Asynchronous Javascript: Analyse und Implementation von Mechanismen zum asynchronen Laden von Daten in Webbrowsern

Bachelor-Thesis

zur Erlangung des akademischen Grades B.Sc.

Kristin Groschoff

2141708

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

Erstprüfer: Prof. Dr. Nils Martini

Zweitprüfer: David Henkensiefken

Hamburg, 29.08.2016

Inhaltsverzeichnis

Abstract

The aim of this bachelor's thesis is to analyze different methods for asynchronous loading of data for the company veeseo GmbH. It considers the progressive development in dealing with asynchrony in JavaScript, starting with callbacks, promises and generators and leads to the future use of async and await. Furthermore, their particular advantages and disadvantages will be evaluated. Finally it will be applied to a specific common problem of the company. This is intended to show which possibilities exist to optimize the current code for delivering the so-called widgets in the future.

Zusammenfassung

Im Rahmen dieser Bachelor-Thesis werden für das Unternehmen veeseo GmbH verschiedene Methoden zum asynchronen Laden von Daten analysiert. Es wird die fortschreitende Entwicklung im Umgang mit Asynchronität in JavaScript betrachtet, beginnend bei Callbacks, über Promises und Generatorfunktionen, bis hin zur nahen Zukunft der Verwendung der Schlüsselwörter async und await. Zudem werden deren Vor- und Nachteile abgewägt. Abschließend wird dies auf ein konkretes alltägliches Problem des Unternehmens angewandt. Dies soll aufzeigen, welche Möglichkeiten bestehen, den aktuellen Code zur Auslieferung von sogenannten Widgets in Zukunft zu optimieren.

1 Einführung

1.1 Motivation

Laut einer Studie der Bitkom besitzen aktuell etwa 88 Prozent der Haushalte in Deutschland einen Internetzugang.[1]. Die Nutzung dieses Mediums ist bei den meisten Bürgern bereits ein fester Bestandteil ihres täglichen Lebens. Mit nur wenigen Klicks können Informationen abgerufen werden. Die Internetverbindungen werden immer schneller und immer mehr Webanwendungen fühlen sich bereits wie Desktopanwendungen an. Besonders extern eingebundene JavaScripte verlangsamen jedoch den Seitenaufbau. Die Frustrationsgrenze ist daher schnell erreicht, wenn eine Seite zu lange zum Laden der Inhalte benötigt. Nicht selten wird die Seite direkt wieder geschlossen.

Für Unternehmen ist die Geschwindigkeit ihrer Webanwendungen besonders wichtig. Speziell im IT-Bereich sind diese oftmals in Verwendung, da sie ortsungebunden sind und daher jederzeit und von überall im Browser abgerufen werden können. Um einen schnellen Datenaustausch zu gewährleisten, werden Daten asynchron geladen.

An dieser Stelle setzt diese Arbeit an. Die veeseo GmbH erstellt Widgets mit Empfehlungen für einen Artikel. Für die Generierung dieser Widgets wurde der sogenannte Widgetgenerator entwickelt. Der dazugehörige Widgetcode wurde bereits vor drei Jahren geschrieben und verwendet noch immer Callbacks[2]. Da er eine Vielzahl von teils auch älteren Browsern unterstützen muss, ist der Einsatz von neueren Technologien noch nicht möglich, weil eben diese noch nicht in allen Browsern unterstützt werden. Mit dieser Arbeit soll jedoch schon einmal ein Einblick gegeben werden, wie die Umsetzung des asynchronen Ladens in der Zukunft gestaltet werden kann.

1.2 Aufgabenstellung

In dieser Arbeit sollen verschiedene Methoden analysiert werden, die das asynchrone Laden von Daten unter Verwendung von JavaScript ermöglichen. Dazu muss vorab erläutert werden, was der Unterschied zwischen synchronem und asynchronem Laden ist und weshalb die Asynchronität angestrebt werden sollte.

[1] vgl. Bitkom (2016)

[2] Eine Methode zum asynchronen Laden von Daten. Genauer beschrieben im Abschnitt 4.1

Die gewonnenen Kenntnisse sollen schließlich prototypisch umgesetzt werden. Als Vorlage soll hierbei die Logik des schon bestehenden Widgetcodes dienen.

Ziel der Arbeit ist es, ein vereinfachtes, aber funktionsfähiges Modell zu erstellen, welches die wichtigsten Funktionen des bereits vorhandenen Widgets aufweist.

1.3 Inhaltlicher Aufbau der Arbeit

Die Arbeit ist in sechs Abschnitte gegliedert. Den ersten Teil bildet die Einleitung, welche einen kurzen Einblick in das Thema gibt. Des Weiteren werden hier die Aufgabenstellung und das daraus resultierende Ziel der Arbeit definiert. Der zweite Abschnitt erklärt die Grundlagen. Hier wird das Unternehmen, für das die Bachelor-Thesis geschrieben wird, vorgestellt und alle wichtigen Begrifflichkeiten beschrieben, die im weiteren Verlauf für die Verständlichkeit benötigt werden. Nachdem die Grundlagen erläutert wurden, werden in Abschnitt drei die Ladeverhalten von synchronen und asynchronen Anwendungen erklärt. Die Analyse und Erklärung von den verschiedenen Methoden zum asynchronen Laden findet in Abschnitt vier statt. Dies sind Callbacks, Promises, die Generatorfunktion mit ihrem Schlüsselwort yield und die Verwendung von async und await. Der Abschnitt fünf besteht schließlich aus der Implementierung des prototypischen Codes. An dieser Stelle wird zunächst die aktuelle Vorgehensweise erklärt und anschließend dessen Schwachstellen beschrieben. Im Anschluss wird der Prototyp mit den aktuellen Methoden vorgestellt. Die Arbeit endet mit einem Fazit, welches die Ergebnisse zusammenfasst und einen Ausblick auf möglicherweise weitere Entwicklungen und Verbesserungen gibt.

2 Grundlagen

2.1 Das Unternehmen

Das Unternehmen, für das die Bachelor-Thesis geschrieben wird, trägt den Namen veeseo GmbH[3] – im Folgenden veeseo genannt – und wurde 2010 mit Sitz in Hamburg gegründet. Im Jahre 2013 wurde veeseo als weiterhin eigenständiges Unternehmen von der Ligatus GmbH, einem Tochterunternehmen der Gruner+Jahr GmbH & Co. KG, übernommen.

veeseo generiert Content Recommendations für Verlage und andere Webseiten, indem es die Inhalte eines Artikels oder eines Videos analysiert, Schlüsselwörter herausfiltert und darauf aufbauend passende Inhalte empfiehlt. Diese erscheinen meist unmittelbar unter dem jeweiligen Artikel in einer kleinen Box, in der auch Werbung integriert ist. Ein Webseitenbesucher klickt entweder auf einen ähnlichen Artikel und bleibt dem Publisher als Besucher erhalten, was für höhere Klickzahlen sorgt, oder aber er klickt auf eine Werbung, bei denen der Publisher von den Werbeeinnahmen profitiert, obwohl der Besucher die Seite bereits verlassen hat.

2.2 JavaScript

JavaScript gehört zu der Klasse der Skriptsprachen, was bedeutet, dass die Skripte während der Laufzeit durch einen Interpreter ausgeführt werden und nicht kompiliert werden müssen. Sie wurde ursprünglich entwickelt, um Dokumente interaktiv und dynamisch zu gestalten, zum Beispiel um Benutzeraktionen auszuwerten oder Inhalte zu verändern. Heutzutage ist in fast jeder Webseite ein JavaScript vorhanden, das in ein HTML-Dokument eingebettet ist. Änderungen zum Beispiel durch Benutzereingaben finden nur im Browser statt, sodass kein neues Dokument vom Web-Server abgerufen werden muss. Aus diesem Grund wird JavaScript auch als clientseitige Programmiersprache bezeichnet.

2.2.1 Entstehung

Das US-amerikanische Softwareunternehmen Netscape Communications veröffentlichte im September 1995 die neue Version ihres Internetbrowsers – dem Navigator 2.0. In diesem war LiveScript implementiert, eine Skriptsprache, die von Brendan

[3] http://www.veeseo.com

Eich entwickelt worden war. Kurz darauf verkündeten Netscape und Sun Microsystems eine Kooperation, die zum Ziel hatte, dass LiveScript und Java-Applets direkt miteinander interagieren. Hierfür wurden von Sun die benötigten Java-Klassen und von Netscape die Schnittstelle LiveConnect enwickelt. Außerdem wurde die Sprache in JavaScript umbenannt, die seit der Übernahme von Sun Microsystems eine Marke des Unternehmens Oracle ist.

Die Nützlichkeit von JavaScript erkannte auch der Konkurrent Microsoft schnell und baute eine eigene JavaScript-Engine für ihren Internet Explorer 3.0, die JScript hieß. Diese sollte die Lizenzvorgaben von Netscape umgehen und gleichzeitig mehr Funktionen bieten. Unter den Programmierern sorgte dieser Wettstreit für viel Frust, da vieles sehr umständlich programmiert werden musste, weil die verschiedenen Browser verschiedene Interpreter benutzten.[4]

Um eine gemeinsame Lösung zu finden, wurde das Gremium zur Standardisierung der Techniken im Internet – das World-Wide-Web-Consortium (kurz W3C) – eingeschaltet. Sie erarbeitete ein allgemeines Modell für Objekte eines Dokuments, welches die Bezeichnung Document Object Model (kurz DOM) erhielt. Es regelt den lesenden und verändernden Zugriff auf das Dokument.[5]

2.2.2 ECMAScript

Zusammen mit der European Computer Manufacturers Association (kurz ECMA) wurde ein Standard für die Grundelemente der Sprache entwickelt und 1996 unter dem Namen ECMA-262 (ECMAScript 1) veröffentlicht. Es erstellt damit eine plattformübergreifende und universelle Programmiersprache. Seitdem wurde das ECMAScript stetig weiterentwickelt, wie in Tabelle 2.1 zu sehen.

Die vierte Version wurde nie veröffentlicht, weil es Unstimmigkeiten für die weiteren Entwicklung der Sprache gab. Die Standards wurden anfangs mit Nummern versehen. Mit der sechsen Version des ECMAScripts gibt es auch eine Neuheit bei der Benamung: Statt der sonst üblichen Nummern, werden die Standards mit den Jahreszahlen versehen, in denen sie verabschiedet worden sind.

Der aktuellste Standard ist das ECMAScript 2016, welches jedoch erst nach und nach in die verschiedenen Browserversionen implementiert wird. Die 8. Version, das ECMAScript 2017, ist bereits in Arbeit.

[4] vgl. Schäfer, Mathias (2015)
[5] vgl. W3C (2012)

Versionen des ECMAScript	
Version	**Publiziert**
1, ECMA-262	Juni 1997
2	Juni 1998
3	Dezember 1999
4	abgebrochen
5	Dezember 2009
5.1	Juni 2011
6, ECMAScript 2015	Juni 2015
7, ECMAScript 2016	Juni 2016

Tabelle 2.1: Versionen des ECMAScript[6]

2.3 Ajax

Ajax ist das Apronym für Asynchronous JavaScript and XML, welches erst im Jahre 2005 von Jesse James Garrett in seinem Essay „Ajax: A New Approach to Web Applications“[7] geprägt wurde. Es ist ein Konzept zur asynchronen Datenübertragung, das heißt, es ermöglicht eine Veränderung der Seite, ohne diese komplett neu zu laden. Um Ajax zu realisieren, werden mehrere verschiedene Technologien verwendet.

- HTML und CSS werden für die Gestaltung der Webseite eingesetzt. CSS wird genutzt, um den Inhalt des HTML-Dokuments zu stylen. Dies hat gleichzeitig den Vorteil, dass Designelemente in einer zentralen Datei zu finden sind, die bei Bedarf ohne großen Aufwand angepasst werden kann.
- DOM ist die Gliederung, die für Dokumente wie HTML oder XML benutzt wird. Diese Struktur ist auch als Baumstruktur bekannt. Mithilfe von JavaScript kann diese manipuliert beziehungsweise verändert werden.
- XML beziehungsweise heute üblicherweise JSON ist das Format, welches für den Datenaustausch genutzt wird.
- Der XMLHttpRequest ist eine Anwendungsschnittstelle (API), der für den Datenaustausch zwischen Client und Server sorgt. Sie kommuniziert asynchron mit dem Webserver, wodurch der Browser nicht blockiert ist.

[6] vgl. ECMA International (2016)
[7] Garrett, Jesse James (2005)

- JavaScript wird genutzt, um das Ganze miteinander zu vereinen.

Da eine Ajax-Anwendung heutzutage nicht zwangsweise XML enthalten muss, steht das X in der Praxis für verschiedene Datenformate, die vom Server zum Client geladen werden. Ajax wird daher heute eher als ein feststehender Begriff verwendet.[8]

2.4 jQuery

jQuery ist eine schlanke, aber mächtige JavaScript Bibliothek, die von John Resig entwickelt und im Januar 2006 veröffentlicht wurde. Seitdem wird sie von der jQuery Foundation weiterentwickelt. Mithilfe dieser Bibliothek ist es möglich, mit verhältnismäßig wenig Codezeilen einen komplexen JavaScript Code umzusetzen. Außerdem wird die Programmierung zur Erreichung der Cross-Browser-Kompatibilität erleichtert, was bedeutet, dass Elemente in verschiedenen Webbrowsern richtig angezeigt werden. Passenderweise lautet das Motto von jQuery *„write less, do more“* [9].

jQuery ist sehr schnell zu erlernen und bietet folgende Funktionen:

- Elemente auf einer Webseite können ausgewählt und manipuliert werden, um beispielsweise dessen Aussehen oder Position zu verändern.
- Mit dem Event-System ist es möglich, auf das Nutzerverhalten zu reagieren.
- Animationen und Effekte können einfach umgesetzt werden.
- Durch die Einbindung von Ajax können Daten vom Server schnell abgerufen und weiterverarbeitet werden.
- Es gibt viele Plug-Ins, die die Gestaltung der Benutzeroberfläche erleichtern.

Es spart also durchaus Zeit, jQuery zu nutzen. Dafür muss lediglich die jQuery-Bibliothek, die aus einer JavaScript-Datei besteht, ins HTML-Dokument eingebunden werden. Das Listing 2.1 zeigt einen jQuery Aufruf.

```
1 $("div.enter").on("click", function() {
2     alert("Welcome!");
3 }
```

Listing 2.1: Beispielaufruf mit jQuery

In diesem Beispiel wird dem `div`-Element mit der Klasse `enter` ein Event-Listener zugewiesen. Das Dollarzeichen am Anfang weist dabei auf ein jQuery Objekt hin. Möglich und oftmals genutzt ist stattdessen auch das ausgeschriebene Wort „jQuery“. Wird nun auf dieses Element geklickt, wird eine Nachricht ausgegeben. Diese Umsetzung hat den Vorteil, dass sämtliche Verhalten an einer zentralen Stelle festgelegt werden können und nicht im HTML-Element selber.

[8] vgl. Wenz, Christian (2007)
[9] Vdovkin, Andreas (2011)

2.5 JSON

JSON steht für JavaScript Object Notation und ist ein Textformat, welches besonders für den Datenaustausch geeignet ist. Es wurde 2005 veröffentlicht und wurde schnell zum neuen Standard, wodurch XML (Extensible Markup Language) abgelöst wurde.[10]

Die folgenden beiden Listings zeigen den gleichen Beispielcode. Dabei wird das Listing 2.2 mit XML umgesetzt und das Listing 2.3 mit JSON.

```
1 <film>
2     <titel>Harry Potter</titel>
3     <datum>2001</datum>
4     <regie>
5         <name>Columbus</name>
6         <vorname>Chris</vorname>
7     </regie>
8 </film>
```

Listing 2.2: Beispieldatei mit XML

```
1 {
2    "Titel": "Harry Potter",
3    "Jahr": 2001,
4    "Regie": {
5               "Name": "Columbus",
6               "Vorname": "Chris"
7             }
8 }
```

Listing 2.3: Beispieldatei mit JSON

Der Vorteil bei JSON gegenüber XML besteht darin, dass Daten auf einfache Weise strukturiert werden können. Im Gegensatz zu XML, welches das Tag-System verwendet, ist JSON textbasiert und ist somit leichter zu lesen und zu verstehen. Es ist zudem komplett plattform- und sprachenunabhängig.

Ein JSON-Objekt ist eine Sammlung von Schlüssel-Wert-Paaren, bei denen links der Schlüssel und rechts der Wert steht. Durch die in Tabelle 2.2 beschriebenen Datenstrukturen können Daten beliebig verschachtelt werden. Auch dies ist eine große Stärke von JSON.

[10] vgl. Strang, Martin (2007)

Datentypen in JSON	
Nullwert	null
Boolescher Wert	true, false
Zahl	0 bis 9, auch mit negativen Vorzeichen oder Dezimalpunkt möglich
Zeichenkette	beginnt und endet mit doppelten Anführungszeichen
Array	beginnt und endet mit eckigen Klammern, mehrere Werte werden durch ein Komma getrennt
Objekt	beginnt und endet mit geschweiften Klammern, mehrere Eigenschaften werden durch ein Komma getrennt Eigenschaften bestehen aus einem Schlüssel und einem Wert, die durch einen Doppelpunkt getrennt werden. Der Schlüssel ist dabei eine Zeichenkette; für den Wert sind alle oben genannten Datentypen zulässig.

Tabelle 2.2: Datentypen in JSON[11]

Da JSON bereits valides JavaScript darstellt, lässt sich ein JSON-Objekt direkt mit der `eval()`-Funktion in ein JavaScript-Objekt übersetzen. Da dies jedoch nicht mehr gerne gesehen ist[12] und `eval()` gegebenenfalls auch schädliche Programmanweisungen ausführen kann, gibt es als Alternative zwei Methoden:

- `JSON.parse()`
- `JSON.stringify()`

`JSON.parse()` übersetzt einen String in ein JavaScript-Objekt. `JSON.stringify()` ist die Umkehrung und konvertiert ein JavaScript-Objekt in einen String.

Doch auch JSON hat Grenzen im Datenaustausch. So ist es nicht möglich, Daten über Domaingrenzen hinweg zu übertragen. Dies liegt an der Same-Origin-Policy – einem Sicherheitskonzept, das den Zugriff auf Objekte untersagt, wenn diese von fremden Webseiten stammen. Es ist in allen modernen Browsern implementiert und soll vor Angriffen schützen.

Dieses Problem kann jedoch mit JSONP, dem JSON mit Padding, umgangen werden. Es wurde 2005 von Bob Ippolito vorgestellt und wird in den meisten Webanwendungen unterstützt. Dafür muss es auf der Clientseite eine Callback-Funktion geben,

[11] vgl. SELFHTML-Wiki (2016)
[12] vgl. mediaevent (2016)

die die übertragenen Daten annimmt. Durch einen Skriptaufruf werden der Name dieser Funktion und die zu übertragenen Daten an den Server übermittelt. Der fertige `script`-Tag muss anschließend in die Webseite eingeschleust werden. Dies nennt sich „script tag injection“ und löst die Datenübertragung aus. Der Server übernimmt daraufhin die Daten und extrahiert den Namen der Callback-Funktion. Dieser wird genutzt, um die Daten anschließend durch einen Funktionsaufruf zu umklammern und an den Client als Skript zurückzuschicken. Da es aus einem Funktionsaufruf besteht, wird diese direkt ausgeführt und bekommt die Daten als Parameter übergeben.

2.6 Webbrowser

Webbrowser sind spezielle Computerprogramme, die Webseiten im Internet darstellen können und stellen daher die Benutzeroberfläche für Webanwendungen dar. Es können jedoch nicht nur HTML-Seiten angezeigt werden, sondern auch verschiedene Inhalte, wie Bilder, Musik oder Filme. Diese sind teilweise durch externe Plugins eingebunden. Neben den bekannten Webbrowsern wie Google Chrome, Mozilla Firefox, Internet Explorer, Opera und Safari gibt es noch viele weitere, die allerdings eher unbekannt sind. Auf den ersten Blick verrichten sie alle die gleiche Arbeit, bei genauerer Betrachtung gibt es jedoch Unterschiede. So laden manche Webbrowser Seiten schneller als andere oder haben mehr Plugins zur Auswahl.

3 Ladeverhalten

3.1 Request-Response-Paradigma (HTTP-Request-Cycle)

Damit HTML Dokumente zwischen dem Server und dem Client ausgetauscht werden können, gibt es das sogenannte Hyper Text Transfer Protocol (kurz HTTP). Es ist ein objektorientiertes und zustandsunabhängiges Protokoll, das für viele Aufgaben eingesetzt werden kann. HTTP folgt dem Request-Response-Paradigma – auch als HTTP-Request-Cycle bekannt – und besteht aus vier Phasen:

1. Connect
2. Request
3. Response
4. Close

Das bedeutet, dass der Client im ersten Schritt eine Verbindung aufbaut, zum Beispiel indem er im Browser einen Button anklickt. Dieser stellt anschließend ein HTTP-Request an den Server, der ihm daraufhin antwortet, indem er die Anwendung ausführt und zum Beispiel eine neue HTML-Seite generiert. Diese Daten werden dem Client daraufhin im Browser angezeigt, sodass ein neuer Zyklus beginnen kann.

Die Anfrage des Clients – der Request – kann beispielsweise aussehen wie in Listing 3.1.

```
1 GET /widgets HTTP/1.1
2 Host: localhost
3 Accept: text/html
4 Date: Mo, 29 Aug 2016 09:05:42 GMT
```

Listing 3.1: Ein Request

In der ersten Zeile stehen dabei alle Informationen, die für die Anfrage wichtig sind. Sie besteht immer aus einer Methode, einem Uniform Resource Identifier (kurz URI) und einer Protokoll Version.

In diesem Beispiel wird die `GET`-Methode verwendet. Diese ermöglicht es, Dokumente vom Server abzurufen. Weitere Methoden sind die `POST`-Methode, die dafür zuständig ist, Formulardaten vom Client an den Server zu übermitteln, die `PUT`-Methode, die eine Datei auf einen Server laden kann und die `DELETE`-Methode, die ein Dokument auf dem Server löschen kann.

Der URI – hier /widgets – ist der Ort, von dem der Client sein Dokument anfordert. Dazu wird jedoch auch der Host aus Zeile zwei benötigt, sodass sich daraus die Adresse „http://localhost/widgets“ ergibt.

Die Protokoll Version ist schließlich HTTP/1.1.

Unter dieser ersten Zeile können weitere optionale Informationen übergeben werden, wie beispielsweise die Inhaltstypen, die der Client verarbeiten kann oder das Datum, an dem die Anfrage gesendet wurde. Sobald der Server den Request erhalten hat, weiß er, welche Ressource der Client benötigt. Die Antwort – die Response – könnte aussehen wie in Listing 3.2.

```
1  HTTP/1.1 200 OK
2  Date: Mon, 29 Aug 2016 08:05:53 GMT
3  Content-Type: text/html
4
5  <html>
6  <head>
7  <title>widgets</title>
8  </head>
9  <body>
10 <p>Just an example</p>
11 </body>
12 </html>
```

Listing 3.2: Eine Response

In der Response ist zunächst der Statuscode enthalten, der eine Information darüber gibt, ob alles funktioniert hat oder ob Fehler aufgetreten sind. In diesem Beispiel ist es der Code 200, was bedeutet, dass die Anfrage erfolgreich bearbeitet wurde und das Ergebnis in der Antwort übertragen wird. Weitere Codes sind im Anhang A.1 zu finden.

In der Antwort können außerdem, wie in der Anfrage auch, zusätzliche optionale Informationen, wie etwa das Datum der Antwort oder der Inhaltstyp, übermittelt werden. Anschließend folgt der Body-Part, in dem der angeforderte Inhalt übermittelt wird.

3.2 Synchrones Laden

Im klassischen Modell einer Webanwendung findet der HTTP-Request-Cycle bei jedem Klick statt. Der Benutzer klickt auf einen Link, die Seite baut sich auf, der Benutzer entnimmt alle wichtigen Informationen und klickt auf einen neuen Link, sodass der Zyklus von vorne beginnt. Die Datenübertragung findet synchron hintereinander statt, was recht starr erscheint. Die Abbildung 3.1 zeigt den Vorgang.

Synchrones Laden hat jedoch zum Nachteil, dass der Nutzer unter Umständen sehr lange warten muss, zum Beispiel wenn auf eine Datei zugegriffen werden muss, die auf einem fremden Server liegt und nicht direkt zur Verfügung steht.

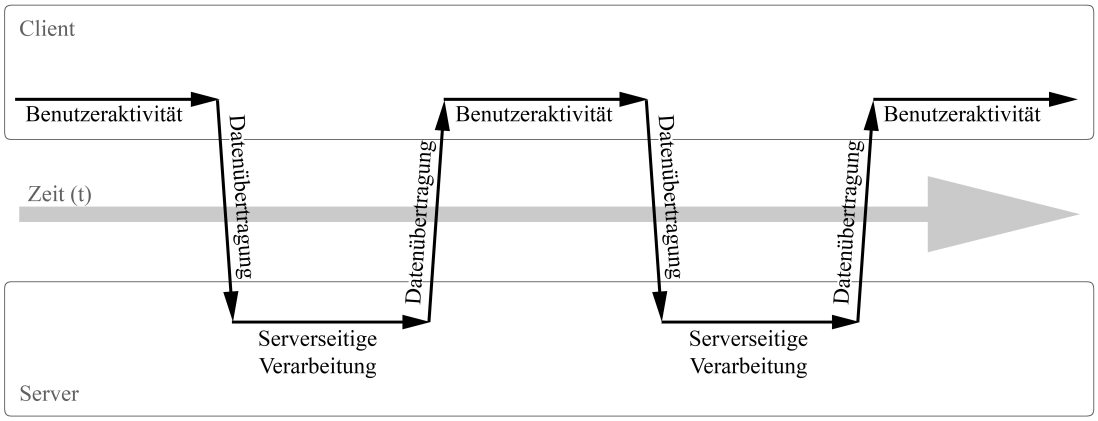

Abbildung 3.1: Synchrone Datenübertragung[13]

3.3 Asynchrones Laden

Bei asynchronen Webanwendungen sind Client- und Serveraufgaben voneinander getrennt. Der Nutzer kann auf diese Weise mit der Oberfläche interagieren, während im Hintergrund Daten zum Beispiel zur Speicherung an den Webserver gesendet werden. Die Abbildung 3.2 zeigt den Vorgang.

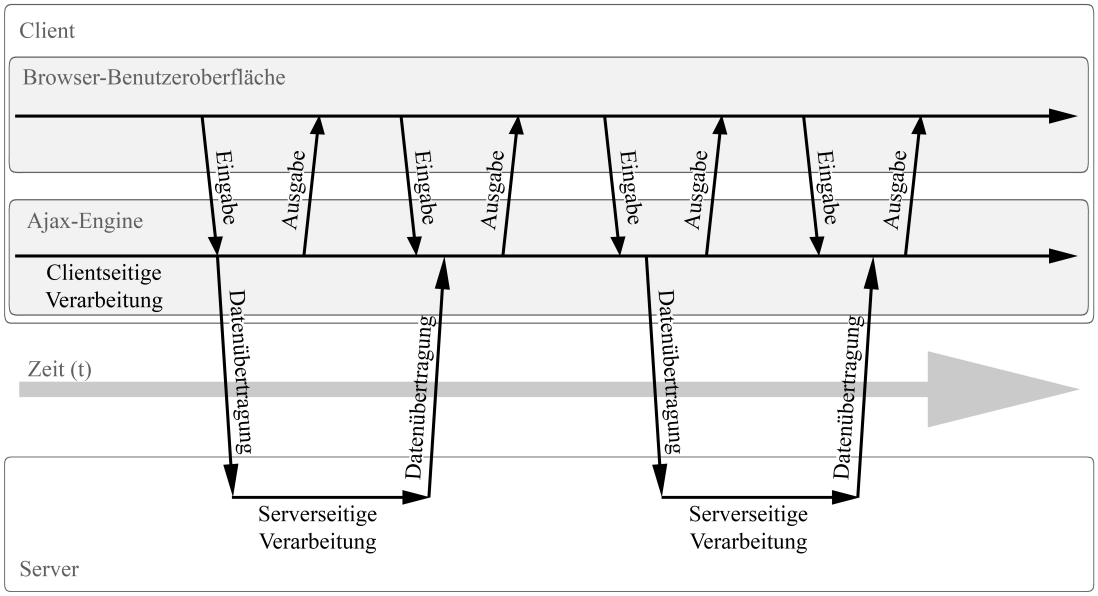

Abbildung 3.2: Asynchrone Datenübertragung[14]

[13] Abbildung in Anlehnung an Garrett, Jesse James (2005)
[14] Abbildung in Anlehnung an Garrett, Jesse James (2005)

Diese parallele Vorgehensweise ist für den Nutzer sehr angenehm, da er nicht warten muss, bis eine Seite komplett neu geladen wurde. Auch wenn noch nicht alle Inhalte zur Verfügung stehen, weil Bilder beispielsweise noch nachgeladen werden, kann er sich bereits einen ersten Eindruck der Seite machen. Somit verbessert sich die Ladezeit und außerdem kann Bandbreite wird eingespart.

Nachteilig kann sich das asynchrone Laden jedoch auf die Leistung des Servers auswirken, je nachdem, wie viele Requests gestellt werden. Wichtig ist außerdem, dass das Caching nicht in Vergessenheit gerät, damit Inhalte nicht doppelt geladen werden und damit unnötiger Traffic entsteht.

Damit eine Seite am Ende durch das Nachladen von Inhalten nicht langsamer wird, wird auf Ajax zurückgegriffen, welches bereits im Abschnitt 2.3 erläutert wurde.

Dabei sollte darauf geachtet werden, dass nur Inhalte nachgeladen werden,

- die selten gebraucht werden, beziehungsweise nur auf speziellen Seiten,
- die durch ihre Größe eine längere Ladezeit haben, wie beispielsweise Bilder oder
- die erst im spätem Verlauf der Seite gebraucht werden und nicht bereits bei Seitenaufruf sichtbar sein müssen.

Für den Benutzer ist es außerdem hilfreich, eine optische Hilfe zu schaffen, die ihn über das Fortschreiten des Ladevorgangs informiert, zum Beispiel indem eine Ladeanimation implementiert wird. Hier ist zu beachten, dass diese nicht zu langsam ausgeführt wird und den Benutzer dadurch behindern.

3.3.1 Lazy Loading

Das sogenannte Lazy Loading ist das bekannteste Entwurfsmuster für asynchrones Laden. Es ist auch als Deferred Loading oder Delayed Loading bekannt. Dabei werden Inhalte nach Bedarf nachgeladen, wie es von Seiten bekannt ist, die scheinbar endlos scrollbar sind. Erst beim Scrollen werden neue Inhalte geladen. Dies beschleunigt die Ladezeit für den Nutzer jedoch nur subjektiv, da für das Laden insgesamt betrachtet genauso viel Zeit gebraucht wird, als wenn es direkt geladen werden würde. Der Nutzer sieht allerdings zunächst schneller ein Ergebnis.

Ein Nachteil bei diesen endlos scheinenden Seiten ist jedoch, dass der Nutzer das Gefühl bekommt, niemals ans Seitenende zu gelangen. Für eine angenehmere Handhabung ist es sinnvoll, nach entsprechender Länge eine Paginierung einzufügen. Außerdem kann das Lazy Loading für eine höhere Last sorgen, wenn mehrere Anfragen gestellt werden, um die Inhalte zu laden. Besonders bei Datenbanken ist dies der Fall, da sie dafür konzipiert sind, viele Daten mit wenigen Anfragen zu bearbeiten. Mehrere Abfragen lassen daher die Belastung der Datenbank steigen.

Das Gegenteil von Lazy Loading ist das Eager Loading, bei dem sofort alle Daten geladen werden, die wahrscheinlich benötigt werden.

3.3.2 Priorisiertes Laden

Beim priorisierten Laden wird eine Reihenfolge vorgegeben, in der die Inhalte geladen werden. Damit lässt sich gut steuern, welche Informationen ein Besucher wahrnehmen soll und worauf seine Aufmerksamkeit gelenkt wird. Dies kann beispielsweise ermöglicht werden, indem besondere Aktionen mit leichter Zeitverzögerung eingeblendet werden.

Ein Nachteil des priorisierten Ladens ist, dass eine Seite hierbei schnell als unruhig und langsam wahrgenommen werden kann. Dies kommt daher, dass sich die Seite während des Ladens immer mehr aufbaut und verändert.

3.3.3 Prediction

Bei der Prediction geht es um die Vorhersage von Inhalten. Dies ist dann hilfreich, wenn man mit großer Wahrscheinlichkeit sagen kann, welche Aktion der Nutzer als nächstes ausführen wird. Die Inhalte werden also nicht bei Bedarf direkt nachgeladen, sondern auf Verdacht vorgeladen. Sie stehen einem Nutzer in diesem Fall dann direkt zur Verfügung. Diese Inhalte werden dabei möglichst in Zeiträumen geholt, in denen aktuell nichts anderes geladen wird. Dadurch hat der Nutzer das Gefühl, dass die Seite eine kurze Ladezeit hat.

Bei der Verwendung von Predictions sollte jedoch genau überlegt sein, ob dieses Vorgehen sinnvoll ist, da gegebenenfalls Inhalte geladen werden, die gar nicht gebraucht werden und somit unnötigen Traffic erzeugen.

3.4 Vor- und Nachteile

Der größte Vorteil beim asynchronen Laden besteht darin, dass Daten nachträglich verändert werden können, ohne die Webseite komplett neu zu laden. Dadurch kann auch auf Benutzereingaben besser und schneller reagiert werden. Außerdem kann der Benutzer bereits relevante Inhalte sehen und nutzen, obwohl noch gar nicht alle Elemente geladen sind. Auch die Handhabung für den Benutzer ist einfach, da die Ajax-Technologie bereits von den Webbrowsern unterstützt wird. Es ist daher keine weitere Installation eines Plugins notwendig. Einzige Voraussetzung ist, dass JavaScript aktiviert sein muss.

Die Tatsache, dass beim asynchronen Laden die Funktionalität der „Zurück“-Schaltfläche im Browser nur schwer zu gewährleisten ist, ist ein Nachteil.[15] Dies ist dadurch bedingt, dass die Browser in der Regel nur statische Seiten in ihrer Historie speichern. Ein Benutzer erwartet, dass er einen vorherigen Schritt durch die Schaltfläche wieder herstellen kann. Dies ist jedoch zusammen mit dynamisch nachgeladenen Inhalten unter Umständen sehr schwer nachzuvollziehen. Dem Browser muss dazu jede Zustandsänderung separat mitgeteilt werden.

[15] vgl. Wenz, Christian (2006)

Wichtig für den Benutzer ist auch eine Rückmeldung über den aktuellen Status des Ladefortschritts, beispielsweise durch einen Ladebalken. Ist dieser nicht vorhanden, kann schnell der Eindruck entstehen, dass die Seite nur sehr zähflüssig lädt.

4 Methoden zum asynchronen Laden

4.1 Callback

Der Begriff Callback ist eine Konvention für die Nutzung von JavaScript Funktionen. Es ist eine Funktion, die einer anderen Funktion als Parameter übergeben wird. Erst nachdem diese andere Funktion ausgeführt wurde, wird die Callback-Funktion aufgerufen. Diese Vorgehensweise beansprucht also etwas mehr Zeit zur Bearbeitung, dafür ist sie asynchron. Dies wird genutzt, um beispielsweise Netzwerkaufrufe oder Benutzereingaben zu verarbeiten.

```
1 $("#btnSave").click(function() {
2     alert("Btn save clicked");
3 });
```

Listing 4.1: Anonyme Callback Funktion

Listing 4.1 zeigt eine bekannte Verwendung in jQuery, nämlich den Klick auf einen Button. Dabei wird der `click()`-Methode eine Funktion als Parameter übergeben, die daraufhin aufgerufen und ausgeführt wird. In diesem Fall ist es eine anonyme Funktion.

Problematik

Es ist möglich, mehrere Callback-Funktionen ineinander zu verschachteln. Dabei entsteht eine pyramidenförmige Struktur im Code, die sogenannte Pyramid of Doom, wie in Listing 4.2 zu erkennen ist. Der Code wird dadurch aber schnell sehr unübersichtlich und lässt sich nur noch schwer lesen. Dieser Umstand wird auch als Callback Hölle (engl. Callback Hell) bezeichnet. Die Funktionalität des Codes leidet zwar nicht darunter, jedoch wird die Wartbarkeit und die Lesbarkeit erschwert.

```
1  loadData(function(paraA){
2      loadMoreData(paraA, function(paraB){
3          loadMoreData(paraB, function(paraC){
4              loadMoreData(paraC, function(paraD){
5                  loadMoreData(paraD, function(paraE){
6                      [...]
7                  });
8              });
9          });
10     });
11 });
```

Listing 4.2: Vereinfachte Darstellung der Pyramid of Doom

Bei der Verwendung von Callbacks lässt sich diese Tatsache nicht umgehen, allerdings ist es möglich, sie etwas aufzubrechen. Gerade wenn bestimmte Funktionen häufiger innerhalb eines Codes genutzt werden, ist es sinnvoll, seperate Funktionen zu schreiben, die dann über den Namen aufgerufen werden. Nimmt man das Beispiel aus Listing 4.1, könnte dies wie folgt aussehen:

```
1 $("#btnSave").click(clickBtn);
2
3 function clickBtn() {
4     alert("Btn clicked");
5 }
```

Listing 4.3: Callback Funktion mit Name

Wird eine Funktion also benannt, wird der Code lesbarer und abstrakter. Außerdem wird eine Wiederholung von gleichen Codezeilen verhindert, sodass das Don't Repeat Yourself-Prinzip (kurz DRY) Anwendung findet.

In Callback Funktionen muss außerdem jeder Error einzeln behandelt werden. Dabei geht es weniger um die Fehler, die ein Programmierer verursacht, sondern um Fehler, die durch das System verursacht werden. Dies sind zum Beispiel Timeouts, ein aufgebrauchter Speicher oder eine fehlgeschlagene Verbindung zum Server. Als Konvention dafür wurde das erste Argument einer Callback Funktion für einen Error reserviert. Eine Funktion mit Fehlerbehandlung ist in Listing 4.4 dargestellt.

```
1 var content = content.load("beispiel.html", loadPage);
2
3 function loadPage(err, content) {
4     if (err) {
5         console.log("Oops, something went wrong", err);
6     } else {
7         console.log("Page loaded");
8     }
9 }
```

Listing 4.4: Callback Funktion mit Fehlerbehandlung

Eine weitere Tücke bei der Verwendung von Callback Funktionen besteht in der Verwendung vom `this`-Keyword, da es nicht immer auf den erwarteten Kontext zeigt. Dazu zeigt Abbildung 4.1 ein einfaches Schema.

Das `this`-Keyword zeigt immer auf die nächste umschließende Funktion. Eine Funktion `function()`, die direkt auf der Seite liegt, verweist mit `this` auf das Fenster. Liegt hingegen zum Beispiel ein Klickhandler in einem bestimmten HTML-Element, so verweist das `this` auf eben dieses. Manchmal ist es jedoch notwendig, auf ein höher liegendes Element zu verweisen. Eine einfache Lösung hierfür ist das Erstellen einer neuen Variable, die ebenfalls auf das gewünschte Objekt zeigt. Diese kann beliebig

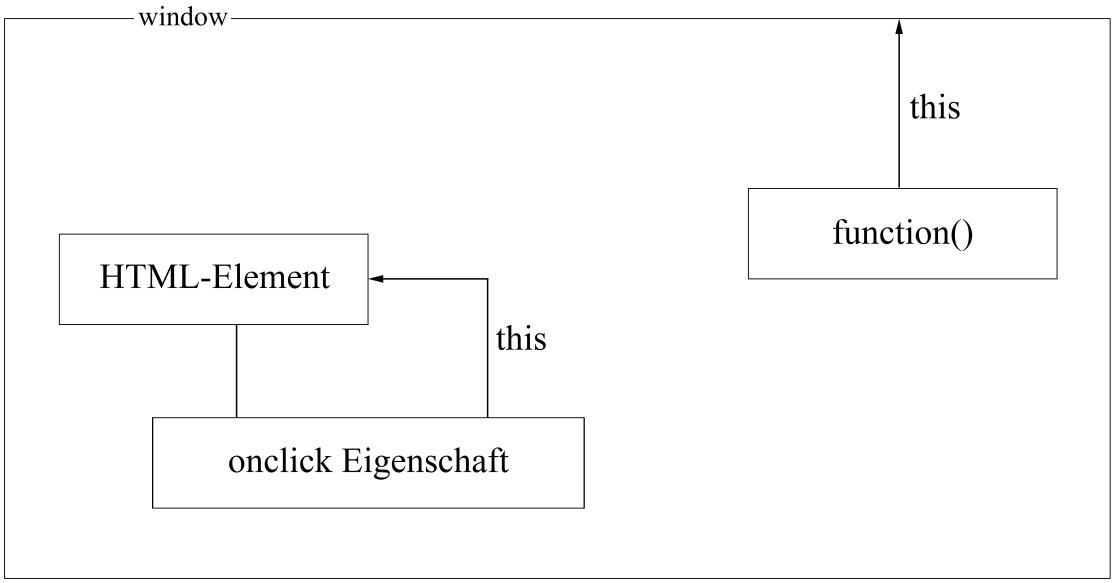

Abbildung 4.1: Schema von this[16]

benannt werden, besonders häufig wird dafür `self` oder `that` verwendet. Dies ist in Listing 4.5 implementiert.

```
1 function MyFunction(data) {
2     this.data = data
3     var self = this;
4     $("#btn").click(function() {
5         console.log(self.data);
6     });
7 }
```

Listing 4.5: Umgang mit this

Callback Funktionen sind ein fundamentaler Teil von JavaScript und bilden den Grundbaustein für alle weiteren Methoden zum asynchronen Laden. Es ist hilfreich, Callbacks zu verstehen, bevor andere Methoden genutzt werden.

4.2 Promise

Ein Promise stellt eine Verbesserung gegenüber Callbacks dar. Beim Aufruf einer asynchronen Funktion wird direkt ein Promise zurückgeliefert, welches entweder durch das Ergebnis einer Operation aufgelöst wird oder mit einem Error verworfen wird. In der Zwischenzeit kann das ausstehende Promise als Platzhalter genutzt werden.

Ein Promise kann einen der folgenden drei Zustände annehmen:

[16] vgl. Koch, Peter-Paul (2006)

1. `pending` – initialer Status
2. `fulfilled` – Operation erfolgreich
3. `rejected` – Operation gescheitert

Außerdem gibt es einen Term mit dem Zustand `settled`. Hierbei ist das Promise entweder `fulfilled` oder `rejected`, aber nicht `pending`.

Das Listing 4.6 zeigt die allgemeine Syntax.

```
1 new Promise(executor);
```

Listing 4.6: Allgemeine Syntax für ein Promise

Der `executor` ist eine Funktion mit zwei Argumenten `resolve` und `reject`, wie in Listing 4.7 dargestellt. `resolve` führt das Promise aus, während `reject` es verwirft.

```
1 new Promise(function(resolve, reject) {...});
```

Listing 4.7: Syntax für ein Promise

Jedes Promise bekommt eine `then()`-Methode, welche es möglich macht, mit dem Promise weiterzuarbeiten. Sie wird aufgerufen, sobald das Promise `resolved` ist. Wird sie `rejected`, wird die `catch()`-Methode aufgerufen. Der Aufbau eines vollständigen Promise wird in Listing 4.8 gezeigt.

```
1  var p = new Promise(function(resolve, reject) {
2      // Do async task
3      if(/* good condition */) {
4          resolve("Success!");
5      } else {
6          reject("Failure!");
7      }
8  });
9  p.then(function() {
10     // Do something with the result
11 }).catch(function() {
12     // Error
13 });
```

Listing 4.8: Vollständiges Promise

Da die Methode `then()` immer ein Promise zurückliefert, können mehrere Methodenaufrufe aneinandergereiht werden, die dann nacheinander ausgeführt werden. Um eine Benachrichtigung darüber zu erlangen, wann alle Ergebnisse vorhanden sind, kann `Promise.all()` genutzt werden. Eine Implementierung zeigt das Listing 4.9. Dabei ist die Eingabe ein Array von Promises und die Ausgabe ein einzelnes Promise, welches ein Array der Ergebnisse enthält.

```
1 Promise.all([func1(), func2()])
2 .then(function([result1, result2]) {
3     // Some Code
4 }).catch(function(err) {
5     //Some Code
6 });
```

Listing 4.9: Verwendung von Promise.all()

Sehr nützlich ist auch die Funktion `Promise.race(iterable)`. Anstatt auf alle Promises zu warten, bis sie ausgeführt oder verworfen wurden, gibt sie das erste Promise zurück, welches `resolved` oder `rejected` wurde.

Die Funktion `Promise.reject(reason)` gibt das Promise zurück, welches aus dem angegebenen Grund verworfen wurde.

Im Gegensatz dazu gibt die Funktion `Promise.resolve(value)` ein Promise zurück, das mit `value` aufgelöst wird. Besitzt das Objekt eine `then()`-Methode, wird diese ausgeführt und übernimmt dessen Status. Ist dies nicht der Fall, wird das Promise auf `fulfilled` gesetzt.

Im Vergleich zu Callbacks haben Promises folgende Vorteile:

- Eine Verkettung wird einfacher realisierbar. Wenn die `then()`-Methode auf eine andere asynchrone Funktion verweist, die wieder ein Promise zurückliefert, wird dieses zurückgegeben, sodass direkt eine neue `then()`-Methode folgen kann.
- Das Zusammensetzen von asynchronen Aufrufen wird erleichtert, da direkt Platzhalter vorhanden sind, mit denen vorerst gearbeitet werden kann.
- Die Fehlerbehandlung ist einfacher.
- Es gibt keine Umkehrung der Steuerung (engl. Inversion of Control), da eine Funktion mit Promise – im Gegensatz zu einem Callback – mit Sicherheit immer ein Ergebnis zurückliefert. Bei einem Callback wird lediglich darauf vertraut, dass die Funktion aufgerufen wird. Es gibt keine Garantie, dass die mitgelieferten Parameter richtig sind oder dass überhaupt ein Ergebnis zurückgeliefert wird.

Probleme

Da Promises ein ECMAScript 6 Feature sind, sind sie nicht mit allen Browsern kompatibel. Der Internet Explorer unterstützt diese nicht. Promises/A+ hingegen ist ein offener Standard, der bereits im ECMAScript 5.1 als Zusatzbibliothek hinzugefügt werden konnte und mehr Browser unterstützt als das ECMAScript 6. Eine weitere Bibliothek ist zum Beispiel die Q-Library, die es ermöglicht, Promises auch mit jQuery zu nutzen.

4.3 Generatorfunktion und yield

Die Generatorfunktionen und das Schlüsselwort `yield` sind weitere neue Sprachmerkmale aus dem ECMAScript 6. Dabei soll bei letzterem eine Funktion während der Ausführung unterbrochen werden, um einen Wert bereits vorzeitig auszugeben. Dies ist besonders sinnvoll, wenn es sich um eine Funktion handelt, bei der größere Werte oder Intervalle berechnet werden müssen, wie zum Beispiel die Berechnung der Primzahlen. Hier würden schon alle bereits berechneten Zahlen ausgegeben werden, während die Berechnung der verbleibenden Zahlen noch stattfindet.

Das Schlüsselwort `yield` verhält sich also wie eine `return`-Anweisung. Im Gegensatz dazu speichert es jedoch den Zustand, sodass die Funktion zu einem späteren Zeitpunkt fortgesetzt werden kann. Dies funktioniert allerdings nicht in jeder beliebigen Funktion, sondern nur in den sogenannten Generatorfunktionen, die in JavaScript mit `function*()` statt `function()` definiert werden. Das Listing 4.10 zeigt eine solche Generatorfunktion.

```
1 let getPrimes = function*(min, max) {
2     for(let num = min; num <= max; num++) {
3         if(isPrime(num)) {
4             yield num;
5         }
6     }
7 }
```

Listing 4.10: Aufbau einer Generatorfunktion

Eine Generatorfunktion gibt zunächst ein Iteratorobjekt zurück und führt nicht, wie bei normalen Funktionen, zunächst den enthaltenen Code aus. Auf diesem kann bis zum ersten Aufruf vom Schlüsselwort `yield` die Funktion `next()` aufgerufen werden, die die eigentliche Funktion ausführt. Solange diese nicht aufgerufen wird, wird die Ausführung also nicht fortgesetzt, wodurch eine Verzögerung beim Erstellen der Daten erreicht wird.

```
1 let iterator = getPrimes(1,10);
2
3 console.log(iterator.next());
4 // => {value: 2, done: false}
```

Listing 4.11: Iterator mit Aufruf von next()

Der Wert, der von `next()` zurückgegeben wird, ist ein Objekt mit einem value und einem done-Flag, wie in Listing 4.11 zu sehen ist. Wird die `next()`-Funktion erneut aufgerufen, wird das Programm an der Stelle weiter ausgeführt, an der es unterbrochen wurde. Trifft es dabei auf ein `yield` hält es erneut an. Ebenso stoppt das Programm, wenn es auf das Codeende trifft. Ein weiterer Aufruf von `next()` nach Programmende wirft eine Exception. Diesen Vorgang zeigt das Listing 4.12.

```
1 console.log(iterator.next()); // => {value: 3, done: false}
2 console.log(iterator.next()); // => {value: 5, done: false}
3 console.log(iterator.next()); // => {value: 7, done: false}
4 console.log(iterator.next()); // => {value: undefined, done: true}
```

Listing 4.12: Weitere Aufrufe von next()

Mit Generatorfunktionen ist es außerdem möglich, in einer Schleife eine endlose Zahlenreihe ausgeben lassen. Dabei wird der Browser jedoch nicht blockiert, da die Schleife nach jedem Durchlauf mit `yield` verlassen wird. Erst mit dem nächsten Aufruf von `next()` wird sie wieder betreten.

Zur Vereinfachung ist in ECMAScript 6 die in Listing 4.13 genutzte for-of-Schleife implementiert, die einen Iterator erzeugt und anschließend durchläuft.

```
1 for(let prime of getPrimes(1, 10)) {
2     console.log(prime);
3 }
4 // => 2, 3, 5, 7
```

Listing 4.13: Die for-of-Schleife

In der `next()`-Funktion können außerdem Parameter übergeben werden, die einen Abbruch von außen möglich machen. Diese stehen als Rückgabewert von `yield` zur Verfügung. Ein Programm zur Berechnung der Primzahlen kann beispielsweise nach der Ausgabe der ersten drei Zahlen abgebrochen werden.

Oftmals ist es erforderlich, mehr als eine Aufgabe parallel auszuführen. Mit dem Schlüsselwort `yield` kann jedoch nur ein einzelner Abschnitt pausiert werden. Es ist allerdings möglich, Generatorfunktionen und Promises gemeinsam zu nutzen. Ein asynchroner Code erhält dadurch die Optik eines synchronen Codes und kann besser nachvollzogen werden. Dazu werden Promises ausgeliefert, die erst weiterverarbeitet werden, sobald sie erfüllt sind.

```
1  function request(url) {
2      return new Promise(function(resolve,reject){
3          makeAjaxCall(url, resolve);
4      });
5  }
6
7  runGenerator(function *main(){
8      var result1 = yield request("http://some-url.com");
9      var data = JSON.parse(result1);
10     var result2 = yield request("http://some-url.com?id=" + data.id);
11     var resp = JSON.parse(result2);
12     console.log("The value you asked for: " + resp.value);
13 });
```

Listing 4.14: Die Verwendung von Promises in einer Generatorfunktion

Das Listing 4.14 ist ein Beispiel, welches die Generatorfunktion zusammen mit einem Promise zeigt. In der Generatorfunktion `runGenerator()` wird die Funktion `request()` aufgerufen und in `result1` gespeichert. Die Funktion `request()` liefert ein Promise zurück, sobald die Funktion `makeAjaxCall()` erfüllt wurde. Solange dies nicht der Fall ist, wird die weitere Ausführung der `runGenerator()` Funktion durch das Schlüsselwort `yield` gestoppt. Wurde ein Promise zurückgeliefert, wird dessen Inhalt in `data` in ein JSON-Objekt übersetzt. Für die nächste Anfrage wird schließlich die `id` aus `data` benötigt. Da die Generatorfunktion bei der Anfrage für `result1` solange pausiert hat, bis dort ein Ergebnis vorhanden war, kann die `data.id` entsprechend für `result2` eingesetzt und richtig ausgeführt werden. Die Konsole gibt schließlich den gewünschten Inhalt aus.

Probleme

Generatorfunktionen sind nicht kompatibel mit dem Internet Explorer und Safari.

4.4 async und await

Das ECMAScript 7 brachte die Schlüsselwörter `async` und `await` mit sich, um asynchronen Code zu schreiben, der jedoch an die synchrone Codestruktur angelehnt ist. Die Grundlage für diese Vorgehensweise bilden Promises, denn jede `async` Funktion gibt ein Promise zurück und jedes `await` ist für gewöhnlich ein Promise. Dabei funktioniert das `await` wie das `then()` bei einem Promise. Das folgende Listing 4.15 zeigt eine Funktion mit Promise, das Listing 4.16 zeigt den gleichen Ablauf mit async und await.

```
1 function getFirstUser() {
2     return getUsers().then(function(users) {
3         return users[0].name;
4     }).catch(function(err) {
5         return {name: "default user"};
6     });
7 }
```

Listing 4.15: Beispielfunktion mit Promise

```
1 async function getFirstUser() {
2     try {
3         let users = await getUsers();
4         return users[0].name;
5     } catch (err) {
6         return {name: "default user"};
7     }
8 }
```

Listing 4.16: Beispielfunktion mit async und await

Anhand dieser Beispiele ist gut zu erkennen, dass das `await` wie die `then()`-Methode verwendet wird. Es stoppt die Ausführung solange, bis der Wert vorhanden ist. Außerdem kann wie bei synchronem Code ein `try/catch`-Block verwendet werden, um Fehler abzufangen.

Ein häufig gemachter Fehler besteht darin, ein `await` alleine zu benutzen. Ähnlich wie bei den Generatorfunktionen und dem `yield` muss die Funktion für ein `await` als `async` deklariert sein. Andernfalls wird ein Fehler geworfen. Es ist jedoch möglich, eine `async` Funktion ohne ein `await` zu benutzten. Dabei muss allerdings bedacht werden, dass die Funktion dann selbstverständlich nicht auf einen Wert wartet, sondern stattdessen ein Promise zurückgibt.

In jeder `async` Funktion kann jedoch nur ein einziges `await` zur Zeit ausgeführt werden.

```
1 let users = await getUsers();
2 let countries = await getCountries();
```

Listing 4.17: Verwendung von zwei Aufrufen mit await

Beim Listing 4.17 würde also zunächst die erste Zeile ausgeführt werden und erst wenn diese fertig ist, würde die zweite Zeile ausgeführt werden. Dies kann allerdings nützlich sein, wenn eben diese vorherigen Daten für den weiteren Ladeverlauf wichtig sind.

4.5 Zusammenfassung

Da sich das ECMAScript in einer ständigen Weiterentwicklung befindet, werden auch die Methoden zum asynchronen Laden immer ausgereifter. Besonders deutlich zu erkennen ist die aufgeräumtere Struktur, wodurch das Lesen des Codes erleichtert wird.

Die vorgestellten Methoden sind jedoch nicht völlig unabhängig voneinander. Viel mehr bauen sie aufeinander auf und ergänzen sich gegenseitig. In Kombination miteinander führen sie zu einem eleganten Weg, Inhalte asynchron zu laden.

Allerdings sind nicht alle Browser dazu fähig, die verschiedenen Standards des ECMAScripts mit ihren Eigenschaften zu unterstützen. Besonders der Internet Explorer ist hier noch recht schlecht aufgestellt. Für die Nutzung der vorgestellten Methoden sollte entsprechend darauf geachtet werden, wer die Zielgruppe der Anwendung ist. Möglicherweise ist dadurch die Nutzung von neueren Methoden nicht sinnvoll.

Die Tabelle 4.1 stellt noch einmal alle Vor- und Nachteile der jeweiligen Methoden gegenüber.

Vor- und Nachteile der Methoden zum asynchronen Laden

Methode	Vorteile	Nachteile
Callbacks	- Callbacks stellen die erste Möglichkeit zum asynchronen Laden dar	- Die Callback Hölle: der Code wird schnell unübersichtlich, wenn mehrere Funktionen ineinander verschachtelt werden - Jeder Error muss manuell behandelt werden - Die Verwendung von this() benötigt große Achtsamkeit, damit auf das richtige Element gezeigt wird - Es ist unklar, wann und ob ein Ergebnis zurückgeliefert wird
Promises	- Durch die `then()` -Methode wird eine Verkettung mehrerer asynchroner Funktionen einfacher - Ein Promise liefert immer Daten zurück, mit denen gearbeitet werden kann, sodass das Zusammensetzen von asynchronen Aufrufen erleichtert wird - Die Fehlerbehandlung wird erleichtert - Es besteht keine Gefahr der Inversion of Control	- Der Codefluss ist nicht intuitiv, da er vom synchronen Code abweicht

Generators und yield	- Funktionen lassen sich anhalten, um Werte bereits vorzeitig auszugeben - Die Verwendung von Iteratoren erleichtert die Programmierung - Der Code sieht synchron aus	- Generatorfunktionen werden nur im Zusammenspiel mit Promises wirklich asynchron
async und await	- Einfache Fehlerbehandlung - Der Code ist deutlich lesbarer und einfacher zu verstehen	

Tabelle 4.1: Vor- und Nachteile der verschiedenen Methoden zum asynchronen Laden

5 Praktischer Anwendungsfall

5.1 Anforderungen

Widgets sind bei veeseo kleine Boxen, die meist unterhalb eines Artikels eingebaut sind und generierte Artikel- oder Videoempfehlungen beinhalten. Zusätzlich können diese jedoch auch Sponsored Posts und/oder Werbung enthalten. Für die Umsetzung muss es einen Skriptaufruf mittels JSONP geben, der die angeforderten Daten vom Server holt. Diese müssen anschließend in einer entsprechenden Funktion weiterverarbeitet und dargestellt werden.

Dabei gibt es folgende Dinge, die es zu beachten gilt:

- die Quellen dieser drei Inhalte kommen von drei verschiedenen Domains
- die Rückmeldungen der Domains erfolgen in unterschiedlichen Geschwindigkeiten
- die Reihenfolge der Rückmeldungen ist nicht vorhersagbar
- die Quellen dürfen nicht voneinander abhängig sein

Die Reihenfolge spielt jedoch für die Anzeige der Widgets eine große Rolle, denn Empfehlungen müssen immer angezeigt werden, während die Sponsored Posts und die Werbung nicht zwingend notwendig sind. Nutzer, die Adblock verwenden, verhindern sogar, dass die Werbung eine Rückmeldung liefern kann, da durch Adblock keine Methode dafür aufgerufen werden kann. Sponsored Posts sind keine speziellen Empfehlungen zu einem Artikel, sondern bestehen aus Inhalten, die promoted werden sollen.

Die Reihenfolge soll daher folgende sein:

1. Empfehlungen
2. Werbung
3. Sponsored Posts

Gibt es keine Empfehlungen, soll auch das Widget nicht angezeigt werden.

Kommen die Sponsored Posts vor der Werbung zurück, sollen diese zunächst zurückgehalten werden und eine gewisse Zeit warten, damit die Werbung geladen werden kann.

Entsteht nun der Fall, dass die Werbung erst nach Ablauf dieser Zeit eine Rückmeldung liefert, weil der Server beispielsweise gerade überlastet ist, dann müssen

alle Elemente umsortiert werden. Dies hat zur Folge, dass alle Elemente noch einmal angefasst und überprüft werden müssen, um sie richtig anzuordnen.

Um diese Reihenfolge zu sicher zu bewahren, wäre es einfach, die Quellen voneinander abhängig zu machen. Dies ist jedoch für den Widgetcode nicht möglich, da bei einem Ausfall einer Quelle auch die nachfolgenden davon beeinträchtigt wären. Im schlimmsten Fall würde das Widget gar nicht angezeigt werden, weil der Code niemals zum Ende gelangen würde. Dieses Verhalten ist natürlich unerwünscht, weshalb es eine Anforderung ist, dass die Quellen nicht voneinander abhängig sein dürfen.

Sobald das Widget unter einem Artikel angezeigt wird, wird das Tracking ausgelöst, damit die Click-Through-Rate (kurz CTR) berechnet werden kann.

5.2 Ist-Zustand

Die Darstellung der Widgets wird nach Kundenwünschen im Widgetgenerator – einem internen Tool – festgelegt. Dort finden sämtliche CSS Anpassungen statt, aber auch Angaben über die Platzierung und Anzahl der Werbung und Sponsored Posts. Mithilfe des Widgetgenerators wird ein Code generiert, den der Kunde anschließend auf seiner Seite einbaut. Da auf einer Seite auch mehrere Widgets an unterschiedlichen Stellen eingebaut sein können, hat jedes Widget eine eindeutige ID, sodass es wiedererkannt werden kann.

In Abbildung 5.1 ist der vereinfachte Ablauf dargestellt.

Der Widgetgenerator erzeugt für jeden Kunden eine Konfigurationsdatei mit allen wichtigen Informationen, die angegeben wurden. Anschließend wird die Funktion `setConfig()` aufgerufen. Sie nimmt alle übermittelten Daten entgegen und holt sich die aktuelle Seitenadresse.

Trifft die Bedingung

```
1 if (typeof this.config.refreshWidgetsAfterLoad !== "undefined" && this
    .config.refreshWidgetsAfterLoad === "1")
```

Listing 5.1: Bedingung für den Aufruf von showNewWidgets()

zu, wird die Funktion `showNewWidgets()` aufgerufen, die sich mit der Funktion `getActiveWidgets()` alle aktiven Widgets holt und für diese anschließend ihren Inhalt mit `getWidgetContents()` abruft.

Ist die Bedingung nicht zutreffend, wird direkt `getActiveWidgets()` aufgerufen. Wenn alle aktiven Widgets abgerufen wurden, gibt es die Bedingung

```
1 if (this.findWidgetHideTag() === false && this.checkCondition())
```

Listing 5.2: Bedingung für den Aufruf von getWidgetContents()

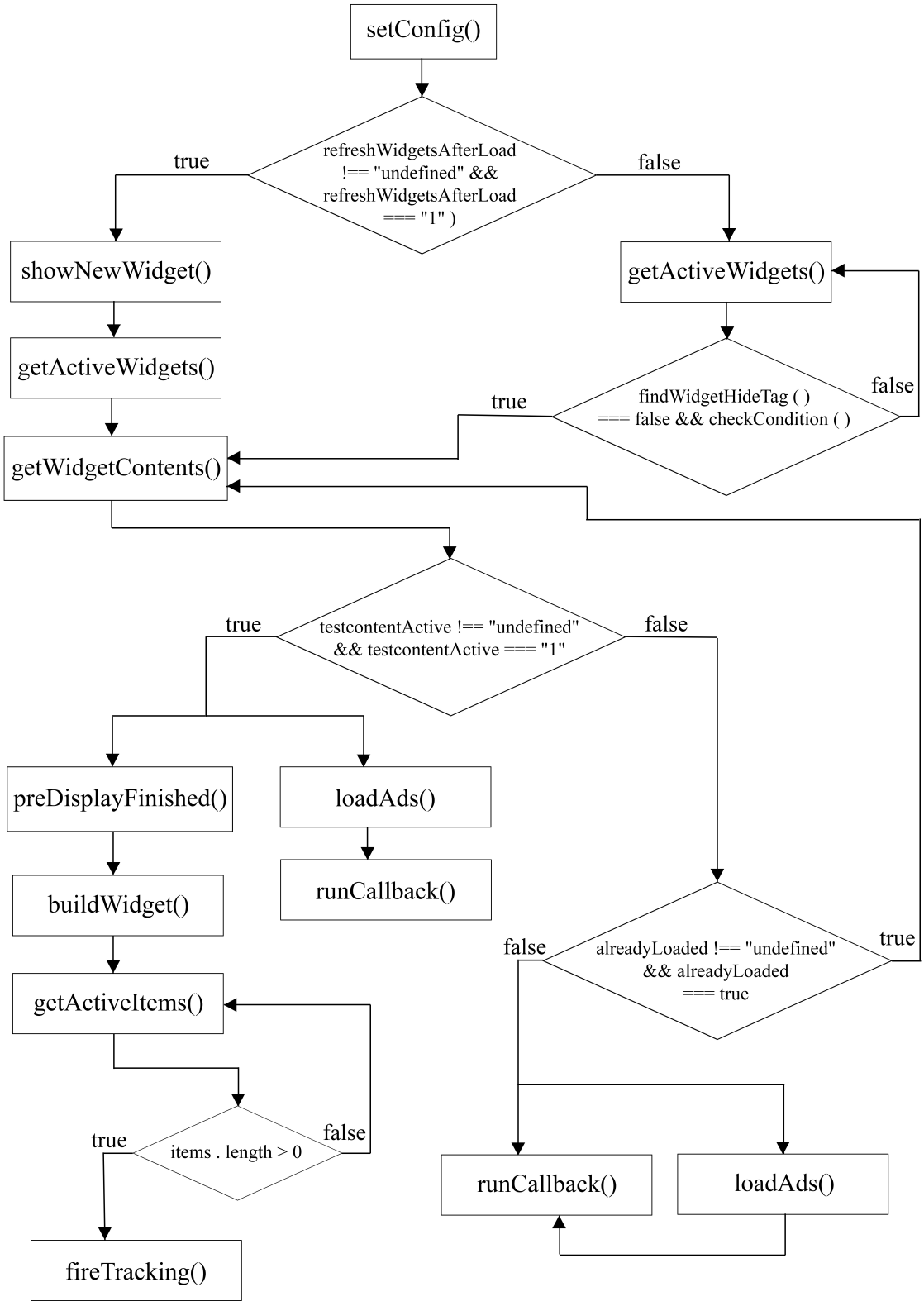

Abbildung 5.1: Ablauf des Widgetcodes

Die Funktion `findWidgetHideTag()` gibt `true` zurück, wenn auf der Seite kein spezieller HTML-Tag gefunden wurde, der dem Widget signalisiert, dass es auf dieser Seite nicht angezeigt werden soll.

Die Funktion `checkCondition()` gibt `true` zurück, wenn sich das Widget nicht auf einer speziell gefilterten URL befindet und wenn es auf der Artikelseite ein sichtbares Datum gibt.

Trifft die Bedingung zu, wird `getWidgetContents()` aufgerufen.

getActiveWidgets()

Die Funktion `getActiveWidgets()` erstellt eine Liste mit allen aktiven Widgets. Dafür werden alle Widgets durchsucht, ob sie eine veeseo ID haben. Ist dies der Fall, wird weiter geprüft, ob das jeweilige Widget eine Klasse `veeseoWidget` besitzt. Trifft auch das zu, wird `alreadyLoaded` auf `true` gesetzt.

getWidgetContents()

Die Funktion `getWidgetContents()` lädt für das entsprechende Widget die Empfehlungen, sowie eventuell Sponsored Posts und die Werbung. Hier werden zunächst alle aktiven Widgets abgerufen und prüft für jedes einzelne Widget, ob es eine Klasse `veeseoWidget` hat. Für jedes Widget gibt es anschließend zwei Fälle:

Fall 1: Test-Content ist aktiviert

Test-Content sind einzelne Artikel aus der Seite des Kunden, die im Widgetgenerator angelegt werden. Dieser Inhalt dient dazu, dem Kunden vorab ein Beispielbild zu zeigen, wie das Widget am Ende auf seiner Seite eingebunden aussehen würde. Erst wenn der Kunde die endgültige Zustimmung gibt, werden Artikel aus der Seite ausgelesen und im System mit Keywords versehen, die für die Empfehlungen benötigt werden. Ist der Test-Content jedoch noch aktiv, werden keine Empfehlungen ausgespielt, sondern entsprechend der Test-Content. Dabei wird `isFinal` auf `false` gesetzt. Um trotzdem das Widget sehen zu können, wird der Test-Content per JSON geparst. Außerdem werden die Funktionen `preDisplayFinished()` und `loadAds()` aufgerufen. Dabei wird in `preDisplayFinished()` die Variable `preDisplayDone` auf `true` gesetzt und die Funktion `buildWidget()` aufgerufen. Die Funktion `loadAds()` entnimmt die aus dem Widgetgenerator eingetragene Quelle für die Werbung. Diese ist zumeist von Ligatus und hat eine sogenannte LigatusID. Aus diesen Informationen wird eine URL erzeugt und anschließend der Funktion `runCallback()` übergeben. `runCallback()` wiederum erstellt mittels JSONP ein Skript zum Datenaustausch.

Fall 2: Test-Content ist nicht mehr aktiviert

An dieser Stelle werden einige Variablen überprüft und eventuell angepasst. Schließlich kommt folgende Bedingung:

```
1 if (typeof widget.alreadyLoaded !== "undefined" && widget.
    alreadyLoaded === true)
```

Listing 5.3: Bedingung für das Laden neuer Widgets

Trifft dies zu, wird der komplette vorher beschriebene Vorgang für das nächste Widget ausgeführt. Wird die Bedingung jedoch nicht erfüllt, so wird zunächst eine URL aus allen gesammelten Informationen zusammengestellt und `runCallback()` übergeben. Dann wird geprüft, ob Sponsored Posts aktiviert sind und wenn ja, wird ebenfalls eine URL erstellt und an `runCallback()` übermittelt. Zum Schluss wird `loadAds()` aufgerufen, welche sich wieder die Quelle der Werbung holt und daraufhin mithilfe der LigatusID eine URL erstellt, die ebenfalls an `runCallback()` übermittelt wird.

buildWidget()

In der Funktion `buildWidget()` wird das Widget erstellt. Hier wird geprüft, welche Inhalte in der zu generierenden HTML- und CSS-Struktur enthalten sein müssen, wie zum Beispiel ein Titel oder eine Beschreibung. Außerdem wird die Funktion `getActiveItems()` geladen, die alle bereits geladenen Empfehlungen in einem Objekt zurückliefert.

Anschließend wird die Länge der Items geprüft und wenn sie größer Null ist, wird das Tracking ausgelöst.

```
1 if (items.length > 0) {
2     widget.tracking.fireTracking("load", items);
3 }
```

Listing 5.4: Aufruf der fireTracking() Funktion

Die Funktion `fireTracking()` erstellt die URL, mit der jedes Item verfolgt werden kann.

Schließlich wird das CSS erstellt. Es gibt Kunden, die ihr eigenes CSS verwenden möchten. Für diesen Fall gibt es im Widgetgenerator ein Feld, für das ein Haken gesetzt werden kann, sodass das Widget ohne einen Style ausgeliefert wird.

5.3 Schwachstellen

- Die Quellen für die Empfehlungen, Sponsored Posts und die Werbung werden zwar unabhängig voneinander geladen, allerdings kann nicht sicher gesagt werden, wann und ob sie eine Rückmeldung geben. Das Widget wird deshalb eventuell sogar gar nicht angezeigt, nämlich in dem Fall, wenn die Empfehlungen zu lange benötigen. Dadurch besteht die Gefahr, dass der Besucher die Seite bereits wieder verlassen hat und der Kunde keine weiteren Klicks bekommt.

- Mit der Anzeige des Widgets wird das Tracking ausgelöst. Dies stellt ein Problem dar, wenn beispielsweise die Werbung länger braucht und mit einiger Verzögerung erst geladen werden kann. Durch das dann notwendige Umsortieren der Elemente, werden die dann überschüssigen Items weggekürzt. Das ist deshalb problematisch, weil auch diese Elemente bereits ins Tracking aufgenommen worden sind und nun keine Chance mehr haben, angeklickt zu werden. Die daraus resultierende CTR ist daher sehr schlecht, sodass der Artikel nach einiger Zeit aus dem System entfernt wird, da er nicht oft genug geklickt wird und daher keinen Umsatz erzeugt. Möglicherweise ist der Inhalt dieses Artikels jedoch sehr interessant und hätte das Potential, viele Klicks zu erzeugen.

5.4 Prototypische Implementierung

Die Vorlage für diese Umsetzung bildet das aktuelle Widget. Da es jedoch zunächst nur eine prototypische Implementierung ist, wurden hier nur die wichtigsten Bestandteile berücksichtigt, sowie Vereinfachungen vorgenommen. Als Plattform für die Entwicklung wurde die App Engine von Google genutzt, die eine Webanwendung auf einem Server darstellen kann. Aus Datenschutzgründen werden keine Livedaten verwendet, sondern Beispieltexte genutzt, die, wie in Listing 5.5 gezeigt, aus einer JSON Datei geladen werden.

```
[
    {
        "Title": "Example Title 1",
        "Description": "Example Description 1",
        "Thumbnail": "/static/img/IMG_2335.jpg"
    },
    {
        // Weitere Beispielempfehlungen
    }
]
```

Listing 5.5: JSON-Datei der Beispielempfehlungen

Zu den Vereinfachungen zählen neben der Datenbeschaffung auch die Konfigurationen, die der Widgetgenerator sonst übernimmt. Da für dieses Beispiel nur die Anzahl der Empfehlungen und die Position der Werbeanzeigen wichtig ist, wird dieses im Konstruktor definiert.

```
constructor() {
    var self = this;
    this.ItemsToDisplay = 3;
    this.AdPositions = [1];
    this.init()
}
```

Listing 5.6: Der Konstruktor des Codes

Das Listing 5.6 zeigt den Konstruktor. Dabei gibt `ItemsToDisplay` die Anzahl der Empfehlungen an, die die Länge des Widgets bestimmt. Dies ist deshalb wichtig, da für einen Artikel mehrere Empfehlungen generiert werden. Die Variable `AdPositions` ist eine Liste, in der die Position der Werbung festgelegt wird. Gleichzeitig kann hier bestimmt werden, wie viele Werbeanzeigen ausgespielt werden sollen. Die Position des ersten Items ist dabei null. Daraus folgt, dass die Werbung aus diesem Ausschnitt das zweite Item sein soll. In der Funktion `init()` wird die Logik zum Anzeigen und Sortieren ausgeführt. Diese Funktion weist das Schlüsselwort `async` auf.

Der weitere Vorgang wird in Listing 5.7 gezeigt.

```
1  let allRecommendations = await Data.get("/js/sampledata/
       recommendations.json");
2
3  let html = await Template.load("widget");
4  $("widgetContainer").html(html({}));
5
6  let recs = [];
7  for(let i = 0; i < this.ItemsToDisplay; i++) {
8      recs.push(allRecommendations[i]);
9  }
10
11 html = await Template.load("rec");
12 $("widgetContainer widget ul").append(html({Recommendations: recs}));
13
14 let displayRecs = [];
15 let finalRecList = [];
16 $("widgetContainer widget ul li").each(function() {
17     displayRecs.push(this);
18     finalRecList.push(this);
19 });
```

Listing 5.7: Laden der Empfehlungen

Durch das Schlüsselwort `await` wird sichergestellt, dass die Daten vollständig vorhanden sind, ehe damit weitergearbeitet wird. Dabei werden zunächst die Empfehlungen geholt, da ein Widget nur angezeigt wird, wenn es auch Empfehlungen zu diesem Artikel gibt. Außerdem wird das Widget-Template geladen, welches bereits gestyled ist. Auch diese Informationen werden sonst im Widgetgenerator festgelegt und daraus generiert. Dieses Template wird zunächst unbefüllt in die Artikelseite eingebunden.

Da für einen Artikel oftmals mehr Empfehlungen vorhanden sind, als ausgespielt werden sollen, werden nun in einer for-Schleife so lange Empfehlungen in eine Liste `recs` geschoben, bis die angegebene Zahl aus ItemsToDisplay erreicht ist. Nach Abschluss werden diese dann in das Template eingefügt. Außerdem werden zwei Listen `displayRecs` und `finalRecList` angelegt, die beide mit den gerenderten Empfehlungen befüllt werden, die im Widget angezeigt werden. Dies ist notwendig, damit es beim späteren Umsortieren nicht zu Schwierigkeiten kommt. Die Liste `finalRecList` wird benötigt, damit aus ihr die Elemente entnommen werden können, die nach einer

erfolgreichen Sortierung gelöscht werden können.

Es ist nicht ungewöhnlich, dass die Werbung erst fertig geladen ist, nachdem das Widget mit den Empfehlungen schon angezeigt wird. Dies ist dadurch zu erkennen, dass sich der Inhalt des Widgets nochmal verändert und an einer oder auch mehreren Positionen plötzlich Werbeanzeigen platziert werden. Dazu müssen die Items umsortiert werden, da die Empfehlungen nach Relevanz sortiert sind. An der ersten Position steht also die Empfehlung mit der Übereinstimmung mit dem Artikel. Aus diesem Grund werden die Elemente verschoben und nicht einfach überschrieben.

Nachdem die Anzeigen vollständig vorhanden sind, wird in einer weiteren Schleife jeder einzelnen Werbeposition, die aus `AdPosition` entnommen wird, ein Item zugeordnet. Dieses wird schließlich an die richtige Stelle in der Liste `displayRecs` eingefügt und zwar vor dem Element, das aktuell an dieser Stelle positioniert ist. Dadurch werden jedoch mehr Items ins Widget eingespeist, als in `ItemsToDisplay` festgelegt sind. Damit die Anzahl im Widget dennoch stimmt, muss die Empfehlung an der letzten Position entfernt werden. Wird diese Empfehlung jedoch direkt gelöscht, könnten Probleme bei mehreren Anzeigen auftreten, da die Position, vor die die Werbung geschoben werden soll, möglicherweise nicht mehr vorhanden ist. Um dies zu umgehen, werden die Items, die später gelöscht werden sollen, zunächst in einer weiteren Liste `recsToRemove` abgelegt. Erst nachdem alle Anzeigen an ihrer richtigen Position angeordnet sind, werden die Items, die sich in der Liste `recsToRemove` befinden, gelöscht. Somit stimmt die Anzahl der Items im Widget wieder.

Dieser Vorgang wird in Listing 5.8 aufgezeigt.

```
1  for(let k in this.AdPositions) {
2      let pos = this.AdPositions[k];
3      $(displayRecs[pos]).before(html({Ad: ads[k]}));
4      recsToRemove.push($(finalRecList[finalRecList.length-1]));
5      finalRecList.pop(finalRecList.length-1);
6  }
7
8  for(let k in recsToRemove) {
9      $(recsToRemove[k]).remove();
10 }
```

Listing 5.8: Umsortieren der Items

Da die Daten in diesem Prototypen alle von der gleichen Domain kommen, gibt es kaum Zeitverzögerung bei der Darstellung. Damit jedoch der Effekt des Nachladens und Umsortierens zu Vorzeigezwecken sichtbar gemacht werden kann, wird mit Timeouts gearbeitet, die die weitere Ausführung des Codes verzögern.

Das fertige Widget unter einem Artikel ist in Abbildung 5.2 zu sehen.

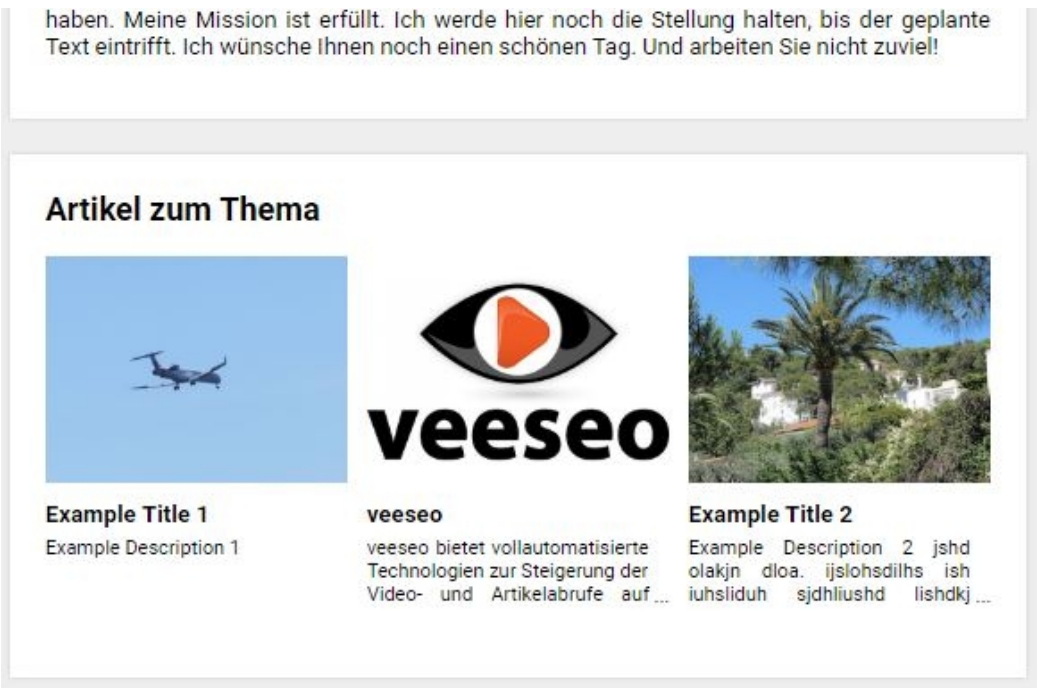

Abbildung 5.2: Das Widget unter einem Artikel, komplette Ansicht in Anhang A.2

6 Schluss

6.1 Evaluation

Im Rahmen der Bachelor-Thesis wurden verschiedene Methoden analysiert, die das asynchrone Laden ermöglichen. Diese sind im einzelnen Callbacks, Promises, Generatorfunktionen und ihr Schlüsselwort yield, sowie die noch sehr neue Methodik von async und await. Es ist deutlich geworden, dass jede von ihnen eine Weiterentwicklung darstellt, die vor allem durch eine einfachere Syntax die Verwendung vereinfacht. Für den Gebrauch ist es jedoch trotzdem hilfreich, auch ältere Methoden zu kennen, da sie alle aufeinander aufbauen und sogar miteinander kombiniert werden können.

Durch asynchrones Laden werden Webseiten zudem schneller in ihrer Laufzeit und können interaktiver auf den Benutzer eingehen. Es besteht außerdem die Möglichkeit, den Fokus des Benutzers auf bestimmte Elemente zu lenken, indem beispielsweise die Reihenfolge der zu ladenden Inhalte bestimmt wird. Im Webbereich ist die Verwendung von Methoden, die asynchron Daten laden, nicht mehr wegzudenken.

Des Weiteren wurde ein Prototyp erstellt, bei dem der bestehende Widgetcode als Vorlage diente. Hier sind die Erkenntnisse aus der Analyse eingeflossen und umgesetzt worden. Der Prototyp lädt Empfehlungen und Werbung und positioniert diese richtig im Widget. Hier sind allerdings noch lange nicht alle Funktionalitäten und Anforderungen des aktuellen Codes implementiert, was jedoch auch nicht das Ziel für diese Arbeit war. Auch wenn die Umsetzung stark vereinfacht stattgefunden hat, ist bereits deutlich zu erkennen, wie viel lesbarer der Code durch die Verwendung von neueren Methoden wird. Während im bestehenden Code noch mit Callbacks gearbeitet wird, werden die Datenabfragen im Prototypen bereits mit async und await ausgeführt.

6.2 Ausblick

Der aktuell verwendete Widgetcode ist in seinem Umfang sehr komplex und bietet nach wie vor viel Spielraum für Verbesserungen. Die Optimierung des Codes ist ein dauerhaftes Ziel.

Außerdem soll auf lange Sicht betrachtet das Laden und Zusammensetzen der Datenquellen bereits im Backend geschehen, sodass nur noch das Widget auf der Artikelseite eingebunden werden muss, mit Inhalten, die bereits so vorbereitet wurden, dass sie keiner weiteren Bearbeitung benötigen. Diese Implementation hätte den Vorteil, dass die Items, die bei der Umsortierung von nachgeladener Werbung aus dem

Widget gekürzt werden, nicht mehr getrackt werden. Somit bekommen sie aus diesem Grund auch keine schlechte CTR mehr.

Da das ECMAScript 2016 gerade erst als Standard verabschiedet worden ist, ist dieser noch nicht mit allen Browsern kompatibel. Der Widgetcode muss allerdings viele verschiedene Browser unterstützen, weshalb es zunächst noch nicht möglich ist, in diesem die Funktionalitäten von async und await zu implementieren. Sobald jedoch mehr Browser fähig sind, mit dem ECMAScript 2016 umzugehen, steht der Nutzung von async und await nichts mehr im Wege.

A Anhang

A.1 HTTP-Statuscodes

1xx - Informationen

100 Continue
101 Switching Protocols
102 Processing

2xx - Erfolgreiche Operation

200 OK
201 Created
202 Accepted
203 Non-Authoritative Information
204 No Content
205 Reset Content
206 Partial Content
207 Multi-Status
208 Already Reported
226 IM Used

3xx - Umleitung

300 Multiple Choices
301 Moved Permanently
302 Found (Moved Temporarily
303 See Other
304 Not Modified
305 Use Proxy
307 Temporary Redirect
308 Permanend Redirect

4xx - Client-Fehler

400 Bad Request
401 Unauthorized
402 Payment Required
403 Forbidden
404 Not Found
405 Method Not Allowed
406 Not Acceptable
407 Proxy Authentication Required
408 Request Time-out
409 Conflict
410 Gone
411 Length Required
412 Precondition Failed
413 Request Entity Too Large

414 Request-URL Too Long
415 Unsupported Media Type
416 Requested range not satisfiable
417 Expectation Failed
420 Policy Not Fulfilled
421 Misdirected Request
422 Unprocessable Entity
423 Locked
424 Failed Dependency
425 Unordered Collection
426 Upgrade Required
428 Precondition Required
429 Too Many Requests
431 Request Header Fields Too Large
451 Unavailable For Legal Reasons

5xx - Server-Fehler

500 Internal Server Error
501 Not Implemented
502 Bad Gateway
503 Service Unavailable
504 Gateway Time-out
505 HTTP Version not supported
506 Variant Also Negotiates
507 Insufficient Storage
508 Loop Detected
509 Bandwidth Limit Exceeded
510 Not Extended
511 Network Authentication Required

A.2 Ein Artikel mit Widget

Abbildung A.1: Ein Artikel mit Widget

Abkürzungsverzeichnis

Ajax	Asynchronous JavaScript and XML
CSS	Cascading Style Sheets
CTR	Click-Through-Rate
DOM	Document Object Model
DRY	Don't Repeat Yourself
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ID	Identifikator
JSON	JavaScript Object Notation
JSONP	JavaScript Object Notation mit Padding
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
XML	Extensible Markup Language

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Literaturverzeichnis

Acodemy: *Learn JSON In A Day: The Ultimate Crash Course to Learning the Basics of JSON In No Time*, 1. Aufl., CreateSpace Independent Publishing Platform (2015)

Archibald, Jake: „JavaScript Promises“, http://www.html5rocks.com/en/tutorials/es6/promises/, 2014, letzter Zugriff: 31.07.2016

Basset, Lindsay: *Introduction to JavaScript Object Notation*, 1. Aufl., O'Reilly Media Inc, USA (2015)

Bevacqua, Nicólas: „Understanding JavaScrip's async await“, https://ponyfoo.com/articles/understanding-javascript-async-await, 2016, letzter Zugriff: 15.08.2016

Bitkom: „Internetzugang“, https://www.bitkom.org/Marktdaten/Konsum-Nutzungsverhalten/Digitalisierung/Internetzugang.html, 2016, letzter Zugriff: 20.08.2016

Bongers, Frank: *jQuery: Das Praxisbuch*, 3. Aufl., Galileo Computing 2013

Brain, Daniel: „Understand promises before you start using async/await“, https://medium.com/@bluepnume/learn-about-promises-before-you-start-using-async-await-eb148164a9c8#.6m7j2h7d3, 2016, letzter Zugriff: 24.07.2016

Brückmann, Manuel: „Endsstation Ladezeit: 3 Geheimtipps für ein optimales Ladeverhalten“, https://www.konversionskraft.de/tipps/endstation-ladezeit-3-geheimtipps-fuer-ein-optimales-ladeverhalten.html, 2014, letzter Zugriff: 28.07.2016

callbackhell: „Callback Hell“, http://callbackhell.com/, 2012, letzter Zugriff: 15.07.2016

Carl, Denny: *Praxiswissen Ajax*, 1. Aufl., O'Reilly Verlag GmbH & Co. KG 2006

Catuhe, David: „JavaScript Goes Asynchronous (and It's Awesome)“, https://www.sitepoint.com/javascript-goes-asynchronous-awesome/, 2015, letzter Zugriff: 23.08.2016

ECMA International: „ECMAScript® 2016 Language Specification“, http://www.ecma-international.org/ecma-262/7.0, 2016, letzter Zugriff: 05.07.2016

Flanagan, David: *JavaScript - kurz & gut*, 4. Aufl., O'Reilly Verlag GmbH & Co. KG, 2012

Fox Powell, David: „Why Can't Anyone Write a Simple es6 Generators Tutorial“, https://medium.com/@dtothefp/why-can-t-anyone-write-a-simple-es6-generators-tutorial-ec2bbdf6ff45#.t4819ep2a, 2015, letzter Zugriff: 02.08.2016

Garrett, Jesse James: „Ajax: A New Approach to Web Applications“, http://adaptivepath.org/ideas/ajax-new-approach-web-applications, 2005, letzter Zugriff: 28.06.2016

Havoc: „Callbacks, synchronous and asynchronous“, http://taligarsiel.com/Projects/howbrowserswork1.htm, 2011, letzter Zugriff: 12.07.2016

Koch, Paul-Peter: *ppk on JavaScript*, 1. Aufl., New Riders (2006)

Krager, Helmut: „JavaScript Object Notation with Pa.dding“, http://jsonp.eu/, 2012, letzter Zugriff: 10.08.2016

Kröner, Peter: „ECMAScript 5, die nächste Version von JavaScript – Teil 1: Ein Überblick“, http://www.peterkroener.de/ecmascript5-die-nachste-version-von-javascript-teil-1-ein-uberblick/, 2011, letzter Zugriff: 20.07.2016

Kröner, Peter: „ECMAScript 6: Generators, http://www.peterkroener.de/ecmascript-6-generators/, 2013, letzter Zugriff: 01.08.2016

mediaevent: „Javascript JSON: Datenaustausch zwischen Client und Server“, http://www.mediaevent.de/javascript/json.html, 2016, letzter Zugriff: 19.07.2016

Naumov, Alexander: „Javascript Promises: Asynchrones Programmieren“, https://www.alexandernaumov.de/blog/javascript-promises-asynchrones-programmieren, 2015, letzter Zugriff: 30.07.2016

Porto, Sebastian: „Asynchronous JS: Callbacks, Listeners, Control Flow Libs and Promises“, 2012, letzter Zugriff: 30.07.2016

Promises/A+: „Promises/A+“, https://promisesaplus.com/, letzter Zugriff: 17.08.2016

Reibold, Holger: „Hypertext Transfer Protocol“, http://www.tecchannel.de/a/hypertext-transfer-protocol,401210,6, 2001, letzter Zugriff: 29.06.2016

Roden, Golo: „Einführung in die asynchrone JavaScript Programmierung“, http://www.heise.de/developer/artikel/Einfuehrung-in-die-asynchrone-JavaScript-Programmierung-2752531.html, 2015, letzter Zugriff: 12.07.2016

Schäfer, Mathias: „Eine kurze Geschichte von JavaScript“, http://webkrauts.de/artikel/2015/eine-kurze-geschichte-von-javascript, 2015, letzter Zugriff: 30.06.2016

SELFHTML-Wiki: „HTTP/Statuscodes“, https://wiki.selfhtml.org/wiki/HTTP/Statuscodes, 2016, letzter Zugriff: 22.08.2016

SELFHTML-Wiki: „JavaScript/JSON“, https://wiki.selfhtml.org/wiki/JavaScript/JSON, 2016, letzter Zugriff: 13.07.2016

Simpson, Kyle: „Going Async With ES6 Generators“, https://davidwalsh.name/async-generators, 2014, letzter Zugriff: 06.08.2016

Stephan: „JavaScript: Callback-Funktionen erstellen & nutzen“, http://www.smart-webentwicklung.de/2013/05/javascript-callback-funktionen-erstellen-nutzen/, 2013, letzter Zugriff: 15.07.2016

Strang, Martin: „Kritik an Web-Technologien“, http://www.macwelt.de/news/Kritik-an-Web-Technologien-3023625.html, 2007, letzter Zugriff: 12.07.2016

Vdovkin, Andreas: *jQuery - kurz & gut*, 2. Aufl., O'Reilly Verlag GmbH & Co. KG (2011)

W3C: „A Short History of JavaScript“, https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript, 2012, letzter Zugriff: 30.06.2016

Walsh, David: „JavaScript Promise API“, https://davidwalsh.name/promises, 2015, letzter Zugriff: 30.07.2016

Wenz, Christian: *AJAX schnell + kompakt*, 2. Aufl., Entwickler.Press 2007

Wenz, Christian: *JavaScript und AJAX: Dass umfassende Handbuch*, 7. Aufl., Galileo Computing 2006

Zakas, Nicholas C.: *Professional JavaScript for Web Developers*, 3. Aufl., John Wiley & Sons 2012

Zimmermann, Joe: „Simplify Asynchronous Coding with ES7 Async Functions“, https://www.sitepoint.com/simplifying-asynchronous-coding-es7-async-functions/, 2015, letzter Zugriff: 02.08.2016

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

Ort, Datum Kristin Groschoff