Ausarbeitungen zum Seminar

Rich Internet Applications w/HTML and Javascript

Wintersemester 2016/2017

06. Februar 2017

Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik
Abt. Systemsoftware und verteilte Systeme

www.uni-oldenburg.de/svs

Inhaltsverzeichnis

Responsive Webdesign

Nina Schneider
Carl von Ossietzky Universität Oldenburg
Oldenburg, Germany
nina.schneider@uni-oldenburg.de

***Abstract*—Dies ist eine Ausarbeitung zum dem Thema „Responsive Webdesign in Zusammenhang mit Bootstrap“ von Nina Schneider. Diese Ausarbeitung vergleicht Ansätze der Layoutgestaltung bei Webseiten und stellt des Weiteren Bootstrap als ein mögliches CSS-Framework für die Gestaltung von responsive Websites zusammen mit Beispielen verschiedener Elemente von einer Webseite vor.**

***Index Terms*—Responsive Webdesign, Bootstrap, Navigation, Layoutgestaltung**

I. Was ist Responsive Webdesign?

Responsive Webdesign stellt eine aktuelle Technik zur Verfügung, mit der Webseiten so angelegt werden, dass sie sowohl auf dem Computer-Desktop, Smartphone oder Tablet ohne entsprechenden Qualitäts- oder Informationsverlust dargestellt werden. Würde es keine responsive Websites geben, müsste man auf kleineren Endgeräten sowohl seitlich als auch nach oben oder unten scrollen, um den Inhalt vollständig lesen zu können, außer es wurde extra eine Webseite für kleinere Geräte erstellt, welches einen hohen Aufwand bedeutet. Eine responsive Website ist nun also eine Website, die sich entsprechend der Bildschirmgröße anpasst und trotzdem alle Inhalte erhalten bleiben. Eine solche Website besteht aus einem fluiden Layoutraster, anpassungsfähigen Inhalten und Layoutumbrüchen durch Media Queries (Medienabfragen), welche eine Liste an Kriterien beinhaltet, die das Ausgabemedium erfüllen muss.

Der Begriff des Responsive Webdesign trat erstmals 2010 in einem Artikel von dem Designer und Entwickler Ethan Marcotte auf, der darin für ein neues Verständnis von Webdesign aufgrund der Zunahme von mobilen Endgeräten aufrief. Er stieß damit auf große Resonanz, da die Gestaltung solcher Webseiten ein bis dahin ungelöstes Problem darstellte, da der Anspruch an die Darstellung von Webseiten wuchs und weiterhin wächst - und das auf allen Endgeräten.

Die ersten Webseiten im World Wide Web waren bereits flexibel und anpassungsfähig, da dort lediglich Text stand und noch kein Design vorhanden war. Mit der Zeit wurden die Ansprüche an Webseiten immer höher, die Seiten wurden komplexer. Die Layouts stießen aufgrund der Vielfalt an Bildschirmgrößen an ihre Grenzen, welche bis heute noch größer geworden ist. Das Ziel war es also, Webseiten wieder so flexibel zu gestalten wie in den Anfängen des Word Wide Web, ohne dabei aber auf die Vorzüge des Designs verzichten zu müssen.

Beim responsive Webdesign ist es wichtig, zwischen einer tatsächlichen responsive Website und verschiedenen Versionen der Webseite für die verschiedene Bildschirmgrößen zu unterscheiden. Die Mobilversion einer Webseite beispielsweise ist eine extra neu erstellte Version der Webseite, die für kleine Bildschirme angepasst wurde, wenn die Original-Webseite eine Desktopversion ist. Hierbei sind die Inhalte der Mobilversion und der Desktopversion nicht zwingend gleich. Hingegen gibt es bei einer responsive Website einen bestimmten Inhalt der Website, der auf allen Geräten gleich ist und lediglich anders dargestellt wird. Der Vorteil von responsive Websites ist, dass sie auf allen Bildschirmgrößen optimal aussieht. Des Weiteren ist der Aufwand der Wartung deutlich geringer als bei entsprechenden Mobilversionen der Webseiten, da der Inhalt sich bei den unterschiedlichen Versionen nicht unterscheidet.

Ein Layout, welches nicht responsive ist, kann die Unterschiede zwischen der Displaygröße eines Smartphones und eines Desktops nicht abdecken. Daher ist es notwendig, dass das Layout flexibel und kontrollierbar ist.

Es gibt grundlegend vier verschiedene Arten des Layouts bei der Gestaltung einer Webseite.

A. *Fixed Layout*

Zum einen gibt es das fixed Layout. Dies ist ein Layout, das auf festen Pixelgrößen basiert, welche sich auch beim Verändern der Bildschirmgröße nicht verändert. Dieses Layout hat den Vorteil, dass es leicht zu planen und ebenfalls leicht in der technischen Umsetzung ist. Es hat aber den Nachteil, dass es nur auf einem Viewport gut sichtbar ist und vor allem auf kleineren Geräten der Inhalt nicht mehr vollständig oder nur durch Scrollen lesbar ist.

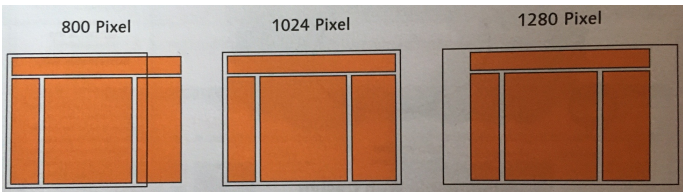

Abbildung 1 - Fixed Layout [1]

B. *Fluid Layout*

Zum anderen gibt es das fluide Layout, welches auf flexiblen Pixelgrößen basiert. Somit passt sich der Inhalt der entsprechenden Bildschirmgröße an und der ganze Platz wird ausgenutzt. Hierbei gibt es aber einige Einschränkungen in der Gestaltung, da der komplette Inhalt der Webseite flexibel sein muss. Das fluide Layout ist somit ein Layout, bei dem sich die Dimensionen des Layouts verändern, sobald sich die Bildschirmgröße ändert, wobei Bilder und Texte in ihrer ursprünglichen Größe erhalten bleiben.

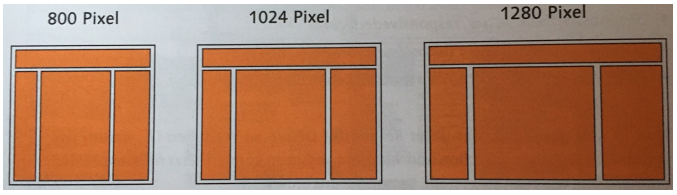

Abbildung 2 - Fluid Layout [1]

C. *Elastic Layout*

Und weiterhin gibt es das elastische Layout, wobei die Seitenelemente sowohl in der Höhe als auch in der Breite variabel sind. Hierbei skaliert das Design immer proportional zur Größe des Bildschirms und ist daher gut angepasst an alle unterschiedlichen Bildschirmgrößen. Der Nachteil von einem elastischen Layout ist allerdings, dass es in der Planung und technischen Umsetzung sehr komplex ist.

D. *Responsive Layout*

Das responsive Layout ist eine Kombination aus einem adaptiven und einem fluiden Layout. Hierbei ist das adaptive Layout ein Layout, welches in mehreren Versionen für die unterschiedlichen Bildschirmgrößen existiert, wobei diese Versionen jeweils fest sind. Dieses ist also schon recht nah an einem responsive Layout dran. Das Problem bei dem adaptiven Layout ist aber, dass es auf den Zwischengrößen nicht optimal aussieht.

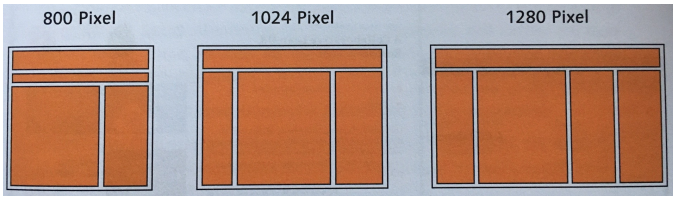

Abbildung 3 - Adaptive Layout [1]

Das responsive Layout besitzt nun also wie das adaptive Layout Werte, ab denen sich das Layout verändert, aber dazwischen verhält es sich dann wie das fluide Layout. Somit wird es auf allen Bildschirmgrößen optimal dargestellt.

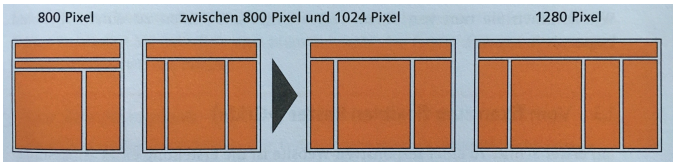

Abbildung 4 - Responsive Layout [1]

Für das Erstellen einer Webseite benötigt man zum einen HTML, um den Inhalt der Webseite zu erstellen, zum anderen CSS zum Erstellen des Layouts der Webseite und JavaScript, um das Verhalten der Webseite zu programmieren.

HTML steht für Hypertext Markup Language und ist eine textbasierte Auszeichnungssprache zur Darstellung von Texten, Grafiken und weiteren Inhalten. Ein HTML-Dokument kann ebenfalls Informationen enthalten, die nicht im Dokument angezeigt werden, wie beispielsweise den Verfasser des Dokuments. Die neueste Version ist HTML5, welche neue Elemente für beispielsweise Video und Audio bietet.

CSS steht für Cascading Style Sheets und ist eine Gestaltungssprache für elektronische Dokumente. Style Sheets steht hierbei für eine Sammlung von Formatvorlagen für HTML-Elemente. Man behandelt die Strukturierung und das Layout einer Webseite getrennt voneinander. Während HTML-Elemente also dazu dienen, den Inhalt zu strukturieren, wird CSS für das Stylen verwendet. Der HTML-Code liegt hierbei in einer anderen Datei als der CSS-Code. Dies hat den Vorteil, dass das Layout auf allen Seiten zugleich verändert werden kann, indem die CSS-Datei geändert wird. Die aktuelle Version ist CSS3.

Weiterhin ist JavaScript die clientseitige Skriptsprache des Webbrowsers. Mit JavaScript können HTML und CSS um Benutzerinteraktionen erweitert werden, um beispielsweise Inhalte auszuwerten und dynamische Webseiten zu erstellen.

Es wird also das HTML geschrieben und dann je nach den verschiedenen Parametern mit einem unterschiedlichem CSS ausgeliefert. Das Layout passt sich so den unterschiedlichen Umgebungen an. Diese verschiedenen Parameter werden mit Media Queries abgefragt. Media Queries sind Medienabfragen, welche jeweils aus einer Liste an Kriterien bestehen, die das entsprechende Ausgabemedium erfüllen muss, damit das Stylesheet eingebunden wird. Hierbei wird also unter anderem die entsprechende Größe des Bildschirms überprüft, aber auch die Auflösung und Farbfähigkeit.

Insgesamt wird das Fundament einer responsive Website mit HTML gelegt, welches für die Struktur verantwortlich ist und in dem beispielsweise Überschriften und Texte definiert werden. Für die Formatierung und das Layout der Website ist CSS verantwortlich. Hierbei werden zum Beispiel Farben oder Schriften definiert. Während HTML also für die Inhalte der Website verantwortlich ist, definiert CSS das Aussehen dieser Inhalte. Um dann aus einer statischen Website eine dynamische Website zu gestalten, ist zusätzlich noch JavaScript nötig.

Bei der Erstellung des HTML, CSS und JavaScript für Webseiten gibt es mehrere mögliche Herangehensweisen: Eine Möglichkeit ist „Desktop First“. Hierbei beginnt man mit dem mehrspaltigen Layout für große Bildschirme. Danach wird es auf kleinere Bildschirme reduziert. Eine zweite Möglichkeit ist „Mobile First“, wobei erst mit der kleinsten Variante begonnen wird und danach für große Bildschirme erweitert wird.

Für die Gestaltung und Erstellung von responsive Websites hat die Herangehensweise „Mobile First“ die Vorteile, indem beispielsweise eine platzsparende „Einklapp-Navigation“ eingebaut werden kann, die dann im Browser nebeneinander dargestellt wird.

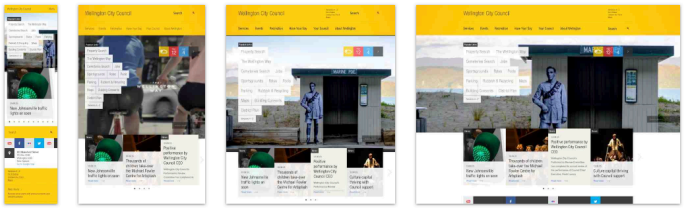

Abbildung 5 - Ein Beispiel für eine responsive Website [2]

II. BOOTSTRAP

A. *Was ist Bootstrap?*

Bootstrap ist ein fertiges CSS-Framework, welches die Erstellung einer responsive Website erleichtert. Ein Framework ist ein bereits fertiger Baukasten mit dynamischen Gestaltungsmöglichkeiten. Ein solches Framework enthält eine Sammlung von Hilfsmitteln für den einfachen Einsatz von CSS im Webdesign. Diese Frameworks bieten eine Kombination aus HTML5 und CSS3. Bootstrap verwendet außerdem die Stylesheet-Sprache Less, welche dafür da ist, Code-Wiederholungen zu vermeiden. Less bietet dafür noch weitere

Anweisungen als CSS, welche in Abschnitt E genauer erläutert werden. Dieser Less-Code wird dann zu CSS kompiliert.

Bootstrap ist 2011 als eine Lösung bei Twitter entstanden, um die internen Analyse- und Verwaltungswerkzeuge weiterzuentwickeln. Im August 2011 veröffentlichte Twitter die Ergebnisse und seitdem hat es sich zu einem populären Projekt entwickelt.

Mittlerweile gibt es die neueste Version Bootstrap 3, welche für die Gestaltung von responsive Websites verwendet wird. Hierbei ist es nicht mehr nötig, mobile Elemente später hinzuzufügen, da diese direkt mit eingebaut sind. Somit verfolgt Bootstrap den Mobile-First-Ansatz. Hierbei überlegt man sich zuerst, welche Inhalte wichtig sind und wie diese am besten auf kleinen Endgeräten dargestellt werden können.

Damit auf der Website die Inhalte auch auf mobilen Webseiten optimal angezeigt werden können, ist es wichtig, in den <*head*> der Datei die Zeile

<meta name="viewport" content="width=device-width, initial-scale=1">

einzubauen.

Weiterhin sind verschiedene Zeilen in der Datei nötig, um Bootstrap dann korrekt zu implementieren, beispielsweise die CSS-Datei, um den CSS-Teil von Bootstrap nutzen zu können. Eine einfache Datei sieht dann so aus:

```
<html>
    <head>
        <meta charset=„utf-8">
        <link rel="stylesheet" href="https://
        maxcdn.bootstrapcdn.com/
        bootstrap/3.3.5/
        css/bootstrap.min.css">
        <link rel="stylesheet" href="main.css">
        <link rel="stylesheet" href="https://
        maxcdn.bootstrapcdn.com/font-
        awesome/
        4.7.0/css/font-awesome.min.css">
    </head>
 <body>
    <h1>Hello World!</h1>
    <script src="https://ajax.googleapis.com/ajax/libs/
    jquery/1.11.3/jquery.min.js"></script>
    <script src="https://maxcdn.bootstrapcdn.com/
    bootstrap/3.3.5/css/bootstrap.min.css"></script>
 </body>
</html>
```

Wenn diese Dateien in die HMTL-Datei eingebunden wurden, kann man das Bootstrap-Framework vollständig verwenden.

B. Das Layout mit Bootstrap

Bootstrap verwendet ein sogenanntes Rastersystem für die Gestaltung des Layouts einer Webseite. Für die Gestaltung einer Webseite ist so ein Rasterlayout sehr hilfreich, da man die Seite dann besser strukturieren kann. Das Prinzip des Rasterlayouts beruht darauf, dass die gesamte Breite des Layouts auf gleich große Teile aufgeteilt wird. Bootstrap verwendet hierfür die Aufteilung in 12 Spalten, was bei einer Layoutbreite von 960 Pixeln einem einzelnem Raster jeweils einen Pixelwert von 80 zuordnet. Hinzu kommen aber noch entsprechende Abschnitte zwischen den einzelnen Rastern von 10 Pixeln, woraus sich ein Pixelwert von 60 für die einzelnen Raster ergibt.

Entsprechend kann man aber beispielsweise auch nur zwei Spalten benutzen, die dann jeweils aus 6 Rastern mit einer Breite von 60 Pixeln bestehen. Die verschiedenen Spalten für das Layout müssen aber nicht immer gleich groß sein. Es gibt ebenfalls die Möglichkeit, beispielsweise das Layout aus einer Spalte mit einer Breite von 4 Rastern mit einer Größe von je 60 Pixeln und der anderen Spalte dann entsprechend mit einer Breite von 8 Rastern mit einer Größe von 60 Pixeln zu erstellen.

Das folgende Beispiel zeigt, wie man ein Layout in Bootstrap realisieren kann [3]. Man legt als erstes fest, ob das Layout in einer festen Breite oder der ganzen Breite platziert werden soll. Die feste Breite wird mit der Klasse *.container* und die ganze Breite mit der Klasse *.container-fluid* realisiert. Der komplette Inhalt der HTML-Seite wird nun in ein <*div*>-Element verpackt, welches die entsprechende Klasse enthält. Man kann natürlich nicht nur eine Zeile mit verschiedenen Spalten versehen, sondern mehrere Zeilen und diese einzelnen Zeilen dann wiederum mit einer anderen Verteilung der Raster. Um diese Zeilen zu realisieren, arbeitet man innerhalb des Containers mit *.row*-Elementen. Die HTML-Elemente werden in die Zeilen der Spalten geschrieben, wobei für jede Zeile eine eigene Rasteraufteilung gewählt werden kann. Möchte man beispielsweise eine Zeile mit drei gleichgroßen Teilen des Layouts belegen und in jedes der drei Teile einen Artikel schreiben, wird das so realisiert:

```
<div class=“row“>
        <article class=“col-md-4“>...</article>
        <article class=“col-md-4“>...</article>
        <article class=“col-md-4“>...</article>
</div>
```

Das Layout passt sich nun auf allen Geräten aufgrund der Media Queries optimal an.

Zusätzlich gibt es in Bootstrap die Möglichkeit, das Layout für verschiedene Bildschirmgrößen anders darzustellen. In dem Beispiel wurde die Rasterklasse *.col-md-4* verwendet, was in dem Fall bedeutet, dass auf sogenannten mittelgroßen Geräten ein drei-spaltiges Layout angezeigt wird. Es gibt aber auch noch die Rasterklassen *.col-xs-n, .col-sm-n* und *.col-lg-n*. Das bedeutet, dass man sich aussuchen kann, bei welcher Bildschirmgröße man das entsprechende Layout anwenden möchte.

Bei dem obigen Beispiel wird das Layout bzw. die Webseite auf Bildschirmen mit einer Größe von 992 Pixeln oder größer nebeneinander dargestellt und bei einer Größe kleiner als 992 Pixeln untereinander dargestellt, welches die zwei folgenden Abbildungen zeigen:

Abbildung 6 - Das Layout auf einem Bildschirm mit mehr als 992 Pixeln

Abbildung 7 - Das Layout auf einem Bildschirm mit weniger als 992 Pixeln

Man kann dieses Layout dann noch erweitern, wenn man beispielsweise möchte, dass auch auf der nächstkleineren Bildschirmgröße die Spalten nebeneinander dargestellt werden und nicht untereinander. Man muss den Teil aus dem oberen Beispiel dann wie folgt erweitern:

```
<div class="row">
        <article class="col-sm-12 col-md-4">...</article>
        <article class="col-sm-6 col-md-4">...</article>
        <article class="col-sm-6 col-md-4">...</article>
</div>
```

Das bedeutet dann, dass auf einem Bildschirm mit einer Größe zwischen 768 Pixeln und 992 Pixeln der erste Artikel über die gesamte Breite dargestellt wird und die anderen beiden Artikel darunter in zwei Spalten. Bei einer Größe unter 768 Pixeln werden die Artikel wie in Abbildung 7 alle untereinander dargestellt. Und ab der Größe über 992 Pixeln werden alle drei Artikel wie in Abbildung 6 nebeneinander dargestellt. Diese oben erwähnte „Sonderbehandlung" der Zwischengröße sieht im Ergebnis dann wie folgt aus:

Abbildung 8 - Das neue Layout auf Bildschirmgröße zwischen 768 und 992 Pixeln

C. Bilder mit Bootstrap

Man kann Bilder in sein HTML-Dokument einbinden, diese sind dann aber nicht automatisch für kleinere Bildschirme angepasst. Damit die Bilder sich ebenfalls verkleinern und auch auf entsprechend kleinen Bildschirmen optimal aussehen, gibt es bei Bootstrap die Klasse *.img-responsive*. Die Bilder werden also beim Verkleinern des Bildschirms nicht einfach an den Seitenrändern abgeschnitten, sondern sie verkleinern sich mit, je kleiner der Bildschirm wird, was durch das responsive Layout von Bootstrap möglich ist. Es gibt somit keine Bildschirmgröße, auf der das Bild entweder zu klein oder zu groß dargestellt wird.

Abbildung 9 - Bilder auf einer großen Bildschirmgröße

Abbildung 10 - Bilder auf einer kleinen Bildschirmgröße

D. Navigation mit Bootstrap

Es gibt verschiedene Möglichkeiten für die Gestaltung einer Navigation auf einer Webseite mithilfe von Bootstrap. Die Navigationselemente werden alle mithilfe von Listen erstellt. Man kann eine einfache Navigation mit *Pills* oder *Tabs* realisieren, wobei man zum einen die Klasse *.nav* als auch *.nav-pills* bzw. *.nav-tabs* benötigt, was in Bootstrap standardmäßig dann so aussieht, wenn zusätzlich noch die sogenannte „aktive" Seite farblich unterlegt ist:

Abbildung 11 - Eine Navigation mit Tabs

Abbildung 12 - Eine Navigation mit Pills

Eine weitere Möglichkeit für eine Navigation ist die sogenannte *navbar*, welche sowohl Elemente der Navigation beinhalten kann, aber ebenso beispielsweise die Suchfunktion. Diese Navigationsleiste kann dann auf kleineren Bildschirmen wie Smartphones eingeklappt werden und wird auf breiten Bildschirmen wie gewohnt nebeneinander angezeigt.

Eine mögliche Realisierung kann sein, dass auf kleinen Bildschirmen ein Button angezeigt wird, der beim Anklicken das JavaScript *collapse* zum Ausklappen verwendet.

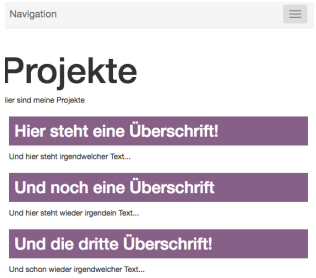

Abbildung 13 - Die eingeklappte Navigation auf einem kleinen Viewport

Wie erwähnt ist es möglich, noch weitere Komponenten auf der *navbar* zu platzieren. Mit der Klasse *.navbar-form* kann man Formulare zu der Navigationsleiste hinzufügen, wie zum Beispiel ein Suchfeld.

E. Bootstrap individualisieren

Man kann die vorgefertigte Bootstrap-Datei aus dem Internet laden oder aber man kann auch eine eigene Bootstrap-Version herunterladen. Bootstrap bietet an, dass man vor dem Herunterladen einzelne Komponenten, Variablen und Erweiterungen anpassen, aber auch weglassen kann, wenn man die für seine Webseite nicht benötigt. Zusätzlich dazu kann man ebenfalls ein benutzerdefiniertes Stylesheet erstellen, wobei man Angaben zu der Größe oder der Schriftart machen kann und verschiedene HTML-Elemente und Komponenten anpassen kann. Hierbei passt man die sogenannten Less-Variablen an seine eigenen Bedürfnisse an.

Less-Variablen sind häufig verwendete Werte, die dann nur einmal definiert werden müssen und für das ganze HTML-Dokument gelten. Less ist eine Stylesheet-Sprache. Sie wird dazu verwendet, zentrale Variablen für beispielsweise CSS festzulegen, welche man dann an mehreren Stellen wiederverwenden kann. Damit man einen solchen CSS-Prozessor verwenden kann, schreibt man entsprechenden Code in der Stylesheet-Sprache und mithilfe eines Präprozessors wird diese Less-Syntax dann in eine CSS-Datei umgewandelt und kann verwendet werden. Das Ziel von Less ist vor allem also, Code-Wiederholungen zu vermeiden. Man kann Bootstrap also mit dem bereits kompilierten CSS-Code nutzen oder mit den ursprünglichen Less-Dateien. Bootstrap verwendet die Less-Variablen. Der Vorteil solcher Variablen ist, dass man entsprechende Änderungen nur an einer zentralen Stelle ändern muss und nicht im gesamten Code. Mit Less hat man zusätzlich die Möglichkeit, sogenannte Mixins zu verwenden. Mit solchen Mixins kann man verschiedene Eigenschaften unter einem Bezeichner zusammenfassen, wobei es dann reicht, nur diesen Bezeichner aufzurufen, wenn man die restlichen Eigenschaften verwenden möchte. Ebenfalls ist es mit Less möglich, verschiedene Sektoren ineinander zu verschachteln, womit der Code verkürzt werden kann.

III. FAZIT

Insgesamt war die Erfindung des responsive Webdesign unumgänglich in Hinblick auf die steigende Vielfalt von Größen und Auflösung bei mobilen Endgeräten. Bootstrap als CSS-Framework macht es dem Webentwickler leicht, solche Webseiten zu entwickeln, wobei Bootstrap zudem noch so personalisiert werden kann, dass die Webseiten individuell gestaltet werden können und nicht alle das gleiche Design haben.

QUELLEN

[1] Ertel, Andrea; Laborenz, Kai. Responsive Webdesign. Anpassungsfähige Websites programmieren und gestalten. 1. Auflage. Galileo Press. Bonn 2014.

[2] http://mediaqueri.es/

[3] Wolf, Jürgen. HTML5 und CSS3. Das umfassende Handbuch. 1. Auflage. Rheinwerk Verlag GmbH. Bonn 2015.

[4] Müller, Peter. Flexible Boxes. Eine Einführung in moderne Websites. 2. Auflage. Rheinwerk Verlag. Bonn 2015.

[5] https://wiki.selfhtml.org/wiki/CSS/Media_Queries

[6] https://de.wikipedia.org/wiki/Bootstrap_(Framework)

[7] http://holdirbootstrap.de/

[8] https://blog.kulturbanause.de/2012/10/die-layout-typen-einer-website-fixed-fluid-elastic/

Angular 2

Linus Barth
Carl von Ossietzky Universität von Oldenburg, Deutschland
Department für Informatik
linus.barth@uni-oldenburg.de

***Abstract*—Dieses Dokument soll Problematiken der Webentwicklung aufzeigen und eine mögliche Lösung dieser mithilfe des "Angular 2"-Frameworks darstellen. Die Webseiten werden aufwändiger und bedürfen Hilfsmittel bei der Entwicklung. Angular 2 ist genau das und verwirft die klassische Struktur der HTML-Dokumente, die jeweils eine gesamte Seite aufbauen. Das Framework baut "Single-Page"-Webseiten. Auf der Hauptseite angekommen werden also nur noch Inhalte ausgetauscht. Welche Konzepte dahinterliegen und die Anwendung dieser wird gezeigt. Abschließend soll die Einbettung sowie Problemlösung durch Angular betrachtet werden.**

***Keywords*—Rich Internet Application, Javascript, Typescript, Angular 2**

I. Einleitung

Das Internet verändert sich. Die Webseiten entwickeln sich zu Anwendungen, die denen auf dem Computer stark ähneln. Sie sind interaktiv und funktionsreich und werden Rich-Internet-Applications genannt. Das Element, das diese früher so statischen Seiten mit Dynamik versieht, ist meistens Javascript. Kurzgefasst handelt es sich um clientseitigen Programmcode, der die Darstellung der Seite dynamisch ändert und dazu noch mit dem Server kommunizieren kann. Die Clientunterstützung ist weitreichend und die Vorteile groß. Nutzer haben von all ihren Geräten Zugriff auf die Applikation und ihre damit erzeugten Daten. Die Erstellung solch mächtiger Webseiten wirft einige Probleme auf. Es benötigt ein Werkzeug, dass bei der Lösung dieser hilft. Angular 2 ist eines davon.

A. Was ist Angular 2?

Angular 2 ist ein Framework, das moderne Konzepte in die Welt der Javascriptentwicklung bringt. Konzepte, die teilweise schon längst in der Desktop- und Smartphone-Appentwicklung Einzug fanden. Ein solches Framework bietet Richtlinien und Bibliotheken, um die mitgebrachten Konzepte zu erfüllen. In diesem Dokument werden diese Konzepte beschrieben und die beispielhafte Anwendung kurz veranschaulicht.

B. Problem der Webentwicklung

Die Internetpräsenzen bekommen immer mehr Funktionalität. Die benötigte Menge Code wächst und bedarf guter Struktur. Ohne diese potenziert sich der Aufwand zum Verstehen des Codes. Aus fehlendem Verständnis könnten dann Fehler folgen. Das Erzielen von Fortschritt wird mühsam und auch die Wartung wird viel Zeit in Anspruch nehmen. Nutzer werden außerdem anspruchsvoller. Schnelles Laden der Webseiten sowie des Feedbacks bei der Bedienung sind wichtig, um die Benutzer nicht ungeduldig werden zu lassen. Sie werden dank technischem Fortschritt an die Schnelligkeit bei der Verwendung von Software gewöhnt. Webseiten müssen aufgrund der Abhängigkeit mit dem Internet besonders effizient arbeiten, um mit lokalen Anwendungen mithalten zu können.

II. Grundkonzept

Zur Bekämpfung des Problems wurde folgende Grundidee entworfen. Lange redundanzüberhäufte HTML-Dokumente an denen Javascript anhängt werden vermieden. Stattdessen wird ein modulares System eingeführt. Ein besonderes Modul bildet den Einstiegspunkt der Anwendung. Es wird als Root-Modul bezeichnet und verknüpft weitere Module und sich selbst. Für die Darstellung der Webseite sind die sogenannten Komponenten zuständig. Sie beschreiben ihre Darstellung mittels HTML- und CSS-Code. Das Root-Modul legt die Komponente fest, die der Browser anzeigen soll. Genannt wird sie Root-Komponente. Diese hat die Aufgabe, die Grundstruktur der Webseite aufzubauen. Um dies zu tun, können andere Komponenten eingebettet werden. Diese können wiederum weitere Komponenten in ihrer Darstellung verwenden. Dadurch entsteht eine hierarchische Webseitenstruktur. Die Gesamtseite wird somit aus Teilstücken von HTML zusammengefügt. Durch die Komponenten wird das Problem der Darstellung der Webseite in kleinere Teilprobleme unterteilt. Diese Teilung ist bei der Problemlösung ein wichtiger Schritt. Bei steigender Komplexität kann mit feinerer Unterteilung des Problems eine geringe Komplexität der Teilprobleme beibehalten werden. Die Komponenten können zusätzlich an ihren HTML-Code dynamische Aspekte anfügen. Es kann sowohl der Klartext als auch die Attribute eines HTML-Tags an Variablen gebunden werden. Das bedeutet, dass jede Veränderung der gebundenen Variable Einfluss auf den HTML-Inhalt nimmt. Das Propagieren des neuen Wertes der Variable an die Webseite übernimmt das Angular-Framework. Dies ist ein Feature, das bei Rich-Internet-Applications einen großen Vorteil bewirkt. Solche Anwendungen kommunizieren oft mit einem Server und präsentieren erhaltene Daten dem Benutzer. Die Seite neuzuladen würde einen großen Aufwand bedeuten, um unter Umständen lediglich einen veränderten Wert darzustellen. Mittels dem sogenannten Data-Binding funktioniert dies dann sparsam und ganz automatisch. Analog dazu existiert das Event-Binding. Dies ermöglicht es, auf Veränderungen der Webseite durch den Nutzer zu reagieren. So kann ein Textfeld beispielsweise an einen String gebunden werden und jede Eingabe darin wird die String-Variable verändern. Diese Bindungen lassen sich beliebig herstellen und auch in beide Richtungen realisieren. Das Textfeld bekommt bei solch einem Two-Way-Data-Binding dann alle Veränderungen des Strings

und umgekehrt. Weiterhin besitzt Angular 2 ein fundamentales Konzept, um Elemente des HTML-Codes zu verändern oder Logik an diese anzuheften. Genannt werden die Platzhalter Direktiven, die im HTML-Code der Komponenten die Stelle der Transformation markieren. Die Navigation zwischen den Komponenten wird mithilfe der sogenannten Router bewerkstelligt. Sie tauschen nicht nur die Komponenten nach Betreten der Seite aus, sondern regeln ebenfalls das korrekte Routing zu einer Unterseite (z.B.: test.de/UNTERSEITE). Außerdem ist das Konzept der Dienste vorhanden. Diese stellen Daten zur Verfügung oder führen anderweitige Hintergrundaktivitäten aus.

III. Konzepte

A. Modules

Module sind die grundlegendsten Elemente in Angular 2. Unterschieden wird grob in Feature-Module und dem Root-Modul. Unter ersterem versteht man Module, die eine gewisse Funktionalität bieten. Zum Beispiel ermöglicht das HttpModule, das von den Angular 2-Entwicklern bereitgestellt wird, eine einfache Möglichkeit zur Kommunikation über HTTP. Es lassen sich aber auch eigene bauen und genau so wie die bereits existierenden in das Root-Modul importieren. Das Root-Modul benennt alle Module, die die Anwendung benötigt. Aus Konvention wird es AppModule genannt. Hinter einem Modul steckt eine Typescript-Klasse mit einem NgModule-Dekorator. Diese sogenannten Dekorator-Funktionen sind eine Typescript-Funktionalität um Metadaten mit einer Klasse zu verknüpfen. Das Grundgerüst eines Moduls kann damit wie folgt aussehen:

```
1  import { NgModule } from '@angular/core';
2
3  @NgModule({
4    imports: [ ],
5    declarations: [ ],
6    exports: [ ],
7    providers: [ ],
8    bootstrap: [ ]  // Nur vom Root-Modul angegeben
9  })
10 export class AppModule { }
```

Listing 1. app.module.ts - Root-Modul-Gerüst

In dem Code-Auszug besitzt der Dekorator die wichtigsten Parameter der NgModule-Funktion. Das imports-Array beinhaltet eine Menge an Modulen, auf dessen Funktionalitäten zugegriffen wird. Zum Beispiel ist es typisch für ein Root-Modul das BrowserModule zu importieren, um eine Browseranwendung erstellen zu können. Das declarations-Array beinhaltet alle Komponenten, die diesem Modul zugeordnet sind. Damit andere Module die Komponenten, Dienste und weiteres verwenden können, die diesem Modul zugeordnet sind, werden sie in den exports eingetragen. Importiert ein Modul ein anderes, so sind die Exports dieses für das Modul verwendbar. Die providers sind die Services, die die Komponenten dieses Moduls benötigen. Sie werden als Singleton instanziert und in die Komponenten injiziert. Der letzte Parameter bootstrap benennt die Root-Komponente, also die Komponente, die als erstes mit der Anwendung instanziert werden soll.

B. Components

Komponenten bilden das hierarchische System von Angular 2. Demnach lassen sie sich ineinander verschachteln. Die Root-Komponente stellt die Wurzel dieser Hierarchie dar. Im Folgenden ist das Grundgerüst dieser zu sehen. Der Konvention zufolge wird sie AppComponent genannt.

```
1  import { Component } from '@angular/core';
2
3  @Component({
4    selector: 'my-app',
5    template: '',
6    styles: []
7  })
8  export class AppComponent { }
```

Listing 2. app.component.ts - Root-Komponenten-Gerüst

Wie zu sehen handelt es sich wieder um eine Typescript-Klasse mit einem Dekorator. Dieser unterliegt jedoch diesmal der Component-Funktion. Wie es auch bei den Modulen der Fall war, werden den Komponenten wichtige Metadaten über die Parameter des Dekorators zugewiesen. Die wichtigsten sind im Listing 2 aufgeführt. Der selector ist ein CSS-Selektor und legt eine Identifikation für diese Komponente fest. Das template beinhaltet den HTML-Aufbau der Komponente und das styles-Array enthält CSS-Anweisungen, die auf diese Komponente wirken. Damit kann jede Komponente seinen Bereich aufbauen und stilisieren. Außerdem wird später gezeigt, wie das Verhalten definiert werden kann. Eine Komponente wird immer von genau einem Modul deklariert. Es besitzt sozusagen einen Vater, der die nötigen Provider und Imports des Kindes berücksichtigt. In der Beispielanwendung gibt es drei Komponenten. Die AppComponent (Root-Komponente) und die BasicComponent gehören dem AppModule (Root-Modul). Das HttpSampleComponent wird vom TestModule deklariert. So finden sich die Module und ihre Kinder auch in unterschiedlichen Ordnern wieder. Das sorgt für Übersicht. In der Rootkomponente des Beispiels sind lediglich Routerlinks zu den anderen beiden Komponenten und die Komponente des Routerziels im Template festgelegt. Anderweitig ist nichts zu bemerken. Die Root-Komponente hat für gewöhnlich nur die Aufgabe, die nächst-tieferen Komponenten in ein grobes Layout zu ordnen. Im Beispiel beinhaltet die Rootkomponente dazu noch das Menü. Auch dies wird idealerweise ausgelagert werden, sobald die Applikation wächst. Auf die Funktionsweise der Router wird später eingegangen. Die BasicComponent beinhaltet sowohl mehr Template als auch CSS-styles und Klassenvariablen. Eine Reihe von Angular 2-Funktionen wird hier auf kleinem Raum verwendet und in diesem Dokument nach und nach erklärt werden. Die HttpSampleComponent dagegen konzentriert sich ganz auf eine kleine Demonstration einer HTTP-bezogenen Darstellung von Informationen.

C. Variables & Data Binding

Die Syntax von Konstanten, Variablen, Arrays und Methoden ist TypeScript-typisch und wird nicht näher erläutert. Mit etwas Erfahrung in Programmiersprachen sollte dies aber intuitiv anhand der Beispiele verständlich sein. Um nun einige Möglichkeiten des Templates aufzuzeigen, kann folgender Ausschnitt der BasicComponent betrachtet werden:

```
1  @Component({
2   template:
3    `<h3 [class.special]="vis"> // Property-Binding
4    Hallo, {{member}}</h3> // One-Way-Binding
5    <input [(ngModel)]="member"/>` // Two-Way-Binding
6  })
```

```
export class BasicComponent {
  member: string = 'John Doe';
  vis: boolean = false;
}
```

Listing 3. basic.component.ts - Data-Binding

Das Property ”class.special” des h3-Tags wird an den Boolean ”vis” gebunden. Wenn das Property gesetzt wird, dann gehört das Tag zur CSS-Klasse ”special” und erbt dessen Stilisierungen. Die Klasse ”special” sollte also im angehängten CSS-Code der Komponente definiert sein. Verändert sich die Variable ”vis”, dann wird dem Tag die Klasse ”special” zugewiesen beziehungsweise entzogen. Stile werden also auch dynamisch ein- und ausschaltbar. Das One-Way-Data-Binding in der nächsten Zeile ist eine Methode, um Daten aus den Objekten in das Template und somit in die entstehende Webseite einzubinden. Bei Veränderung dieses Datenwertes wird automatisch auch der Wert im HTML-Code angepasst, daher spricht man von einem Binding (dt.: Bindung). Für die Darstellung dynamischer Inhalte ist es optimal. Für die Umsetzung ist zuerst eine Variable nötig. In der BasicComponent wird eine Zeichenkette (String) in der Klasse deklariert und auch direkt initialisiert. Im Template kann nun mittels `{{VARIABLE}}` innerhalb eines beliebigen Taginhaltes der Wert von VARIABLE angezeigt werden. Das dahinterliegende Konzept des Propertybindings kann auch direkt angewandt werden und bietet darüber hinaus mehr Möglichkeiten. So kann mit der Syntax `<TAG [PROPERTY]= "VARIABLE"/>` ein beliebiges Attribut eines HTML-Tags an eine Variable gebunden werden. Jegliche Änderung dieser wird dann propagiert und das Tag entsprechend angepasst. Dieses Binding ist nicht nur der Standardbibliothek vorenthalten. Auch selbst erstellte Komponenten können mithilfe von Annotationen Variablen als Eingabeparameter definieren. Bei der Verwendung der Komponente im Template wird dann die Bindung von Vaterkomponente zu Kindkomponente über ein Property Binding definiert. Komponenten können aber auch Ausgabeparameter festlegen. Diese sind eventgesteuert und befüllen die gebundene Variable des Vaters bei einem emit, also einer Aussendung eines aktualisierten Wertes. Die Syntax ist wie folgt: `<TAG (EVENT)="VARIABLE"/>`. Nun kann man sowohl Eingabe- als auch Ausgabebindungen zwischen Komponenten herstellen. Und das auch zugleich. In dem Fall gibt es eine verkürzte Schreibweise, die beides anwendet. Vorrausgesetzt die Bezeichnung der Eingabevariable ist gleich dem Eventsender (anschaulich gesprochen der Ausgabevariable) konkateniert mit ’Change’. Dann kann dieses sogenannte Two-Way-Data-Binding über die Kurzform verwendet werden. Im Beispiel wird mit der Input-Komponente der Angular 2-Bibliothek eines eingegangen. Sowohl dies als auch Property- und One-Way-Binding sind im folgenden Listing zu sehen (stark verkürzt).

D. Directives

Direktiven sind Markierungen im Template, an die Angular 2 die entsprehende Direktivenklasse eine Transformation ausführen lässt. Angular 2 unterscheidet zwischen drei verschiedene Typen. Komponenten sind eine davon. Sie fügen an der Stelle ihres Tags ihr eigenes Template an. Der Selektor ist übrigens eine Bedingung der Direktiven. Alle müssen einen angeben. So auch die Strukturdirektiven. Die bekanntesten von Angular 2 sind ngIf, ngFor und ngSwitch und ermöglichen es, ein HTML-Tag konditional anzuzeigen oder (mit der ngFor-Schleife) zu vervielfachen. Dabei ist interessant, dass Elemente, die die Kondition nicht erfüllen, nicht nur ausgeblendet, sondern ganz aus dem Objekt-Modell entfernt werden. Der Vorteil liegt sowohl in dem Speicherverbrauch, als auch in der Fehleranalyse. Eine ausgeblendete Komponente könnte im Hintergrund laufen und mit seinem Verhalten ungewünschte Auswirkungen haben. Dies ist nicht sofort offensichtlich. Diese Entscheidung kann aber auch schlechten Einfluss auf die Performanz nehmen. Wird zum Beispiel eine Komponente öfter konditional geschaltet, so wird auch der unter Umständen hohe initiale Resourcenaufwand jedesmal erneut aufgebracht. Der Entwickler kann jedoch bewusst die ”kritischen” Komponenten verstecken oder den Aufwand in einen Service auslagern, der die Daten im Speicher behält. Die Beispielanwendung zeigt die Anwendung einer IF-Bedingung sowie der FOR-Schleife.

```
<div *ngIf="vis">
 <span *ngFor="let text of texts">{{text}} </span>
</div>
```

Listing 4. basic.component.ts - Direktiven

Das ngIf bekommt als Übergabe einen booleschen Wert und das ngFor eine Anweisung, mit der für jedes Element des Arrays `texts` ein eigenes span-Element erzeugt wird. Das Element, das die ngFor-Direktive enthält, wird also vermehrfacht. Hierbei wird jedem erzeugten span das Arrayelement der jeweiligen Position als ”text” verfügbar gemacht. Die Syntax ist also `let ELEMENT of ARRAY`. Das Ergebnis des Beispiels zeigt also bei wahrem Boolean ”vis” drei span’s. Jedes gibt über One-Way-Binding den Text ihres Arrayelementes aus. Das * an den Direktiven ist ein wichtiges syntaktisches Merkmal. Es bedeutet, dass das zu strukturierende Tag nochmals in ein Template-Tag gewickelt wird. Dies ist notwendig, damit Angular 2 die Elemente bei nicht erfüllter Bedingung vollständig entfernen kann. Die dritte und letzte Art von Direktiven sind die Attributdirektiven. Sie verändern das Aussehen oder Verhalten des Tags, das sie beschreiben. Alle drei Arten von Direktiven sind selbst erstellbar. So kann beispielsweise auch die ngIf-Direktive nachgebaut werden.

E. Routers

Das Einbinden der Routing-Funktionalität erfordert die Festlegung der Routen, die einen Pfad auf eine Komponente weisen. In der Beispielanwendung wird dies in dem SampleRoutingModule gemacht.

```
const routes: Routes = [
 {path:'', redirectTo:'/basic', pathMatch:'full'},
 {path:'basic', component:BasicComponent},
 {path:'http-sample', component:HttpSampleComponent}
];
@NgModule({
  imports: [ RouterModule.forRoot(routes) ],
  exports: [ RouterModule ]
})
export class SampleRoutingModule {}
```

Listing 5. sample-routing.module.ts - Routes

Die erste Route beschreibt die typische Umleitung. Wegen dieser wird beim Betreten der Webseite automatisch auf den /basic-Link weitergeleitet. Die Zweite definiert nun den /basic-Pfad und routet ihn auf die BasicComponent. Analog dazu

wird der /http-sample-Pfad auf die HttpSampleComponent geroutet. Diese Routen werden nun der statischen forRoot-Methode übergeben, die ein entsprechend konfiguriertes Modul zurückgibt. Wie im Listing zu sehen, wird diese Rückgabe importiert. Das anschließende Exportieren macht diese Konfiguration für das RootModul importierbar. Die so in der ganzen Anwendung verfügbaren Routing-Funktionen werden im Beispiel in der AppComponent genutzt.

```
<a routerLink="/basic">Basic</a>
<a routerLink="/http-sample">Http-Sample</a>
<router-outlet></router-outlet>
```

Listing 6. app.component.ts - Routing

Die a-Tags starten den Router mit dem jeweiligen routerLink-Attribut. Dieser initialisiert dann die Komponente, auf die geroutet wurde und zeigt sie im router-outlet-Tag an. Danach wird die Adressleiste im Browser mit dem neuen Pfad aktualisiert. Das ist wichtig, damit Benutzer die durch den Router veränderte Seite über einen Link aufrufen können und nicht von der Hauptseite aus zur gewünschten Seiten navigieren müssen. Diese Router-Funktionalität ermöglicht eine Überarbeitung des klassischen Ansatzes. Anstatt per Hyperlink immer wieder gesamte HTML-Dokumente anzufragen, werden nur die geänderten Teile ausgetauscht. Dies vermindert häufig die Menge der Daten, die übertragen werden muss. Gerade im volumenbegrenzten Mobilinternet ist das ein wichtiger Vorteil. Dazu bedeuten weniger Daten auch weniger Zeit, um diese zu übertragen. Das macht die Darstellung schneller und die Webseite damit angenehmer bei der Bedienung.

F. Services & Dependency Injection

Für die Dienstbereitstellung ist es wichtig, auf dieselbe Instanz zugreifen zu können. Dies wird mithilfe des Singleton-Patterns realisiert, um Nutzen aus hergestellten Verbindungen oder gecacheten Daten zu ziehen und die Aktivität von mehreren Orten aus steuern zu können. Angular 2 übergibt den Komponenten die vom Modul bereitgestellten Dienste mittels Dependency Injection. So muss eine Dienst-Klasse zunächst den Injectable-Dekorator tragen. Im Modul kann diese dann als Provider festgelegt werden. Alle Komponenten dieses Moduls können dann dieselbe Instanz des Dienstes erhalten, wenn sie in ihrem Konstruktor den zu injizierenden Dienst angegeben haben. Wie im Auszug der Beispielanwendung zu sehen, ist die Syntax schlicht gehalten.

```
export class HttpSampleComponent {
  constructor(private httpSampleService:
    HttpSampleService) { }
}
```

Listing 7. http-sample.component.ts - Dienste

Ist der HttpSampleService ein Dienst und wurde im Quellcode der Komponente importiert, so wird dieser als neue globale Variable deklariert. Wie die Verwendung eines solchen Dienstes aussehen kann, wird in dem nächsten Abschnitt gezeigt.

G. HTTP & Interfaces

Für die Kommunikation mit einem Server bietet Angular 2 den bereits erwähnten HTTP-Dienst. Mit ihm kann man dann mit wenig Zeilen Code eine HTTP-Anfrage senden. Die Antwort wird über eine intelligente Lösung zurückgegeben. Auf eine HTTP-Anfrage wird ohne warten ein sogenanntes Observable-Objekt zurückgegeben. Auf das kann man Code (genauer Lambda's) registrieren, der beim Erhalt der eigentlichen Antwort ausgeführt wird. Zusätzlich sollte beachtet werden, dass Serveranfragen nie über den Konstruktor geschehen sollten. Wenn also Serverdaten direkt bei Initialisierung der Komponenten benötigt werden, muss die entsprechende Anfrage woanders geschehen. Hierfür ist das Interface OnInit geeignet, welches eine Methode bereitstellt, die bei Initialisierung der Komponente aufgerufen wird. Die HttpSampleComponent des Beispiels implementiert dieses. Angular 2 ruft dann die ngOnInit()-Methode zu gegebener Zeit auf. Die Beispielanwendung nutzt dann den injizierten HttpSampleService, um einen Observable der erwarteten Rückgabe zu erhalten. Dieser wiederum bekommt den HTTP-Dienst von Angular 2 injiziert und sendet damit die Anfrage wie folgt heraus.

```
getData() {
  return this.http.get('http://httpbin.org/ip')
    .map(response => response.json());
}
```

Listing 8. http-sample.service.ts - HTTP

Die get(URL)-Methode des HTTP-Dienstes sendet eine GET-Anfrage an den Server. Die Rückgabe wird ein Observable sein, das auf die Antwort des Servers reagiert. Die map-Methode ändert den Typ des Observables. Anstatt einer HTTP-Response wird ein Json-Objekt beobachtet, indem die Json-Daten aus der Antwort extrahiert werden. Dadurch ist die HTTP-Anfrage besser gekapselt. Die HttpSampleComponent kann nun mittels einem subscribe auf dem Observable das Json-Objekt wie folgt verarbeiten:

```
export class HttpSampleComponent implements OnInit {
  ngOnInit(): void {
    this.httpSampleService.getData().subscribe(
      clientIp => this.clientIp = clientIp,
      error => console.error('Error: ' + error),
      () => console.log('Completed!'));
  }
}
```

Listing 9. http-sample.component.ts - Interface & Observable

In diesem Fall erwarten wir ein Json-Objekt vom Typ ClientIp. Die Variable "origin" des Objektes ist dann mit der eigenen externen IP gefüllt und wird über ein Binding im Template letztendlich ausgegeben.

IV. Einbettung in der Praxis

Webseiten, die mit Angular erstellt worden sind, müssen Javascript auf dem Client ausführen können. Trotz der weitreichenden Clientunterstützung entschließen sich viele dazu, Javascript zu deaktivieren. Oft sind Sicherheitsbedenken der Grund. Moderne Browser schützen jedoch immer besser vor den Gefahren möglicher Sicherheitslücken der Skriptsprache. Da Angular vorallem die Verwendung von Javascript vereinfacht, kann man annehmen, dass in vielen Projekten die Javascript-Verwendung ohnehin geplant ist. Die Entscheidung Angular zu verwenden schränkt unter der Annahme also nicht die Clientunterstützung ein. Solche Projekte können auch beliebige Javascript-Bibliotheken weiter verwenden. Lediglich

andere Javascript-Frameworks, dessen Regeln sich mit denen von Angular überschneiden, werden Probleme bereiten.

V. Fazit

Angular 2 bettet zeitgemäße Überlegungen in die klassische Webentwicklung ein. Ein Webseitenprojekt erhält einen ähnlichen Aufbau wie gewöhnliche Desktop-programme. Bei Wachstum der Codemenge steigt nicht im gleichen Maße das benötigte Verständnis und damit die Wahrscheinlichkeit, Fehler zu machen. Zudem wird dank Konzepten wie Komponenten und Routing eine performante Erfahrung möglich. Die Generationen, die mit der schnellen Entwicklung des Internets aufgewachsen sind und niedrige Ladezeiten gewöhnt sind, werden nicht durch lange Wartezeiten abgeschreckt. Die entstehenden Rich Internet Applications holen großes Potenzial aus der Cloud-Technik. Nutzer bekommen eine plattformübergreifende Applikation, um direkt mit ihren Daten weiter zu arbeiten. Werkzeuge wie Angular 2 fördern die Entwicklung und bringen die Zukunft schneller in die heutige Zeit.

React and Redux

Matthias Kevin Caspers
Carl von Ossietzky University of Oldenburg, Germany
Department of Computer Science
matthias.kevin.caspers@uni-oldenburg.de

***Abstract*—Many popular JavaScript frameworks for developing single page applications (SPAs) and rich internet applications (RIA) use mutable state, templating, a two-way data-binding system and impose the imperative programming paradigm on the developer. This paper will show some problems that the aforementioned design choices have and present React and Redux as an alternative for developing SPAs and RIAs. React is a user interface (UI) library, written in JavaScript and developed by Facebook. It's often described as being the V in MVC. Redux is an easy to use predictable state container, created by Dan Abramov in JavaScript. React and Redux are both independent from one another. They do however work very well together in practice. Since React is a library for building UIs and Redux is a state container, they do not come with many of the features that fully fledged SPA/RIA frameworks like Angular JS or Ember JS provide. There exist however dozens of libraries which can provide these features to them. This paper will introduce the reader to React and Redux, as well as the problems which they solve. The first part of this paper will focus solely on React. Redux will be covered in the second part. The third part will cover how React and Redux can be used together. It will be shown, that React and Redux present a powerful way for developing SPAs/RIAs, which in many regards is superior to the way SPAs and RIAs were written prior to the emergence of React.**

***Keywords*—React JS, Redux**

I. Introduction

Developing RIAs [1], while promoting the use of mutable state, templating, two-way binding and the imperative programming paradigm has some drawbacks. Mutable state makes bugs in programs harder to detect. The reason for this is that it's harder to keep track of all the places state is mutated from, especially when the state is mutated by asynchronous code. The imperative paradigm often leads to more complicated code compared to code written in a declarative way, as one has to specify how a problem is solved in the former. Two-way data-binding connects HTML/DOM to JavaScript [1]. It makes it easy for developers to have changes in the View reflect in the Model and vice versa. A big problem which arises from this in larger applications is that if dirty checking (checking if data has changed) is used, checking for changes takes a lot of time. This is due to constantly checking all elements for changes (polling) [1]. An alternative way of checking for changes is to use observables (dependency checking) instead of dirty checking. This however is no silver bullet, as observables require configuration in JavaScript and come with some overhead. It's necessary to setup observables for the DOM elements, which requires memory and cpu cycles. React and Redux are an alternative to the established way RIAs are written. It will be shown where the strengths of React and Redux lie and how they can be used together to develop RIAs.

[1]Everything covered by this paper concerning RIAs also applies to SPAs (single page applications). All of the mentioned frameworks and libraries can be used for developing both. Note that, as with everything in JavaScript, the limitations set by React and Redux can not be enforced. The developer should however adhere to best practices and not circumvent the rules set by the libraries.

II. React

A. Reacts core design principles

React is a UI library. Its primary goals are to render UI and respond to events [2]. The following concepts are the core design principles which React resolves around. They will be covered more in depth later. In React components are used instead of templating [2]. Components combine the display logic written in JavaScript with the markup written in HTML [3]. Re-rendering on every change in state, that should be reflected in the UI [2]. React allows developers to write code in the same way they would, if the whole applications would get re-rendered on every update. For efficient re-rendering React utilizes a system known as virtual DOM [4]. React also comes with its own custom event system (synthetic events which utilizes automatic event delegation), which wraps browser events, to improve the cross browser development experience [3].

B. JSX - an optional syntax extension for JavaScript

JSX allows developers to embed HTML tags and HTML-like tags, used for React components, directly in JavaScript [3]. The following is a React element.

```
var element = <h1>An element</h1>
```

This is neither valid JavaScript nor HTML, it's a JSX expression. Such an element can be rendered into the DOM. JSX syntax allows developers to write:

```
<MyButton color="blue" shadowSize={2}>
Click Me
</MyButton>
```

which will be compiled to

```
React.createElement(
MyButton,
{color: 'blue', shadowSize: 2},
'Click Me'
)
```

[3] The former being more concise and easier to read. A big advantage of this over using templates is that a developer can

utilize the entirety of JavaScripts features instead of relying on the far more limited set of features a templating library provides [2]. Developers can for example utilize JavaScript for conditional branching, filtering, mapping, reducing and looping. JSX recognizes arrays containing multiple React elements as such, which enables on the fly generation of components and elements.

C. Components and elements

React is a component centered library for building user interfaces [5]. This is a contrast to most other UI libraries which use templating. Templating allows the developer to access JavaScript data directly from within HTML. It may implement, depending on the templating engine used, some advanced features. The handlebars JavaScript library for example implements a templating system which enables developers to use for-each looping functionality inside the HTML-markup. It also enables the creation of shared templates (Partials), which allow for the reuse of existing templates [6]. In React, DOM generation and display logic are combined into components [2]. React components are written in a declarative manner [3]. Stateful components are required to be either a class extending React.Component or a variable returned by the React.createClass method. Stateless components may also be creating this way, but may be implemented as a function instead. Note that all components must implement a render method which returns an element. The only exception to this rule are components implemented as functions themselves, these have an element as their return value. React elements are plain old JavaScript objects describing how the HTML should look. Apart from the tag, specifying the type of DOM element or React component they represent, they may also contain props and children which are elements themselves [3]. A simple element:

```
var reactElement = <div><p>Two paragraphs
    contained by a div</p><p>the paragraphs
    are children of the div</p></div>;
```

A stateless component:

```
function statelessComponent(props) {
  return <p>This is a paragraph containing
      {props.data}</p>;
}
```

1) State and Props: Props are just like properties in HTML tags. Props can be passed down the component tree and should never be mutated. This leads to a clean uni-directional data flow. There are multiple types of props which can be passed to components including JavaScript primitives and React elements. Change handlers can be passed down as props, thus the components which receive them do not require state themselves [3].

```
<h1 onClick={this.propPassedDown}>click</h1>
```

The handler function that gets passed as a prop must have its ”this” value bound to the component which contains the state that should be changed. Assuming that the function is supposed to alter the state of that particular component. A component may have its own state. To assign the initial state a class constructor is used. Using setState as the sole way of updating the state is necessary for the virtual DOM, which will be covered later, to work correctly [3].

```
class aComponent extends React.Component {
  constructor(props) {
  super(props);
  this.state = {myData: 1};
  }
  // ...
  someMethod() {
   this.setState({
   myData: props.data
   });
  //...
}
```

React, unlike some popular libraries (e.g. Angular 1), doesn't use two-way data-binding. Applications using two-way data binding, grow very complex as they evolve. Two-way data binding leads to cascading updates in complex applications. Cascading updates are updates which cause large parts of the application to be re-rendered, even parts, which do not have a direct relation to the originally intended update. Cascading updates lead to unpredictable code [5]. React uses unidirectional data-binding, utilizing props, instead. This mitigates the complexity that comes with two-way data-binding.

2) Lifecyle hooks: When a component gets mounted, it gets inserted into the DOM and its componentDidMount method will be called by React. On removal from the DOM, React will call the components componentWillUnmount method [3]. These methods allow developers to write logic that is executed whenever a component mounts or unmounts.

3) Container and Presentational Components: Components should be divided into container (smart) and presentational (dumb) components [7] [8]. This leads to better separation of concerns [9]. Presentational components are reusable as they contain generic logic and markup. They seldom have state and if they do it's state concerning UI. They typically receive data and callback functions in the form of props. Presentational components should be written as functions whenever possible [8]. They should not have any application dependencies to be truly reusable. As their output depends solely on their props, they can easily be unit tested. Container components on the other hand contain application logic such as data fetching and typically only contain markup needed to wrap other components.

D. Virtual Dom

It is very convenient to re-render on every state change, to reflect user actions and offer a smooth user experience. Before explaining how the virtual DOM works, the problems of re-rendering on every state change using the actual browser DOM will be illustrated.

1) Problems of re-rendering using the browser DOM: Re-rendering the whole DOM on every update is problematic for several reasons. Firstly repainting is very slow [10]. On every change styles and sizes have to be recalculated and reapplied which can lead to cascading reapplications of CSS. Layouting

is the main reasons why working directly with the DOM is considered slow [11]. Secondly data will be lost. For example re-rendering the DOM will mess up form fields and reset the scrollbar position [2]. Thirdly animations get reset. Fourthly re-rendering without buffering will cause screen-flashing [2].

2) Virtual DOM explained: Working on a virtual DOM is much faster than working on the real DOM, as layouting is not necessary. The setState method of a React component, marks it as dirty [12]. Calling setState on individual components allows the virtual DOM to use selective sub-tree re-rendering [13]. React calculates the difference between the virtual DOM and the changed subtrees to generate patches which are applied to the browser DOM. At the end of an event loop all components marked as dirty are re-rendered. React using batched updates to efficiently write to the browser DOM. This means, that the DOM is updated just once every event loop [13]. This makes re-rendering with React very fast out of the box, which is difficult to achieve using other frameworks which do not utilize a virtual DOM or a comparable system like the incremental DOM. Developers don't have to take care of synchronous and/or asynchronous re-rendering and the complexity which comes with optimizing it. The shouldComponentUpdate hook can be used to let React know if a component should be updated depending on its state. This can be used to further optimize the operations done by the virtual DOM [3].

3) Virtual DOM and manipulation of the browser DOM directly or by third party libraries: React works mainly with the virtual DOM, some data like form field data, is however held in the actual DOM. React doesn't usually compare the virtual DOM tree to the actual DOM. This means that using JavaScript libraries like jQuery, which work directly on the DOM, will lead to unpredictable behavior, if they manipulate the browser DOM. The reason for this being that the virtual DOM doesn't know about changes made to the DOM by those libraries. This can be avoided through usage of the aforementioned lifecycle hooks when accessing third party APIs.

E. Synthetic Event System

React comes with its own event system known as the "synthetic event system". It wraps browser specific events inside synthetic events [3]. Each synthetic event contains a generic set of properties in addition to event specific properties [14]. This makes it easier to develop for multiple browsers. The names of synthetic events are written in camelCase starting with a lower case letter, e.g. "¡button onClick=handlerMethod¿". This is different from browser native events which have lower case names. The synthetic event system does not cover all DOM events [14]. The most commonly used types of events are covered though.

F. Libraries providing features of full blown RIA frameworks to React

As mentioned before, React is a UI library, it misses the features fully blown RIA frameworks provide out of the box. There are however many libraries which can provide these feature to React. A lot of these libraries were written specifically for React or with React in mind (e.g. React-Router, a routing library for keeping the UI in sync with an URL).

G. Server-side rendering with React

React makes it possible to render components on the server. This is due to React being able to be run in JavaScript environments other than a web browser. The most popular example of such an environment is NodeJS. A problem with pure client side rendering is that Search Engine Bots do not execute JavaScript, therefore making React applications written without server-side-rendering impossible to index appropriately (unless using a third party SEO-service). Server-side rendering has three main advantages. Firstly, all components can be read by a search engine bot, as they appear as static HTML. Secondly the amount of bytes transferred is reduced, as no further fetch requests are necessary. This results in an improved user experience, since less HTTP request make the page load faster. Usage of less bandwidth also reduces the bandwidth cost.

III. Redux

Redux is a predictable state container. This means that the cause of state change can easily be identified. Redux uses a single store instead of many different stores like other flux inspired libraries do. This is often called "single source of truth", as the store is the only place where state can be accessed from [15]. As applications become more complex, they have to manage more state. Redux makes state changes predictable. It achieves this by setting several rules. The first of which being that state must be immutable. Secondly the information about state changes must be contained by actions. And thirdly the use of reducers, which are pure functions, specifying how actions transform the state [16]. References to multiple states can be stored, as the state in a Redux application is immutable. This makes implementing a history easy [16].

A. Store

The store should contain all state in a Redux application. The state is usually implemented as a tree shaped hierarchy of plain old JavaScript objects, referred to as the state tree. The store provides a getState method returning the state and a dispatch method for dispatching actions. It also provides a subscribe method allowing developers to subscribe their own state change listeners to the store [16].

B. Actions

Actions are plain old JavaScript objects, representing the intention to change state. All data that is going into the store must be contained by an action object. Actions are required to have a type field to make them identifiable by reducers (see subsection: Reducers). They may contain additional fields, a popular pattern is to use a payload field to hold the data of an action. The actions are dispatched via the stores dispatch method [16]. Storing actions is similar to using the event sourcing pattern, because actions are descriptions of change. Actions can be generated by Action creators, which are functions returning an action. An example is a subscribe-with-email-button on a page, an action creator would generate a new action object for each unique email address. Actions can be dispatched from anywhere, including callbacks.

C. Reducers

A reducer is a pure function which takes the previous state and an action as its parameters and transforms the state tree according to that action. It generates a new state or returns the current state unmodified. As Redux's createStore function only takes one function as a parameter (and also an initial state as another parameter), it is a common pattern to pass in a function which iterates over multiple distinct reducers, each handling one concern [16]. Redux's combineReducers function returns such a function. It takes an object as its parameter. The object maps the sub-state a reducer operates on as a field name to that reducers. Note that every reducer will be called with every action.

D. Middleware

Middleware allows custom logic to be placed between an action being dispatched and that action being forwarded to the reducers. This is for example useful to easily add logging of actions to the application [16].

E. Asynchronous Actions

While it's possible for functions to dispatch actions, the functions in question need a reference to the stores dispatch method to do so. Passing the dispatch method as a parameter to every function which needs to dispatch actions leads to unclean code. An alternative is to hold the store as a global variable, making it accessible for all functions. This would however make it impossible to use server-side rendering, as a separate store is necessary for each request. Several middlewares addressing this problem have been created, Redux-Thunk being the most popular among them. It allows functions to be dispatched via the stores dispatch method. These function receive the stores dispatch and getState methods as parameters.

IV. Integrating React with Redux through the React-Redux library

React-Redux makes it possible to automatically generate container components which render the presentational component they wrap. To do this React-Redux provides a connect function. This function takes in two parameters, a mapStateToProps and a mapDispatchToProps method. These have to be written for each component for which a container component should be generated. The mapStateToProps method maps the part of the state needed by a component to its props. It does so by taking the state as its parameter, calculating the props for the presentational component and returning them. The props will be updated on every state change. The mapDispatchToProps method maps the dispatch method of the store to the components props. It takes the stores dispatch method as its parameter and returns callback functions which will be passed to the presentational component as props. The connect function will then return a function. The returned function takes as its parameter the component for which the mapStateToProps and a mapDispatchToProps methods were written. It will then return a generated container component for the presentational component which was passed in [18].

V. Conclusion

The declarative nature of React makes it easy to create and understand components. It promotes the creation of testable and reusable components. Reacts virtual DOM takes care of rendering to the browser DOM making it easy to build well performing applications [13]. Redux makes state predictable by enforcing immutable state and actions as the only way to communicate change. The predictability and simplicity gives confidence to developers, which leads to better code [5]. Implementing functionality equivalent to event sourcing can easily be done by storing actions. Powerful developer tools for React and Redux allow developers to make use of hot-reloading and time-travel debugging [17].

References

[1] *Marius Gundersen: A comparison of the two-way binding in AngularJS, EmberJS and KnockoutJS*. JSConf Europe 2013.

[2] *Pete Hunt: React - Rethinking Best Practices*. JSConf Asia 2013.

[3] *React Official Documentation*. https://facebook.github.io/react/docs. Accessed: 2016-11-19.

[4] *Boris Dinkevich: ReactJS: Under The Hood*. ReactNext 2016, Tel Aviv, Israel.

[5] *Tom Occhino: Introducing React Native*. React.js Conf 2015 Keynote.

[6] *Handlebars Js: Getting Started*. http://handlebarsjs.com/. Accessed: 2016-11-19.

[7] *Jake Trent: Smart and Dumb Components in React*. http://jaketrent.com/post/smart-dumb-components-react/. Accessed: 2016-12-14.

[8] *Dan Abramov: Presentational and Container Components*. https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0. Accessed: 2016-11-18.

[9] *Learn React with chantastic*. https://medium.com/@learnreact/container-components-c0e67432e005. Accessed: 2016-12-16.

[10] *Mark Wilton-Jones: Efficient JavaScript*. https://dev.opera.com/articles/efficient-javascript/?page=3. Accessed: 2016-12-14.

[11] *Chris Minnick: The Real Benefits of the Virtual DOM in React.js*. https://www.accelebrate.com/blog/the-real-benefits-of-the-virtual-dom-in-react-js/. Accessed: 2016-11-15.

[12] *Harshit Pandey: ReactJS — Learning Virtual DOM and React Diff Algorithm*. http://www.oyecode.com/2015/09/reactjs-learning-virtual-dom-and-react.html. Accessed: 2016-11-15.

[13] *Christopher Chedeau: React's diff algorithm*. http://calendar.perfplanet.com/2013/diff/. Accessed: 2016-12-13.

[14] *Kirupa: Events in React*. https://www.kirupa.com/react/events_in_react.htm. Accessed: 2016-12-14.

[15] *Dan Abramov: The Redux Journey*. React Europe 2016.

[16] *Official Redux Documentation*. http://redux.js.org/docs/. Accessed: 2016-11-15.

[17] *Dan Abramov: Live React: Hot Reloading with Time Travel*. react-europe Conf 2015.

[18] *Dan Abramov: Getting Started with Redux*. Egghead.io Course.

Building Rich Internet Applications with Node.js and Express.js

Christian Peters
Carl von Ossietzky University of Oldenburg, Germany
Department of Computer Science
christian.peters1@uni-oldenburg.de

***Abstract*—While there are many different frameworks for back end web development Node.js is unique in that it puts JavaScript on the back end. Before Node.js JavaScript was mostly known for creating and manipulating web user interfaces within a browser. With Node.js and frameworks built on top of it, like Express.js, web developers can build applications that are rich Internet applications or even only API endpoints completely within a JavaScript technology stack. With this stack that relies on event-driven design creating scalable applications with a relatively simple architecture for different use cases is possible.**

***Keywords*—back end web development, event-driven development, Node.js, Express.js**

I. Introduction

Node.js (from here on only Node) is a runtime environment for the JavaScript language which is build upon the JavaScript engine V8 [1]. Node provides the ability to write back end JavaScript applications and it relies heavily on non-blocking I/O. It makes it possible to write e.g. a server application in the same language that the front end is written in. This combination makes it unique in the web applications technology stack. The reason Node.js gained so much popularity might lie in the fact that JavaScript is a widely used language in web development. Front end development relies on JavaScript for interacting and modifying the Document Object Model in the browser. Now with Node developers do not need to switch context, at least in the sense of the programming language, if they are developing the front or back end of an application. But familiarity with JavaScript is not the only reason why Node became popular.

The other reasons that are often stated are Nodes low memory footprint compared to other frameworks and its ability to scale out of the box [2]. Another reason for Nodes adoption is the interoperability with the serialization format JavaScript Object Notation (JSON). JSON is used for interchanging data between application endpoints. It is a lightweight text format that can be transmitted over a network without depending on a specific programming language [3]. JSON is used frequently in web applications e.g. for sending only the data in a JSON file from the server to the application and letting the client side integrate the data into the application's UI. This can be done independently of the clients implementation as long as the client knows how to parse the JSON data. In that case the server can send the same JSON file to a mobile client as to a desktop browser. Since Node is an engine for JavaScript and JSON is a subset of JavaScript, developers can use JSON natively without depending on any third party libraries. NoSQL solutions such as MongoDB use a format similar to JSON as their data model. These data models are stored as documents and can be queried just as a JSON document instead of in a relational way. Thus a native interaction with JSON and not relying on external libraries helps in further streamlining the communication with data models in JavaScript web applications throughout the whole application stack.

This paper examines the core parts of Node.js and Express.js a framework built on top of the Node.js platform. It will describe at first what the foundation for Nodes speed and efficiency is and then go into details about how to build server applications using only plain Node.js and then refining it with the help of Express.js to make developing with Node.js more comfortably.

II. Node.js

A. V8 Engine

Since Node is built upon Chrome's V8 JavaScript engine it is good to get an understanding about what V8 is and how it operates. Chrome's V8 is an engine for JavaScript built in C++ and is used for the Chrome browser. To achieve a similar performance like static languages such as its own host language C++ V8 was developed to increase the performance of the dynamic language JavaScript. For that V8 compiles JavaScript and is in charge of memory allocation and garbage collection. [4]

Because JavaScript is dynamically typed it is possible to modify properties of objects very easily without having strict syntactic or semantic rules which might be violated by doing so. For example a JavaScript object can be created with a given set of variables and functions. At runtime this very same object can be modified by adding additional variables or functions, without any errors. On the one hand, this approach makes it possible for developers to modify code on the go without worrying about type violations. but on the other hand it is slower to look up properties of objects. This is because a dynamic lookup is necessary every time the value needs to be retrieved from memory.

Increasing the speed of the property retrieval is done by V8 via the implementation of hidden classes for every instantiated object. Dynamic modifications to one object create a new hidden class, but the old hidden classes are still left intact. These seemingly unused classes are used whenever an object corresponds to the form of the hidden class [5]. This usage of hidden classes makes the execution of JavaScript faster because code can be optimized for retrieving a hidden class. More specifically, whenever two different objects are represented by

the same hidden class the same optimized code can be used for both objects. This optimized code needs to be written only once by the compiler which provides a boost in speed and property retrieval [6].

Besides hidden classes, V8 supports types for numbers and handles Arrays in two different ways. As for numbers, the technique of tagging is used to use pointers for two different types of numbers. The last bit of a pointer is used to differentiate between a pointer to a 31-bit signed integer and an object pointer. This way an increase in speed can be gained if numerical values are used in an application that can be referenced by a 31-bit signed integers pointer.[6] Arrays are also internally represented in two different ways, namely as a linear storage and as a hash table. A better performance can be achieved by creating an array that uses elements of the same type, so a change of the underlying hidden class from linear to hash table is not performed.[6] And lastly an other big optimization factor of V8 are optimized compilers that can produce efficient code generation for JavaScript [6].

B. Asynchronicity and Events

The underlying performance optimization by the V8 engine is not the only part of Node, that makes JavaScript very scalable and efficient. Traditionally servers are using multiple threads to handle incoming traffic and heavy I/O operations. Contrary to that Node does not use multiple threads in dealing with its operations. Instead Node relies on a single thread but uses the event driven programming paradigm. This means that almost every operation of Node is asynchronous and uses higher order functions as event listeners [7].

Event driven design is often found in areas such as GUI implementations. A sequential order of instructions is impossible to predict in an user's interaction with the GUI because e.g. it cannot be predicted on which button the user will press at any given time. Instead the application has event listeners and waits for the user's actions. These listeners are waiting e.g. for a click event on a button and code for this event is executed as soon as the button click was registered. In Node the same pattern of event driven programming applies and this paradigm is predominant throughout all of Node's API. As an example, when a Node application waits for an incoming HTTP connection, a callback function is defined in the Node application which will be executed as soon as a connection is registered. [8] In the meantime, the application can execute other code or just be idle waiting for events. It is not blocking up execution by busy waiting for an incoming HTTP request. When the operating system notifies Node about any HTTP connections, only then this callback function will be scheduled for execution and called by Node to execute the steps for processing a HTTP connection defined in the callback function. With this a non blocking flow of execution is achieved.

The following code example illustrates this mechanism by implementing a simple server. For that the *http* module is imported that serves as a wrapper around the HTTP protocol and parses incoming messages into headers and a body, but does not process them any further [9]. In this example the server only sends an HTTP response with the status code 200, a header specifying the Content-Type of the response header and the data in the form of a text string, every time a request comes in.

Listing 1. SIMPLE SERVER APPLICATION

```
const http = require('http');

let server = http.createServer((req, res) =>{
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello, World!');
}).listen(8080, '127.0.0.1');
```

JavaScript handles asynchronous behavior by utilizing higher order functions as the previous mentioned callbacks and an event loop to support event based programming. This pattern is also a core component of Node and the goal of writing an effective Node application should be to use asynchronous methods while interacting with incoming requests and thus creating a non-blocking IO flow. In the previous example this can be illustrated by the *http.createServer()* function. That function takes as the parameter an anonymous function as the request listener callback and returns an instance of a *Server* object. The callback function will be executed every time an event of type *request* occurs while the *Server* object is listening for incoming requests and then the String *Hello, World!* is sent as a response.

To further understand how this works, it is necessary to get a sense of the event loop. In JavaScript or in the environment where JavaScript code is executed, be it a browser or an engine like Node, exists a FIFO queue that functions as an event loop. Callback functions that should be executed after their corresponding event was registered will be put into this event loop. For example, if a the application waits for the disk to finish writing some data into a buffer, the JavaScript engine will postpone the execution of the callback that e.g. processes the content of the buffer, until the content from the disk is completely written into the buffer. As soon as it is read and ready to be processed, the engine puts the callback into the event loop. If other callback functions are already in the queue, the buffer processing will have to wait. Only after all previous callbacks in the event loop are executed, the callback which will be dealing with the buffer will start its execution [10]. The built in modules of Node are asynchronous by nature, but there are also functions, e.g. in the *fs* module that are synchronous and allow a sequential order of execution. Since they are an exception it is necessary to get used to the pattern to provide a callback as an argument to Nodes core functions while working with Node.

C. Buffers

When dealing with binary data like incoming TCP streams or other data such as video and audio files it is necessary to have data structures that provide the capability to read and manipulate the bytes of this data directly. JavaScript provides native support for easily working and manipulating binary files only since ECMAScript 2015 (ES6) in the form of the TypedArray type. But before that Node had already an implementation of a *Buffer* type which allowed manipulation of binary files and making it easy to create custom protocols and working with other protocols like TCP streams [11]. One simple example for Buffer usage would be the following

scenario. A client connects to a server and the server sends the client an image, which is saved on disk of the client as can be seen in Listing 2 and Listing 3.

Listing 2. CLIENT APPLICATION FOR RECEIVING IMAGES

```
// client.js
const net = require('net');
const fs = require('fs');

let client = new net.Socket();
client.connect(8080, '127.0.0.1', () => {
    console.log('Connecting to server');
});
client.on('data', (imageBuffer) => {
    // write buffer to disk
    fs.writeFile('clientNode.png', imageBuffer,
            (err) => {
                if (err) throw err;
        console.log(
            "Image received and saved!");
        });
    client.end();
});
```

Listing 3. SERVER APPLICATION FOR SENDING IMAGES

```
// server.js
const net = require('net');
const fs = require('fs');

let server = net.createServer( (socket) => {
    // read an image from disk
    fs.readFile('./nodejs.png',
            (err, data) => {
        // send the buffer to the client
        socket.write(data);
    });
}).listen(8080, '127.0.0.1');
```

In Listing 3 the code for the server is similar to the previous example in Listing 1, but this time it is a TCP server to illustrate that the same principles easily apply to working with TCP streams. What the server does is, it reads an image from disk and stores it internally as an object of type *Buffer* and as soon as the image is read, streams it directly as a buffer to the client. The client listens for incoming data on the connected socket and after the incoming object is written to disk writes a message as output.

Nodes Buffer API provides several ways to construct a *Buffer* object from scratch or from existing data. After a *Buffer* object is created, this object can be manipulated in regards of the size or the encoding and can be converted back into a different type like a *String*.

D. Streams

In the previous section about buffers the server reads in an image and then sends it to the client. This is fine for files that are relatively small and can be processed rather quickly. But large files that would be read in as whole first into memory and then send over the network or saved to disk only after the whole file was received, could tie up the server's resources pretty quickly. The previous example is a good demonstration of that because the server buffers the whole image and sends this *Buffer* over to the client. Streams are a way to go around this problem by splitting the buffer into several smaller chunks and *stream* the chunks one by one to the client. To be more precise the chunks from a file that are read are send as soon as they are retrieved from disk [12]. This way the client can already start processing the incoming chunks e.g. by writing them to disk even if the whole file has not arrived yet.

There are five different streams in Node namely readable, writable, transform, duplex and classic. A readable stream can output data into a writable stream and a writable stream consumes data. Transform and duplex streams are both readable and writable and a classic stream is part of Nodes earlier API. To connect one stream to another the *.pipe()* function is used. This function is called from a readable stream and outputs its data into a writable stream. For example if *rs* is a readable source stream and *ws* is a writable destination stream the call *rs.pipe(ws)* would write the data from rs into ws. *nix users may know this behavior from the pipe command and analogous to the above example this is what would be entered into a shell `rs | ws`.

To enhance the previous example of the server and instead of buffering the whole image and sending it to the client, the following code example creates a readable stream to read in the image and then sends it to the client.

Listing 4. READING A STREAM FROM DISK

```
const fs = require('fs');
const http = require('http');

let server = http.createServer( (req, res) => {
    // read in video file from disk as a stream
    let stream =
            fs.createReadStream('./stream.mp4');
    stream.pipe(res);
    });
}).listen(8080, '127.0.0.1');
```

E. Filesystem

Node provides an API for accessing the file system of the underlying operating system via the *fs* module. This is handled by wrappers around standard POSIX functions [13]. Contrary to the other core libraries provided by Node, the *fs* module has both asynchronous and synchronous methods.

In the section about buffers, the *fs* module has already been used to read in an image as a *Buffer*. In Listing 3 the *fs.readFile()* method was used, which is an asynchronous method to read in a file from disk. The synchronous method would be *fs.readFileSync()*. The difference between these methods is that the asynchronous method gets a callback function as a parameter which gets executed after the file is read completely. While in the synchronous method no callback function is provided and the flow of execution will be sequential and won't depend on the event loop.

Listing 5. ASYNCHRONOUS AND SYNCHRONOUS FILE ACCESS

```
fs.readFile(fileName,
        (err, data) => {
        console.log("Async File Access");
});

console.log("Between functions");
```

```
fs.readFileSync(fileName);
console.log("Synchronous File Access");
```

In Listing 5 a possible output might be indeed one where *Between functions* and *Synchronous File Access* is written before *Async File Access* is printed. However the last *console.log()* will not be executed while *fs.readFileSync()* is still reading the file.

III. Express.js

Express.js (from here on only *Express*) is a framework built on top of Node. It provides a simplified API for some of Node's core functionality. It can be described as an abstraction layer over the HTTP module of Node's core API [14]. Express provides a lot of additional functionality over the HTTP module that doesn't need to be rewritten from scratch for common tasks in handling requests, defining routes or rendering static assets.

A. Middleware

Middleware is a feature of Express that deals with requests, responses and subsequent middleware functions, which will be described below. A server created in Node is usually listening on a given port for incoming requests which will be put into the event loop and handled accordingly. But this also means, that there is one single monolithic request handler that deals with every request. A middleware function is a function that splits up this monolithic request handler into several individual steps. This function can execute any code, make changes to the request and response objects, end the request-response cycle and call the next middleware function from the middleware stack to continue working on the request object. [15].

Listing 6. Simple Express Application

```
const express = require("express");
const http = require("http");

let app = express();

app.use((req, res, next) => {
    console.log("Incoming request url: "
            + req.url);
    res.end("Hello, World!");
});

http.createServer(app).listen(8000);
```

In Listing 6 the main differences between plain Node and Express are already visible. First of all, the call to *express()* creates a new Express application. This application object is used to start the server via the *createServer(app)* function call. The middleware function can be seen in the *app.use(callback)* function call. It takes three parameters namely the request object, the response object and the next middleware function. The names of the parameters can be chosen arbitrarily but using the above names is the convention and it is recommended to do so [15].It is also important to note, that it is possible to attach several middleware functions to the app with the *app.use(callback)* function call. In the example above only the requests url was printed to standard output and then the response containing the String *Hello, World!* was sent back. But it might also be of use to first do some form of authentication on the incoming request, before continue processing it as can bee seen in Listing 7.

Listing 7. Creating Express Middleware Functions

```
app.use( (req, res, next) => {
    let isAuth = auth(req);
    if (isAuth) {
        next();
    } else {
        res.end(error);
    }
});

app.use((req, res, next) => {
    console.log("Incoming request url: "
            + req.url);
    res.end(data);
});
```

In the above code listing, it is assumed that *auth(req)* is a function that processes the request object for some authentication and returns a boolean value. If the request passes the authentication, the next middleware function is called with the *next()* function call. Otherwise, it is assumed that some kind of error is send in the response. The above order of the middleware functions is important for the order of execution. In the example, the authentication handling was loaded into the application context before sending the data. If the methods would have been switched around the data would be sent without any authentication on the server, because the *next()* function wouldn't have been called so that no next middleware would be executed. This can also be seen in the authentication middleware function. If the request wouldn't authenticate the request-response cycle would be shut down without getting the next middleware handler.

B. Routing

A web application has rarely only one page that is linked to one URL. Even if it is a single page application, there are usually pages that are addressed from a specific URL. Even parameters for queries or user IDs are often contained within a URL. When a server process an incoming request it should route this request to the desired part of the application. For example when a request comes in for an about page, the server should not serve the main page or any other page but the requested.

Express has built in functions for handling incoming HTTP requests with a specified URI that will be mapped to a request handler. For example an incoming GET request can be explicitly handled by the *app.get(path, callback)* function. The *get()* function is one of the *app.METHOD()* functions, where *METHOD* stands for a HTTP method. Some of the most important methods are *get()*, *post()*, *put()* and *delete()*. With these methods an express app can handle the basic Create Read Update Delete(CRUD) operations.

The arguments *path* and *callback* stand for the URI and respectively for one or several middleware functions that will handle the incoming request [16].

To build on the previous examples, it would be possible to handle the authentication, only when someone visits a log in page which is located at the */login* end point.

Listing 8. ROUTING AND MIDDLEWARE

```
let isAuthenticated = function
(req, res, next) {
    let isAuth = auth(req);
    if (isAuth) {
        next();
    } else {
        res.end(error);
    }
} );

app.get('/login', isAuthenticated,
    (req, res, next) => {
        res.end(data);
});
```

In Listing 8 the routing works only for the */login* URI. Every other incoming URI will be ignored and therefore it's necessary to create handlers for the individual pages or endpoints that could be visited by some form of a user and also a handler for incoming URIs that are not supported by the application.

Listing 9. USGING ROUTING ORDER AND MIDDLEWARE ORDER FOR REQUEST WORKFLOW

```
app.use( (req, res, next) => {
    console.log("Request URL: " + req.url);
    console.log("Request Date: " + new Date());
    next();
});

app.get('/', (req, res, next) => {
        /* show homepage */
});

app.get('/login', isAuthenticated,
    (req, res, next) => {
        /* proceed to login page */
    });

app.get('/users/:userid',
    (req, res, next) => {
        /* Parse :userid and show user's page*/
});

app.use( (req, res, next) => {
        res.status(404).send(
          "404 Page not found!");
});
```

Adding Listing 9 to the previous code examples, the application deals now with incoming GET requests for the homepage indicated by the root slash, for the login page as well as for a specific user's page. Every other HTTP Method or URI will be handled by a 404 page not found response. The function dealing with users id's will accept parameters in the form of */users/2* or */users/some_user*. To capture users parameters more strictly it is possible to use regular expression instead of *:userid* to enforce e.g. only numerical user id values in the query parameter [14]. Also because of the ordering of the middleware functions at first the logging middleware function will be executed on every incoming request followed by the HTTP methods. With that a basic server logging functionality is built using Express' middleware.

C. Static Files

A rich internet application usually has some form of static files that need to be send to the user. This static files might be HTML, CSS or JavaScript files e.g. containing a template that might be filled with data on the client side. But even if they are populated on the client side the back end server needs to able to provide them these very files in some kind of way. It is possible to do that in plain Node but Express has a helper functions regarding just that.

First, the path to the directory where the static files are located needs to be configured. To prevent failures between different operating system it's helpful to use Node's *path.resolve()* function from the *path* module. After that, the static file directory can be attached to the application's context just like any other middleware function via the *app.use()* and Express' *express.static()* function.

Listing 10. SERVING STATIC FILES

```
const staticPath = path.resolve(__dirname,
        "static");

app.use(express.static(staticPath));
```

Same as with other middleware functions several paths to different static file directories might be passed to the context and can be processed during the process of a request. Here again, the order is of importance for using the *app.use()* function. The path that is listed first will be first resolved and if on this path a match for a file is found, the following paths won't be further considered for that request. This means, if there exist two files with identical names, but are in two different directories, as soon as a first match is found the second file will not be send to the request, even if it was initially intended [14].

IV. THE FUTURE OF NODE

Regarding Node's stability and maturity a solution for future releases of Node was found. Under the umbrella of the Node.js Foundation a Long Term Support (LTS) release cycle was introduced. This ensures that Nodes API will be reliable at least between the LTS versions and makes Node even more interesting for various enterprises.

From a technical view point Node's future lies therein to bring an updated V8 to Node and also incorporate the ECMAScript 2016 Language Specification and future specifications. As of this writing, the last update to Node did exactly that namely bringing V8 to version 5.4 and implementing operators and Object properties from ES2016 and ES2017 [17]. From Nodes Roadmap one of the most immediate priorities is debugging and tracing [18]. These areas will be improved by further iterating on *AsyncWrap* and a debugger from Chrome which will be able to work as a standalone debugger with Node even without a browser. *AsyncWrap* is an API for creating hooks into objects and as such provide tracing information about the life cycle of those objects. Since event driven programming has pitfalls and difficulties in debugging, e.g. nested callbacks are difficult to debug, because their order of execution cannot be predicted as a synchronous execution, this is a feature that is very anticipated in Nodes community.

V. Conclusion

Node makes JavaScript a viable choice to create scalable back end web applications with its non-blocking IO operations. The choice of the language might be irritating for some, since JavaScript is mostly known as a language that lives only in the browser. Previously this language was essentially used to create dynamic UIs for web applications. But since most developers who are developing web applications tend to know at least some JavaScript the learning curve for adapting a complete JavaScript stack from back end to front end declines.

On top of that, the simplicity of creating highly scalable applications was one of the key features that made Node very popular. With only a few lines of code a web server can be created from scratch. With the help of the quickly expanded ecosystem around Node and libraries such as Express it is easy to grow this simple server to a fully functional application that can scale very well for every kind of scenario.

References

[1] *Node.js*. URL: https://nodejs.org/en/ (visited on 12/04/2016).

[2] "Node at LinkedIn: The Pursuit of Thinner, Lighter, Faster". In: *Commun. ACM* 57.2 (Feb. 2014), pp. 44–51.

[3] *Introducing JSON*. URL: http://json.org/ (visited on 01/04/2017).

[4] S. Thompson and M. Hablich. *V8 Wiki*. URL: https://github.com/v8/v8/wiki (visited on 11/16/2016).

[5] Thibault Laurens. *How the V8 engine works?* URL: https://thibaultlaurens.github.io/javascript/2013/04/29/how-the-v8-engine-works/ (visited on 11/27/2016).

[6] Daniel Clifford. *Breaking the JavaScript Speed Limit with V8*. URL: https://v8-io12.appspot.com (visited on 11/27/2016).

[7] S. Tilkov and S. Vinoski. "Node.js: Using JavaScript to Build High-Performance Network Programs". In: *IEEE Internet Computing* 14.6 (Nov. 2015), pp. 80–83.

[8] A. Young and M. Harter. *Node.Js in Practice*. Manning Publications Company, 2015.

[9] *HTTP — Node.js v6.9.1 Documentation*. URL: https://nodejs.org/dist/latest-v6.x/docs/api/http.html (visited on 11/24/2016).

[10] K. Simpson. *You Don't Know JS: Async &Performance*. Mar. 2015. URL: https://github.com/getify/You-Dont-Know-JS/blob/master/async%5C%20%5C&%5C%20performance/README.md#you-dont-know-js-async--performance (visited on 11/20/2016).

[11] *Buffer — Node.js v6.9.1 Documentation*. URL: https://nodejs.org/dist/latest-v6.x/docs/api/buffer.html (visited on 11/24/2016).

[12] J. Halliday. *stream-handbook*. URL: https://github.com/substack/stream-handbook/ (visited on 03/12/2016).

[13] *File System — Node.js v6.9.1 Documentation*. URL: https://nodejs.org/dist/latest-v6.x/docs/api/fs.html (visited on 11/25/2016).

[14] E. M. Hahn. *Express in Action*. Manning Publications Company, 2016.

[15] *Writing middleware for use in Express apps*. URL: http://expressjs.com/en/guide/writing-middleware.html (visited on 11/13/2016).

[16] *Express 4.x - API Reference*. URL: http://expressjs.com/en/4x/api.html (visited on 11/13/2016).

[17] Michael Zasso. *Node.js v7 has updated V8 to 5.4*. Dec. 2016. URL: https://nodejs.org/en/blog/community/update-v8-5.4/ (visited on 04/12/2016).

[18] *Node.js Roadmap*. URL: https://github.com/nodejs/node/blob/master/ROADMAP.md (visited on 10/12/2016).

HTML5 Local Storage

David Specht
Carl von Ossietzky University of Oldenburg, Germany
Department of Computer Science
david.specht@uni-oldenburg.de

***Abstract*—Mit der Veröffentlichung von HTML5 im Oktober 2014 wurde eine neue Form der lokalen Speicherung von Informationen im World Wide Web, welches sich unter dem Begriff "Local Storage" verbirgt, präsentiert. In diesem Aufsatz wird zunächst die geschichtliche Entwicklung des lokalen Speicherns von Informationen im World Wide Web beschrieben. Ebenso wird in diesem Aufsatz aufgezeigt, inwieweit die Entwicklung dieser Technologie notwendig bzw. unvermeidbar war. Zusätzlich wird ein Überblick über den Aufbau und die Funktionsweise des "Local Storage" geliefert. Im Anschluss wird anhand ausgewählter Beispiele überprüft, inwieweit die Verwendung des "Local Storage" gängige Praxis ist und eine Zukunftsprognose erstellt.**

Keywords*—*Browserspeicher, Web Storage, localStorage

I. Speicherformen in der Entwicklung des World Wide Web

Die Entwicklung von Websites für den Otto-Normal-Verbraucher war ursprünglich nicht zu mehr gedacht als zur Darstellung von Informationen. Diesen Zeitraum nennt man üblicherweise das Web 1.0. Die Technologien waren simpel. Die Browser benötigten mit der Darstellung von Websites kaum mehr Technologien als die "Hypertext Markup Language" (HTML) und die "Cascading Style Sheets" (CSS). Diese Technologien haben sich über Jahre hinweg durchgesetzt. HTML dient dabei als Auszeichnungssprache für Elemente innerhalb der Website und CSS als dessen Definition für die graphische Darstellung. Innerhalb HTMLs war keine Speicherung von Daten notwendig. Die Programmiersprache JavaScript, die 1995 veröffentlicht wurde[1], galt als allgemeine Sprache für Browser, sodass die Funktionalität der Websites ihren Grundstein erhielt. Mit JavaScript war es Anbietern von Websites möglich die Seite genauer zu spezifizieren, indem der Code zum Client mitgesendet und dieser je nach Bedarf, ausgeführt wurde. Im Zeitverlauf erfolgte eine Weiterentwicklung von Internet-Technologien. Diese wurden effizienter und umfangreicher. Die Nutzung beschränkte sich nicht nur auf den Konsum statischer Websites. Dem User wurde die Möglichkeit geboten, neben dem Senden von Anfragen auch mit der Website zu interagieren. Diese Entwicklung wurde mit dem Begriff Web 2.0 beschrieben. Neben der Möglichkeit Daten hochzuladen wurden auch Foreneinträge und Login-Systeme ermöglicht. Mit der Entwicklung von Login-Systemen bestand erstmals in der Geschichte des Webs die Notwendigkeit, Daten des Servers im Browser zu speichern. Schließlich sollte der Konsument auf einer Website zwischen unterschiedlichen Webpages navigieren können, ohne das er sich erneut anmelden muss. Gelöst wurde diese Problemstellung mit HTML-Cookies, deren Funktionsweise im folgenden geschildert wird. Im Oktober 2014 wurde mit der Veröffentlichung von HTML5 eine alternative Speicherungsform präsentiert: "Local Storage". Den Schwerpunkt dieses Aufsatzes bildet die Analyse von Local Storage. Neben der Funktionsweise wird auch die Sicherheit analysiert. Anhand von ausgewählten Beispielen wird zusätzlich überprüft, inwieweit Local Storage in der Praxis verwendet wird und eine Zukunftsprognose geliefert.

A. Cookies

HTML-Cookies dienten ursprünglich dazu, dass der Client, der sich auf einer Website angemeldet hat, mit jeder weiteren HTTP-Anfrage den zur Website zugeordneten Cookie mitsendet. Dieser Cookie enthält die Informationen über den Client, so definiert, dass dieser eindeutig zuzuordnen ist. Folglich kann der User nun mit der Navigation innerhalb einer Website über verschiedene Webpages angemeldet bleiben, ohne dass er sich erneut anmelden muss.
Technisch betrachtet umfasst ein Cookie einen Speicherbereich vom Webbrowser, der üblicherweise bis zu 4kB pro Cookie großsein kann.[2]. Pro Domain sind 20 Cookies möglich[2]. Ein Cookie ist durch einen Namen und dem zugeordneten Wert aufgebaut, beide als Zeichenketten (Strings). Cookies werden ausschließlich clientseitig verwaltet und können mit JavaScript beeinflusst werden. So kann der User theoretisch selbst entscheiden, welche Cookies er behalten möchte, welche gelöscht werden sollen usw. Typischerweise haben Cookies kein Ablaufdatum, sofern nicht anders vorgesehen. Bestimmt der JavaScript-Code einen Löschvorgang für einen bestimmten Zeitraum bzw. Zeitpunkt, so werden die Cookies gelöscht, falls nicht, bleiben Cookies auf unbegrenzte Zeit gespeichert.
Nun ist mit der Speichergröße und Anzahl der Cookies pro Domain weitaus mehr möglich als "nur" eine Speicherung der Session. Der übrige Speicherplatz kann je nach Bedarf mit unterschiedlichen Daten gefüllt werden. Durch den rasanten Zuwachs der Websites (von 2011 bis 2014 fast verdreifacht[3]) und dessen Ansprüche an die Funktionalität, wurden viele Schnittstellen (APIs), für unterschiedliche Technologien innerhalb der Web-Welt entwickelt. Dieses ermöglichte den Anbietern technologieübergreifende Funktionalitäten zu bieten. Dieser Zeitraum der Vernetzung wird auch Web 3.0 genannt. Die Nutzung von Cookies wird damit auf verschiedene Bereiche ausgeweitet:

1) Die Verwendung von Applikationen bedarf häufig einer Speicherung des Verlaufes. Ein klassisches Beispiel hi-

erfür sind Spiele. Eine Website kann damit den Spielstand des Users lokal abspeichern und erspart sich damit Netzwerk- und Rechenauslastung.
2) Es sollen HTTP-Anfragen an den Server geschickt werden, während das Netzwerk ausfällt, was häufig bei Chat-Applikationen passieren kann. Hierbei speichert eine JavaScript Funktion die Anfrage beim Scheitern der Übertragung in einem Cookie zwischen und probiert eine erneute Übertragung nach regelmäßigen Zeitabschnitten bzw. bei neuer Session.
3) Ein Warenkorb ist für jeden Online-Shop wichtig. Dieser ist dem User eindeutig zugeordnet und kann sessionübergreifend lokal beim User gespeichert werden.

Nicht selten kam es vor, dass der Cookie-Speicher ausgelastet wurde und die Domain keinen weiteren Speicherplatz zur Verfügung hatte. Im Zuge der HTML Entwicklung zu ihrer fünften Version (HTML5), stellte die Entwicklungsgruppe des "World Wide Web Consortiums" (W3C) die HTML-API "localStorage" bereit. Diese Schnittstelle sollte das genannten Problem lösen und dem Web neue Möglichkeiten der clientseitigen Informationsspeicherung liefern.

II. LocalStorage

Local Storage, auch "Super-Cookies" genannt, wird im Allgemeinen als Nachfolger der Standard-Cookies auf lokaler Ebene betrachtet. Während Cookies bei der HTTP-Anfrage mit an den Server übertragen werden, ist das Local Storage nur clientseitig verfügbar und kann mit JavaScript beeinflusst werden. Local Storage bietet mindestens 5MB (Firefox) für jede Domain im zugeordneten Speicherbereich. Die Größe ist allerdings abhängig vom Browser. Auch Local Storage selbst wird erst von bestimmten Browser-Versionen unterstützt[4].

A. Interface Storage

Local Storage selbst unterliegt dem Interface "Storage". Dieses gibt die Methoden vor, die Local Storage anwendet. Des Weiteren existiert eine zusätzliche Klasse, die dem Storage Interface unterliegt. Die Klasse wird als "Session Storage" bezeichnet. Grundsätzlich bestehen keine Unterschiede im Aufbau zwischen dem Local Storage und Session Storage. Allerdings beschränkt sich Session Storage auf die Laufzeit einer Session. Wenn die Website geschlossen wird, wird auch das Session Storage gelöscht. Beim Aufruf einer Website wird erst überprüft, ob bereits Speicherplatz für die Domain reserviert wurde (beispielsweise beim Öffnen eines neuen Tabs). Wenn die Domain bereits Speicherplatz beansprucht, wird dieser weiterhin genutzt. Local Storage und Session Storage befinden sich beide in dem Window-Objekt der HTML-DOM[5]. Um mit dem JavaScript Code an das Local Storage Objekt zu kommen schreibt man:

window.localStorage

Analog kommt man an das Session Storage Objekt heran.

B. Methoden und Attribute

Aus dem Storage Interface und damit geltend für Session Storage und Local Storage existieren Methoden und Attribute, die zur Bedienung des Speichers dienen oder diesen beschreiben. Generell gilt für die beiden Storage Klassen, wie für Cookies auch, dass sowohl "Key", als auch "Value" im Datentyp "String" gespeichert werden. Dabei gilt Key als Identifikator für Value. Gespeichert werden die Key-Value Paare in der Reihenfolge der Erstellung. Dabei bekommt jeder Eintrag einen Index.
Für die folgenden Methoden gehen wir davon aus, dass sich im Local Storage bereits ein Eintrag mit dem Key "Schluessel" und dem Value "beliebiger Eintrag" befindet. Alle Methoden und Attribute sind analog auf Session Storage anwendbar[5].

Local Storage Einträge:

Index	Key	Value
0	"Schluessel"	"beliebiger Eintrag"

1) Attribut: length: Length als "unsigned long" definierte Zahl gibt die Anzahl der Key-Value Paare aus, die gespeichert werden. Gleichzeitig ist das length-Attribut nur lesbar und nicht überschreibbar[5].

window.localStorage.length

Output: 1

2) Methode: key(): Die Methode key gibt den Namen des Eintrags des übergebenen Indexes zurück. Übergibt man der Funktion einen Index vom Typ Integer, bekommt man den Key als String vom entsprechenden Key-Value Paar zurück [5].

window.localStorage.key(0)

Output: "Schluessel"

3) Methode: getItem(): Die Methode getItem gibt den Wert des Eintrages als Referenz des übergebenen Keys zurück. Übergibt man der Funktion einen Key vom Typ String, bekommt man den Value als String vom entsprechenden Key-Value Paar zurück [5].

window.localStorage.getItem("Schluessel")

Output: "beliebiger Eintrag"

4) Methode: setItem(): Die Methode setItem setzt den Wert eines Eintrages anhand des übergebenen Keys auf den Wert des übergebenen Value, wenn es keinen Eintrag mit dem Key gibt, wird ein neuer erschaffen. Also wird erst überprüft, ob es einen Eintrag gibt, bei dem der übergebene Key-String einem Key innerhalb der Local Storage gleicht. Wenn dies der Fall ist, wird der Value innerhalb des Key-Value Paares überschrieben. Wenn dies nicht der Fall ist, wird ein neuer Eintrag geschaffen, der das Key-Value Paar nach den übergeben Strings setzt. Dabei wird der Methode erst der Key gegeben und anschließend der Value. Beim Erschaffen eines neuen Eintrags wird diesem der Index 0 gegeben. Der Index aller anderen Einträge wird um 1 erhöht [5].

window.localStorage.setItem("Schluessel","anderer Eintrag")
window.localStorage.setItem("new", "newEntry")

Hierbei wird erst der Eintrag mit dem Key "Schluessel" überschrieben und anschließend ein neuer Eintrag mit dem Key "new" und dem Value "newEntry" an Index 0 geschaffen. Der Eintrag mit dem Key "Schluessel" bekommt Index 1.

Index	Key	Value
0	"new"	"newEntry"
1	"Schluessel"	"anderer Eintrag"

5) Methode: removeItem(): Die Methode removeItem löscht einen Eintrag anhand des übergebenen Keys. Bei allen Einträgen, die einen höheren Index haben als der zu löschende Eintrag, wird der Index um 1 verringert [5].

window.localStorage.removeItem("new")

Dies führt dazu, dass der Eintrag mit dem Key "new" gelöscht wird.

Index	Key	Value
0	"Schluessel"	"anderer Eintrag"

6) Methode: clear(): Die Methode clear löscht alle Einträge innerhalb des Local Storage.

window.localStorage.clear()

Index	Key	Value

III. Nutzung des Local Storage

A. Das String Problem und JSON

Die lokal Datenspeicherung des Local Storage bietet für die Speicherung lediglich den Datentypen des Strings. Wenn man nun das Beispiel einer Warenkorbfunktionalität betrachtet, so müssen die Preise verschiedener Waren aufeinander addiert werden, damit man den Gesamtpreis erhält. Für die Umwandlung von Zahlen existieren in JavaScript verschiedene parse-Funktionen, die bei Übergabe eines Strings eine Zahl zurückgeben. Es gibt die parse Funktionen *parseInt()* und *parseFloat*, um Strings in Zahlen zu definieren. Wie üblich gibt parseInt einen ganzzahligen Zahlenwert zurück und parseFloat einen dezimalkomma-Zahlenwert zurück.
Die Warenkorbfunktionalität kann nun die Preise verschiedener Produkte zusammen addieren und ihren Gesamtpreis angeben. Wir wählen als Beispiel für den Warenkorbinhalt ein Paar Schuhe der Marke Adidas, der ProduktID 17, der Größe 40, den Preis 50 und die Farbe schwarz. Diese Daten müssen nun im Local Storage abgespeichert werden. An dieser Stelle tritt nun das Problem auf, dass alle diese Attribute als Strings gespeichert werden müssen. Dafür bietet JavaScript den Datentypen JSON. JSON ist entwickelt worden, um verschiedene Attribute innerhalb eines Objektes zu speichern. Anstatt, dass wir die oben genannten Attribute alle einzeln abspeichern, erstellen wir nun ein JSON Objekt, was alle Daten enthält [6].

var saveWarenkorbEintrag = {
"Marke" : "Adidas",
"ProduktID" : 17,
"Größe" : 40,
"Preis" : 50,
"Farbe" : "schwarz"}

Um dieses Objekt nun im Local Storage abspeichern zu können, muss es als String abgespeichert werden. Dafür hat JSON die Funktion *stringify()*[6].

window.localStorage.setItem("Warenkorbeintrag1",
JSON.stringify(saveWarenkorbEintrag)

Nun ist der gesamte Eintrag im Local Storage abgespeichert. Um an die Daten heranzukommen, die nun als String abgespeichert sind, liefert JSON die Funktion *parse()* [6].

var loadWarenkorbEintrag = JSON.parse(window.
localStorage.getItem("Warenkorbeintrag1"))

B. Local Storage auf bekannten Websites

Statistiken zur Nutzung von Local Storage sind zur Zeit kaum existent. Es bleibt die Frage, ob die Anbieter verschiedener Websites Local Storage nutzen. Um mir ein Bild davon zu machen, suchte ich mir einige Websites heraus und überprüfte, wie viele Local Storage und Session Storage Einträge ich in meinem Browser zu der jeweiligen Domain finde. Dafür liefert der Browser Firefox zu den Entwickler-Tools eine Konsole, in der man JavaScript Befehle ausführen und JavaScript Ausgaben ansehen kann. Mit dem length Befehl betrachte ich die Anzahl an Local Storage Einträgen. Zuvor gehe ich sicher, dass ich keine gespeicherten Einträge, die ich aus früheren Besuchen der Websites habe, indem ich den clear Befehl ausführe. Die gewählten Websites sind *google.de*, *facebook.de*, *amazon.de*, *ebay.de* und *uni-oldenburg.de*.

Local Storage Einträge:

Website	localStorage.length	sessionStorage.length
Google	0	16
Facebook	2	1
Amazon	0	4
Ebay	0	0
Uni Oldenburg	0	0

Die erste Betrachtung der Ergebnisse zeigt, dass zumindest die Technologie bereits genutzt wird. 50% der gewählten Websites nutzen Local Storage oder Session Storage. Auffällig ist, dass es mehr Session Storage Einträge gibt, als Local Storage Einträge. Nun betrachten wir den Fall, dass man sich schon einige Zeit auf der Website umgeschaut hat und einige Funktionalitäten nutzte. Ich klicke mich durch unbestimmte Webpages innerhalb der Webseite, wobei es keine Rolle spielt, wo ich genau lande, da ich nur überprüfen möchte, dass Local

Storage bzw. Session Storage genutzt wird und zeige hier die Ergebnisse.

Local Storage Einträge:

Website	localStorage.length	sessionStorage.length
Google	8	174
Facebook	4	1
Amazon	0	15
Ebay	1	0
Uni Oldenburg	0	0

Man erkennt, dass alle Websites, außer die der Uni Oldenburg, Local Storage bzw. Session Storage vermehrt nutzen. Gerade Google zeigt mit 174 verschiedenen Session Storage Einträgen dieser Technologie eine Menge Aufmerksamkeit. Bewusst wählte ich bei Amazon ein beliebiges Produkt und legte dieses in meinen "Warenkorb". Gerade dort wo die Cookies mit ihrem geringen Speicherplatz eine schlechte Alternative für die Funktionalität eines Warenkorbs liefern, erwartete ich, dass Local Storage genau dies aufgreift. Tatsächlich blieb der Local Storage trotz nicht-leerem Warenkorb leer. Auch nachdem ich meinen Browser geschlossen habe, um anschließend Amazon wieder aufzurufen, blieb das Local Storage leer und mein Warenkorb gefüllt.
Ähnliches stellte ich auch bei Ebay fest. Die Website wiederholte seine Empfehlungen nach meiner Suche, gerichtet nach den aufgerufenen Produkten, auch nachdem ich mein Local Storage durch die clear Funktion gelöscht habe. Man hätte auch hier vermuten können, dass Ebay meine Interessen lokal abspeichert, um den Traffic zu verringern.
Auch wenn die Verwendung vom Local Storage, im Besonderen bei Online-Shops, gering erscheint, so lässt sich dies simpel erklären. Gerade Online-Shops sind daran interessiert, Nutzerdaten auf ihren eigenen Systemen zu sammeln um ihre Verkaufsstrategien analysieren und anpassen zu können. Wenn das Userverhalten lokal auf dem Rechner abgespeichert wird, kommt der Server nicht ohne Zusatzfunktionalitäten an die Daten heran. Folglich erscheint es sinnvoll genau dafür Cookies zu verwenden, da diese bereits mit der HTTP-Anfrage an den Server gesendet werden und dieser sie direkt verarbeiten kann. Es ist zu beachten, dass die Local Storage Technologie mit HTML5 publiziert wurde. Auch wenn Browser sehr regelmäßig ihre Versionen aktualisieren, sollte gewährleistet sein, dass alle Nutzer, unabhängig von der Browser Version, die Website nutzen können.

IV. Sicherheit

Local Storage Einträge sind an den Hostnamen, das Protokoll und an den Port gebunden. Damit ist es Website A nicht möglich, unerlaubt auf die Local Storage Einträge von Website B zuzugreifen. Hierbei ist auch zu beachten, dass eine Website, die von verschiedenen Anbietern geteilt wird (bzw. von verschiedenen Usern modifiziert werden kann) sich auch die Local Storage Einträge teilt. Bietet Person A beispielsweise die Webpage *Beispiel.de/A* und Person B *Beispiel.de/B* an, so können sie gegenseitig die Local Storage Einträge lesen und schreiben. Dies gilt allerdings auch für Cookies. Dadurch, dass Local Storage Einträge immer nur clientseitig bleiben und Cookies mit HTTP Anfragen an den Server gesendet werden können, bleibt die Geheimhaltung vor dem Abhören erhalten. Nichtsdestotrotz können die Einträge mit einem Virenangriff wie alle anderen Dateien auf dem Computer ausgelesen werden.

V. Fazit

Durch den rasanten Anstieg an Websites in den letzten Jahren wurden Cookies zur lokalen Datenspeicherung im World Wide Web entwickelt. Die Idee, die sich zunächst hinter Cookies verbirgt, war die Identifikation des Clients zur Erhaltung einer Session. So konnte der User mit der Navigation innerhalb einer Website über unterschiedliche Webpages weiterhin angemeldet bleiben, ohne dass eine erneute Anmeldung auf jeder Webpage erforderlich war. Mit der Entwicklung hin zum Web 3.0 wurde auch der Nutzungsumfang von Cookies erweitert. Durch diese Funktionsausweitung ergaben sich Probleme, da der Cookie-Speicher ausgelastet wurde und sich für die jeweilige Domain kein weiterer Speicherplatz bot. Mit der Entwicklung von HTML5, und damit dem Local Storage, wurde die Problemstellung der clientseitigen Informationsspeicherung hinreichend gelöst. Während Cookies bei der HTTP-Anfrage ebenfalls an den Server übertragen werden, ist das Local Storage nur clientseitig verfügbar und kann mit JavaScript angesteuert werden.
Betrachtet man den Verlauf der Speichergröße auf lokaler Ebene bei den beiden vorhandenen Technologien der Cookies und des Local Storage, so wird die unumgängliche Weiterentwicklung von Cookies zu Local Storage deutlich. Während bei Cookies der Gesamtspeicher 4kB$\cdot$ 20 = 800kB pro Domain belegt werden kann, wuchs es beim Local Storage zu mindestens 5MB, mit dem Session Storage zu mindestens 10MB. Dennoch entwickelt sich die Informationstechnik nicht linear und deshalb ist, zumindest vorerst, keine Weiterentwicklung des Local Storage zu erwarten. Sicher ist jedoch, dass irgendwann eine kommen wird, in der vielleicht mehr Speicherarten zur Verfügung gestellt werden als nur Strings.

References

[1] Shelley Powers. *Einführung in JavaScript*. O'Reilly Germany, 2007.

[2] Dr.-Ing. Dietrich Boles. "JavaScript". Vorlesungsfoliensatz JavaScript zum Modul Internet-Technologien der Uni Oldenburg. 2015.

[3] *Anzahl der Webseiten weltweit in den Jahren 1992 bis 2015*. Statista. 2016. URL: https://de.statista.com/statistik/daten/studie/290274/umfrage/anzahl-der-webseiten-weltweit/.

[4] *HTML5 Local Storage*. W3Schools. URL: http://www.w3schools.com/html/html5_webstorage.asp.

[5] *Storage*. Mozilla Developer Network. URL: https://developer.mozilla.org/en-US/docs/Web/API/Storage.

[6] im Bereich JavaScript. W3Schools. URL: http://www.w3schools.com/.

Über GraphQL

Hendrik Jordan
Carl von Ossietzky University of Oldenburg, Germany
Department of Computer Science
hendrik.jordan@uni-oldenburg.de

***Abstract*—Einführung und Beispiele in GraphQL**

***Keywords*—GraphQL, Query Language, V**

I. Einführung

GraphQL ist eine Spezifikation, die es Anwendungen ermöglicht sehr genau spezifizierte Anfragen an einen Server zu schicken. [1] Zu den Vorzügen zählen, dass die Rückmeldung der Anfrage ähnelt und dass die Technik nicht auf einer Sprache beruht, sondern als Spezifikation in vielen Programmiersprachen umgesetzt werden kann und wurde. [1] GraphQL ist dabei keine Programmiersprache, sondern ein Schema um Anfragen an Server, die dieses Schema unterstützen, zu senden. GraphQL stellt also eine standartisierte Schnittstelle zur verfügung. GraphQLs Vorzüge im Vergleich zu herkömmlichen REST APIs sind vielfältig, heraus sticht jedoch die Möglichkeit, mehrere REST Aufrufe zu einem zu kombinieren, was die Verbindung besser ausnutzt und Zeit spart, da nicht mehrere aufeinander aufbauende Anfragen jeweils gesendet und abgewartet werden müssen.

A. Einbindung von GraphQL

GraphQL ist nicht an eine spezifische Datenbank oder Speicherung der Daten gebunden, was bedeutet, dass GraphQL auf praktisch jedem Server mit wenig Aufwand eingesetzt werden kann. Dadurch, dass in einem GraphQL Schema selber definiert werden muss, wie auf ein Attribut zugegriffen wird, kann man jedes Objekt sehr einfach an die Datenhaltung jedes Servers anpassen. Gleiches gilt für die Mutations, also Anfragen, die Daten schreiben. Jede dieser Mutations benötigt eine eigene Implementierung, sodass der Server selber definiert, was mit den Daten passiert. Es existieren Bibliotheken für viele Programmiersprachen, die das Aufsetzen und Benutzen vereinfacht, doch generell liefert GraphQL JSON Daten zurück [2], also benutze ich auch hier JSON.

II. Nutzung von GraphQL

A. Abfrage mit GraphQL

Eine Anfrage an einen Server könnte mit GraphQL so aussehen:

```
query {
  concert {
    name
    id
    place
  }
}
```

Woraus ein (korrekt aufgesetzter) Server antworten könnte:

```
{
  "data": {
    "concert": {
      "name": "Peter Lustig in Concert"
      "id" : 27
      "place": "Bauwagen"
    }
  }
}
```

In diesem Falle wird also nach allen Daten mit ”concert” gefragt, von diesen wollen wir nun den Namen, die ID und den Ort wissen. Bei dieser Abfrage existiert nur ein Konzert auf dem Server, aber normalerweise würde man so mehr Datensätze zurückbekommen. In dieser Ausarbeitung benutze ich die ”GraphQL Schema Sprache” [3], da GraphQL sprachunabhängig ist und die Schemasprache sehr einfach zu lesen ist. Der Server, der in dieser Sprache geschrieben ist, müsste für das Konzertbeispiel folgende Datenstrukturen und die dazugehörigen Funktionen definieren:

```
type Query {
  concert : Concert
}

type Concert {
  id: ID
  name: String
  place : String
}

function Query_concert(request) {
  return request.getConcert();
}

function Concert_name(concert) {
  return concert.getName();
}

function Concert_place(concert) {
  return concert.getPlace();
}
```

Wie zu sehen ist, definiert GraphQL eine Baumartige Datenstruktur und dazugehörige Getter/Setter für die Felder, die nun abgefragt werden können. Die Funktionen sich Serverspezifisch, sind je nach Datenhaltung unterschiedlich und beinflussen GraphQL nicht, also werden sie in dieser Ausarbeitung weggelassen.

Eine REST-API könnte nun genauso die Daten zurückliefern, etwa an einem Endpoint ”Concert”, jedoch kann REST nur alle Felder komplett zurückliefern, während GraphQL wählen kann welche benötigt werden. Die Vorteile, die GraphQL liefert werden jedoch später deutlich.

B. Ändern von Daten mit GraphQL

Genauso wie von dem Server mit einer Query Daten verlangt werden, können auch Daten bereit gestellt werden. Dazu muss der Server einen Mutation Type bereitstellen. Hier wird auch noch ein Input definiert, sodass komplexe Daten, wie etwa ein komplettes Konzert akzeptiert werden können. Hierbei wird angenommen, dass die ID eines Konzertes automatisch vergeben wird und deshalb nicht mit angegeben ist.

```
mutation newConcert($con: Conc) {
  createConcert(Conc : $con) {
    name
    place
    }
}
input Conc {
  name: String
  place: String
}
```

Diese Mutation gibt die Möglichkeit, ein neues Konzert auf dem Server zu erstellen. Um ein Konzert zu erstellen könnte eine Anfrage so aussehen:

```
mutation {
  "con": {
    "name": "Demetori",
    "place": "Tokyo, Japan"
  }
}
```

Es würden der Name und Ort wieder zurückgegeben werden. Die Möglichkeit, die eingetragenen oder abgeänderten Daten zu beobachten ist sehr nützlich, da sofort der Einfluss auf dem Server ersichtlich und überprüfbar ist. Das REST Gegenstück zu Mutations ist der POST befehl, aber mit GraphQL könnten mehrere Mutations in eine Abfrage gesteckt werden. Wichtig ist, dass man gleichzeitig Daten abfragen kann, aber verändert werden sie nacheinander.

C. Spezifische Abfragen

GraphQL unterstüzt natürlich auch das spezifische Abfragen von Daten. Eine Abfrage die die Daten des Konzertes mit dem Namen ”Demetori” heraussucht:

```
query {
  concert(name: "Demetori") {
    name
    id
    place
  }
}
```

Spezifizierende Eigenschaften werden in Klammern hinter dem Objekt geschrieben, von welchem ein Attribut abgefragt werden soll, hier der Name eines Konzertes. Wobei hierbei nun nur Werte zurückgegeben werden die zu dem Konzert gehören und nicht alle Konzerte auf dem Server.

D. Serverkonfiguration

Die Serverseite einer GraphQL-Anwendung ist natürlich genau so wichtig wie die Clientseite, deshalb hier ein kurzes Beispiel, wie ein Anwendungsfall in ein GraphQL Schema umgesetzt werden könnte. Dazu erweitern wir das bisherige Schema um Tickets, die zu einem Konzert gehören. Jedes Ticket hat eine Id, einen Preis, eine Sitznummer und den Namen des Käufers. Da zur erstellung eines Tickets ein Konzert benötigt wird, benutzt dieses Beispiel das Konzept von Variablen. Eine Variable lässt nur Werte von dem Server zu, hier ein Konzert. Beim laden der Website könnte ein Auswahlfeld mit allen verfügbaren Konzerten gefüllt werden, sodass nicht beliebige Werte eingegeben werden können. Eine Realisierung als GraphQL Schema könnte nun so aussehen:

```
type Query {
  concert:Concert
}

type Concert {
  id: ID
  name: String
  place : String
  tickets : Ticket[];
}
type Ticket {
  id: ID
  seat: Int
  name: String
  price: Int
}

input Ticket {
  id: ID!
  seat: Int!
  name: String
  price: Int
}
mutation Create($co:Concert, $ti: Ticket!) {
  createTicket(Concert: $co, ticket: $ti) {
    name
    seat
    price
  }
}
```

Diese Implementierung stellt eine GraphQL Schnittstelle zur verfügung, über welche nun der Name und Ort eines Konzertes abgerufen werden kann. Zusätzlich kann man Tickets für ein Konzert erstellen und bekommt den Namen des Käufers, den Preis und den Sitzplatz zurückgegeben. Das Eintragen eines verkauften Tickets könnte also so aussehen:

```
mutation {
  "co": "Sonic Hybrid Orchestra",
  "ti": {
    "seat": 301,
```

```
    "name": "Zun Korindo",
    "price": 20
  }
}
```

Ein Vorteil gegenüber REST ist die hierachische Struktur von GraphQL. Während mit GraphQL die Anfrage, wer das Ticket mit der Nummer 42 des Konzertes ”Woodstock” gekauft hat, so aussieht:

```
query {
  concert(name:"Woodstock") {
    tickets(id:"42") {
      name
      }
    }
  }
```

Würde mit REST eine Anfrage verschickt werden welche Konzerte es gibt, in der Antwort gesucht werden welches den Namen ”Woodstock” trägt, die ID kopiert werden und noch eine Anfrage geschickt werden, um herauszubekommen welche Attribute das Ticket mit der ID 42 hat. Bei allen diesen Anfragen entsteht viel Overhead, also Daten, die nicht benötigt werden, werden mitgeschickt und verworfen. [4] Sollte sich der Baum noch weiter fortstrecken (z.B. ein Käufer Objekt mit Adresse) wird der Aufwand mit REST empfindlich höher als mit GraphQL grade in mobilen Netzwerken. [1]

III. Höhere Möglichkeiten GraphQLs

A. Variablen

Wie schon im Kapitel ”Serverkonfiguration” gesehen, gibt es nicht nur die Möglichkeit statische Werte als Argumente zu übergeben, sondern auch sogenannte Variablen. Variablen stellen sicher, dass Werte die übergeben in einem bestimmten Bereich liegen, z.B. nur Namen von bestehenden Konzerten. Die Benutzung von Variablen ist relativ einfach. 1. Definieren aller Variablen und ihres Types. 2. Jeder Variablen einen Namen in der Anfrage zuweisen. 3. In der Anfrage den Namen der Variable mit einem Wert verbinden. Im Code des Servers sähen Schritt 1 und 2 etwa so aus:

```
query ConcertByPlace($place : String) {
  concert(place: $place) {
    name
  }
}
```

Diese Abfrage gibt für einen Ort die Namen der Konzerte zurück, die dort stattfinden. Die beiden ersten Zeilen zeigen wie man Schritt 1 bzw. 2 durchführt. Eine Abfrage müsste nun Schritt 3 so durchführen: (Bisher wurde der Name der Query weggelassen, was möglich ist, hier wird er aber für Klarheit mit angegeben.)

```
query ConcertByPlace{
  "place": "Tokyo"
}
```

B. Fragmente

Fragmente sind sozusagen vordefinierte Gruppen von Feldern, die mehrfach abgefragt werden müssen. So wäre für das Konzertbeispiel ein Fragment, welches den Ort und den Namen gleichzeitig abfragt, also nicht viel spart, aber das Konzept verdeutlicht und eine dazugehörige Abfrage so:

```
fragment placeName on Concert {
  place
  name
}
query {
  concert
    ...placeName
}
```

Das Ergebnis sieht genauso aus, also würde man anstelle ”...placeName” den Inhalt des Fragmentes einsetzen. Fragmente sind unter anderem sinnvoll, um zusammengehörige Felder in häufigen Kombinationen zusammenzufassen.

C. Aliases

Da bisher die Rückgabewerte immer dem Namen des Feldes belegt waren, wäre es unmöglich gewesen in einer Abfrage den Ort des Konzertes mit Namen ”A” und den Ort des Konzertes mit Namen ”B” abzufragen. Dazu gibt es Aliases, die das Ergebnis einer Abfrage umbenennen, sodass mehrere Abfragen gebündelt werden können, beispielsweise für Vergleiche.

```
query {
  io : concert(name: "IOSYS") {
    place
  }
  dmt : concert(name: "Demetori") {
    place
  }
}
```

D. Introspection

Eine starke Fähigkeit von GraphQL ist es, das Schema selber abzurufen. [5] So könnte eine Anwendung nachfragen, welche Attribute man von einem Konzert anfragen kann:

```
{
  __type(name: "Concert") {
    fields {
      name
      }
  }
}
```

Der Server wird nun den Namen jedes Attributes von Concert ausgeben, in unserem Falle also:

```
{
  "data": {
    "__type": {
      "fields": [
        {
```

```
          "name": "id"
        },
        {
          "name": "name"
        },
        {
          "name": "place"
        },
        {
          "name": "tickets"
        }
      ]
    }
  }
}
```

Zusätzlich kann man noch die verschiedenen Eigenschaften der Felder abfragen, wie ”kind”, also den typ des Attributes und vieles weitere. Dadurch, dass alles abrufbar ist, kann die Vollständigkeit und Beschaffenheit der Serverstruktur überprüft werden. Dies ermöglicht auch automatisierte Systeme, die durch Introspektion eines Server eine Liste der Möglichkeiten des Servers erstellen könnte, was die Handhabbarkeit des Systems stark verbessert.

E. Pagination

Ein normales Szenario ist, dass man durch ein Objekt und seine Beziehungen zu anderen Objekten durchlaufen will. Nun könnte man nach allen Tickets fragen, die für ein Konzert existieren, doch meistens möchte man auf eine Liste bestimmte Filter anwenden, denn eine Liste die sehr lang ist (z.B. Posts eines Twitter Benutzers) wird meistens nicht benötigt oder ist zu unhandlich. Dafür existiert Pagination, ein System zur besseren Handhabung von Verbindungen zwischen Objekten. So würde man anstelle einer einfachen Liste der Tickets ein Connection Objekt zurückgeben, dass spezielle Operationen erlaubt, wie einen Curser, der speichert an welcher Stelle man stehengeblieben ist. [5] Somit könnte man eine Abfrage, die immer 5 Tickets abruft so formulieren:

```
query {
  concert {
    tickets(first:5 after after:$cursor) {
      node {
        name
        price
      }
      cursor
      totalCount
    }
  }
}
```

Dazu müsste natürlich der Server diese Funktionalität unterstützen und der der Type Concert geändert werden, sodass er anstelle der Liste ”tickets” nun ein Objekt zurückgibt, dass einen Curser und ”nodes” mit den eigentlichen Daten benutzt. Jeder Aufruf dieses Codes gibt 5 Tickets zurück, erhöht den Curser auf das zuletzt ausgegebene Element und gibt beim nächsten Aufruf die nächsten 5 zurück. Auch kann man mit so einem Connection Objekt Informationen liefern, die sonst nirgendwo hingehören, zum Beispiel die Anzahl von Freunden oder eine bestimmte Sortierung.

F. Vorzüge von GraphQL gegenüber REST

GraphQL bietet als Spezifikation keine neuen Möglichkeiten im Vergleich zu REST, sondern verbessert die bestehenden Techniken durch einfacherer Handhabung und erweiterte Freiheiten für Anfragesteller. GraphQL hat Clientspezifische Anfragen, was bedeutet, dass jeder Client eigene Anforderungen auf sein jeweiliges Problem zuschneiden kann. REST hat Serverspezifische Anfragen, welche ein Client dann benutzen muss um sich seine Anforderungen zusammen zu suchen.

Versioning ist das Updaten von Schnittstellen, denn normalerweise ändert sich durch jedes Update etwas in einer REST Umgebung, was alle Anwendungen zwingt, auf die neue Version umzustellen oder den Server zwingt ältere Schnittstellen noch verfügbar zu halten. Dies ist jedoch sehr schwer bei Zwangsupdates, wie bei Sicherheitslücken oder bei Umstrukturierung der Datenhaltung des Servers. Bei GraphQL muss kein aufwendiges Versioning oder Zuschneiden des Servers auf spezielle Clients durchgeführt werden, da sich die Anfragen und Ergebnisse nicht strukturell ändern. [1]

GraphQL ist Hierachisch aufgebaut, was eine einfache Handhabung mit Hierarchisch strukturierten Views ermöglichen soll.

GraphQL ist stark Typisiert, sodass zur Laufzeit die Korrektheit einer Anfrage überprüft werden kann. Dies ermöglicht auch, dass der Server die Struktur und Korrektheit seiner Antwort garantieren kann.

GraphQL ist ein Protokoll, keine Vorschrift wie die Daten vorliegen müssen. Dies bedeutet, dass GraphQL zusätzlich oder aufbauend auf Servern aufgesetzt werden kann.

IV. Conclusion

GraphQL ist eine sehr nützliche Technologie, um weniger, aber nützlichere Anfragen an Server stellen zu können. Sie ermöglicht es, Probleme, die eine sehr große Schnittstelle hat zu umgehen oder zu lösen, wie Versioning. Die Technik, auf der GraphQL aufbaut ist nichts neues, aber viele Ideen, die GraphQL kombiniert sind sehr nützlich und haben sehr starken praktischen Bezug. Da GraphQL universell für strukturierte Daten eingesetzt werden kann, wird GraphQL wahrscheinlich in Zukunft sehr häufig eingesetzt werden.

References

[1] *GraphQL: A data query language*. URL: https://code.facebook.com/posts/1691455094417024/graphql-a-data-query-language/.

[2] *Learn GraphQL, Best Practices*. 2016. URL: http://graphql.org/learn/best-practices/.

[3] *Learn GraphQL*. 2016. URL: http://graphql.org/learn/.

[4] *REST Web Services*. 2007. URL: http://www.oio.de/public/opensource/t3n_nr8_rest_webservices.pdf.

[5] *GraphQL*. URL: http://facebook.github.io/graphql/.

REST

Timo Bühring
Carl von Ossietzky University of Oldenburg, Germany
Department of Computer Science
timo.buehring@uni-oldenburg.de

***Abstract*—Dieses Dokument behandelt die API Design-Vorschrift REST - Representational State Transfer. Es erklärt dabei die Regeln, denen REST Zugrunde liegt, die Vor- und Nachteile von REST, und gibt dann an, wie in der Praxis eine RESTful API basierend auf dem HTTP Protokoll geschrieben werden kann.**

Keywords*—*REST, HTTP

I. Was ist REST

REST - Kurz für Representational State Transfer - ist eine Design-Vorschrift für die Schnittstelle und Implementation eines Webservices und siedelt sich somit zwischen dem Client und dem Server an. Sie wurde von Roy Thomas Fielding - einer der beteiligten beim definieren der HTTP-Spezifikation - im Jahre 2000 formal in einer Dissertation definiert. REST war das Prinzip, an das sich beim Definieren der HTTP Spezifikation orientiert wurde.

Eine API, die sich an diese Vorschriften hält, wird RESTful genannt. Da die Eigenschaften einer RESTful API oft nicht volllständig eingehalten wurden, existiert zudem das Richardson Maturity Model, welches eine API in vier Level einteilt, je nach dem wie streng sich an die Vorschriften gehalten wurden.

Zwar ist REST sehr allgemein definiert, in der Praxis ist mit REST allerdings fast immer die Kommunikation von zwischen Client und Server mittels HTTP gemeint. In diesem Kontext ist REST eine vollständige und objektbasierte Nutzung des HTTP Protokolls und deren Funktionen, um Daten aller Art von einem Server zu fordern und in einer einheitlichen Form, wie etwa XML oder JSON, zu erhalten.

II. Wann sollte REST benutzt werden

Das Abstrakte REST ist grundsätzlich für die Verwendung mit Websites oder ähnlichen Diensten gedacht, die selten große Mengen an Daten versenden. Dabei nimmt der Server eine passive Rolle ein, in der er Anfragen vom Klienten beantwortet. Hauptmerkmale von REST sind ein großer Fokus auf ein einheitliches, Objektbasiertes Interface und die Zustandslosen, in sich abgeschlossenen Transaktionen. Dadurch kann die Implementation von Aufgaben leicht ausgetauscht werden, neue Funktionen eingegliedert werden, oder neue Komponenten, z.B eine Art Cache, zwischen zwei Komponenten eingefügt werden. Dies führt zu einer guten Flexibilität und Skalierbarkeit des Systems. REST ist für große, langlebende und oft erweiterte Systeme gedacht. [1]

Da allerdings so ein relativ großer Overhead einsteht, sind RESTful Systeme nicht für Systeme geeignet, in denen oft Anfragen mit kleinen Antworten kommen, beispielsweise ein Sensornetzwerk. [2]

In der Praxis wird REST oft als alternative zu Protokollen wie SOAP oder JSON-RPC gesehen. Diese Protokolle werden oft auf HTTP mittels der POST Nachricht aufgebaut. Wenn reines HTTP genutzt wird, ist das Strukturieren nach REST ratsam.

Durch die weite Verbreitung von HTTP ist REST besonders für öffentliche APIs geeignet, die so mit einer möglichst großen Gruppe von Clients kompatibel sind. Auch sind REST basierte Systeme leichter einsehbar, da die meisten Webbrowser nativ HTTP verstehen. Gerade für Websites erhöht das Einhalten von REST zudem die Synergie mit dem Browser. Eine Website, die sich nicht an REST hält könnte beispielsweise sämtliche Interaktionen unter der selben URL durchführen und Session-Informationen auf dem Server speichern, wodurch der Zurück-Knopf nicht mehr dazu in der Lage ist, auf die vorherige Seite zu führen. REST ist so ausgelegt, dass solche Probleme nicht entstehen.

III. Die 6 Constraints

Die Basis von REST bilden 6 Contraints. Diese Constraints definieren REST auf eine abstrakte Weise und sind auch auf andere Bereiche, abseits von HTTP, anwendbar. Es folgt eine kurze Übersicht über die Constraints sowie die Vorteile die sie mit sich bringen, um eine Idee über die Voraussetzungen eines RESTful Systems zu geben. Die konkrete Umsetzung in HTTP folgt später.

A. Client-Server

Die Aufgaben sind zwischen einem Client und einem Server getrennt. Dabei gilt das Prinzip der ”Seperation of Concerns”: das Userinterface sollte ausschließlich auf dem Client zu finden sein. So kann das Userinterface von den Daten entkoppelt werden, wodurch beide Komponenten unabhängig voneinander aktualisiert und verbessert werden.

B. Statelessness

Der Server soll keine Zustandsinformationen halten. Das bedeutet, dass jede Anfrage, die vom Client getätigt wird, allein durch die Daten in dieser Anfrage vollständig verarbeitet werden kann; es müssen keine zusätzliche Daten (wie etwas Session-Daten) zu Rate gezogen werden. Neben dem offensichtlichen Vorteil, dass der Server weniger Daten speichern muss, erleichtert dies das Verteilen der Aufgaben des Servers auf mehrere Komponenten.

C. Cache

Bestimmte Resourcen können als Cachable markiert werden. Der Client kann diese Resourcen in einem Cache halten,

und falls die selbe Resource ein zweites mal angefordert wird, stattdessen die gespeicherte Resource aus dem Cache zurückgeben. Dadurch kann die Last auf den Server verringert werden.

D. Layer-Basierter Aufbau

Das System soll in verschiedene Komponenten unterteilt werden, die in Schichten angeordnet werden. Dabei kann eine Schicht nur mit ihren Nachbarschichten kommunizieren. Eine einzelne Komponente (z.B der Client) ist es dadurch egal, ob er direkt mit dem Server, oder einem dazwischengeschalteten Cache-Server redet. Dadurch wird das System übersichtlich und flexibel gehalten. Zudem erleichtert es die Verteilung des Systems auf mehrere Physische Server.

E. Code-On-Demand

Dieser Constraint ist Optional und muss nicht unbedingt eingehalten werden. Hierbei soll der Client Code nachladen und ausführen können, in der Regel als Javascript oder Applet.

F. Uniform Interface

Der laut Fielding wichtigste Constraint wird durch vier weitere Constraints erreicht. Dies sind die Eigenschaften, die REST von anderen Paradigmen abheben und daher am ehesten unter REST verstanden werden. Erstens soll das Interface **Resource-Based** sein. Dabei werden die Informationen, die der Server liefern soll, als Ressourcen klassifiziert - zum Beispiel kann ein Film, eine Person, ein Video oder ein Wetterservice eine Ressource darstellen. Anfragen an den Server liefern dann eine Repräsentation dieser Ressource zurück, beispielsweise als JSON oder HTML. Diese Repräsentationen **enthalten alle Informationen die benötigt werden um Operationen, wie modifizieren oder löschen, auf ihnen Auszuführen**.

Zudem soll jede Antwort **selbstbeschreibend** sein; das heißt sie enthält alle Informationen die benötigt werden um sie zu verstehen, wie etwa den MIME Typen, aber auch ob diese Ressource Cachebar sein soll.

Als letztes gilt das HATEOAS Printzip - **Hypermedia as the Engine of Application State**. Dies ist ein relativ komplexes Konzept und wird später in einem eigenen Abschnitt erläutert.

IV. Umsetzung in der Praxis

In diesem Abschnitt wird die Implementierung eines RESTful Systems in der Praxis besprochen. Dies bedeutet insbesondere das Korrekte Benutzen der vom HTTP Protokoll angebotenen Features. Als Beispiel dient ein einfacher online-Shop.

V. Ressourcen und URI-Design

Ein Wichtiges Konzept in REST ist das der Ressourcen. Eine Ressource ist ein beliebiges Datenobjekt, zum Beispiel ein Video, ein Artikel, aber auch eine (echte) Person. Dabei bezieht sich diese Ressource rein auf das Konzept dieser Information und trägt keine Implementationstechnischen Eigenschaften mit sich. Erst wenn eine Operationen auf diese Resource ausgeführt wird, ergibt sich eine Anfrage, die eine Reaktion vom Server hervorruft. Eine GET Operation gibt beispielsweise eine Liste der verfügbaren Elemente zurück - dazu später mehr.

Eine URI - Uniform Resource Identifier - identifiziert dabei im HTTP Protokoll eine solche Ressource. Das Generelle Schema einer URI lautet wie folgt:

```
scheme : authority / path ? query # fragment
```

Ein Beispiel:

```
http://www.myshop.com/products?category=books#nr=10"
```

Hierbei ist

- `http:` das Schema
- `//www.myshop.com/` die Authority
- `products` der Path
- `?category=books` die Query
- `#page=10` das Fragment

Scheme steht hierbei für das Protokoll, über das die Daten Übertragen werden - bei REST fast immer HTTP. Authority ist der hierarchische Name der Website oder des Services, der von der ICANN vergeben wird. Dieser Teil kann außerdem Zusatzinformationen enthalten, wie eine Nutzernamen-Passwort Kombination oder ein Port, der beim Zielserver angesprochen wird.

Path, Query und Fragment können dagegen vom Nutzer definiert werden.

Die Resourcen können dabei in folgende Archetypen unterteilt werden: [3]

Ein **Dokument** ist ein einzelnes Objekt. Es enthält Einträge sowie Verweise zu anderen Ressourcen, mit denen es zu Tun hat. Die anderen Archetypen fallen alle unter die Definition eines Dokumentes; es kann also als eine Art Grund-Typ gesehen werden.

Ein Dokument wird durch ein Nomen im Singular benannt. Beispiele:

```
http://www.myshop.com/products/
http://www.myshop.com/customers
```

Eine **Sammlung** ist ein Verzeichnis das vom Server verwaltet wird und enthält mehrere Einträge. Elemente können durch den Server hinzugefügt oder entfernt werden; die neuen URIs werden rein vom Server bestimmt.

Eine Sammlung wird durch ein Nomen im Plural benannt. Beispiele:

```
http://www.myshop.com/products
http://www.myshop.com/customers
http://www.myshop.com/customers/251/orders
```

Ein **Speicher**, im Kontrast zu einer Sammlung, wird vom Nutzer verwaltet. Der Nutzer Selber bestimmt, wann neue Elemente unter welcher URI hinzugefügt oder gelöscht werden.

Ein Speicher wird durch ein Nomen im Plurar benannt. Beispiele:

`http://www.myshop.com/customers/251/favourites`

`PUT http://www.myshop.com/customers/251/favourites/SICP` erstellt so einen neuen Eintrag

Ein **Controller** beschreibt eine Prozedur, die nicht durch eine der CRUD Operationen (Create, Retrieve, Update, Delete) abgedeckt sind. Sie haben keine Kinder in der Hierarchie und werden durch Verben beschrieben.

Ein Controller wird durch ein Verb beschrieben. Hier ein Beispiel von PayPal:

`http://www.myshop.com/customers/1614/account/withdraw https://api.paypal.com/v1/payments/sale/36C38912MN9658832/refund`

Die Query-Parameter bilden den Nicht-Hierarchischen Teil einer URI. Sie sind meistens, aber nicht zwingend, ein Satz von Identifier-Werte Paaren, die an das Ende des Ressource Pfades abgehangen werden. Dabei sind sie mit einem Fragezeichen vom Rest des Pfades getrennt, sowie untereinander mit einem Und-Zeichen (&).

`http://www.myshop.com/products?format=JSON&encoding=UTF8`

Ein GET-Request auf diese URI würde beispielsweise die JSON Repräsentationen der Produkte in UTF8 zurückgeben. Diese Eigenschaften sind nicht Hierarchisch in ihrer Natur. Die Kodierung und das Format stehen in keiner Beziehung zueinander und können unabhängig voneinander verändert werden.

Im obigen Beispiel wurde das Format in dem sich die Antwort befinden soll spezifiziert. Ein GET Request auf könnte so zum Beispiel eine gefilterte Liste erlauben, statt immer die gesamte Liste zurückzugeben

`http://www.myshop.com/products?category=books`

`http://www.myshop.com/products?name="SICP"`

Da eine Hierarchie auf mehrere Weisen definiert werden kann, ist es oft Möglich, sowohl Query-Parameter als auch den Pfad zur identifikation bestimmter Ressourcen zu Nutzen. Im oberen Beispiel wurde der Pfad

`http://www.myshop.com/products/1`

benutzt, um das Produkt mit der ID 1 zu Referenzieren, während

`http://www.myshop.com/products?name="SICP"`

die Liste der Produkte auf das Produkt mit dem namen SICP reduziert hat. Sofern der Name jedes Produktes eindeutig ist, wäre es auch denkbar, die hierarchische Struktur als

`http://www.myshop.com/products/SICP`

zu definieren, und durch

`http://www.myshop.com/products? id=1`

auf das Produkt mit der ID 1 zu kommen.

Hier ein paar Beispiele für URIs, die nicht RESTful sind:

`http://www.api.myshop.com/1.5/products/json/utf8`

In diesem Beispiel wurden json und utf8 zum teil des Hierarchischen Pfades, obwohl sie keinen hierarchischen Zusammenhang haben.

`http://www.api.myshop.com/1.5/customers/1536/update`

Diese URI sieht zunächst aus, als falle sie in den Archetypen des Controllers. Allerdings sollten Controller nur für Operationen benutzt werden, die nicht unter die CRUD Operationen fallen; update ist einer dieser Operationen.

`http://www.myshop.com/api?operation=remove_product&pid=16142&refund_orders=true`

Zwar klassifiziert sich eine API als Ressource, allerdings wird hier am HTTP Protokoll vorbei spezifiziert. In HTTP und damit in einem RESTful System werden die HTTP Operationen auf eine Ressource, die durch die URI identifiziert wird, angewandt; diese URI spezifiziert allerdings die Operation und ähnelt somit eher einem RPC Ansatz.

Der Letzte teil der URI, das Fragment, ist nur für den Client von Bedeutung, und hat keinen Einfluss auf eine mögliche Operation, die auf die URI ausgeführt wird. Er beschreibt einen Teil der Repräsentation, die erhalten wurde, und hilft somit hauptsächlich dem Browser, sich in der Seite zurechtzufinden. Die URI

`http://www.myshop.com/customers/1251/reviews#5`

beschreibt beispielsweise die Reviews des Kunden mit der Nummer 1251, während das Fragment das fünfte Review fokussiert. In einem Browser aufgerufen würde dieser Link automatisch die Seite zum fünften Review scrollen.

VI. Operationen: GET, POST, PUT, DELETE

Während Ressourcen durch Nomen beschrieben werden, bilden die Operationen die Verben, die für eine Transaktion nötig sind. Verschiedene Operationen haben dabei verschiedene Bedeutungen, wenn sie auf die selbe Ressource angewandt werden.

Da REST normalerweise mit HTTP verwendet wird, sind die Operationen hier die von HTTP definierten Operationen GET, POST, PUT und DELETE. Anders als bei aufgesetzten Protokollen, die häufig nur POST benutzen, hat bei REST jeder dieser Operationen eine eigene Bedeutung, die im folgenden näher erläutert wird.

A. GET

Die GET Operation liefert eine Repräsentation der angegebenen Ressource zurück. Sie ist die Operation die vom Browser verwendet wird, wenn eine URL in die Address-Zeile eines Browsers eingeben wird.

GET muss eine sichere Methode sein. Das bedeutet, dass das Aufrufen von GET keine Veränderungen an Ressourcen

verursachen sollte, sodass das mehrfache Aufrufen von GET auf eine URI ohne Probleme getan werden kann.

Wird GET auf eine Ressource angewandt, so sollte das Ergebnis eine Liste von Repräsentationen aller Elemente dieser Ressource sein.

```
GET http://www.api.myshop.com/products
```

könnte beispielsweise folgende Antwort geben

```
{
{"id" = 1, "name" = "SICP - Structure and
Interpretation of Computer Programs",
"category"="books" },
{"id" = 2, "name" = "Microsoft Office",
 "category"="software" }
}
```

Eine URI, die ein spezifisches Element repräsentiert, soll dagegen nur dieses Element zurückgeben. Gibt man beispielsweise den Request

```
GET http://www.api.myshop.com/products/1
```

an, wäre die Antwort nur

```
 {"id" = 1, "name" = "SICP - Structure
and Interpretation of Computer Programs",
"category"="books" }
```

Diese Voraussetzungen können leicht gebrochen werden, wenn die URIs nicht nach RESTful Prinzipien benannt worden sind. Ein GET Request der Form

```
GET http://www.api.myshop.com/products/
add?name="Visual_Studio_2015"&category=
"software"
```

würde ein neues Produkt zum Shop hinzufügen, was eine Änderung auf Serverseiten entspricht - eine solche GET-Methode ist nicht sicher.

B. PUT

PUT ist eine idempotente Operation. Das bedeutet, dass mehrfaches anwenden den Selben Effekt hat wie einmaliges Anwenden. Ein berühmtes Beispiel aus der Mathematik für diese Eigenschaft ist die Multiplikation mit 0: x*0 = x*0*0 = x*0*0*0. Diese Eigenschaft erlaubt ein sichereres Benutzen dieser Operation: wird beispielsweise ein PUT Request gesendet, um einen neuen Artikel einzufügen, aber die Antwort geht auf dem Weg verloren, so weiß der Client nicht, ob die Operation korrekt ausgeführt wurde, oder der Request auf dem Weg zum Server verloren gegangen ist. Da aber mehrmaliges ausführen kein unterschiedlichen Effekt hat, kann ein der Request ein zweites mal gesendet werden, ohne dass der selbe Artikel zwei mal hinzugefügt wird.

Um dies zu erreichen hat PUT zwei verschiedene Semantiken, je nachdem ob die angegebene Ressource bereits existiert oder nicht. Existiert die Ressource nicht, so wird sie mit der übergebenen Repräsentation angelegt. Existiert sie bereits, so ist die übergebene Repräsentation eine modifizierte Version der existierenden Ressource, und die existierende Ressource wird mit dieser Repräsentation überschrieben.

Grundsätzlich wird also PUT zum Anlegen oder Modifizieren einer Ressource benutzt, die ihre eigene URI hat.

C. POST

Anders als PUT ist die POST Operation nicht idempotent. Viele Browser warnen daher den Nutzer, wenn er eine POST Operation wiederholen würde, wenn er eine Seite Zurück geht.

Die POST Operation dient dazu, eine untergeordnete Ressource zu erstellen. POST darf daher keine URI referenzieren, die noch nicht existiert; wird also durch eine POST Operation eine neue URI angelegt, so liegt diese URI in der hierarchischen Struktur unter der URI, auf die die Operation ausgeführt wird.

Gleichzeitig muss eine POST Operation nicht unbedingt eine neue URI erstellen. Zum Beispiel kann POST benutzt werden, um einen Beitrag in einem Internet-Forum hinzuzufügen.

Da die URI der erstellten Ressource nicht bekannt sein muss, eignet sich POST dazu, Ressourcen anzulegen, bei denen die URI noch nicht bekannt ist, und erst durch Algorithmen auf dem Server gebildet werden kann. Möchte beispielsweise ein Nutzer in unserem Online-Shop ein Review hinzufügen, kann er das mit PUT nicht tun, da er die URI des Reviews kennen müsste. Würde er sein Review durch

```
PUT http://www.myshop.com/products/
1/reviews/2
```

hinzufügen, selbst wenn das Review mit der ID 2 gerade nicht existiert, liefe er in Gefahr das Review einer anderen Person zu überschreiben, die ebenfalls auf dieser ID ein GET aufruft. Durch ein POST Request auf

```
GET http://www.api.myshop.com/products
/1/reviews
```

erstellt der Server selber die neue URI. Dadurch kann er dafür sorgen, dass die ID noch nicht belegt ist.

D. DELETE

DELETE wird dazu genutzt, eine Ressource, die durch eine URI identifiziert wird, zu Löschen. DELETE ist ebenfalls eine idempotente Operationen - da die Ressource explizit durch die URI angegeben werden muss, eine nicht existierende Ressource aber nicht gelöscht werden kann, hat mehrfaches Ausführen der Operation auf die selbe URI keinen Effekt.

VII. HATEOAS

Ein Laut Fielding sehr wichtiges und oft verletztes Konzept [1] in REST ist Hypermedia as the Engine of Application State. Die Idee hierbei ist dass sich der Client durch das gesamte System bewegen kann, ohne Annahmen über die Struktur der verfügbaren URIs zu machen.

Zu diesem Zweck soll jede URI als Teil der Antwort URIs zu anderen Teilen der Systems enthalten, gekoppelt mit einer Beschreibung, welche Operationen auf dieser URI ausgeführt werden können, und welche nicht.

Bei einer normalen Website ist dies in der Regel von Natur aus gegeben. Die Links sind hierbei in der HTML Seite

eingebettet und werden vom Nutzer geklickt, um sich durch die Website zu navigieren.

Bei einer API drückt sich HATEOAS dadurch aus, dass in den Antworten zu Anfragen URIs zu möglichen Operationen finden. Die offizielle PayPal API fügt ihren Antworten Beispielsweise folgende HATEOAS Informationen hinzu [4]:

```
{
  "links": [{
    "href": "https://api.paypal.com/v1/
    payments/sale/36C38912MN9658832",
    "rel": "self",
    "method": "GET"
  }, {
    "href": "https://api.paypal.com/v1/
    payments/sale/36C38912MN9658832/refund",
    "rel": "refund",
    "method": "POST"
  }, {
    "href": "https://api.paypal.com/v1/
    payments/payment/
    PAY-5YK922393D847794YKER7MUI",
    "rel": "parent_payment",
    "method": "GET"
  }]
}
```

Dieses Objekt besteht aus einem Array aus Links; jeder Link enthält die URI, eine Beschreibung der Relation, in der dieser Link zur angeforderten Ressource steht, sowie die Operation, die auf ihr ausgeführt werden kann.

VIII. Statelessness und Sessions

Einer weiterer großer Punkt in REST ist, dass der Server keinerlei Zustandsinformationen speichern soll. Der Client muss also alle Informationen halten, die nötig sind, um eine Session aufrechtzuerhalten.

Das HTTP Protokoll unterstützt die Basisauthentifizierung. [5] Ein Server kann seine URIs dabei in Bereiche aufteilen - sogenannte Realms - auf die ohne gültige Nutername-Passwort Kombinationen nicht zugegriffen werden kann. Sollte dies passieren, fragt der Server nach einer Authentifizierung. Ein Webbrowser öffnet in diesem Fall oft ein Popup, das einen dazu auffordert, Nutzername und Daten anzugeben. Die eingegebenen Daten werden von diesem Zeitpunkt als der Teil des HTTP-Headers mit gesendet. Dabei werden Nutzername und Passwort mit einem Doppelpunkt (:) verbunden und der resultierende String in base64 encodiert.

Grundsätzlich ist es auch möglich, einen Cookie anzulegen, der Nutzername und Passwort enthält. Der Cookie wird mit jeder HTTP Operation, die auf die Website ausgeführt wird, mitgesendet, sodass der Server die Daten überprüfen kann.

Keiner dieser Methoden ist von Natur aus sicher, da die Nutzerdaten - insbesondere das Passwort - sich unverschlüsselt in jeder Nachricht finden. Daher ist diese Art der Authentifizierung nicht für unverschlüsselte Verbindungen geeignet, sondern sollte nur über verschlüsselte Wege, beispielsweise HTTPS, benutzt werden.

IX. Richardson Maturity Model

Das Richardson Maturity Model[6] bewertet, wie RESTful ein Service ist, indem er es in eine von vier Stufen einteilt.

- Level 0: Verwendet eine einzige URI und eine einzige HTTP Operation, meißtens POST
- Level 1: Verwendet verschiedene URIs für verschiedene Ressourcen, aber nur eine HTTP Methode
- Level 2: Verwendet verschiedene URIs und mehrere HTTP Operationen
- Level 3: Verwendet verschiedene URIs, mehrere HTTP Operationen und HATEOAS

Laut Richardson sind die meisten APIs, die sich RESTful nennen, auf Level 2; HATEOAS sei ein relativ komplexes Konzept und wird daher oft nicht implementiert. Es ist also erstrebenswert, mindestens Level 2 zu erreichen. Zwar ist streng genommen nur Level 3 eine wirklich RESTful API, es sollte aber gut darüber nachgedacht werden, ob der zusätzliche Aufwand, HATEOAS zu implementieren, wirklich von Nöten ist.

X. Fazit

Eine RESTful API zu Designen erfordert viel Planung und schränkt einen, insbesondere durch die Zustandslosigkeit des Servers, eventuell etwas ein. Insbesondere durch das HATEOAS Prinzip kann zudem eine Formal korrekte Implementierung sehr viel Arbeit bedeuten. Für kleinere Projekte ist REST meiner Meinung nach nicht zu Empfehlen.

Wenn dagegen nach einer Möglichkeit gesucht wird, einen Robusten Webservice zu schreiben, der sich über Jahre entwickeln und verändern können soll und bei erhöhter Nachfrage leicht Skalierbar sein soll, ist REST eine gute Möglichkeit, dies zu garantieren.

References

[1] Roy Thomas Fielding. *REST APIs must be hypertext-driven*. URL: http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven.

[2] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. URL: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[3] Mark Masse. *REST API Design Rulebook*. O'Reilly Media, 2011.

[4] PayPal. *The REST APIs and HATEOAS*. URL: https://developer.paypal.com/docs/api/hateoas-links/.

[5] J. Franks et al. *HTTP Authentication: Basic and Digest Access Authentication*. URL: https://tools.ietf.org/html/rfc2617#section-2.

[6] Leonard Richardson. *Act Three: The Maturity Heuristic*. URL: https://www.crummy.com/writing/speaking/2008-QCon/act3.html.