mit den Userdaten über alle anfordernden Komponenten einmalig verfügbar sein muss.

Eine Stelle, wo der authentifizierte Zustand geprüft wird, ist der Router. Alle Routes im System müssen den Benutzer zurück auf das Login Formular weiterleiten, wenn dieser nicht eingeloggt ist (siehe Quellcode 18).

```
//routes.js
function routesConfig($stateProvider, $urlRouterProvider) {
  $stateProvider
  // ...
  .state('photos', {
    url: '/photos/:page?search',
    template: photosTemplate,
    controller: 'photosController',
    controllerAs: '$ctrl',
    resolve: { authenticate: authenticate }
  })
  // ...
}

function authenticate($q, $state, $timeout, session) {
  if (session.isAuthenticated()) {
    return $q.when();
  }

  $timeout(function() {
    $state.go('session');
  })

  return $q.reject();
}
```

Quellcode 18: Route Autorisierung

Über das *resolve* Keyword lässt sich in der Router Konfiguration eine *authenticate* Funktion

5 Implementierung

anhängen. Diese wird immer aufgerufen, wenn eine Routeänderung auf die entsprechende Route passiert. Der Rückgabewert dieser Funktion - ein erfolgreich/nicht erfolgreich aufgelöstes [Promise](#) Objekt - bestimmt darüber, ob die Routeänderung wirklich durchgeführt werden darf.

Diese Entscheidung kann die *authenticate* Funktion einfach treffen, indem es die API des injizierten Session Objektes *session.isAuthenticated()* in Anspruch nimmt.

Das oben beschriebene Verfahren der Route Autorisierung ist offensichtlich keine Sicherheitsmaßnahme, da das Ganze in dem Client Code passiert. Es dient, wie angedeutet, lediglich der notwendigen User Experience - der User soll immer auf das Login Formular umgeleitet werden, falls nicht richtig autorisiert.

Wie in Kapitel [5.6.1](#) geschildert, muss ein Server jede HTTP Anfrage auf das valide Session Token prüfen, da diese auch ohne den originalen Webclients von einem potentiellen Angreifer durchgeführt werden können. Das Session Objekt muss demnach bei allen HTTP Aufrufen bekannt sein.

Angular bringt einen HTTP Client mit, welcher lediglich einen bequemeren Wrapper über die native Browser *fetch* API darstellt. Über einen sog. *HTTP Interceptor* lässt sich das Setzen des nötigen Autorisierungshaders einmalig konfigurieren (siehe Quellcode [19](#)).

```
//http-interceptor.js
export default function (session) {
  return {
    request: (config) => {
      if (session.isAuthenticated()) {
        config.headers.Authorization = `Bearer ${session.user.token}`;
      }
      return config;
    }
  };
}
```

Quellcode 19: HTTP Interceptor

Die Abhängigkeit zu dem HTTP Client lässt sich über den von Angular mitgelieferten DI-Container auflösen. Nach der korrekten Registrierung des HTTP Interceptor muss bei der Benutzung des Clients nichts weiter beachtet werden (Siehe Quellcode [20](#)).

```
//photos/client.js
class PhotosClient {
  constructor($http) {
    this.$http = $http;
  }

  // ...

  findById(id) {
    return this.$http.get(`/api/photos/${id}`);
  }

  // ...
}
```

Quellcode 20: HTTP Client Verwendung

5.7 Menüführung

In Kapitel 4.3 wurde ein Seitenmenü für die Hauptnavigation der Applikation und jeweils ein Kontextmenü in der Tab-Leiste für die Unterbereiche, wie die Galerie- oder Photo-Detail-Ansicht, entworfen. Das Hauptseitenmenü bleibt während des gesamten Nutzererlebnisses unverändert, während die Kontextmenüs sich je nach Kontext ändern können.

In Quellcode 12 wurde der Lösungsansatz für die Menüführung bereits deutlich. Die Applikation benötigt ein Hauptlayout bzw. ein übergreifendes Frame. Dieses Frame enthält das Seitenmenü, dessen Komponente über die [AngularJS Direktive side-menu](#) im Markup initialisiert wird. Neben dem Seitenmenü ist ein Platzhalter für die Unterbereiche, dessen Inhalt von dem Router bestimmt wird. Die verschiedenen Kontext Untermenüs hingegen werden von der jeweiligen Hauptkomponente als Direktive in Markup eingebunden. Dies hat den Grund, dass der Kontext sich mit dem Wechsel in einen Unterbereich unterscheidet. Die Fotogalerie benötigt beispielsweise ein Suchfeld, während die Detailansicht unter anderem ein Button für das Anzeigen der Metadaten enthält.

Die Implementierung des Seitenmenüs gestaltet sich recht trivial. Der SideMenuController (siehe Quellcode 21) initialisiert Menüeinträge. Es sind lediglich Einträge mit Namen und URL-Route zu dem Unterbereich. Der Controller muss somit nicht auf die Navigation selbst reagieren, da diese von dem Router übernommen wird.

5 Implementierung

```
import angular from 'angular';
import template from './index.pug';

//side-menu/index.js
class SideMenuController {

    /** @ngInject */
    constructor($mdSidenav) {
        this.$mdSidenav = $mdSidenav;
        this.initMenu();
        this.initAdminMenu();
    }

    initMenu() {
        this.menu = [
            {
                link: '/photos',
                title: 'Photos',
                icon: 'image',
            },
            {
                link: '/settings',
                title: 'Settings',
                icon: 'settings',
            },
            //...
        ];
    }

    close() {
        this.$mdSidenav('left').close();
    }
}

export default angular.module('pika.side-menu', [])
.component('sideMenu', {
    template: template(),
    controller: SideMenuController,
    controllerAs: 'sideMenuController',
});
```

Quellcode 21: SideMenuController

```
//side-menu/index.pug
md-sidenav.md-sidenav-left()
// ...
md-content
  md-list(flex=' ')
    md-list-item(ng-repeat='item in sideMenuController.menu')
      md-icon.blue_grey_400.material-icons.step {{item.icon}}
      p a(href='item.link')
        {{item.title}}

  md-button(ng-click='sideMenuController.close()', class='md-primary')
    Close
```

Quellcode 22: Side Menu View

Das Verhalten des Seitenmenüs inklusive der Ein- und Ausklappen Animation bringt die Angular *mdSideNav* Komponente mit, welche schließlich im Markup erzeugt wird (siehe Quellcode 22). Paradoxerweise ist der SideMenuController nicht für das Ausklappen des Seitenmenüs zuständig. Diese Aufgabe übernehmen die Kontextmenüs. Dies hat den einfachen Grund, dass der Ausklappregler sich in dem jeweiligen Kontextmenü links und nicht in dem Seitenmenü selbst befindet. Der Button zum Ausklappen lässt sich ebenfalls in eine extra Komponente auslagern und in jede Kontext-Toolbar integrieren.

5.8 Foto Galerie

5.8.1 Laden

Die Fotogalerie ist ein Beispiel für die Kommunikation über mehrere Controller und Wiederverwendbarkeit von Models. Die Einzelansicht des Fotos ist ebenfalls ein Art Galerie, da hier ein Sliden zum nächsten und vorherigen Foto möglich ist. Beide Ansichten - Gallerie- und Einzelansicht - laden die Fotos stapelweise (siehe 5.8.2). Dieses gemeinsame Verhalten für das Laden von Fotos wird an ein separates Model, die *PhotosGallery* ausgelagert.

Das *PhotosGallery* Objekt wird zwischen den *PhotosController* und *PhotoDetailsController* wiederverwendet. Es speichert den aktuell geladenen Stapel an Fotometadaten sowie das aktuell ausgewählte Foto. Das *PhotosGallery* Objekt lädt einen entsprechenden Stapel an Fotos, wenn der Benutzer auf eine Galerieseite navigiert. Das Sliden zum nächsten Foto in der Einzelansicht geschieht innerhalb des geladenen Fotostapels. Will der Benutzer ein weiteres Foto betrachten als im Stapel verfügbar, geschieht das Nachladen über den selben Mechanismus des paginierten Ladens in der Fotogalerie.

5 Implementierung

Der *PhotosController* und der *PhotoDetailsController* beobachten den Zustand des *PhotosGallery* Objekts. Unabhängig davon, welcher der beiden Controller eine Zustandsänderung im *PhotosGallery* Objekt ausgelöst hat, bleibt die Darstellung immer synchronisiert. Der Benutzer kann somit mehrere Fotobündel weit in der Einzelansicht滑动. Wenn er am Ende wieder in die Galerieansicht zurückkehrt, wird das aktuelle Fotobündel mit letztem betrachteten Foto dargestellt.

Die Navigation zu *photos* URL rendert die Fotogalerie. Dies übernimmt *PhotosController#showPhotos*. Der Controller deklariert eine Abhängigkeit auf die *PhotosGallery*, welche wiederum über den *PhotosClient* den HTTP GET Aufruf an den Server durchführt.

Die Antwort ist ein Array in [JSON](#) Format mit Fotometadaten mit Titel, Beschreibung, diversen Kameraeinstellungen und der URL zu dem eigentlichen File. [JSON](#) wird in JavaScript durch einen Aufruf von *JSON.parse* direkt in ein Objekt umgewandelt. Der Angular HTTP Client macht es automatisch. Da die Darstellungslogik in den Controllern (Presentern) liegt, werden diese Rohdaten-Objekte direkt als Models verwendet, ohne extra Klassen dafür anzulegen. Die Fotometadaten werden in der Instanzvariable *photos* in der *PhotosGallery* abgelegt.

```
//photos/controller.js
class PhotosController {
  /** @ngInject */
  constructor($scope, $stateParams, $state, photosGallery) {
    // ...
    this.photosGallery = photosGallery;
    this.showPhotos();
  }

  showPhotos() {
    //...
    this.photosGallery.showPhotos({ search, page });
  }
}

//photos/gallery.js
class PhotosGallery {
  /** @ngInject */
  constructor(photosClient) {
    this.client = photosClient;
    this.photos = [];
    // ...
  }

  showPhotos({ search, page = 1 } = {}) {
    const _this = this;
    //...
    return this.client.find({ search, page }).then(res => {
      _this.photos = res.data.photos;
      //...
      return _this.photos;
    });
  }
}
```

Quellcode 23: Photo Galerie

5.8.2 Gruppieren

Nachdem die Fotos erfolgreich in dem *PhotosGallery* Service geladen wurden, müssen die Rohdaten für die eigentliche Darstellung verarbeitet werden. Dieses Kapitel zeigt die weitere Verarbeitung exemplarisch anhand automatisch nach dem Monatsdatum gruppiertener Darstellung der Fotos.

Diese Logik ist spezifisch für die Galerieansicht und wird im *PhotosController* angewendet. Die Gruppierung der Photos muss mit jeder Änderung des Arrays in der *PhotosGallery* passieren.

5 Implementierung

Der *PhotosController* trägt sich hierfür als Beobachter auf Änderungen des *photos* Arrays im *PhotosGallery* Service ein.

Nach einer Änderung des *photos* Arrays in der *PhotosGallery* führt der Controller eine Gruppierungsfunktion aus. Das Endresultat legt der Controller in einem eigenen assoziativen Objekt als separate Instanzvariable *photosByMonth* mit Monatsnamen als Schlüssel und Fotos für den jeweiligen Monat als Array ab. Der View rendert die Monatsgruppen aus der neuen Instanzvariable in der natürlichen vorsortierten Schlüsselreihenfolge und die einzelnen Fotos der Gruppen in der Reihenfolge der Monatsgruppen-Arrays.

```
//photos/controller.js
class PhotosController {
    /** @ngInject */
    constructor($scope, $stateParams, $state, photosGallery) {
        // ...
        this.photosGallery = photosGallery;
        this.photos = {};

        this.showPhotos();
        this.initWatchers();
    }

    initWatchers() {
        this.$scope.$watch(
            () => { return this.photosGallery.photos },
            this.groupPhotosByMonth.bind(this)
        );
    }

    /* group the photos array to a key value object of month name to photos
     e.g. { 'Jan 2021' => [{id: 1, ...}, {id: 2, ...}, ... ] }
     */
    groupPhotosByMonth(photos) {
        const res = {};
        photos.forEach((photo) => {
            const month = this.monthLabel(photo);
            if (!res[month]) {
                res[month] = [];
            }
            res[month].push(photo);
        });
        this.photosByMonth = res;
    }
}
```

Quellcode 24: Foto Gruppierung

5.8.3 Paginieren

Für das Laden von Fotos in Stapeln wird in dem Benutzerinterface eine simple Paginierung verwendet. Es werden Links auf zwei vorherige und zwei folgende Seiten mit jeweils 50 Bildern angezeigt. E.g. [5, 6, 7, 8, 9].

Das Laden des nächsten Fotostapels passiert durch das Klicken des entsprechenden Seitenlinks und wird in der Browser-Adresszeile abgebildet. Es kann aber auch durch das Sliden zum nächsten bzw. vorherigen Foto in der Detailansicht passieren. Somit ist wiederum der *PhotosGallery* Service für die Paginierung verantwortlich.

Der *PhotosClient*, über den letztendlich die Daten angefragt werden, akzeptiert einen *page* Parameter, welchen er per HTTP Query-Parameter wiederum an den Server senden kann. Basierend auf der aktuell angefragten Seite, generiert der *PhotosGallery* Service ein Array aus Seitenzahlen in einer Instanzvariable nach jedem Laden eines Fotostapels.

Die Paginierung geschieht folgenderweise:

- Der Benutzer lädt die */photos* URL ohne *page* Parameter
- Der *PhotosGallery* Service lädt den Initialbatch 1 mit 50 Fotos
- Der *PhotosGallery* Service generiert den ersten Bereich aus Seitenlinks [1,2,3,4,5]
- Der *PhotosController* rendert die Fotos und die Seitenlinks
- Der Benutzer klickt auf einen der Links, z.B. 5
- Der *PhotosController* nimmt die Aktion mit dem Seitenparameter entgegen, sorgt darauf für eine Routeänderung, welche die Seitenzahl enthält, z.B. *photos5* (Dieser Schritt ist notwendig, damit sich die Seitenzahl ebenfalls in der Adresszeile widerspiegelt und damit auch über die Adresszeile navigiert werden kann).
- Der *PhotosController* wird durch den Router neu initialisiert, liest den *page* Parameter aus dem Router, fordert *PhotosGallery* auf, den neuen Stapel zu laden
- Der *PhotosGallery* Service fragt den entsprechenden Stapel vom *PhotosClient* an, legt diesen in einer Instanzvariable ab, generiert neue Paginierungslinks

5 Implementierung

```
//photos/controller.js
class PhotosController {
  constructor(/**/) {
    //...
    this.showPhotos();
  }

  showPhotos() {
    const page = parseInt(this.$stateParams.page, 10) || undefined
    const search = this.$stateParams.search

    this.photosGallery.showPhotos({ page, search });
  }

  // triggered on page link click
  showPage(page) {
    this.$state.go(
      'photos',
      { page, search: this.photosGallery.search },
      { location: 'replace' }
    );
  }
}

//photos/gallery.js
class PhotosGallery {
  showPhotos({ search, page = 1 } = {}) {
    const _this = this;
    return this.client.find({ search, page }).then(res => {
      _this.photos = res.data.photos;
      _this.totalSize = res.data.totalSize;
      _this.currentPage = page;
      _this.search = search;

      _this.paginate();
      return _this.photos;
    });
  }

  paginate() {
    let start = this.currentPage - 2;
    let end = start + 5;
    // ... handle edge cases ...
    this.pages = _.range(start, end);
  }
}
```

Quellcode 25: Paginierung

5.8.4 Rendering

Das Markup der Fotogalerie ist ein logikfreies Rendering der oben erläuterten Datenstrukturen. Pro Monat der Aufnahmen wird eine GridList mit den Fotokacheln gerendert und unten auf der Seite eine Zeile mit den Paginierunglinks angehängt. Pro Foto wird eine Grid-Kachel *md-grid-tile* erstellt. Das Foto selbst ist ein CSS *background-image* Attribut der Kachel. Die Kachel erhält ebenfalls eine *ng-click* Direktive mit entsprechender Aktion und Foto-ID, um bei einem Klick auf das Foto auf die Detailansicht zu gelangen.

```
//photos/index.pug
photos-toolbar

md-content(...)
  div(ng-repeat='(monthLabel, photos) in $ctrl.photosByMonth')

    md-content(...) {{monthLabel}}

    md-grid-list(...)
      md-grid-tile(
        ng-repeat='photo in photos'
        ng-style="{ 'background-image': 'url(' + photo.src + ')' }"
        ng-click="$ctrl.showPhoto(photo._id)"
      )

    div(class='pagination', ...)
      md-button(
        ng-repeat='page in $ctrl.photosGallery.pages'
        ng-click='$ctrl.showPage(page)'
      ) {{page}}
```

Quellcode 26: Foto Gallerie Markup

5.9 Foto Suche

Die Fotosuche verwendet für die Darstellung den exakt gleichen Code aus Kapitel 5.8. Die einzige Erweiterung, welche für die Suchefunktion notwendig ist, ist eine Weitergabe eines Suchbegriffes an den *PhotosGallery* Service und wiederum an den HTTP Aufruf im *PhotosClient*.

Das Suchfeld wird in die galeriespezifische Toolbar platziert. Nach der Eingabe wird eine Routenänderung über den Angular Router erzwungen. Die Route führt wie in dem Standardfall auf den *PhotosController*, enthält jedoch einen Suchbegriff im Route-Parameter, z.B. */photos?search=hamburg*.

Der *PhotosController* liest beim Laden der Fotos, wie in Quellcode 25 gezeigt, durch die *showPhotos* Methode zusätzlich den *search* Parameter aus. Dieser wird dann an *PhotosGellary.showPhotos* weitergereicht.

Letzendlich ist die Aufgabe des Servers entsprechend ein gefiltertes Ergebnis zurückzugeben. Der Suchbegriff muss zusätzlich zwischengespeichert werden, damit die Paginierung den Suchfilter beachtet.

5.10 Foto Details

In Kapitel 5.8.4 wird erwähnt, dass jede Fotokachel eine Klickaktion zugewiesen bekommt, welche zur der Detailansicht eines Fotos führt. Diese Aktion wird vom *PhotosController* verarbeitet und ist eine triviale Routeänderung auf den *PhotoDetailController* mit entsprechenden Photo-ID-Parameter in der URL. Die Routeänderung sorgt dafür, dass der *PhotoDetailController* neu instanziert wird. Bei jedem Instanziieren lädt der *PhotoDetailController* das entsprechende Foto aus dem Photo-ID-Parameter. Der *PhotoDetailController* verwendet ebenfalls wie der *PhotosController* den *PhotosGallery* Service, um die Daten für das einzelne Foto zu laden.

Im Standardfall hat der *PhotosGallery* Service die Daten für das einzelne Foto bereits in dem *photos*-Array. Diese befinden sich dort aus der vorherigen Anfrage des *PhotosController* für die Darstellung der Fotoübersicht. Da ein Fotostapel eine überschaubare Größe hat, reicht hier eine lineare Suche nach dem Eintrag mit den Fotos mit entsprechender ID. Das gefundene Fotoobjekt inklusive der Fotometadaten wird in der Instanzvariable *currentPhoto* abgelegt und per Databinding gerendert.

Die Anzeige der Metadaten ist hinter einer weiteren, von rechts ausklappbaren Sidebar, (siehe 4.7) versteckt. Editierbare Werte wie der Titel und die Beschreibung werden in Formularfelder dargestellt. Ein bidirektionales Databinding kopiert die Benutzereingaben in die entsprechende Felder des *currentPhoto*-Objekts.

Der *PhotoDetailController* registriert sich auf die *onChanged* -Events der Formularfelder und sendet nach einem kurzem Timeout die neuen bekannten Daten an den Server mit Hilfe des *PhotoClient* Objektes. Somit ergibt sich ein sog. „In Place Editing“-Benutzererlebnis.

5.11 Foto Slider

Die Foto-Detailansicht bietet gleichzeitig das Sliden zur Detailansicht des nächsten Fotos an. Grundsätzlich wird das Sliden zum nächsten bzw. vorherigen Foto durch ein Inkrementieren bzw. Dekrementieren eines Indexes implementiert. Hierfür speichert das *PhotosGallery*-Objekt den aktuellen Index (*currentIndex*) des angezeigten Fotos im *photos* Stapel.

Die Slide-Aktion zum nächsten bzw. vorherigen Foto ruft die *PhotosGallery*-Methode *next* bzw. *prev* auf. Dies ist wiederum ein Aufruf von *slide* mit einem positiven bzw. negativen Wert. Dabei wird *currentIndex* um den Wert verändert und der berechnete *nextIndex* mittels *setIndex* gesetzt. *setIndex* speichert gleichzeitig die Referenz auf die Fotometadaten unter dem Index in der Variable *currentPhoto*. Wenn sich der Wert von *currentPhoto* ändert, sorgt Databinding für ein Rendering des entsprechenden Fotos (siehe Quellcode 27).

Ferner findet in *slide* eine Ausnahmebehandlung statt, wenn der Index außerhalb der Grenzen des zwischengespeicherten Stapels an Fotos wandert. Es wird der nächste bzw. vorherige Stapel an Fotos geladen. Hierfür wird die *PhotosGallery#showPhotos* Methode aus Quellcode 25 wiederverwendet. *showPhotos* wird nun mit dem entsprechenden Parameter *page* der nächsten bzw. vorherigen Seite aufgerufen. Das Verfahren aus Quellcode 25 wird nun für den weiteren Stapel an Foto Metadaten wiederholt.

Der Pseudocode in Quellcode 27 veranschaulicht das Verfahren unter Verzicht auf die Details der asynchronen JavaScript Programmierung.

5 Implementierung

```
//photos/gallery.js
class PhotosGallery {
    showPhotos({ search, page = 1 } = {}) {
        /* ... */
    }

    // ...

    next() {
        return this.slide(1);
    }

    prev() {
        return this.slide(-1);
    }

    slide(step = 1) {
        const nextIndex = this.currentIndex + step;
        const nextPage = this.currentPage + Math.sign(step);

        if (this.isValid(nextIndex)) {
            this.setIndex(nextIndex);
            return;
        }

        if (nextPage <= 0) {
            return;
        }

        this.showPhotos({ page: nextPage });
        // ...
        // reset index to 0 (for next) or photos.length - 1 for (prev)
    }

    isValid(index) {
        return index >= 0 && index < this.photos.length;
    }

    setIndex(index) {
        this.currentPhoto = this.photos[index];
        this.currentIndex = index;
        // ...
    }

    // ...
}
```

Quellcode 27: Sliding

6 Fazit und Ausblick

6.1 Fazit

Die Arbeit schaute auf die historische Entwicklung des World Wide Webs vom klassischen Client-Server-Modell bis zur Entstehung immer mächtigerer Mechanismen für Webanwendungen, die eines reichhaltigeren Benutzererlebnises auf den Endgeräten bedürfen.

Für einen solchen potentiell an Benutzerinteraktionen reichen, nativ ähnlichen Client einer Single Page Webapplikation in Form einer Fotoverwaltungssoftware wurden charakteristische Anforderungen aufgestellt. Anhand der Analyse und Implementierung dieser Anforderungen zeigten sich wesentliche Problematiken des Software Designs eines solchen Clients im Detail.

Es wurde gezeigt, wie eine solche Software on-the-fly an den Benutzer durch normale Webmechanismen ohne extra Installation ausgeliefert werden kann. Ohne zur Implementierungszeit etwas über das Endgerät des Benutzers zu wissen, kann sich die realisierte Fotoverwaltungsapplikation an das tatsächliche Endgerät mit Hilfe von Responsive Webdesign anpassen.

Im Kontrast zur der oft üblichen Praktik der Anreicherung des Benutzererlebnisses mit Hilfe von einzelnen Skripten, wurde der Fokus der Arbeit darauf gelegt, die gewachsene Komplexität im Client durch saubere Trennung von Verantwortlichkeiten zu handhaben. Dabei entstanden einzelne Software-Komponente aus mehreren Entitäten für die jeweiligen Domain-Bereiche der Applikation. Zum einen wurde verdeutlicht, dass sich nun ein Bedarf für eine Kommunikation und Wiederverwendung der Logik zwischen einzelnen dieser Komponenten ergibt, und zum anderen wurde gezeigt, wie ein Anwendungszustand zwischen diesen Komponenten verwaltet werden kann.

6.2 Ausblick

Die Anforderung an Client-Server-Kommunikation wurde im Ansatz betrachtet. In großen Webanwendungen mit verschiedenen Endgeräten ergibt sich oft ein Bedarf an eine weit komplexere

6 Fazit und Ausblick

Datenschnittstelle mit Optionsparametern und Beziehungen zwischen einzelnen Datenstrukturen.

Für die Autorisierung und Authentifizierung wurde eine minimal hinreichende Lösung durch JWT Tokens aufgezeigt, welche zwar über den Browser-Standard-Cookie-Mechanismus hinausgeht, jedoch bei weitem nicht die Breite des Themas von Sicherheit in Webanwendungen erfasst. Zum Beispiel ergibt sich eine Fragestellung, wie Sicherheitstoken valide sein sollen und wie sie invalidiert werden. Ferner sind Webanwendungen meistens Mehrbenutzersysteme und bedürfen komplexerer Benutzer- und Rechtelogik.

Ein Thema, welches außer Acht gelassen wurde, ist die Testbarkeit von reichhaltigen Clients. In einer Single Page Webapplikation ergeben sich ebenfalls Besonderheiten im Gegensatz zur nativen Offline-Software wegen der Browserumgebung und des Simulierens der Client-Server-Kommunikation unter Test.

Webanwendungen haben den Vorteil einer on-the-fly Auslieferung und erlauben dem Benutzer, eine komplexe Software ohne eine lokale Installation zu benutzen. Sie haben allerdings Grenzen. Eine weitere Fragestellung ist, wie weit diese Grenzen reichen und welche Szenarien auch mit reichhaltigen Webanwendungen schwer bis gar nicht realisierbar sind.

Glossar

Angular Factory Komponenten, welche Services erzeugen. <https://docs.angularjs.org/guide/providers#factory-recipe>. 49

Angular Service Ersetzbare Objekte, welche zusammengebunden werden um Code zu organisieren und zwischen Modulen zu teilen. <https://docs.angularjs.org/guide/services>. 49

AngularJS Direktive Marker auf dem DOM Element, welcher dem AngularJS Renderer mitteilt spezifisches Verhalten an das DOM Element zu binden. <https://docs.angularjs.org/guide/directive>. 36, 39, 51

Databinding Eine Technik, welche die Benutzeroberfläche der App mit den dort angezeigten Daten verbindet und synchronisiert . 21, 46

Dependency Injection Ein Entwurfsmuster der objektorientierten Programmierung, bei dem Abhängigkeit zwischen Objekten erst zur Laufzeit hergestellt werden . 49

ES2016 Spezifische Version des ECMAScript standardisierten Sprachkerns von JavaScript. <http://www.ecma-international.org/ecma-262/7.0/>. 38, 39

JSON Javascript Object Notation. <http://json.org/> . 54

LocalStorage HTML5 Web Storage Schnittstelle. https://www.w3schools.com/html/html5_webstorage.asp . 47

Promise Eine Repräsentation einer eventuellen Ausführung einer asynchronen Operation . 50

Pug Javascript HTML Template Engine. <https://pugjs.org/> . 39, 45

Glossar

SASS Stylesheet-Sprache, die als CSS-Präprozessor, mit Variablen, Schleifen und vielen anderen Funktionen, die Cascading Style Sheets (CSS) nicht mitbringen, die Erstellung von CSS vereinfacht und die Pflege großer Stylesheets erleichtert . [38](#), [39](#)

Webpack Javascript module bundler. <https://webpack.js.org/>. [38](#)

Abbildungsverzeichnis

3.1 Client-Server-ModellSingSing. Quelle: Parikh u. a. (2015), Kap. Background	13
3.2 Serverseitiges MVC	18
3.3 Clientseitiges MVC	21
3.4 Webdesign Raster. Quelle: Marcotte (2011), S.27	22
3.5 Flexibles Webdesign Raster. Quelle: Marcotte (2011), S.33	23
4.1 Farbpalette	26
4.2 Authentifizierung	27
4.3 Foto Galerie Large	28
4.4 Foto Galerie Normal	29
4.5 Foto Galerie Small	30
4.6 Paginierung	31
4.7 Foto Details Large	32
4.8 Foto Details Bild Small	33
4.9 Foto Details Metadaten Small	34
5.1 Layout Container. Quelle: material.angularjs.org	37

Quellcode Verzeichnis

1	HTTP GET Request	14
2	HTTP GET Response	15
3	"Hello user"CGI script	16
4	Raster CSS	22
5	Flexibles Raster CSS	23
6	Container mit Bild Markup	23
7	Container mit Bild CSS	24
8	Bildskalierung	24
9	Bildausschnitt	24
10	index.html	38
11	webpack.config.js	39
12	Hauptlayout	40
13	Routing Konfiguration	41
14	Directory Structure	43
15	Modul Export	44
16	Modul Zusammenbau	44
17	Session Handling	48
18	Route Autorisierung	49
19	HTTP Interceptor	50
20	HTTP Client Verwendung	51
21	SideMenuController	52
22	Side Menu View	53
23	Photo Galerie	55
24	Foto Gruppierung	56
25	Paginierung	58
26	Foto Gallerie Markup	59
27	Sliding	62

Literaturverzeichnis

[Bekman und Cholet 2003] BEKMAN, Stas ; CHOLET, Eric: *Practical Mod_PERL*. Sebastopol, CA, USA : O'Reilly & Associates, Inc., 2003. – ISBN 0596002270

[Crockford 2008] CROCKFORD, Douglas: *JavaScript: The Good Parts*. O'Reilly Media, Inc., 2008. – ISBN 0596517742

[Dubost 2012] DUBOST, Karl: *HTTP - an Application-Level Protocol*. 2012. – URL <https://dev.opera.com/articles/http-basic-introduction/>. – letzter Zugriff: 13. 07. 2021

[Federico u.a. 2021] FEDERICO ; CULLOCA u.a.: *How the Web works*. 2021. – URL https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/How_the_Web_works. – letzter Zugriff: 13. 07. 2021

[Fielding und Reschke 2014] FIELDING, Roy ; RESCHKE, Julian: *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. 5177 Brandin Court Fremont, CA 94538: IETF (Veranst.), 2014. – URL <https://tools.ietf.org/html/draft-ietf-httpbis-p2-semantics-26#section-4>. – letzter Zugriff: 13. 07. 2021

[Garrett 2005] GARRETT, Jesse J.: *Ajax: A New Approach to Web Applications*. 2005. – URL <https://web.archive.org/web/20181231042432/https://adaptivepath.org/ideas/ajax-new-approach-web-applications/>. – letzter Zugriff: 13. 07. 2021

[Kukic 2014] KUKIC, Ado: *AngularJS Best Practices: Directory Structure*. 2014. – URL <https://scotch.io/tutorials/angularjs-best-practices-directory-structure>. – letzter Zugriff: 13. 07. 2021

[Lahres und Rayman 2009] LAHRES, B. ; RAYMAN, G.: *Objektorientierte Programmierung: Das umfassende Handbuch*. Rheinwerk Verlag, 2009. – URL <https://books.google.de/books?id=ZusSnQAACAAJ>. – ISBN 9783836222280

Literaturverzeichnis

- [LaunchSchool 2016] LAUNCHSCHOOL: *Working with Web APIs*. Launch School, 2016. – URL https://launchschool.com/books/working_with_apis. – letzter Zugriff: 13. 07. 2021
- [Marcotte 2011] MARCOTTE, Ethan: *Responsive Web Design*. A Book Apart, 2011. – ISBN 9780984442577
- [MSDN 2009] MSDN: *The MVVM Pattern*. 2009. – URL <https://docs.microsoft.com/en-us/archive/msdn-magazine/2009/february/patterns-wpf-apps-with-the-model-view-viewmodel-design-pattern>. – letzter Zugriff: 13. 07. 2021
- [MSDN 2010] MSDN: *The Model-View-Presenter (MVP) Pattern*. 2010. – URL <https://msdn.microsoft.com/en-us/library/ff649571.aspx>. – letzter Zugriff: 13. 07. 2021
- [Parikh u. a. 2015] PARIKH, Aarti ; AGRAM, Albert u. a.: *Introduction to HTTP*. Launch School, 2015. – URL <https://launchschool.com/books/http>. – letzter Zugriff: 13. 11. 2021
- [Peyrott 2018] PEYROTT, Sebastian E.: *The JWT Handbook*. Auth0 Inc, 2018. – URL <https://auth0.com/resources/ebooks/jwt-handbook>. – letzter Zugriff: 13. 11. 2021
- [Schippers 2016] SCHIPPERS, Ben: App Fatigue. In: *TechCrunch* (2016). – URL <https://techcrunch.com/2016/02/03/app-fatigue/>. – letzter Zugriff: 13. 07. 2021

Anhang

Auf der beigelegten CD befindet sich der komplette Quellcode der entwickelten Single Page Webapplikation zur Einsicht.

Eine Kopie des Quellcodes ist zusätzlich auf <https://github.com/st-anastasia/pika> hochgeladen.

Ich versichere, die vorliegende Arbeit selbstständig ohne fremde Hilfe verfasst und keine anderen Quellen und Hilfsmittel als die angegebenen benutzt zu haben. Die aus anderen Werken wörtlich entnommenen Stellen oder dem Sinn nach entlehnten Passagen sind durch Quellenangaben eindeutig kenntlich gemacht.

Ort, Datum

Anastasia Stieb