Asnychrone Programmierung: Moderne Methoden in ECMAScript 6

Seminar 1908 an der FernUni Hagen im WS 2016/17:
Moderne Programmiertechniken und -Methoden

Felix Eckstein*

Dezember 2016

This paper gives an overview on modern methods of asynchronous programming in the 2015 JavaScript Standard "ECMAScript 6".

Firstly, an overview of the JavaScript runtime environment is given to motivate the excessive use of asynchronous programming in JavaScript applications. The principle of asynchronous programming is briefly introduced. Following this clarification of terms, the traditional callback approach to asynchronous programming is presented together with its inherent downsides.

In the main part of the paper, the modern ECMAScript 6 features "promises" and "generators" together with their application to asynchronous programming are introduced and discussed in detail. A special focus is laid on the way they solve problems that were present with the traditional callback approach.

The paper concludes with a discussion whether the new language features are worth using and under what circumstances it might be a good option to waive the new features in favor of compatibility to older runtime environments.

The appendix contains examples of common asynchronous patterns in ECMAScript 6.

* *Student im Bachelor of Science Informatik an der FernUniversität Hagen, Matr.-#: 8161569, felix.eckstein@gmx.de*

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

LISTINGS

1 JAVASCRIPT – DIE HÄUFIGSTE PROGRAMMIERSPRACHE DER WELT

Seit der Erfindung der Sprache „Mocha“ durch Brendan Eich im Mai 1995 und dem ersten Auftreten dieser inzwischen in JavaScript umbenannten Sprache im Netscape Navigator 2.0 im März 1996 (vgl. [Netscape, 1995]), hat diese Sprache eine bemerkenswerte Metamorphose erlebt. Zunächst belächelt und relativ nutzlos (vgl. [Buckler, 2013]) wurde die Sprache stetig weiter entwickelt und schließlich Ende 1996 an die Standardisierungsorganisation ECMA übergeben (vgl. [Netscape, 1996]) und im Jahr 1997 im ECMA-262-Standard bzw. ISO/IEC 16262 genormt und offiziell in ECMAScript umbenannt (vgl. [Ecma TC39 und Steele, 1997]).[1] Inzwischen ist Javascript als eine „echte“ Programmiersprache anerkannt und findet immer stärkere Anwendung. Auch der Einsatzbereich hat sich weit über Scripting im Browser hinaus erweitert, so dass heutzutage insbesondere auch Serverapplikationen häufig in JavaScript realisiert werden.

> The World's Most Misunderstood Programming Language Has Become the World's Most Popular Programming Language [Crockford, 2008]

Laut einer Untersuchung der Entwickler-Website „Stackoverflow“ ist JavaScript inzwischen die am häufigsten nachgefragte Sprache (vgl.[Taft, 2016]). Auch das stabile Rankig innerhalb der Top Ten der beliebtesten Programmiersprachen im TIOBE Index und die „Ehrung“ als „Programming Language of the Year 2013“ belegen die hohe Popularität der Sprache (vgl. [TIOBE, 2016]).

Ein Grund für die Popularität der Sprache dürfte darin liegen, dass sich JavaScript inzwischen als lingua franca des WWW durchgesetzt hat und von jedem aktuellen Browser als clientseitiger Script-Sprache für Webapplikationen unterstützt wird. Da alle aktuellen Browser auch Entwicklerwerkzeuge zum direkten Erstellen, Ausführen und Debuggen von JavaScript Code enthalten, dürfte JavaScript die einzige Programmiersprache sein, für die annähernd jeder Computerbenutzer – von der Bundeskanzlerin bis zum Computernerd – eine vollständige Entwicklungsumgebung inklusive Editor, Compiler und Debugger besitzt.

In JavaScript-Programmen ist asynchrone Programmierung allgegenwärtig. Sie weicht erheblich von dem aus anderen Sprachen gewohnten Programmierpardigma des sequentiellen Programmablaufs mit blockierenden Funktionsaufrufen ab.

In dieser Arbeit soll daher folgender Frage nachgegangen werden: Was ist asynchrone Programmierung in JavaScript und welche mo-

1 Trotz dieser Umbenennung ist der frühere, noch von Netscape und Sun Microsystems eingeführte Name ein weithin gebrauchtes und übliches Synonym und ein Oberbegriff für die Sprachen, die dieser Spezifikation folgen. Auch in der vorliegenden Arbeit sollen die Begriffe „ECMAScript", „ES“, „JavaScript“ und „JS“ synonym gebraucht werden. Lediglich wenn es um Features geht, denen speziell der im Jahre 2015 erschienene Standard „ECMA-262 6th Edition, The ECMAScript 2015 Language Specification“ zugrunde liegt, wird explizit der Ausdruck „ES6“ oder „ECMAScript 6“ benutzt.

dernen Methoden zur asynchronen Programmierung bietet der neue Sprachstandard ECMAScript 6?

Es wird zunächst ein kurzer Abriss über die Funktionsweise der Laufzeitumgebung von JavaScript gegeben. Die sich daraus ergebende Notwendigkeit zur asynchronen Programmierung wird erläutert und der Begriff „asynchrone Programmierung“ skizziert.

Nachfolgend wird erläutert, wie asynchrone Programmierung unter Verwendung von traditionellen Callbacks in Standard-JavaScript vor der Einführung von ECMAScript 6 realisiert wurde. Dabei wird aufgezeigt, welche Nachteile die ausschließliche Verwendung solcher Callbacks für die asynchrone Programmierung mit sich bringt.

Die in ECMAScript 6 neu eingeführten Sprachmittel *Promise* und *Generator* werden detailliert eingeführt und ihre Arbeitsweise erläutert. Dabei werden die Unterschiede dieser neuen Sprachmittel zum traditionellen Ansatz herausgestellt.

Abschließend werden die in ECMAScript 6 neu eingeführten Methoden zur asynchronen Programmierung bewertet um daraus Konsequenzen für deren Einsatz in der Praxis abzuleiten.

In einem Anhang werden Beispiele angegeben, wie typische Muster der asynchronen Programmierung unter Verwendung der ECMAScript 6 Methoden konkret formuliert werden.

2 WICHTIGE EIGENSCHAFTEN VON JAVASCRIPT

Zunächst sollen zwei Grundeigenschaften von JavaScript geklärt werden, in denen sie sich deutlich von anderen Sprachen wie z. B. Java und Pascal unterscheidet.[2]

2.1 Funktionale Eigenschaften

Eine wichtige Eigenschaft der Sprache sind deren funktionalen Elemente und darunter besonders die Eigenschaft, dass Funktionen als ganz normale Objekte angesehen werden.[3]

Funktionen sind damit sogenannte „First-Class-Citizens“ der Sprache und können daher an Variablen zugewiesen werden, an andere Funktionen als Parameter übergeben werden und auch als Rückgabewerte von Funktionen dienen. Diese Eigenschaften sind notwendig um das Konzept der„Higher-Order-Functions“ der funktionalen Programmierung umzusetzen.[4] Darauf basiert der in JavaScript weit verbreitete „Callback-Mechanismus“: Eine Funktion bekommt neben den Daten,

2 Das sind die Sprachen, die standardmäßig an der Fernuni Hagen im Informatikstudium gelehrt werden [FernUni Hagen, 2016].

3 Die ECMAScript 6 Spezifikation hierzu:

> ECMAScript function objects encapsulate parameterized ECMAScript code closed over a lexical environment and support the dynamic evaluation of that code. An ECMAScript function object is an ordinary object and has the same internal slots and the same internal methods as other ordinary objects. [Ecma TC39 und Wirfs-Brock, 2015, §9.2]

4 vgl. [Ganzinger, Six, Voss und Beierle, 2011, S. 148 ff.]

die sie zur Ausführung benötigt, zusätzlich eine Funktion übergeben, die sie ihrerseits aufrufen kann, um bestimmte weitere Aktionen durchzuführen.

2.2 Laufzeitumgebung

JavaScript-Programme werden nicht direkt in ausführbaren Code übersetzt und auf der Zielhardware ausgeführt, sondern als Quelltext ausgeliefert und erst zur Laufzeit interpretiert bzw. „just-in-time“ kompiliert. Zudem verzichtet die ECMAScript Spezifikation vollständig auf die Definition von Ein- und Ausgabefunktionen. Aus beiden Eigenschaften ergibt sich die Notwendigkeit einer Laufzeitumgebung – eines „Host Environments“ – zur Ausführung von JavaScript Code auf der Zielplattform.[5]

Die mit Abstand am häufigsten eingesetzten Laufzeitumgebungen für JavaScript sind die Implementierungen in modernen Webbrowsern wie Firefox, Chrome, Safari oder auch Edge. Jeder moderne Webbrowser kann Javascript Code, der in Webseiten eingebettet ist, ausführen, wodurch komplexe Webanwendungen erst praktikabel werden.

Neben dem Einsatz von JavaScript im Browser hat sich inzwischen auch der Einsatz auf dem Server etabliert. Möglich wurde dies vor allem durch die Verfügbarkeit von „Node.js“, das als serverseitige Laufzeitumgebung für JavaScript erstmals 2009 auf der JavaScriptConf.eu von ihrem Erfinder Ryan Dahl der Öffentlichkeit vorgestellt wurde.[6] Seit dem Erscheinen von Node.js wurden viele große und umsatzstarke Webpräsenzen auf diese Plattform migriert, was für die Stabilität und Performanz der Technologie spricht.

Obwohl die Spezifikation keine spezielle Implementierung der Laufzeitumgebung vorschreibt, so sind doch einige Elemente verpflichtend oder haben sich als Standard durchgesetzt. Diese sollen hier kurz beschrieben werden, um die spezielle Semantik von JavaScript-Programmen zu verdeutlichen.

STACK UND EXECUTION CONTEXT Der ECMAScript Standard schreibt vor, dass mit der Ausführung einer jeden Funktion ein „Execution Context“ auf dem Stack angelegt wird. In diesem Kontext sind alle notwendigen Statusinformationen der Funktion verfügbar, die zur Ausführung des Codes notwendig sind. Mit jedem Funktionsaufruf wird

5 Die Spezifikation dazu:

> ECMAScript is an object-oriented programming language for performing computations and manipulating computational objects within a host environment. ECMAScript as defined here is not intended to be computationally self-sufficient; indeed, there are no provisions in this specification for input of external data or output of computed results. Instead, it is expected that the computational environment of an ECMAScript program will provide [...] certain environment-specific objects [...] and certain functions that can be called from an ECMAScript program. [Ecma TC39 und Wirfs-Brock, 2015, § 4]

6 Die Originalpräsentation ist auf Youtube zu sehen: https://www.youtube.com/watch?v=ztspvPYybIY

ein neuer Kontext auf den Stack gepusht und mit jedem return wird der oberste Eintrag entfernt.[7]

QUEUE In der Queue der Laufzeitumgebung werden die noch anstehenden Aufgaben verwaltet, die in der Zukunft ausgeführt werden müssen. Sobald die Abarbeitung des aktuellen Codeabschnitts beendet ist – also der Stack mit den Execution Contexts geleert wurde – entnimmt die Laufzeitumgebung den nächsten Eintrag aus der Warteschlange und initiiert die Ausführung des damit assoziierten Codes.

Darin werden Nachrichten und Events verwaltet, die entweder von laufenden Javascript-Programmen oder aber von der Laufzeitumgebung dort eingereiht werden können. Das können z. B. Benutzerinteraktionen wie der Click auf ein Webseitenelement sein, die mit einer JavaScript-Funktion zur Ausführung assoziiert sind. Von laufenden Funktionen können aber auch Nachrichten eingereiht werden, die Referenzen auf andere Funktionen (Callback-Funktionen) enthalten, deren Ausführung zu einem späteren Zeitpunkt eingeplant werden soll.

EVENT-LOOP Um die noch anstehenden Aufgaben aus der Queue abzuarbeiten, läuft in der Laufzeitumgebung eine Event-Loop ab. Sobald der Stack nach Ausführung der zuletzt laufenden Funktion wieder geleert ist, „tickt“ die Event-Loop und entnimmt der Queue das nächste Element und bringt die damit verbundene Funktion zur Ausführung. Sobald diese vollständig abgearbeitet wurde findet der nächste Tick statt.

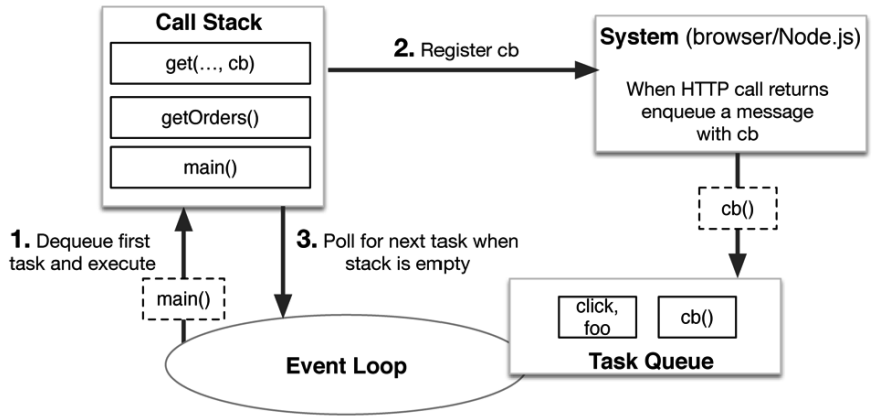

Abbildung 1: Das JavaScript Event Loop Modell (aus [Gallaba u. a., 2015, S. 2])

7 Die Spezifikation dazu:

> An execution context is a specification device that is used to track the runtime evaluation of code by an ECMAScript implementation. At any point in time, there is at most one execution context that is actually executing code. This is known as the running execution context. A stack is used to track execution contexts. The running execution context is always the top element of this stack. A new execution context is created whenever control is transferred from the executable code associated with the currently running execution context to executable code that is not associated with that execution context. The newly created execution context is pushed onto the stack and becomes the running execution context. [Ecma TC39 und Wirfs-Brock, 2015, § 8.3]

Der Ablauf der Nachrichtenverarbeitung und des Einreihens neuer Nachrichten durch die Laufzeitumgebung ist in Abbildung 1 dargestellt.

Der Programmlauf startet mit der ersten Funktion der JavaScript Applikation (Schritt 1). Während der Beispielausführung wird ein HTTP-Request über eine Bibliotheksfunktion der Laufzeitumgebung abgesetzt. Zusammen mit dieser Abfrage wird eine Callback-Funktion registriert (Schritt 2). Sobald eine Antwort auf die Anfrage eintrifft, wird eine entsprechende Nachricht in die Queue eingereiht, die mit der Ausführung der Callback-Funktion assoziiert ist. Wenn die Ausführung des aktuellen Codeabschnitts auf dem Stack beendet ist, tickt die Event-Loop und entnimmt die nächste Nachricht aus der Queue (Schritt 3). Nach einer gewissen Anzahl Ticks wird so auch die vorher eingereihte Nachricht verarbeitet und die assoziierte Callback-Funktion wird ausgeführt.

2.3 Single Thread Modell und Run-to-Completion Semantik

Eine wichtige Eigenschaft von ECMAScript ist in diesem Ablauf die durch den Standard garantierte Beschränkung der Programmausführung auf einen einzigen Thread.[8] Dadurch ist garantiert, dass eine einmal zur Ausführung gekommene Funktion nicht durch andere Funktionen unterbrochen werden kann. Jeder Eintrag in der Queue wird vollständig abgearbeitet, bevor die nächste Nachricht entnommen und verarbeitet werden kann.

Dieses Verhalten ist unter dem Namen „Run-To-Completion“-Semantik bekannt und unterscheidet JavaScript von multi-threaded Programmiersprachen wie z. B. Java, in denen die Programmiererin damit rechnen muss, dass ihr laufender Code zu jeder Zeit vom Code eines anderen Threads innerhalb ihrer Applikation unterbrochen werden kann.

KONSEQUENZEN UND VORTEILE Dass die Codeausführung nicht unterbrochen werden kann, erinnert an alte Zeiten von kooperativen Multitasking Systemen. Dort konnte ein einzelner lang laufender Prozess das System unbedienbar machen oder sogar zum Absturz bringen. Die Situation in JavaScript ist ähnlich: Eine einzelne lang laufende Operation kann die Ausführung anderer, eigentlich nebenläufiger, Prozesse verzögern oder sogar verhindern. Im Browser wird dann die Meldung „Script takes too long to run“ angezeigt (vgl. [Mozilla Developer Network, 03.11.2016]).

Für die Javascript Programmiererin ist es unerlässlich durch geeignete Maßnahmen wie asynchrone Programmierung dafür zu sorgen, dass eine Funktionsausführung den Stack nicht zu lange blockiert. Dies wird nicht durch die Laufzeitumgebung sichergestellt, sondern liegt einzig in

8 Die ECMAScript 6 Spezifikation dazu:

> At any point in time, there is at most one execution context that is actually executing code. This is known as the running execution context. [Ecma TC39 und Wirfs-Brock, 2015, § 8.3]

und

> Execution of a Job can be initiated only when there is no running execution context and the execution context stack is empty. [Ecma TC39 und Wirfs-Brock, 2015, § 8.4]

der Hand der Programmiererin. Sie ist dafür verantwortlich, dass die Event-Loop nicht blockiert wird.

In anderen Programmiersprachen ist es möglich und üblich für lang laufende Operationen einen eigenen Thread abzuspalten, der dann parallel zur weiteren Programmausführung abläuft. Das Umschalten zwischen den Threads einer Anwendung wird durch das Laufzeitsystem bzw. das Betriebssystem übernommen und erfolgt präemptiv. Dadurch bekommen die Threads einen fairen Anteil an Rechenzeit oder können ganz suspendiert werden, wenn sie z. B. auf I/O-Operationen warten.

Das Single-Thread Ausführungsmodell mit seiner Run-to-Completion Semantik scheint auf den ersten Blick nebenläufige Code-Ausführung zu verhindern und ein klarer Nachteil gegenüber multi-threaded Systemen zu sein. Nicht umsonst haben sich auf dem Desktop präemptive Multitasking-Betriebssysteme durchgesetzt, die auch dann noch bedienbar bleiben, wenn ein einzelner Prozess nicht mehr reagiert.

Der Vorteil des Multi-Threading wird aber durch zwei neue Probleme erkauft:

Die Verwaltung und das Scheduling von Threads kosten erhebliche Resourcen. Dies wird von Ryan Dahl bei der Vorstellung des Node.js Konzepts anhand des multi-threaded Webservers Apache im Vergleich zum Event-Loop getriebenen single-threaded Server NGINX eindrücklich präsentiert (vgl. [Dahl, 2009] ab ∼3:00 und den Blogpost [Poli, 2013]).

Das andere Problem in einer multi-thread Umgebung ist die Synchronisation. Jeder Programmteil muss jederzeit damit rechnen, dass die Ausführung unterbrochen wird und zwischenzeitlich anderer Code abläuft. Die Verantwortung dafür, dass die Daten und Objekte der Applikation trotzdem in einem konsistenten Zustand bleiben, liegt bei der Programmiererin. Oft muss erheblicher Aufwand zur Synchronisation getrieben werden. Solche Synchronisationsprobleme können in einer reinen single-thread Umgebung wie JavaScript gar nicht erst auftreten. Dave Herman schreibt dazu sehr treffend:

> Part of the beauty of JavaScript's event loop is that there's a very clear synchronization point for reaching a stable state in your programs: the end of the current turn. You can go ahead and leave things in a funky intermediate state for as long as you like, and as long as you stitch everything back up in time for the next spin of the event loop, no other code can run in the meantime. That means you can be sure that while your object is lying in pieces on the floor, nobody else can poke at it before you put it back together again. [Herman, 2011]

3 ASYNCHRONE PROGRAMMIERUNG

Wie können trotz der Beschränkung auf einen einzigen Thread performante Programme geschrieben werden, die die Event-Loop nicht blockieren und auch bei länger laufenden Operationen reaktiv und bedienbar bleiben?

Die allermeisten Anwendungen sind selber nicht sehr rechenintensiv, benutzen jedoch externe Funktionen, die vergleichsweise lange brauchen, um ein Ergebnis zu liefern. Typischerweise sind dies alle Arten

von I/O-Operationen, Netzwerkzugriffen und Datenbankabfragen. In Tabelle 1 sind Latenzzeiten von beispielhaften Operationen angegeben. Diese werden zwar häufig durch eigenen Code ausgelöst bzw. aufgerufen, sie laufen jedoch nicht innerhalb der eigenen Applikation ab. Damit solche Aufrufe nicht die gesamte Anwendung blockieren, muss eine konkurrenzfähige Programmiersprache eine Form der Nebenläufigkeit (Concurrency) unterstützen.

Operation	**Latency**
L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

Tabelle 1: Vergleich unterschiedlicher Latenzen typischer Operationen. (Daten aus [Dean, 2007, S. 13])

Die JavaScript Event-Loop unterstützt genau solch eine Nebenläufigkeit, wenn es zwar lang dauert bis ein Ergebnis vorliegt, diese Latenz aber nicht durch große Rechenlast verursacht wird, sondern durch das Warten auf externe Operationen, die außerhalb der eigenen Applikation ablaufen.

3.1 Non-Blocking Function Calls

In traditionellen Programmiersprachen ist man es gewohnt, dass ein Funktionsaufruf mittels eines `return` ein Ergebnis zurückgibt. Diese Ergebnisrückgabe an die aufrufende Funktion erfolgt, wenn das Ergebnis auch wirklich vorliegt. Der aufrufende Code ist solange blockiert bis er die Ablaufkontrolle zusammen mit dem Ergebnis explizit per `return` zurück bekommt.

Eine solche Art der Programmausführung führt in einer single-threaded Sprache zu sehr ineffizienten Programmen, da während der Wartezeit keine anderen Aktionen ausgeführt werden können.

Um diese Situation aufzulösen, muss es möglich sein die Rückgabe der Ablaufkontrolle mittels `return` und die Bereitstellung des Ergebnisses einer Funktion voneinander zu entkoppeln. Das Anstoßen einer externen Operation darf den Programmablauf nicht blockieren.

Dieses Verhalten zeigen nicht-blockierende Funktionsaufrufe (non-blocking function calls). Sie stoßen eine externe Aktion lediglich an und kehren sofort zurück, bevor das angeforderte Ergebnis vorliegt. Das Programm kann währenddessen mit anderen Aufgaben fortfahren und

wird informiert, wenn das Ergebnis verfügbar ist. Die angestoßene Aktion läuft dabei in der Regel außerhalb des eigenen Single-Threads ab.[9]

3.2 *Jetzt* und *Später*

Die Applikation kann eine Aktion *jetzt* anstoßen, wird dadurch aber nicht blockiert, sondern kann direkt weiterlaufen. Das Ergebnis liegt erst zu einem noch unbekannten Zeitpunkt in der Zukunft vor und wird *später* verarbeitet.

Ein solches Programmiermodell wird als „asynchrone Programmierung“ bezeichnet. Deren Kern besteht in der Entkopplung zwischen der *jetzt* stattfindenden Auslösung einer Aktion und dem *späteren* Vorliegen und Verarbeiten eines Ergebnisses.

> In fact, the relationship between the *now* and *later* parts of your program is at the heart of asynchronous programming. [Simpson, 2015]

Im Gegensatz zur synchronen Programmierung kann man sich in der asynchronen Programmierung den Programmablauf nicht mehr als linearen Zeitstrahl vorstellen, bei dem jede Ursache (ein Funktionsaufruf) direkt eine Wirkung (die Rückgabe des Ergebnisses) nach sich zieht. Vielmehr wird die Programmabarbeitung in kleinere Einheiten unterteilt, welche zwar kausal voneinander abhängen, deren zeitlicher Ablauf jedoch voneinander entkoppelt ist. Ein Programm kann daher immer weiter laufen, ohne dass es blockierend auf Ergebnisse einer von ihm angestoßenen Aktion warten muss.

Die *spätere* Verarbeitung des Ergebnisses ist garantiert. Es ist jedoch unbekannt, wann genau das Ergebnis tatsächlich verarbeitet wird.[10]

Durch die konsequente Nutzung asynchroner Programmierung mit non-blocking function calls ist es möglich Programme zu schreiben, die durch den Verzicht auf internes Multi-Threading sehr performant und mit hohem Durchsatz ausgeführt werden können. Während externe Ergebnisse bereitgestellt werden kann die Applikation weiterlaufen (vgl. [Dahl, 2009]).

3.3 Asynchrone Programmierung in JavaScript

Die Entkopplung zwischen *jetzt* und *später* wird in JavaScript über die Event-Loop des Laufzeitsystems erreicht. *Jetzt* bezeichnet den aktuellen Tick der Event-Loop, bis der Stack wieder geleert ist. Alle Codeabschnitte, die mit einem weiteren Ereignis der Queue assoziiert sind, werden *später* ausgeführt.

---

9 vgl. dazu:

> [...]the JavaScript code you write all runs on a single thread, but the code that implements the async task ([...]) is not part of that JavaScript and is free to run in a separate thread. [Parker, 2015, S. 7]

10 vgl. dazu:

> After the callback is added to the queue, there is no guarantee how long it will have to wait. How long it takes the current code to run to completion and what else is in the queue controls the time. The queue can contain things such as mouse clicks, keystrokes, and callbacks for other async tasks. [Parker, 2015, S. 7]

Der Code in jedem Tick der Event-Loop läuft dabei wie gewohnt sequentiell ab. Die Ergebnisse von non-blocking function calls werden als Ereignisse in die Queue einsortiert und damit *später* verarbeitet.

In der JavaScript Praxis ist das Konzept der asynchronen Programmierung so erfolgreich, dass z. B. Node.js (fast) vollständig auf non-blocking function calls setzt[11] und auch im Bereich der Webapplikationen im Browser die Zahl der asynchron arbeitenden Application Programming Interfaces (API) rapide zunimmt.[12]

4 CALLBACKS IN TRADITIONELLEM JAVASCRIPT

Um die eben erläuterte asynchrone Programmierung in JavaScript zu realisieren, ist ein Mechanismus notwendig, die *später* vorhandenen Ergebnisse in den Programmablauf einzuspeisen und weiter zu verarbeiten. In traditionellem JavaScript wird dazu der aufgerufenen asynchronen Funktion eine sogenannte „Callback-Funktion“ als Parameterargument übergeben. Diese kann ausgeführt werden, sobald das Ergebnis des asynchronen Aufrufs vorliegt, und dieses verarbeiten.

> The callback function is the async work horse for JavaScript, and it does its job respectably. [Simpson, 2015, S. 26]

4.1 Funktionsweise von Callbacks

Die Callback-Funktion wird beim asynchronen Aufruf übergeben und soll *später* das Ergebnis verarbeiten. Das eigene Programm läuft nach der sofortigen Rückkehr aus dem nicht-blockierenden Funktionsaufruf *jetzt* weiter. Die asynchrone Funktion dagegen wird außerhalb des aktuellen Execution-Context des Hauptprogramms ausgeführt und ermittelt letztendlich ein Ergebnis (Erfolg oder Fehler). Um dieses Ergebnis an das Hauptprogramm zurückzuliefern, wird der Aufruf der Callback-Funktion als Nachricht in die Queue der Laufzeitumgebung eingestellt. Das Ergebnis selber wird der Callback-Funktion als Aufrufparameter übergeben. Die Event-Loop bringt die übergebene Callback-Funktion zu einem *späteren* Zeitpunkt zur Ausführung, so dass sie das Ergebnis im Rahmen des Hauptprogramms verarbeiten kann.[13]

---

11 Ryan Dahl dazu in einem Interview:

> Everything is a callback. So where you would traditionally say: „access the database, write to file, move file over there and do something else“ you kind of do these sequential sort of actions one after another. In Node you can't do those sort of things because you might take some amount of time for you to move a file from one place to another because the disk might have to spin, or if you query your database that might take some milliseconds for you to respond and in Node everything is non-blocking and so it doesn't allow you to just sit there and then return the response. [Synodinos, 2010]

12 vgl. dazu:

> The number of asynchronous JavaScript APIs is rapidly growing. [Parker, 2015, S. 1]

13 Eine interessante Sichtweise dieses Prozederes besteht darin die übergebene Callback-Funktion als „Continuation“ aufzufassen: Die aufrufende Funktion übergibt der aufgerufenen den „nächsten Schritt“ des Hauptprogramms (eine Continuation) zur Verarbeitung des Ergebnisses. Diese Sichtweise und Art der Programmierung

Ein Beispiel für einen asynchronen Aufruf mit Ergebnisübergabe per Callback ist in Listing 1 zu sehen. Der vierte Parameter für asyncCall ist die Callback-Funktion, die an Ort und Stelle als anonyme Funktion inline definiert wird.

```
//Example for trailing anonymous callback with error-first
    protocol

asyncCall("http://someurl.com", secondPar, thirdPar, function
    (err, result) {
    // error?
    if (err) {
        console.error(err);
    }
    // otherwise, assume success
    else {
        console.log(result);
    }
});
```

Listing 1: Beispiel für einen asynchronen Aufruf mit Callback.

KONVENTIONEN Prinzipiell sind als Schnittstelle zwischen eigenem Programm, asynchronen Aufrufen und Callbacks beliebige Funktionen mit beliebigen Signaturen zulässig. In der Javascript Praxis haben sich aber Konventionen herausgebildet, die eine gewisse Durchgängigkeit bringen und den Wildwuchs von Schnittstellen eindämmen. Am Beispiel in Listing 1 lassen sich die Konventionen „trailing callback“ und „error first“ erläutern.

TRAILING CALLBACK In den meisten Funktionen, die einen Callback akzeptieren, wird dieser als letzter Parameter in der Funktions-Signatur angegeben. Diese Konvention wird als „trailing callback“ bezeichnet und ist insbesondere bei Modulen für die Node.js Laufzeitumgebung häufig anzutreffen.

Diese letzte Parameter-Position für den Callback verbessert die Lesbarkeit erheblich, wenn als Callback-Funktion anonyme Funktionen verwendet werden, die sich über mehrere Zeilen erstrecken.

Auch existieren mittlerweile viele APIs, bei denen die Angabe einer Callback-Funktion optional ist. In diesem Fall ist es notwendig, dass die aufgerufene Funktion überprüfen kann, ob ihr eine Funktion als Callback-Parameter übergeben wurde oder nicht. Dies wird wesentlich erleichtert, wenn der optionale Callback-Parameter an einer ausgezeichneten (der letzten) Position in der Parameterliste steht.

ERROR FIRST In sehr vielen APIs hat sich das sogenannte „error-first“ Protokoll für Callbacks durchgesetzt. Diese Konvention besagt, dass eine Callback-Funktion die festgelegte Signatur cb(err, result) besitzt. In der Callback-Funktion selber wird geprüft, ob der err-Parameter angegeben wurde. Wenn dieser vorhanden ist, dann wird eine Fehlerbehandlung durchgeführt. Wenn dieser erste err-Parameter nicht gesetzt

wird auch als "Continuation Passing Sytle"(CPS) bezeichnet. (vgl. [Rauschmayer, 2012])

ist, dann wird das im folgenden result-Parameter übergebene Ergebnis verwendet und verarbeitet.[14]

4.2 Problematik traditioneller Callbacks

Die Übergabe einer Callback-Funktion an die aufgerufene asynchrone Funktion, welche diese dann ihrerseits mit ihrem Ergebnis aufruft, stellt die traditionelle Methode zur asynchronen Programmierung in JavaScript dar. In den JavaScript Versionen vor ECMAScript 6 gab es keine andere Methode asynchrone Programmierung zu realisieren.

Doch ist dieses Vorgehen nicht frei von Problemen, welche im Folgenden aufgezeigt werden sollen.

4.2.1 *„Pyramid of Doom“ und schlechte Nachvollziehbarkeit*

Häufig sind mehrere asynchrone Funktionen voneinander abhängig. Im Beispiel Listing 2 wird zunächst eine Datenbankverbindung aufgebaut. Sobald diese Verbindung aufgebaut ist, wird eine Anfrage an die Datenbank abgesendet. Schließlich wird das angefragte Datum an den zu Beginn übergebenen Callback übergeben und dort verarbeitet.

```
//Example of nested Callbacks (own example)
//Example is over simplified regarding proper error handling
    and closing the database connection

//definition of retrieval function
function retrieveData(dbName, dbRow, dbQuery, finalCallback) {
    openDBConnection(dbName, function (err, connection) {
        if (err) return finalErrorHandler(err);
        connection.retrieveData(dbRow, dbQuery, function (err,
            data) {
            if (err) return finalErrorHandler(err);
            return finalCallback(null, data);
        })
    })
}

//invocation of retireval function
retrieveData(fooDB, "name", "name='Mueller'", function (err,
    data) {
    if (err) return console.error(err);
    console.log("finally retrieved: " + data);
})
```

Listing 2: Beispiel zu „nested Callbacks“ bei voneinander abhängigen asynchronen Funktionsaufrufen

Abhängige asynchrone Aufrufe führen zu verschachtelten Callbacks, deren Code durch die immer weiter voran schreitende Einrückung schnell schwer lesbar und unübersichtlich wird. Diese immer weiter voran schreitende Einrückung im Quelltext ist unter dem treffenden Namen „Pyramid of Doom“ hinlänglich bekannt. (vgl. [Kennedy Kambona, Elisa Gonzalez Boix und Wolfgang De Meuter, 2013, § 1])

14 Die Firma Joyent, welche die initiale Entwicklung von Node.js gesponsort hat, dazu in ihren Best Practices:

> The usual pattern is that the callback is invoked as callback(err, result), where only one of err and result is non-null, depending on whether the operation succeeded or failed. [Joyent]

Bei solchen verschachtelten Callbacks ist jedoch nicht in erster Linie die Formatierung des Quelltexts problematisch. Vielmehr ist es für die Programmiererin schwierig nachzuvollziehen, welche Ausführungspfade tatsächlich eingeschlagen werden und welche zeitliche Abfolge sich ergibt.[15] Insbesondere in Zusammenhang mit der Fehlerbehandlung ergeben sich in der Praxis sehr schnell äußerst unübersichtliche Strukturen.[16]

4.2.2 *Komplizierte Fehlerbehandlung*

In synchronem JavaScript Code wird das Error-Handling über einen try-catch-finally-Mechanismus behandelt. Wenn in einer Funktion ein nicht behebbarer Fehler auftritt, so wird mittels throw die Funktion vorzeitig mit einem Fehlerobjekt verlassen. In einer weiter außen liegenden Programmschicht – also in einem auf dem Stack weiter unten liegenden Execution-Context – kann dieser Fehler mittels einer catch-Klausel abgefangen und behandelt werden. Die Installation eines generischen Error-Handlers, der in einem bestimmten Applikations-Kontext alle bis dahin unbehandelten Fehler fängt und behandelt, ist im synchronen Fall über den try-catch-finally-Mechanismus relativ leicht möglich.

Im Fall der asynchronen Programmierung versagt dieser Sprachmechanismus. Der asynchrone Code läuft in einem anderen Ausführungskontext ab als die aufrufende Funktion. Es ist keine weiter außen liegende Programmschicht vorhanden, welche den Fehler im Kontext des Funktionsaufrufs der asynchronen Funktion behandeln könnte. Ein mit throw geworfener Fehler müsste von einem Error-Handler verarbeitet werden, der keine Kenntnis davon hat, aus welchem Kontext heraus der vorliegende Fehler entstanden ist.[17]

Um einen extern zur Applikation auftretenden Fehler im Applikationskontext behandeln zu können, muss der Fehler genau wie ein Ergebnis wieder zurück in einen anderen Kontext übergeben werden. Dazu dient z. B. das erwähnte Error-First-Protokoll (vgl. Listing 1). Die Fehlerbehandlung muss aber für jeden einzelnen asynchronen Aufruf explizit ausprogrammiert werden, wobei viele verschiedene Ausführungspfa-

15 Das menschliche Gehirn ist auf die Verarbeitung von linearen Prozessen ausgelegt, die nacheinander ablaufen. (vgl. „Sequential Brain“ in [Simpson, 2015, 27ff.]) Durch die Benutzung von asynchronem Code mit Callbacks wird dieser normale Denkablauf stark gestört und es fällt Menschen schwer, die genauen Abläufe mit all ihren zeitlichen und datengetriebenen Abhängigkeiten nachzuvollziehen und zu planen. Siehe dazu [Kennedy Kambona u. a., 2013]:

> Since callbacks are common in JavaScript, programmers end up losing the ability to think in the familiar sequential algorithms and end up dealing with an unfamiliar program structure.

16 In realem Code kommen überwiegend zwei bis drei Schachtelungsebenen vor. Jedoch gibt es Beispiele von bis zu 8 Eben tiefen Callback-Schachtelungen. (vgl. dazu [Gallaba u. a., 2015, § IV.D])

17 vgl. dazu:

> In synchronous JavaScript code the throw keyword can be used to signal an error and try/catch can be used to handle the error. When there is asynchrony, however, it may not be possible to handle an error in the context it is thrown. Instead, the error must be propagated asynchronously to an error handler in a different context. [Gallaba u. a., 2015, S. 2]

de berücksichtigt werden müssen. Dazu muss eine große Menge „Boilerplate-Code“ geschrieben werden, der sowohl unübersichtlich als auch schlecht wiederverwendbar ist. Die Gefahr ist groß, dass bestimmte Fehlerfälle bei dieser expliziten Behandlung vergessen werden und im besten Fall „silently swallowed“ werden, also unbehandelt bleiben und im schlechtesten Fall die Applikation zum Absturz bringen.

4.2.3 *Inversion of Control*

Ein weiteres Problem bei der Übergabe von Callbacks an die asynchrone Funktion ist die Umkehrung der Ausführungskontrolle („Inversion of control“). Durch die Übergabe eines Callbacks wird die Kontrolle über den weiteren Programmablauf vom Hauptprogramm an die aufgerufene Funktion abgegeben. Die aufgerufene Funktion ist für die Ausführung der übergebenen Callback-Funktion (der Continuation) verantwortlich. Das ist eine Inversion of Control, in der das aufrufende Programm keinen Einfluss mehr darauf hat, in welcher Art und Weise der an die aufgerufene Funktion übergebene „nächste Schritt“ ausgeführt wird.

Häufig handelt es sich bei den aufgerufenen Funktionen um fremden Code (3^{rd} Party Code, externe Bibliotheken, ...), der nicht als Quelltext zur Überprüfung vorliegt oder so komplex ist, dass eine Überprüfung aus praktischen Gründen ausscheidet. Die Programmiererin ist also darauf angewiesen der aufgerufenen Funktion zu vertrauen, dass diese den übergebenen Callback korrekt aufruft. Sie gibt damit die Kontrolle über den weiteren Ablauf des Programms aus der Hand.

Natürlich ist beim Aufruf externer Funktionen immer ein gewisses Maß an Vertrauen darauf notwendig, dass die fremde Funktion so funktioniert wie versprochen und ein korrektes Ergebnis gemäß ihrer Spezifikation liefert. Während in synchronem Code jedoch der schlimmste eintretende Fall darin besteht, dass ein inkorrektes Ergebnis zurückgegeben wird oder eine Ausnahme ausgelöst wird, so hat ein falsch verwendeter Callback direkte Auswirkungen auf den weiteren Ablauf des eigenen Programms. Der fremde Code könnte (vgl. [Simpson, 2015, S. 48])

- den Callback zu früh, also synchron anstatt asynchron, aufrufen,
- den Callback zu spät oder gar nie aufrufen,
- den Callback zu selten oder zu oft aufrufen,
- auftretende Fehler oder Ausnahmen verschlucken.

Insbesondere der erste Punkt stellt ein subtiles Problem bei der Übergabe von Callbacks an Fremdcode dar. Es ist einer aufgerufenen Funktion nicht anzusehen, ob sie intern wirklich asynchron arbeitet, den Callback also *später* ausführt oder ob es bestimmte Situationen gibt, in denen der Callback vielleicht doch synchron ausgeführt wird. Durch ein solches Verhalten wird eine Verzweigung im Ausführungspfad des Programms eingeführt, die den Code deutlich komplexer macht. Insbesondere wird die Run-to-Completion Semantik gebrochen, bei der die Programmiererin sich darauf verlassen kann, dass der eigene Code

vollständig abgearbeitet wird, bevor die Callback-Funktion aus dem asynchronen Aufruf heraus ausgeführt wird.[18]

Obwohl Gefahren bei den anderen genannten Punkten offensichtlicher sind, und damit leichter auffallen, stellen auch sie eine erhebliche Fehlerquelle für die korrekte Programmausführung dar.

Bei der Übergabe einer Continuation in Form eines Callbacks an eine asynchrone Funktion muss die Programmiererin darauf vertrauen, dass diese korrekt arbeitet. Selbst die ausführlichsten Tests können allenfalls Indizien liefern, die dieses Vertrauen rechtfertigen.

4.3 Bewertung traditioneller Callbacks

Die vorigen Abschnitte haben deutlich gezeigt, dass die Übergabe von Callbacks zur Verarbeitung asynchroner Ergebnisse zwar einfach zu realisieren ist, aber im Detail einige Probleme bereit hält. Diese reichen von Verständnisproblemen, die eher lästig sind, über eine aufwändige Fehlerbehandlung bis hin zu schwerwiegenden Vertrauensproblemen im Zusammenhang mit dem Code von Dritten.

Obwohl der Callback-Mechanismus das Fundament der asynchronen Programmierung darstellt, führen die angesprochenen Probleme zu der in der JavaScript Community weithin bekannten „Callback Hell“.

> **That** is what „callback hell“ is all about! The nesting/indentation are basically a side show, a red herring. [Simpson, 2015, S. 32]

5 MODERNE METHODEN IN ECMASCRIPT 6

Wie jede lebendige Programmiersprache entwickelt sich auch JavaScript beständig weiter. Es werden laufend neue Konzepte entwickelt, um die Verwendung der Sprache zu erleichtern und sicherer zu machen. In JavaScript werden solche neuen Konzepte häufig zunächst mit den Mitteln des aktuell gültigen Sprachstandards entwickelt und über sogenannte Polyfills[19] implementiert. Wenn sich die neu entwickelten Sprachmittel

18 Das Verhalten, dass eine Callback Funktion in manchen Fällen synchron und in anderen asynchron ausgeführt wird, ist in der Javascript-Community inzwischen weithin als „Release of ZALGO“ bekannt und sollte unbedingt vermieden werden. Die Bezeichnung für das Phänomen geht auf den inzwischen berühmten Blogpost von Isaac Schlüter zurück [Schlueter, 2013].
Um eine Vermischung von synchroner und asynchroner Ausführung zumindest im eigenen Code zu vermeiden, sollte man den folgenden Rat beherzigen:

> You can see just how quickly the unpredictability of Zalgo can threaten any JavaScript program. So the silly-sounding „never release Zalgo“ is actually incredibly common and solid advice. Always be asyncing. [Simpson, 2015, S. 37]

Das bedeutet, dass man den Aufruf eines übergebenen Callbacks immer zur *späteren* Ausführung in de Queue einreihen sollte, selbst wenn das Ergebnis im speziellen Fall schon *jetzt* zur Verfügung steht.
Eine bekannte Funktion, die gegen diesen Rat verstößt und ihren Callback je nach Situation synchron oder asynchron ausführt, ist z. B. die Funktion jQuery.ready(cb).

19 Als Polyfill bezeichnet man ein Stück JavaScript Code, der neuere Sprachfeatures per Bibliotheksfunktion auch in älteren Laufzeitumgebungen verfügbar macht. Diese Technik wird oft benutzt, um z. B. in Webbrowsern die unterschiedliche Unterstützung neuerer Funktionen auszugleichen. Je nach verwendetem Browser wird ein

in der Praxis bewähren, können sie in den Sprachstandard aufgenommen werden.[20]

NEUE SPRACHMITTEL: PROMISES UND GENERATORS In der Javascript Community wurden auch Konzepte entwickelt, um die asynchrone Programmierung zu erleichtern und der Callback-Hell zu entkommen. Insbesondere „Promises“ und „Generators“ sind dazu geeignet, die im vorigen Abschnitt angesprochenen Probleme traditioneller Callbacks zu lösen. Diese beiden Konzepte fanden Aufnahme in den ECMAScript 6 Standard (vgl. [Ecma TC39 und Wirfs-Brock, 2015]) und sind seitdem Teil aller kompatiblen Sprachimplementierungen.

Die Konzepte „Promises“ und „Generators“ sollen in diesem Kapitel als moderne Antwort auf die traditionellen Callbacks in JavaScript detailliert erläutert werden.

5.1 Promises

Wenn die Übergabe von Callbacks an asynchrone Funktionen zur *späteren* Verarbeitung der Ergebnisse offensichtlich problematisch ist, so stellt sich zwangsläufig die Frage nach Alternativen:

> What if instead of handing the continuation of our program to another party, we could expect it to return us a capability to know when its task finishes, and then our code could decide what to do next? [Simpson, 2015, S. 39]

5.1.1 *Die Idee von Promises*

Interessanterweise ist schon lange ein Konzept bekannt, welches genau das leistet: Promises als Platzhalterobjekt für Ergebnisse, die erst in *späterer* Zukunft vorliegen.

Anstatt der asynchron aufgerufenen Funktion die Continuation ihres Programms anzuvertrauen, erhält die Programmiererin ein Platzhalterobjekt zurück, das letztendlich nach Abschluss der asynchronen Operation deren Ergebnis enthält.

Dieses Konzept wurde 1976 in einem Paper [Friedman und Wise, 1976, S. 263] das erste Mal für die Parallelisierung von funktionaler „Lazy Evaluation“ in Mehrprozessorsystemen vorgestellt und dort als „Promise“ benannt:

> in fact [. . .] z is initially bound only to a „promise“ of this result. [Friedman und Wise, 1976, S. 268]

Obwohl das dort vorgestellte Konzept einen anderen Ausgangspunkt hatte, waren die damit gelösten Probleme doch denen der asynchronen Programmierung sehr ähnlich:

> The colonel behaves exactly as a single processor would, except that from time to time it accesses what would have been a suspension and instead finds the result already provided by a sergeant who had passed through earlier. [Friedman und Wise, 1976, S. 269]

passender „Polyfill“ mit der Webseite ausgeliefert, so dass die Programmiererin für alle Plattformen die gleichen Funktionen und APIs nutzen kann. (vgl. [Sharp, 2010])

20 Zur Aufnahme neuer Features in den Sprachstandard vgl. den Prozess in [Ecma TC39 committee, 27.09.2016].

Der sogenannte „Colonel“ Prozess läuft auf einem einzigen Thread und bekommt von parallel in anderen Ausführungskontexten arbeitenden „Sergeants“ Ergebnisse geliefert, mit denen er weiterarbeiten kann.

Die erste bekannte Implementierung von Promises, um explizit asynchrone Prozeduraufrufe aufzulösen, wurde 1988 in [Liskov und Shrira, 1988, S. 260] vorgestellt. Die dort vorgestellten Promises geben schon 27 Jahre bevor die eingangs dieses Abschnitts zitierte Frage gestellt wurde eine zufriedenstellende Antwort darauf:

> Promises allow a caller to run in parallel with a call and to pick up the results of the call, including any exceptions it raises, in a convenient and type-safe manner. [Liskov und Shrira, 1988, S. 260]

GEÄNDERTE AUFRUFSEMANTIK VON ASYNCHRONEN FUNKTIONEN Beim bisherigen Konzept zum Aufruf asynchroner Funktion wurde diesen eine Callback-Funktion übergeben, welche nach Beendigung der asynchronen Aufgabe zur Rückgabe des Ergebnisses aufgerufen wurde. Bei der Verwendung von Promises wird die Situation umgedreht und die aufgerufene Funktion gibt direkt und synchron (also *jetzt*) ein Promise-Objekt als Platzhalter für das zu einem *späteren* Zeitpunkt vorliegende Ergebnis zurück. Diese Promise kann von der eigenen Applikation wie ein normales Objekt verwendet werden. Sie kann z. B. in einer Variable zwischengespeichert oder an eine andere Funktion weitergereicht werden. Wenn das Ergebnis des asynchronen Aufrufs in der Promise schließlich vorliegt, kann dieses weiterverarbeitet werden, ohne dass dazu vorher eine Continuation an die asynchrone Funktion herausgegeben werden müsste.

5.1.2 *Promises nach dem Promise/A+ Standard*

Das Konzept der Promises als Platzhalterobjekt für das Ergebnis einer asynchronen Operation ist auch in der JavaScript Community schon seit längerem bekannt und es gibt eine ganze Reihe von Bibliotheken mit denen es umgesetzt werden kann. Um eine Interoperabilität dieses Konzepts zu erreichen, wurde der „Promise/A+“-Standard (vgl. [Promises A+, 2016]) entwickelt, der eine Semantik für Promises definiert, an den sich die allermeisten Promise-Bibliotheken halten.[21]

Mit der Version 6 der ECMAScript Spezifikation finden Promises nach dem Promise/A+-Standard im Jahr 2015 auch Eingang in den JavaScript Sprachstandard und sind seitdem Teil aller kompatiblen Sprachimplementierungen (vgl. [Ecma TC39 und Wirfs-Brock, 2015]). Die Funktionsweise dieser Promises soll im folgenden detailliert erläutert werden.

Eine Promise hat einen Wert und einen Status. Je nach Status ist der Wert der Promise schon bekannt, oder er wird erst später zugewiesen wobei der Status sich ändert.

21 Eine populäre Ausnahme bildet die jQuery-Bibliothek, die erst ab Version 3.0 (9. Juni 2016) kompatibel zum Promise/A+ Standard ist. Die folgenden Ausführungen sind daher nicht auf ältere Versionen von jQuery anwendbar.

STATUS EINER PROMISE Einem Promise Objekt ist einer von drei Status zugeordnet:

> Any Promise object is in one of three mutually exclusive states: *fulfilled*, *rejected*, and *pending*:
>
> - A promise p is fulfilled if p.then(f, r) will immediately enqueue a Job to call the function f.
> - A promise p is rejected if p.then(f, r) will immediately enqueue a Job to call the function r.
> - A promise is pending if it is neither fulfilled nor rejected.
>
> A promise is said to be *settled* if it is not pending, i.e. if it is either fulfilled or rejected. [Ecma TC39 und Wirfs-Brock, 2015, § 25.4]

Diese drei Zustände einer Promise sind in Abbildung 2 dargestellt.

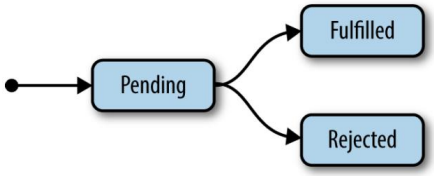

Abbildung 2: Die drei Status einer Promise. (aus [Parker, 2015, S. 16])

Solange in der asynchronen Funktion aus der die Promise zurückgegeben wurde noch kein Wert vorliegt, bleibt die Promise im Status *pending*. Sobald die asynchrone Funktion ein Ergebnis ermittelt hat, wird dieses dem Wert der Promise zugewiesen und die Promise *settled* damit. Ihr Status ändert sich entweder in *fulfilled* oder *rejected*.

Die Zuweisung an das Wertobjekt der Promise kann entweder ein Ergebniswert („value“), ein Fehlerobjekt („reason“) oder aber eine weitere Promise sein. Der Normalfall ist die Zuweisung eines Ergebnisses mit dem Übergang in den Status *fulfilled*. Falls bei der Ausführung der asynchronen Funktion ein Fehler auftritt, so bricht diese mit einem Fehlerobjekt ab und die Promise geht in den Status *rejected* über. In beiden Fällen gilt die Promise als *settled* und wird zu einem unveränderlichen (immutable) Objekt.

Wenn während des Settlements einer Promise diese als Wert eine weitere Promise zugewiesen bekommt, so übernimmt die äußere Promise Status und Wert der neu zugewiesenen inneren. Man spricht von „lock-in“. Verschachtelte Promises werden rekursiv aufgelöst, bis am Ende ein einzelnes Promise Objekt übrig bleibt welches entweder den Status *fulfilled* oder *rejected* annimmt.

Diese Zustandsübergänge sind noch einmal in Abbildung 3 angegeben.

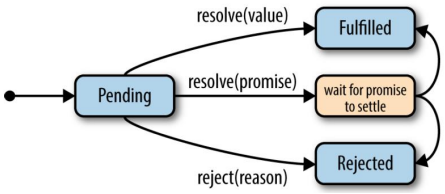

Abbildung 3: Settlement einer Promise mit verschiedenen Werten. (aus [Parker, 2015, S. 17])

SIGNALISIERUNG UND BENUTZUNG DES WERTS Man kann einer Promise von außen nicht ansehen, in welchem Status sie sich gerade befindet. Der Status kann sich jederzeit von *pending* zu *settled* ändern. Um den endgültigen Wert einer erhaltenen Promise im eigenen Code zu verwenden, stellt jede Promise eine Methode p.then(f, r) bereit, die Handler-Funktionen registriert, welche ausgeführt werden, wenn eine Promise im Status *settled* ist. Sie hat die Signatur promise.then([f(val)], [r(val)]) returns promise.

Jede Handler-Funktion f(val) und r(val) nimmt je einen Parameter. In diesem Parameter wird ihr der Wert der zugrundeliegenden (*settled*) Promise übergeben. Dieser Wert kann dann verarbeiten werden. Die beiden Funktionsparameter f und r von then([f], [r]) sind optional.

Der Aufruf p.then(f, r) auf einer Promise p registriert die übergebenen Funktionen als Ergebnis-Handler für diese Promise. Der Handler f wird für die Verarbeitung des Wertes der Promise im Status *fulfilled* registriert, der Handler r für die Verarbeitung des Wertes der Promise im Status *rejected*.

Der Aufruf p.then(f, r) kann beliebig oft und zu beliebigen Zeiten auf einer Promise erfolgen. Bei mehrfachem Aufruf werden weitere Handler-Funktionen registriert. Die Promise selber wird immutable, sobald sie *settled* ist und verändert sich durch weitere Registrierung von Handler-Funktionen nicht mehr.

Wenn die Promise *settled* ist, werden je nach Status entweder alle registrierten Fulfillment-Handler oder aber alle registrierten Rejection-Handler zur Ausführung gebracht. Wenn beim Aufruf von p.then() ein Parameter weggelassen wird und somit kein Handler für *fulfilled* oder *rejected* registriert wird, so wird der leere Standard-Handler ausgeführt. Da es relativ häufig vorkommt, dass nur der Rejection-Handler registriert werden soll, wurde in ES6 die Abkürzung p.catch(r) für den Aufruf p.then(null, r) eingeführt.

Wird eine Handler-Funktion auf einer Promise registriert, die schon *settled* ist, so wird die Ausführung des passenden Handlers direkt in die Queue eingereiht. Der Aufruf einer jeden Handler-Funktion erfolgt immer asynchron durch Einreihung in die Queue. Ein synchroner Aufruf des Handlers ist ausgeschlossen!

PROMISE CHAINING Die Methode p.then() hat als Rückgabewert selber ein Promise Objekt. Wert und Status dieser neuen Promise hängen von der ausgeführten Handler-Funktion ab: Wird eine Handler-Funktion fehlerfrei ausgeführt, so wird die neue Promise mit dem Rückgabewert dieses Handlers *resolved*. Ist die Rückgabe des Handlers ein einfacher Wert (ein „Immediate“), so hat die durch p.then() zurückgegebene Promise sofort den Status *fulfilled* und das Immediate als Wert. Das gilt auch wenn der Rejection-Handler ausgeführt wird und ein Immediate zurückliefert.

Tritt während der Ausführung einer Handler-Funktion ein Fehler auf, so wird die neue Promise mit dem Fehlerobjekt der throw-Klausel *rejected*.

Falls die ausgeführte Handler-Funktion keinen expliziten Wert per return zurückgibt, so wird die ursprüngliche Promise auf der p.then()

aufgerufen wurde zurückgegeben. Das trifft auch zu, wenn einer der optionalen Handler nicht angegeben wurde. Der (leere) Standard-Handler gibt einfach die ursprüngliche Promise zurück.

Dadurch ist es leicht möglich, auch auf der neuen, von p.then() zurück gegebenen Promise mittels eines weiteren (p.then()).then()-Aufrufs wieder Handler zu registrieren. Diese weiteren Handler-Funktionen können zum Beispiel weitere asynchrone Aufrufe auslösen, die von dem zuvor erhaltenen Ergebnis abhängen. Durch dieses sogenannte Promise-Chaining können auf einfache Weise sequentielle Abläufe aus asynchronen Aufrufen formuliert werden.

Ein Beispiel für die Benutzung der then()-Methode zur Verarbeitung und Verkettung von Promise-Werten ist in Listing 3 angegeben.

```
//single Rejection Handler at the end of a Promise chain
//the called functions are promise returning asynch functions

var p = foo();

p.then(result => {
    return bar(result);
}).then(result => {
    return baz(result);
}).then(result => {
    return booze(result);
}).then(result => {
    console.log("Chain ended with success: ", result)
}).catch(e => {
    console.log("chain got stuck with error:", e);
})
```

Listing 3: Eine Kette voneinander abhängiger Promises mit zentralem Rejection-Handler am Ende.

Über die Registrierung der Handler-Funktionen kann der JavaScript Code, der asynchrone Aufrufe nutzt, bestimmen, wie diese Ergebnisse verarbeitet werden sollen. Die Handler-Funktionen werden aufgerufen, sobald das Ergebnis des asynchronen Aufrufs vorliegt. Sie übernehmen die Rolle der traditionellen Callback-Funktionen.

Der wesentliche Unterschied zu asnychronen Aufrufen mit traditionellen Callbacks ist die Tatsache, dass die Continuation des Programms nicht mehr an die asynchrone (3rd Party) Funktion herausgegeben wird, sondern dass die Handler-Funktion auf dem wohldefinierten Promise-Objekt registriert wird und die einmalige, immer asynchrone Ausführung jeder registrierten Handler-Funktion durch den Sprachstandard garantiert ist. Durch die immutability einer *settled* Promise ist garantiert, dass jede Handler-Funktion, die auf einer Promise registriert wird, immer mit dem gleichen Ergebniswert aufgerufen wird.

Auch die Fehlerbehandlung ist dadurch klar definiert, dass zwischen dem normalen und dem fehlerhaften Abschluss eines asynchronen Aufrufs mit den beiden unterschiedlichen Status *fulfilled* und *rejected* unterschieden wird.

5.1.3 *Erzeugung von Promises in ES6*

Natürlich müssen Promises nicht nur konsumiert und von externen Funktionen zurückerhalten werden, sondern es ist auch notwendig, solche Objekte zu erzeugen und deren *settlement* zu steuern.

Um eine Promise zu erzeugen, stellt ECMAScript 6 einen Konstruktor zur Verfügung: new Promise(function(resolve, reject){...}) returns promise.

Der Promise Konstruktor nimmt als einzigen Parameter eine sogenannte resolver-Funktion, die direkt innerhalb des new Promise() Konstruktors synchron ausgeführt [Simpson, 2015, S. 45] wird. Die beiden Parameter resolve und reject dieser resolver-Funktion sind selber Funktionsobjekte, die dazu benutzt werden, die neu erstellte Promise entweder in den Status *fulfilled* oder *rejected* zu überführen. Jede der beiden Funktionen nimmt selber einen Parameter: resolve(val) den Wert, mit dem die Promise *fulfilled* werden soll, und reject(e) das Fehlerobjekt, das zur *rejection* der Promise führt.

Auf jeder Promise kann immer nur eine dieser beiden Funktionen genau einmal aufgerufen werden. Beim Aufruf von entweder resolve(val) oder reject(e) geht die Promise in einen *settled* Status über und wird immutable. Weitere Versuche die Promise zu *settlen* schlagen fehl.

Da die resolver-Funktion innerhalb des Konstruktors synchron aufgerufen wird, obliegt es der Erstellerin der verwendenden asynchronen Funktion, keinen blockierenden Code in der resolver-Funktion zu platzieren, sondern den Aufruf von entweder resolve oder reject über eine Callback-Ebene asynchron auszuführen.

Das Prinzip soll an einem Beispiel in Listing 4 gezeigt werden:

```
//Example of using the new Promise() constructor
//from Parker, Daniel, Javascript with promises, p. 13

function loadImage(url) {
    var promise = new Promise(
        function resolver(resolve, reject) {
            var img = new Image();
            img.src = url;
            img.onload = function () {
                resolve(img);
            };
            img.onerror = function (e) {
                reject(e);
            };
        });
    return promise;
}
```

Listing 4: Asynchrone Funktion mit Promise als Rückgabe. (Code aus [Parker, 2015, S. 13])

Der asynchrone und lang laufende Aufruf im Beispiel wird durch die Verknüpfung der url mit der Property img.src angestoßen. Der Aufruf der Funktionen resolve bzw. reject wird zurückgestellt, bis entweder der Event img.onload oder der Event img.onerror in der Queue der Laufzeitumgebung auftauchen. Das bedeutet, dass zwar die resolver-Funktion synchron aufgerufen wird, dabei aber nicht blockiert, sondern lediglich den Aufruf von resolve und reject zu bestimmten Events registriert. Das neu erzeugte Promise Objekt kann sofort synchron zurückgegeben werden. Die Funktionen zur Zuweisung des Ergebniswerts werden dagegen ähnlich wie bei der Verwendung von traditionellen Callbacks zurückgestellt und *später* aufgerufen, wenn das Ergebnis des asynchronen Aufrufs vorliegt.

WEITERE ES6-API FUNKTIONEN ZUR ERZEUGUNG VON PROMISES Die Erzeugung über den gerade gezeigten sogenannten „revealing Constructor“ ist nicht der einzige Weg in ECMAScript 6 eine neue Promise zu erzeugen. Es gibt noch weitere API-Funktionen, welche häufig vorkommende Muster bei der Erzeugung von Promises abstrahieren und bequemer zu benutzenden syntaktischen Zucker dafür anbieten.

PROMISE.RESOLVE() UND PROMISE.REJECT() Es kommt relativ häufig vor, dass von einer Funktion eine Promise als Rückgabeobjekt erwartet wird, obwohl das Ergebnis gar nicht asynchron ermittelt wird.[22]

In diesen Fällen gibt es die beiden Abkürzungen Promise.resolve(val) und Promise.reject(e), welche eine Promise zurückgeben, die direkt in einem *settled* Status ist. Der Wert dieser Promise entspricht dem Parameter der Funktion.

PROMISE.ALL() UND PROMISE.RACE() Oft kommt es vor, dass verschiedene asynchrone Aufrufe voneinander abhängen und das Endergebnis erst von Interesse ist, wenn diese Abhängigkeiten erfüllt sind. Auch dazu wurden zwei spezielle syntaktische Elemente in die ES6 API aufgenommen, um die Übersichtlichkeit und Benutzbarkeit zu verbessern.

Promise.all(it) nimmt als Parameter eine Kollektion (ein „iterable“) von Promises und liefert eine einzelne Promise an die Aufruferin zurück. Diese einzelne Promise nimmt den Status *fulfilled* an, sobald alle Promises der Kollektion it in den Status *fulfilled* übergegangen sind. Der Wert der resultierenden Promise ist eine Kollektion mit den Werten der einzelnen Promises aus it. Wenn jedoch wenigstens eine einzelne der übergebenen Promises in it den Status *rejected* annimmt, so nimmt die gesamte von Promise.all() zurückgegebene Promise den Status *rejected* an und bekommt als Wert das Fehlerobjekt der ersten Promise aus it zugewiesen, die in den Status *rejected* übergegangen ist.

Mit Hilfe von Promise.all() können auf einfache Weise mehrere asynchrone Aufrufe gleichzeitig abgesetzt werden, die dann nebenläufig ausgeführt werden. Die Endergebnisse werden erst nach erfolgreichem Abschluss aller Aufrufe gesammelt zur Verfügung gestellt.

Promise.race(it) dagegen nimmt zwar auch eine Kollektion (ein „iterable“) von Promises als Parameter, liefert aber selber eine Promise mit nur einem einzigen Ergebniswert zurück. Die zurückgelieferte Promise übernimmt Wert und Status der ersten *settled* Promise aus der übergebenen Kollektion it. Das kann entweder die erste *fulfilled* oder die erste *rejected* Promise sein.

Promise.race(it) wird häufig dazu verwendet, einen asynchronen Aufruf mit einem Timeout zu versehen. Dazu wird als ein Element in it eine Funktion angegeben, die nach einer vorgegebenen Zeitspanne ihre

22 Dies ist z. B. häufig der Fall wenn eine Funktion mehrfach aufgerufen wird und ein bestimmter Wert schon bei einem früheren Aufruf in einem Cache abgelegt wurde. Um nicht asynchrone und synchrone Aufrufsemantiken zu vermischen (vgl. „Release of ZALGO“ [Schlueter, 2013]) muss auch dann eine Promise zurückgegeben werden.

Rückgabepromise *rejected*. In diesem Fall wird die gesamte von Promise.race() zurückgegebene Promise *rejected* und die Aufruferin kann anhand des Fehlercodes im Wert sehen, dass ein Timeout zugeschlagen hat.

5.1.4 *ES6 Promises im Vergleich zu traditionellen Callbacks*

> You've no doubt noticed that Promises don't get rid of callbacks at all. They just change where the callback is passed to. Instead of passing a callback to foo(..) , we get *something* (ostensibly a genuine Promise) back from foo(..), and we pass the callback to that *something*instead. [Simpson, 2015, S. 52]

Nach der Vorstellung der Arbeitsweise von Promises soll nun untersucht werden, welche Vorteile Promises gegenüber der Verwendung traditioneller Callbacks in der praktischen Anwendung haben.

BESSERE LESBARKEIT Einer der Hauptvorteile der Verwendung von Promises gegenüber Callbacks liegt in der deutlich verbesserten Lesbarkeit von sequentiellen, aufeinander aufbauenden asynchronen Aufrufen. Bei der Formulierung aufeinander aufbauender Sequenzen mittels Callbacks baut sich unweigerlich die „Pyramid of Doom“ mit immer tiefer verschachtelten Callbacks und Einrückungen auf. Eine Sequenz mit Promises dagegen kann ganz einfach in einer Promise-Chain geschrieben werden, deren Einrückungstiefe über den ganzen Verlauf konstant bleibt. (Siehe dazu Listing 3.)

Die flache Struktur der Promise-Chain ist aber nicht nur typographisch gefälliger. Sie wird von oben nach unten abgearbeitet und ist damit den gewohnten Denkstrukturen unseres Gehirns wesentlich ähnlicher als eine tief geschachtelte Abfolge von Callbacks, die von außen nach innen abgearbeitet wird.

Immer dann, wenn asynchrone Aufrufe von den Ergebnissen anderer asynchroner Aktionen abhängen, ist die „flache“ Schreibweise der Promises den ineinander geschachtelten traditionellen Callbacks überlegen.

VEREINFACHTES ERROR HANDLING Jede Exception, die innerhalb der resolver-Funktion des Promise-Konstruktors auftritt, führt dazu, dass die zu konstruierende Promise in den Status *rejected* übergeht und als Wert das Error-Object der Exception zugewiesen bekommt. Neben dieser impliziten Rejection kann eine Promise auch explizit *rejected* werden, wenn ein Fehler bemerkt wird. Der „rejection reason“ muss dann explizit gesetzt werden.[23]

Es ist also sehr einfach auf standardisierte Weise den Misserfolg oder Fehler aus einer asynchronen Funktion zurück zu melden. Die zugehörige Promise geht dann in den Status *rejected*.

Für die Behandlung von *rejected* Promises kann mittels p.then(f, r) (bzw. mittels p.catch(r)) die Handler-Funktion r registriert werden.

Jeder Aufruf von p.then() (bzw. von p.catch()) erzeugt als Rückgabewert wieder eine Promise. Es wird also im Innern der Promise Konstruk-

23 Dazu sollte ein Error-Objekt erzeugt werden, welches neben der textuellen Fehlermeldung noch weitere Informationen über den Fehler aufnehmen kann (vgl. [Parker, 2015, S. 22]).

tor aufgerufen. Eine Exception während der Ausführung von p.then() liefert daher automatisch eine *rejected* Promise auf der weitere verkettete then() Aufrufe stattfinden können. Eine *rejected* Promise durchläuft eine solche Kette von then() Aufrufen solange, bis sie auf einen registrierten Rejection-Handler trifft. Aufrufe von then(f, null), in denen lediglich der Fulfillment-Handler und kein Rejection-Handler registriert wird, durchläuft die *rejected* Promise unverändert.

> Rejections and errors propagate through promise chains. When one promise is rejected all subsequent promises in the chain are rejected in a domino effect until an onRejected handler is found. In practice, one catch function is used at the end of a chain [...] to handle all rejections. [Parker, 2015, S. 20]

Diese Verhalten macht eine sehr übersichtliche und zentralisierte Fehlerbehandlung auch in langen Ketten von asynchronen Aufrufen möglich. Es ist lediglich eine einzelne Fehlerbehandlungsroutine für die gesamte Kette zu implementieren. Häufig ist es dabei ausreichend festzustellen, dass die Abarbeitung der Kette nicht fehlerfrei funktioniert hat, um die Applikation danach wieder geordnet weiter ausführen zu können. Es kann aber auch anhand des Rejection-Reason (also des Wertes der *rejected* Promise) festgestellt werden, in welchem Schritt die Abarbeitung der Kette abgebrochen ist, um dann noch spezifische Aktionen, wie z. B. die Freigabe zuvor angeforderter Resourcen, zu veranlassen.[24]

UN-INVERSION OF CONTROL Auch bei der Verwendung von Promises kommen Callbacks zum Einsatz. Diese werden jedoch nicht mehr auf dem Fremdcode direkt registriert, sondern auf dem zurückgegebenen Promise Objekt, dessen Semantik über den Sprachstandard von ECMAScript 6 garantiert ist. Die Erzeugung der Promise im Fremdcode geschieht über einen Callback innerhalb der resolver-Funktion. An dieser Stelle findet eine erste „Inversion of Control“ für den Fremdcode statt. Beim Beobachten der Promise durch den eigenen Code wird mittels p.then() ein eigener Callback (die Handler-Funktion) auf der Promise registriert. Auch diese Registrierung kann als „Inversion of Control“ interpretiert werden. Damit findet eine doppelte Invertierung statt, die im Ergebnis einer „Un-Inversion of Control“ entspricht. Weder

24 Es ist gute Praxis **jede** Promise Chain mit einem p.catch(r) abzuschließen, um auszuschließen, dass innerhalb der Kette ein Fehler auftritt, der still und leise verschluckt wird. Ohne einen einzigen Rejection-Handler würde einfach die Bearbeitung der Promise-Chain abbrechen und die Rückgabe der gesamten Kette würde zu einer *rejected* Promise. Diese Promise würde aber niemals wieder aufgelöst, so dass der Fehler nicht bemerkt würde. Er tritt höchstens durch das Ausbleiben von Aktionen zutage, die nach Auftreten des Fehlers innerhalb von weiteren Fulfillment-Handlern angestoßen werden sollten.
Die gleiche Situation tritt auf, wenn im letzten Rejection-Handler der Promise-Kette eine Exception auftritt. Auch in diesem Fall wird die Rückgabe der Kette zu einer *rejected* Promise, die jedoch niemals aufgelöst wird. Der Fehler wird so im besten Fall still ignoriert.
Dieses Problem wird mit den in ECMAScript 6 standardisierten Promises nicht gelöst. Einige JavaScript-Implementierungen erkennen nicht behandelte *rejected* Promises und berichten darüber zumindest in der Console (vgl. [Rauschmayer, 2016] oder [Simpson, 2015, S. 64]). In diesem Text soll darauf nicht näher eingegangen werden.

die Programmiererin der Applikation noch die Erstellerin des asynchronen Fremdcodes müssen die Continuation ihres eigenen Programms an fremden Code herausreichen. Sie benutzen lediglich vertrauenswürdige und klar definierte Mechanismen des Sprachkerns.

Durch die Verwendung von Promises bekommt die Programmiererin also die Kontrolle über ihren Programmablauf zurück. Die durch die Inversion of Control bei traditionellen Callbacks potentiell auftretenden Probleme werden durch die Verwendung von Promises zuverlässig vermieden. Es wird sichergestellt, dass eine vom eigenen Programm registrierte Ergebnishandler-Funktion genau einmal asynchron aufgerufen wird. Ein zu früher (synchroner) Aufruf kann genauso ausgeschlossen werden wie ein mehrfacher oder gar kein Aufruf.

Selbst wenn der asynchrone FremdCode sein Ergebnis synchron statt asynchron zurück gibt, so wird doch spätestens die Beobachtung mittels p.then() asynchron zurückgestellt und auf einen *späteren* Tick der Event-Loop verschoben. Eine asynchrone Verarbeitung des Ergebnisses ist damit garantiert.

Wenn eine Promise in den Status *settled* übergeht, werden alle darauf registrierten Handler in die Event-Queue eingereiht und damit genau einmal aufgerufen. Selbst Handler, die erst nach dem *settlement* der Promise registriert werden, kommen zuverlässig asynchron zur Ausführung.

Natürlich kann es vorkommen, dass eine externe Funktion ihren Zweck nicht vertragsgemäß erfüllt und die zurückgegebene Promise auf immer im Status *pending* belässt. Doch auch dagegen kann sich die Programmiererin des aufrufenden Programms eigenverantwortlich wappnen, indem sie die Auflösung der Promise mit einem Timeout über Promise.race() versieht. Wenn der Timeout abläuft kann die Promise unter eigener Kontrolle entweder *rejected* oder z. B. mit einem default Wert *fulfilled* werden. Da die *settled* Promise immutable ist, kann es nicht passieren, dass der externe Code nach Ablauf des Timeouts einen weiteren (dann zu späten) Aufruf des Handlers triggert. Es ist eine zuverlässige und eindeutige Ausführungssemantik definiert, über welche die Programmiererin die Kontrolle hat.

5.2 Async Control Flow mit Generators

Die bisher vorgestellten Konzepte Callbacks und Promises drehten sich hauptsächlich darum, wie die Ergebnisse asynchroner Aufrufe wieder in den Programmablauf des Hauptprogramms zurück übertragen werden können. Ein weiterer wichtiger Aspekt bei der Benutzung asynchroner Programmierung ist die Frage, wie der asynchrone Programmablauf gesteuert wird und wie auftretende Abhängigkeiten aufgelöst werden.

Es wurde deutlich, dass komplexere Abläufe bei der Verwendung von konventionellen Callbacks schnell unübersichtlich werden und damit die Programmiererin über viele ineinander geschachtelte Callback-Funktionen mit jeweils eigener Fehlerbehandlung auf direktem Wege in die „Callback-Hell“ befördern können.

Mit der Verwendung von Promises hat sich neben den erwähnten Aspekten der verbesserten Sicherheit auch die Lesbarkeit des Codes dramatisch verbessert. Um sequentielle Abläufe zu beschreiben, können Promise-Chains verwendet werden, die sich fast schon wie die Aneinanderreihung von konventionellem synchronem Code lesen.

Mit dem neuen Sprachstandard ECMAScript 6 wurden Generators als weiteres Sprachkonstrukt eingeführt. Es hat sich gezeigt, dass die Kombination von Promises und Generators ein mächtiges Werkzeug zur Programmablaufkontrolle darstellt, welche den Code noch weiter vereinfacht und asynchronen Code genauso einfach lesbar und verständlich macht wie synchronen Code, ohne dass dabei auf die Vorteile der asynchronen Programmierung verzichtet werden muss. Sogar die gewohnte Fehlerbehandlung mittels try-catch-finally wird in dieser Form möglich.

Da Generators nicht primär zur Flusskontrolle von asynchronem Code eingeführt wurden, muss zunächst die allgemeine Funktionsweise erläutert werden, bevor ihre Anwendung auf die asynchrone Programmierung erklärt wird.

5.2.1 Iterators *und* Iterables

Im ECMA 6 Standard werden die beiden Interfaces *Iterable* und *Iterator* (vgl. [Ecma TC39 und Wirfs-Brock, 2015, § 25.1.1]) definiert. Über die Elemente von Objekten, die *Iterable* implementieren, lässt sich leicht iterieren. Dazu wird ein *Iterator* it benutzt, der die Funktion it.next() zur Verfügung stellt, mit dem jeweils das nächste Element des *Iterable* angefordert werden kann.

Ein Aufruf von it.next() liefert ein Objekt mit den beiden Properties done und value zurück. Mittels des Werts false in done wird signalisiert, dass das zugrundeliegende *Iterable* noch weitere Elemente enthält. Der Wert des angeforderten Elements selber ist in value abgelegt.

Ein Beispiel für ein *Iterable* Objekt sind Arrays, deren einzelne Elemente mittels eines *Iterators* nacheinander verarbeitet werden können.

```
//Example of an iteration over Array using an Iterator

var array = [1, 2, 3];

var it = array[Symbol.iterator](); //get the iterator

while ((element = it.next()).done != true) {
    console.log("Element: ", element.value);
}
console.log(it.next()); //another next() always fails
//Element:  1
//Element:  2
//Element:  3
//{ value: undefined, done: true }
```

Listing 5: Iteration über die Werte eines Arrays mittels eines *Iterator*

Im Listing 5 wird zunächst der *Iterator* des Arrays in der Variable it referenziert. In der folgenden Schleife wird jedes Element angefordert und verarbeitet bis kein weiteres Element mehr vorhanden ist. Zur Demonstration wird danach noch ein weiteres Mal it.next() aufgerufen. Bei

einem erschöpften *Iterable* liefert dieser Aufruf den Wert {done: true, value: undefined}. Die it.next() Methode kann beliebig oft aufgerufen werden, ohne dass es zu einem Fehler kommt.

Für die in Listing 5 explizit ausprogrammierte Schleife ist in ECMAScript 6 auch die Abkürzung for (var v of iterator) definiert. (vgl. [Ecma TC39 und Wirfs-Brock, 2015, § 13.7.5])

Die beschriebene Art der Iteration ist in ECMAScript 6 nicht auf Arrays beschränkt, sondern kann auf allen Objekten verwendet werden, welche das *Iterable* Interface implementieren, also über die Property obj[Symbol.iterator] einen *Iterator* zurückgeben, der eine Methode it.next() bereit stellt.

5.2.2 *Generator Functions in ECMAScript 6*

Aufbauend auf den *Iterables* und *Iterators* wurden in ECMAScript 6 sogenannte Generator-Funktionen eingeführt. Diese Funktionen implementieren sowohl das *Iterable* als auch das *Iterator* Interface und werden zur Erzeugung einer Sequenz von Elementen verwendet.

Der Generator wird als Funktion programmiert, die jedoch eine für JavaScript ungewöhnliche neue Eigenschaft hat: Die Abarbeitung des Generators lässt sich an beliebigen Stellen unterbrechen. Dazu wird das neue Schlüsselwort yield in den Code des Generators platziert. Sobald bei der Abarbeitung eines Generators ein yield erreicht wird, pausiert die Funktion und gibt die Ablaufkontrolle zurück. Die pausierende Generator-Funktion behält ihren Status und wartet auf die nächste Anforderung eines von ihr generierten Elements. Bei einer solchen Anforderung wird der Wert rechts vom Schlüsselwort yield als value-Property zurückgegeben.

```
//Example of a really stupid generator function
function* generate() {
    var i = 0;
    yield i += 1;
    yield i += 1;
    yield i += 1;

    return;
}

var it = generate();

for (element of it) {
    console.log("Element: ", element);
}
console.log(it.next()); //another next() always fails
//Element:  1
//Element:  2
//Element:  3
//{ value: undefined, done: true }
```

Listing 6: Eine einfache Generator-Funktion für 1, 2, 3

Das sehr synthetische Beispiel in Listing 6 soll die Syntax der Generators in JavaScript verdeutlichen: Generators werden wie Funktionen deklariert, jedoch wird das Schlüsselwort function* verwendet um sie von normalen Funktionen (ohne „*“) zu unterscheiden. Wenn die so deklarierte Generator-Funktion aufgerufen wird, so wird sie bis zum

ersten yield ausgeführt und pausiert dort. Als Rückgabewert dieses ersten Aufrufs wird das *Iterator*-Objekt zurückgeliefert, über das Werte aus dem Generator angefordert werden können. Sie sind der Inhalt des zugehörigen *Iterable*.

Wie erwähnt wird bei Generators die Run-to-Completion Syntax gebrochen: Der Generator wird eben **nicht** bis zum Ende ausgeführt, sondern pausiert bei Auftreten eines yield und kann seine Ausführung unter Beibehaltung seines internen Zustands beim nächsten Aufruf von it.next() auf seinem *Iterator* it wieder aufnehmen.[25]

Dieses Verhalten ist eng verwandt mit den aus der funktionalen Programmierung bekannten „Streams“ die verzögert („lazy“) ausgewertet werden und auf Anfrage jeweils das nächste Element liefern. Wie bei Streams können auch Generators prinzipiell unendliche Datenstrukturen – z. B. alle natürlichen Zahlen – erzeugen. Sie benötigen kein explizites Ende, sondern können ihre yield Statements in einer Endlosschleife platzieren.

ZWEI-WEGE-KOMMUNIKATION Um einen Generator anzuhalten und einen Wert an den Aufruf von it.next() zurückzugeben, wird im Generator das Schüsselwort yield benutzt. Der Aufrufer bekommt den Wert des Ausdrucks der im Generator rechts von yield steht zurück, und der Generator pausiert.

Darüber hinaus kann in einer Zwei-Wege-Kommunikation auch ein Wert von außen in den Generator injiziert werden. Dazu dient in der aufrufenden Funktion der Aufruf von it.next(value). Der optionale Parameter value ist dabei gesetzt, und wird innerhalb des Generators genauso verarbeitet, als wenn er direkt an der Stelle von yield stehen würde.

Ein Generator kann auch von außen beendet werden. Dazu wird auf dem zugehörigen *Iterator* it entweder die Funktion it.return(value) oder aber it.throw(e) aufgerufen. Die Parameter sind jeweils optional.

Im Fall von it.return(value) wird der Generator beendet und liefert als letzten Wert value zurück. Der Generator verhält sich so, als ob an der Stelle des yield die Anweisung return(value) stehen würde. Genauso funktioniert die Injektion eines it.throw(e). Innerhalb des Generator wird an der Stelle von yield die Exception e geworfen. Diese kann innerhalb des Generators mit einem try-catch-finally Block abgefangen werden oder der Generator wird beendet und der Fehler wird nach außen an den Aufruf des zugehörigen *Iterators* weiter propagiert.

5.2.3 *Generator Functions und Promises*

Generators können dazu benutzt werden, eine Funktion schrittweise ablaufen zu lassen. Die einzelnen Schritte werden durch die yield Statements definiert, an denen der Ablauf des Generators pausiert, Ergeb-

25 Das ist im Sinne der Datenkonsistenz insofern unproblematisch, als dass Generators nicht preemptiv von außen unterbrochen werden können, sondern nur kooperativ die Kontrolle an ihre aufrufende Funktion zurückgeben können. Generators werden daher als „shallow coroutines“ bezeichnet. Weitere Details dazu finden sich z. B. in [Herman, 2011]

nisse ausgeleitet und bei Wiederaufnahme Werte eingeleitet werden können.

Diese Schritte lassen sich auch für die Ausführung von asynchronen Funktionen nutzen: Ein Generator ruft eine asynchrone Funktion auf, pausiert bis zu deren Abschluss und wird mit dem Ergebnis des asynchronen Aufrufs fortgesetzt.

PROMISES IN GENERATOR FUNCTIONS Ein erstes Beispiel soll zeigen, wie sich eine asynchrone Funktion die eine Promise zurückliefert, aus einem Generator heraus aufrufen lässt.

```
//Example of async call from within a generator
//taken from Simpson, Kyle, Async & performance, p. 97

function foo(x, y) {
    //request() is some async function returning a promise
    return request("http://some.url.1/?x=" + x + "&y=" + y);
}

function* main() {
    try {
        var text = yield foo(11, 31);
        console.log(text);
    } catch (err) {
        console.error(err);
    }
}

var it = main(); //get the iterator object
var p = it.next().value; //get out the first yielded value

// wait for the 'p' promise to resolve
p.then(
    function (text) { //fulfillment handler
        it.next(text); //resume generator with promise value
    },
    function (err) { //rejection handler
        it.throw(err); //throw the error coming from async foo
            into generator
    }
);
```

Listing 7: Generator zum Aufruf einer asynchronen Funktion und zur Auswertung des Ergebnisses (vgl. [Simpson, 2015, S. 97])

Im Beispiel in Listing 7 wird ein Generator definiert, der eine asynchrone Funktion aufruft und dann pausiert. Die Promise, die der asynchrone Aufruf liefert, wird per yield ausgeleitet. Die Ergebnis-Handler werden außerhalb des Generators registriert. Sie speisen den Wert der Promise entweder als einfachen Wert oder aber als Exception zurück in den Generator, der das Ergebnis weiter verarbeitet.

Obwohl der gesamte Ablauf asynchron und nicht blockierend ist, wird innerhalb des Generators der Ergebniswert der Promise einer Variablen zugewiesen, als ob es sich um einen synchronen Aufruf handeln würde. Auch die eventuell auftretende Fehler können mit dem try-catch-Mechanismus behandelt werden, obwohl es sich bei foo() um eine asynchrone Funktion handelt. Möglich wird das, weil die vom asynchronen Aufruf gelieferte Promise außerhalb des Generators aufgelöst wird.

An diesem Beispiel wird deutlich, dass sich Generators grundsätzlich dafür eignen, asynchronen Code ohne den Verlust seiner vorteilhaften

asynchronen Eigenschaften so umzuschreiben, dass er genauso leicht lesbar wird wie synchroner Code.

RUNNER UTILITY Natürlich ist es nicht sinnvoll einen einzelnen asynchronen Aufruf, wie in Listing 7, in einen Generator zu verpacken und externe Funktionalität hinzuzufügen, um die Auflösung der asynchronen Aufrufe zu verarbeiten und wieder in den Generator einzuspeisen. Im angegebenen Beispiel ist genau diese Auflösung der von foo() gelieferten Promise außerhalb des Generators explizit ausprogrammiert, so dass der Code zunächst eher schwerer verständlich wird als einfacher.

Das Muster zur Auflösung einer Promise und zur Rückspeisung des Ergebnisses in den Generator ist allgemeingültig. Es lässt sich einmal generisch programmieren und wiederverwenden. Beliebig komplexe asynchrone Sequenzen innerhalb von Generator-Funktionen lassen sich dann in einer einfachen synchron anmutenden Weise zu formulieren.

Eine Hilfsfunktion, die genau das leistet, soll hier kurz vorgestellt werden:

```
//Sample Async Wrapper utility to run Generators with async
    calls
//taken from Parker, Daniel, JavaScript with promises, p. 77

function async(generator) {
    return function () {
        var iterator = generator.apply(this, arguments);

        function handle(result) {
            if (result.done) return
                Promise.resolve(result.value);
            return Promise.resolve(result.value)
                .then(function (res) {
                    return handle(iterator.next(res));
                }, function (err) {
                    return handle(iterator.throw(err));
                });
        }

        try {
            return handle(iterator.next());
        } catch (ex) {
            return Promise.reject(ex);
        }
    };
}

//add an export statement to use it as module
module.exports = async;
```

Listing 8: Hilfsfunktion zur Ausführung eines Generators mit asynchronen Aufrufen. (Code aus [Parker, 2015, Example 6-24, p. 77])

Über die Hilfsfunktion async in Listing 8 kann ein Generator ausgeführt werden, der an jedem yield Statement eine Promise liefert, die durch den internen Aufruf einer asynchronen Funktion erzeugt wird. Die Hilfsfunktion behandelt die Auflösung der nacheinander gelieferten Promises: Bei *fulfillment* der Promise wird der Wert wieder in den Generator eingespeist. Bei *rejection* dagegen wird eine Exception in den Generator geworfen, der die Chance hat den Fehler intern zu behandeln. Wenn der Generator erschöpft ist, liefert die Hilfsfunktion ihrerseits eine Promise an die aufrufende Umgebung, welche das finale Ergebnis der

im Generator kodierten Sequenz enthält. Falls der Generator erfolgreich abläuft, wird diese Promise *fulfilled*, andernfalls *rejected*.

Durch Einsatz einer solchen Hilfsfunktion lassen sich beliebig komplexe Sequenzen asynchroner Aufrufe innerhalb eines Generators in synchronem Programmierstil aufschreiben. Innerhalb des Generators lässt sich der eigentlich synchrone try-catch-finally-Mechanismus zur Fehlerbehandlung verwenden. Von außen muss lediglich ein einziger Aufruf der Hilfsfunktion erfolgen, die sich selber verhält wie ein einzelner asynchroner Aufruf einer Funktion, deren Endergebnis innerhalb einer Promise gekapselt ist.[26]

GENERATORS FÜR ASYNCHRONE SEQUENZEN Abschließend sei noch in einem Beispiel gezeigt, wie sich die eben eingeführten Generators zusammen mit der vorgestellten Hilfsfunktion zur Abarbeitung einer etwas komplexeren Sequenz von asynchronen Funktionsaufrufen einsetzen lassen:

```
//Example of async sequence using a Generator

runAsync = require("./asyncRunner.js");

function asyncFoo(name, delay) {
    return new Promise(function (resolve, reject) {
        console.log("\t asyncFoo('" + name + "', " + delay + ")
            just started async processing.");
        setTimeout(function (name) {
            resolve(name.toUpperCase());
        }, delay, name);
    });
}

function* generator() {
    try {
        var first = yield asyncFoo("This ", 500);

        //two async functions starting concurrently
        var promiseB = asyncFoo("really ", 500);
        var promiseA = asyncFoo("is ", 1000);

        var secondA = yield promiseA;
        var secondB = yield promiseB;
        var third = yield asyncFoo("cool!", 1000);

        return ([first, secondA, secondB, third]);
    } catch (error) {
        throw error;
    } finally {
        console.log("*** Generator cleanup. ***")
    }
}

runAsync(generator)()
    .then(result => console.log("Generator finished
        successfully: " + result))
    .catch(err => console.log("Ooops, error occured: " + err))

console.log("Main program finished. The rest is async.")
```

Listing 9: Benutzung von Generators zur Abarbeitung einer komplexen Sequenz aus asynchronen Funktionsaufrufen

26 Die gezeigte Hilfsfunktion lässt einige wichtige Aspekte außer acht. Es lassen sich z. B. keine Generators schachteln. Für den praktischen Einsatz empfiehlt es sich, auf eine der vielen frei erhältlichen Bibliotheksfunktionen zurückzugreifen, welche auch diverse Spezialfälle abdecken. Eine weit verbreitete Bibliothek ist das von TJ Holowaychuk geschriebene co. (Vgl. [Holowaychuk, 2013])

In diesem Beispiel wird ein Generator definiert, der insgesamt vier asynchrone Funktionsaufrufe tätigt. An jedem yield leitet der Aufruf-Generator die Promises der asynchronen Aufrufe aus und pausiert. Er wartet darauf, dass die Hilfsfunktion ihm das Ergebnis des asynchronen Aufrufs wieder einspeist, das dann direkt und synchron in einer Variable gespeichert werden kann.

Das Ergebnis des Programmlaufs ist in Listing 10 zu sehen.

```
//  asyncFoo('This ', 500) just started async processing.
//Main program finished execution. The rest is async.
//  asyncFoo('really ', 500) just started async processing.
//  asyncFoo('is ', 1000) just started async processing.
//  asyncFoo('cool!', 1000) just started async processing.
//*** Generator cleanup. ***
//Generator finished successfully: THIS ,IS ,REALLY ,COOL!
```

Listing 10: Ausgabelog für Listing 9

Die Aufrufe first und third werden direkt rechts neben dem yield abgesetzt. Die von der asynchronen Funktion zurückgegebene Promise wird von der Hilfsfunktion ausgewertet. Ihr Wert wird, nachdem sie *settled* ist, wieder in den Generator eingespeist, der diesen dann synchron in eigenen Variablen verwenden kann.

Eine Besonderheit bilden die Aufrufe promiseA und promiseB. Hier werden die zurückgegebenen Promises nicht direkt an die Hilfsfunktion ausgeleitet und der Generator pausiert, sondern sie werden überlappend (concurrently) abgesetzt, so dass sie parallel ablaufen können. Der Aufruf third erfolgt erst, nachdem die beiden asynchronen Aufrufe promiseA und promiseB ein Ergebnis geliefert haben. Welches dieser beiden Ergebnisse zuerst vorliegt, spielt für den Ablauf keine Rolle.

In Grafik 4 sind die Abhängigkeiten der vier asynchronen Aufrufe dargestellt.

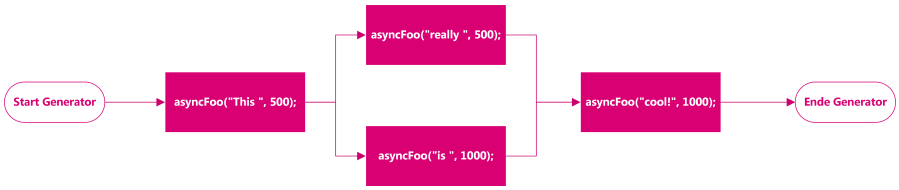

Abbildung 4: Die Abhängigkeiten der asynchronen Aufrufe im Generator aus Listing 9

.

In diesem Beispiel wird deutlich, dass man unter Zuhilfenahme von Generators asynchronen Code sehr übersichtlich strukturieren kann. Er lässt sich genauso einfach aufschreiben und verstehen wie synchroner Code, ohne dabei auf die Vorteile der asynchronen Programmierung zu verzichten. Sogar die Fehlerbehandlung lässt sich innerhalb des Generators über den eigentlich synchronen try-catch-finally-Mechanismus realisieren.

EXKURS: ASYNC-AWAIT IN KÜNFTIGEN VERSIONEN VON ECMASCRIPT Die durch die vorgestellte Benutzung von Generators für

asynchronen Code erreichten Verbesserungen werden aktuell in der JavaScript Community als so wichtig erachtet, dass dieses Konzept in kommenden Versionen des ECMAScript Sprachstandards noch erweitert werden soll. Konkret sollen neue Schlüsselwörter async und await eingeführt werden, welche die Definition von speziellen Funktionen erlauben, die ähnlich ablaufen wie die oben gezeigten Generators mit externer Hilfsfunktion.

In Zukunft wird es möglich sein, solche Funktionen direkt zu definieren, ohne einen Generator von einer Hilfsfunktion ausführen zu lassen. Die Sequenz von asynchronen Aufrufen lässt sich dann in eine async-Funktion schreiben, die direkt aufgerufen werden kann.

```
//Example of using the upcoming async-await syntax
//runs on node 7.1 and later using the --harmony flag

function asyncFoo(name, delay) {
    return new Promise(function (resolve, reject) {
        console.log("\t asyncFoo('" + name + "', " + delay + ")
            just started async processing.");
        setTimeout(function (name) {
            resolve(name.toUpperCase());
        }, delay, name);
    });
}

async function sequence() { //direct declaration of async
    function, no runner needed
    try {
        var first = await asyncFoo("This ", 500);

        //two async functions starting concurrently
        var promiseB = asyncFoo("really ", 500);
        var promiseA = asyncFoo("is ", 1000);

        var secondA = await promiseA;
        var secondB = await promiseB;
        var third = await asyncFoo("async!", 1000);

        return ([first, secondA, secondB, third]);
    } catch (error) {
        throw error;
    } finally {
        console.log("*** Async Sequence Cleanup. ***")
    }
}

sequence() //direct call of async function returning a promise
    .then(result => console.log("Generator finished
        successfully: " + result))
    .catch(err => console.log("Ooops, error occured: " + err))

console.log("Main program finished. The rest is async.")
```

Listing 11: Die gleiche Sequenz wie in Listing 9, jedoch mit async-await anstelle einer Hilfsfunktion zur Ausführung

Diese neue Syntax hat es zwar noch nicht in die neueste Version ECMAScript 7 (vgl. [Ecma TC39 und Terlson, 2016]) geschafft, jedoch kann diese neue Syntax in bestimmten Laufzeitumgebungen wie z. B. Node.js ab Version 7.1 schon ausgeführt werden.[27]

27 Einen guten Überblick über die in Node.js unterstützten Features liefert laufend aktualisiert http://node.green.

6 BEWERTUNG DER NEUEN SPRACHMITTEL

Aufgrund der großen Bedeutung asynchroner Programmierung in JavaScript sind die beiden vorgestellten Sprachmittel der Promises und Generators ein wichtiger Schritt für die Sprachentwicklung. Sie ermöglichen es, deutlich besser lesbaren und damit besser wartbaren Code zu schreiben als dies mit traditionellen Callbacks der Fall war. Auch werden die Probleme gelöst, die durch das Herausreichen der eigenen Programm-Continuation an fremden Code hervorgerufen wurden.

Durch diesen offensichtlichen Zugewinn an Sicherheit und die einfachere Anwendung kann die Einführung dieser modernen Methoden in ECMAScript 6 rundweg positiv beurteilt werden. Sie sind ein wichtiger Baustein weiterhin komplexe JavaScript Anwendungen sowohl auf dem Server als auch im Client zu realisieren.

Leider gibt es aber immer noch eine große installierte Basis an JavaScript Laufzeitumgebungen, die diese modernen Methoden nicht unterstützen.[28]

Ist es trotzdem sinnvoll auf die neuen Sprachstandards und Methoden zu setzen?

PROMISES Für im Browser laufende Webanwendungen ist es wichtig, dass auch ältere Browser unterstützt werden um ein breites Publikum zu erreichen. Hier hat die Programmiererin keine Kontrolle über die Laufzeitumgebung und kann sich daher nicht auf die Kompatibilität zum ECMAScript 6 Standard verlassen.

Trotzdem spricht nichts gegen die Verwendung von Promises: Promises können sehr leicht als Polyfill in älteren Laufzeitumgebungen von JavaScript nachgerüstet werden. Damit ist es möglich die inzwischen standardisierten Features auch dort zu nutzen und von den Verbesserungen zu profitieren. Es muss lediglich eine passende Polyfill-Bibliothek mit ausgeliefert werden.

Sogar ältere APIs, welche mit Callbacks nach dem Error-First-Protokoll und einem trailing Callback arbeiten, lassen sich leicht per promisify() in eine äquivalente API umsetzen, die Promises statt Callbacks nutzt.[29]

GENERATORS Mit Generators wurde eine neue Syntax in JavaScript eingeführt, die sich nicht einfach über Bibliotheksfunktionen nachrüsten lässt.

Wenn man die Laufzeitumgebung unter Kontrolle hat, wie es bei serverseitigen Applikationen üblicherweise der Fall ist, so kann das Konzept der Generators für asynchrone Control-Flows auf jeden Fall genutzt werden.

Der Einsatz von Generators in Webanwendungen sollte dagegen für den Einzelfall abgewogen werden: Programme, die Generators verwen-

28 Stellvertretend sei hier der Microsoft Internet Explorer genannt. Zur Kompatibilität von Laufzeitumgebungen zu verschiedenen Versionen des Standards siehe https://kangax.github.io/compat-table/es6/

29 Details dazu finden sich im Anhang A.1.

den und auch in älteren Laufzeitumgebungen vor ECMAScript 6 laufen sollen, müssen vorher in kompatiblen Code transpiliert werden.[30]

Es ist im Einzelfall zu prüfen, ob der Gewinn durch die Verwendung von Generators den zusätzlichen Build-Step rechtfertigt. Wenn aber vor der Veröffentlichung einer Webanwendung sowieso diverse Build-Steps notwendig sind, ist es in der Regel kein Problem, einen solchen Transpile-Step einzufügen. Der maschinell erzeugte Code ist aber nicht mehr gut lesbar und daher schwerer zu debuggen. Wenn sowieso ein Transpile-Schritt eingefügt wird, dann sollte man gleich den Schritt zu async-await gehen, da der Code dadurch noch prägnanter wird.

Fazit

Schlecht lesbarer Code ist im Umfeld von asynchroner Programmierung häufig anzutreffen. Die „Callback Hell“ ist allgegenwärtig.

Durch die neuen Sprachmittel von ECMAScript 6 wurden jedoch Möglichkeiten geschaffen, auch asynchronen Code lesbar aufzuschreiben und vertrauenswürdig auszuführen. Dabei muss nicht auf die Vorteile der asynchronen Programmierung verzichtet werden.

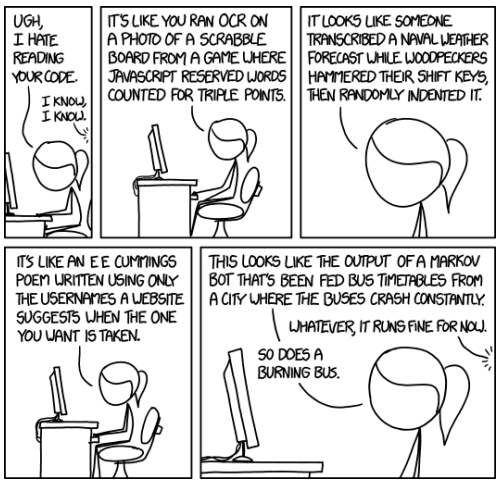

Abbildung 5: xkcd: Code Quality 2 (aus [Munroe, 02.12.2016])

Die vorgestellten modernen Methoden, die ECMAScript 6 für die asynchrone Programmierung bietet, sind daher ein wichtiger Meilenstein, um große Applikationen sicher und effizient zu entwickeln und zu warten.

Es gibt spätestens seit der Einführung des ECMAScript 6 Standards keine Entschuldigung mehr für asynchronen Spaghetti-Code, wie dies noch 2007 in „Web Applications – Spaghetti Code for the 21st Century“ [Mikkonen und Taivalsaari, 2008] beklagt wurde. JavaScript bietet heutzutage alle Möglichkeiten, gut strukturierten, effizienten asynchronen Code zu schreiben. Es ist an den Programmiererinnen diese Möglichkeiten zu nutzen und dazu beizutragen, dass JavaScript sein traditionell schlechtes Image ablegen kann.

30 Unter Transpilation versteht man die Übersetzung von einer Quellsprache in eine andere Quellsprache. In diesem Fall also von ECMAScript 6 in ECMAScript 5, welches von allen aktuellen Browsern unterstützt wird. (vgl. [Sengstacke, 2016])

7 LITERATUR

[Buckler 2013] Buckler, Craig: *JavaScript Comes of Age - SitePoint*. 12 2013. – URL https://www.sitepoint.com/javascript-comes-age/. – Zugriffsdatum: 2016-11-24

[Crockford 2008] Crockford, Douglas: *JavaScript: The World's Most Misunderstood Programming Language Has Become the World's Most Popular Programming Language*. 2008. – URL http://javascript.crockford.com/popular.html. – Zugriffsdatum: 2016-11-26

[Dahl 2009] Dahl, Ryan: *Original Node.js presentation*. 11 2009. – URL https://www.youtube.com/watch?v=ztspvPYybIY. – Zugriffsdatum: 2016-11-25

[Dean 2007] Dean, Jeff: *Software Engineering Advice from Building Large-Scale Distributed Systems*. 2007. – URL research.google.com/people/jeff/stanford-295-talk.pdf

[Ecma TC39 und Steele 1997] Ecma TC39 ; Steele, Guy L. [. ; Ecma International (Hrsg.): *ECMA-262, 1st edition, June 1997*. 1997. – URL http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%201st%20edition,%20June%201997.pdf. – Zugriffsdatum: 2016-12-26

[Ecma TC39 und Terlson 2016] Ecma TC39 ; Terlson, Brian ; Ecma International (Hrsg.): *ECMAScript 2016 Language Specification*. 2016. – URL https://www.ecma-international.org/ecma-262/7.0/. – Zugriffsdatum: 2016-12-26

[Ecma TC39 und Wirfs-Brock 2015] Ecma TC39 ; Wirfs-Brock, Allen ; Ecma International (Hrsg.): *ECMAScript 2015 Language Specification*. 2015. – URL http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf. – Zugriffsdatum: 2016-12-26

[Ecma TC39 committee 27.09.2016] Ecma TC39 committee: *The TC39 Process*. 27.09.2016. – URL https://tc39.github.io/process-document/. – Zugriffsdatum: 2016-12-22

[FernUni Hagen 2016] FernUniversität in Hagen (Hrsg.): *Modulhandbuch Bachelor Informatik*. 2016. – URL http://www.fernuni-hagen.de/imperia/md/content/fakultaetfuermathematikundinformatik/studiengaenge/bachelorinformatik/modulhandbuch.pdf

[Friedman und Wise 1976] Friedman, Daniel P. ; Wise, David S.: THE IMPACT OF APPLICATIVE PROGRAMMING ON MULTIPROCESSING. In: *Proceedings of the International Conference on Parallel Processing* Bd. IEEE Cat. No. 76CH1127–OC, 1976, S. 263–272

[Gallaba u. a. 2015] Gallaba, Keheliya ; Mesbah, Ali ; Beschastnikh, Ivan: Don't Call Us, We'll Call You: Characterizing Callbacks

in Javascript. In: Institute of Electrical and Electronics Engineers (Hrsg.): *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. Piscataway, NJ : IEEE, 2015, S. 1–10. – ISBN 9781467379007

[Ganzinger u. a. 2011] Ganzinger, Harald ; Six, Hans-Werner ; Voss, Josef ; Beierle, Christoph: *Logisches und funktionales Programmieren: Kurseinheit 3: Funktionales Programmieren (1. Teil)*. Hagen, 2011

[Harrington 2015] Harrington, Chris: *Clean promise chains using a pipeline*. 6 2015. – URL http://dapperdeveloper.com/2015/06/09/clean-promise-chains-using-a-pipeline/. – Zugriffsdatum: 2016-12-29

[Herman 2011] Herman, Dave: *Why coroutines won't work on the web*. 12 2011. – URL http://calculist.org/blog/2011/12/14/why-coroutines-wont-work-on-the-web/. – Zugriffsdatum: 2016-11-25

[Holowaychuk 2013] Holowaychuk, T. J.: *co: generator async control flow goodness*. 2013. – URL https://www.npmjs.com/package/co. – Zugriffsdatum: 2016-12-03

[Joyent] Joyent: *Joyent | Error Handling*. – URL https://www.joyent.com/node-js/production/design/errors. – Zugriffsdatum: 2016-11-26

[Kennedy Kambona u. a. 2013] Kennedy Kambona ; Elisa Gonzalez Boix ; Wolfgang De Meuter: *An Evaluation of Reactive Programming and Promises for Structuring Collaborative Web Applications: Proceedings of the 7th Workshop on Dynamic Languages and Applications*. New York, NY : ACM, 2013. – URL http://dl.acm.org/citation.cfm?id=2489798. – ISBN 9781450320412

[Liskov und Shrira 1988] Liskov, Barbara ; Shrira, Liuba: Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In: *ACM SIGPLAN Notices* 23 (1988), Nr. 7, S. 260–267. – ISSN 03621340

[Mikkonen und Taivalsaari 2008] Mikkonen, Tommi ; Taivalsaari, Antero: Web Applications -Spaghetti Code for the 21st Century. In: Dosch, Walter (Hrsg.): *2008 Sixth International Conference on Software Engineering Research, Management and Applications // Sixth International Conference on Software Engineering Research, Management and Applications, 2008*. Piscataway, NJ : IEEE, 2008, S. 319–328. – ISBN 9780769533025

[Mozilla Developer Network 03.11.2016] Mozilla Developer Network: *Concurrency model and Event Loop*. 03.11.2016. – URL https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop. – Zugriffsdatum: 2016-11-24

[Munroe 02.12.2016] Munroe, Randall: *xkcd: Code Quality 2*. 02.12.2016. – URL http://xkcd.com/1695/. – Zugriffsdatum: 2016-12-03

[Netscape 1996] Netscape: *INDUSTRY LEADERS TO ADVANCE STANDARDIZATION OF NETSCAPE'S JAVASCRIPT AT STANDARDS BODY MEETING: (Press Release)*. 11 1996. – URL https://web.archive.org/web/19981203070212/http://cgi.netscape.com/newsref/pr/newsrelease289.html. – Zugriffsdatum: 2016-12-26

[Netscape 1995] Netscape, Sun: *NETSCAPE AND SUN ANNOUNCE JAVASCRIPT, THE OPEN, CROSS-PLATFORM OBJECT SCRIPTING LANGUAGE FOR ENTERPRISE NETWORKS AND THE INTERNET: (Press Release)*. 12 1995. – URL https://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html. – Zugriffsdatum: 2016-11-24

[Parker 2015] Parker, Daniel: *JavaScript with promises*. First edition. Sebastopol, CA : O'Reilly Media, 2015. – URL http://proquest.tech.safaribooksonline.de/9781491930779. – ISBN 9781449373214

[Poli 2013] Poli, Polo-François: *Node.js & Event-driven programming*. 2013. – URL http://www.baloo.io/blog/2013/11/30/node-event-driven-programming/. – Zugriffsdatum: 2016-11-24

[Promises A+ 2016] Cavalier, Brian (Hrsg.) ; Denicola, Domenic (Hrsg.): *Promises/A+*. 2016. – URL https://promisesaplus.com/. – Zugriffsdatum: 2016-10-25

[Rauschmayer 2012] Rauschmayer, Axel: *Asynchronous programming and continuation-passing style in JavaScript*. 6 2012. – URL http://www.2ality.com/2012/06/continuation-passing-style.html. – Zugriffsdatum: 2016-11-27

[Rauschmayer 2016] Rauschmayer, Axel: *Tracking unhandled rejected Promises*. 4 2016. – URL http://www.2ality.com/2016/04/unhandled-rejections.html. – Zugriffsdatum: 2016-11-30

[Schlueter 2013] Schlueter, Isaac Z.: *Designing APIs for Asynchrony*. 8 2013. – URL http://blog.izs.me/post/59142742143/designing-apis-for-asynchrony. – Zugriffsdatum: 2016-11-25

[Sengstacke 2016] Sengstacke, Peleke: *JavaScript Transpilers: What They Are & Why We Need Them*. 4 2016. – URL https://scotch.io/tutorials/javascript-transpilers-what-they-are-why-we-need-them. – Zugriffsdatum: 2016-12-07

[Sharp 2010] Sharp, Remy: *What is a Polyfill?* 10 2010. – URL https://remysharp.com/2010/10/08/what-is-a-polyfill. – Zugriffsdatum: 2016-12-07

[Simpson 2015] Simpson, Kyle: *Async & performance*. First edition. Beijing and Cambridge and Farnham and Köln and Sebastopol and Tokyo : O'Reilly, 2015 (You don't know JS). – ISBN 9781491904220

[Steimann 2010] Steimann, Friedrich: *Objektorientierte Programmierung: Vorlesungsskript zum Kurs 1814*. Hagen, 2010

[Synodinos 2010] Synodinos, Dio: *Deep inside Node.js with Ryan Dahl*. 12 2010. – URL https://www.infoq.com/interviews/node-ryan-dahl. – Zugriffsdatum: 2016-11-26

[Taft 2016] Taft, Darryl K.: *JavaScript Most Popular Language: Stack Overflow Report*. 3 2016. – URL http://www.eweek.com/developer/javascript-most-popular-language-stack-overflow-report.html. – Zugriffsdatum: 2016-11-24

[TIOBE 2016] TIOBE Software BV (Hrsg.): *TIOBE Index for November 2016*. 2016. – URL http://www.tiobe.com/tiobe-index/. – Zugriffsdatum: 2016-11-24

A MUSTER DER ASYNCHRONEN PROGRAMMIERUNG IN ECMASCRIPT 6

In diesem Anhang sollen einige typische Muster der asynchronen Programmierung in ECMAScript 6 inklusive Code-Beispielen vorgestellt werden. Diese Beispiele sollen nicht der ausführlichen Diskussion der Konzepte dienen, sie sind vielmehr als Referenz und Anregung für eigene Projekte gedacht, in denen die neuen Sprachfeatures von ECMAScript 6 eingesetzt werden sollen. Für einen solchen Einsatz sind Anpassungen und Ergänzungen unumgänglich. Vor dem produktiven Einsatz sollte daher auf jeden Fall die umfangreich im Netz vorhandene Dokumentation herangezogen werden.

Auch bei der Programmierung in JavaScript gilt es den Bibliotheksgedanken zu leben und lieber etwas mehr Zeit in die Suche nach einer passenden Bibliotheksfunktion zu investieren als eine Funktion selber neu zu implementieren.

> Jede nicht selbst geschriebene Programmzeile ist ein Gewinn, jede Implementierung einer noch so kleinen Funktion, die es bereits in irgendeiner Bibliothek gibt, ist ein Verlust. [Steimann, 2010, S. 333]

Gerade bei der Programmierung mit Node.js wird der Programmiererin die Benutzung von ausgereiften und getesteten Bibliotheksfunktionen leicht gemacht, da es mit npm (siehe `https://www.npmjs.com/`) ein hervorragend gepflegtes und extrem umfangreiches Paketverwaltungssystem gibt. Eine Suche nach Bibliotheksfunktionen auf npm ist meistens von Erfolg gekrönt. Sollte es einmal kein passendes Paket dort geben, so ist die Programmiererin ermuntert, die eigene Implementierung dort für andere User zugänglich zu machen.

In den nächsten Abschnitten sollen Beispiele zu diesen Themen gegeben werden:

- Umwandlung traditioneller Callback-APIs in Funktionen, die eine Promise zurückgeben
- Error Handling in Promise Chains
- asynchrone Sequenzen mit verschiedenen Parametern
- asynchrone Funktionen mit Timeout
- Zusammensetzen komplexer Abläufe – „functional composition“

A.1 Anpassung älterer APIs an Promises

Obwohl die Benutzung von Promises sich für asynchrone Aufrufe immer mehr durchsetzt, sind in der JavaScript Welt noch sehr viele Bibliotheken verfügbar und gebräuchlich, die ihre Ergebnisse nach dem traditionellen Callback-Verfahren liefern. Um auch solche Funktionen mit den neuen Mechanismen nutzen zu können, muss die asynchrone Funktion, die einen Callback zur Rückgabe des Ergebnisses erwartet, in eine Funktion umgewandelt werden, die statt dessen eine Promise zurückliefert.

Dieser Vorgang wird häufig als „Promisification“ oder „Lifting“ bezeichnet. Durch den ECMAScript 6 Standard wird dieser Vorgang nicht

direkt spezifiziert, aber es existieren viele externe Bibliotheken die entsprechende Funktionen bieten. Diese können problemlos auch im Zusammenspiel mit ES6-Promises genutzt werden.[31]

Auch wenn es angesichts der ausreichend vorhandenen fertigen Lösungen nicht notwendig sein sollte ein solches Lifting selber zu implementieren, so soll doch der Vorgang an einem Stück Beispielcode gezeigt werden. Das Beispiel in Listing A.1 folgt weitgehend der Erklärung in [Simpson, 2015, S. 76].

```
//Polyfill to change callback APIs to promise APIs
//taken from Simpson, Kyle, Async & performance, p.76

// polyfill-safe guard check
if (!Promise.wrap) {
    Promise.wrap = function (fn) {
        return function () {
            var args = [].slice.call(arguments);
            return new Promise(function (resolve, reject) {
                fn.apply(null, args.concat(function (err, v) {
                    if (err) {
                        reject(err);
                    } else {
                        resolve(v);
                    }
                }));
            });
        };
    };
}
```

Listing A.1: Wrapper Funktion zur Umwandlung von Callback APIs in Promise APIs (aus [Simpson, 2015, S. 76])

Der angegebene Code erweitert den Prototyp Promise um eine Methode wrap(), welche eine Funktion erwartet, die den Konventionen Error-First und Trailing-Callback folgt. Als Rückgabe erhält die Programmiererin eine neue Funktion, die keinen Callback mehr erwartet, sondern anstelle dessen eine Promise zurückliefert. Diese *settled* im Status *fullfilled* wenn der zugrundeliegende Aufruf erfolgreich war und im Status *rejected* wenn ein Fehler auftrat.

Die Anwendung des Wrappers wird in Listing A.2 gezeigt. Die Funktion foo ruft eine asynchrone Funktion auf, die einen Callback erwartet. Zunächst wird foo mit traditionellem Callback verwendet und dann mittels des vorgestellten Wrappers in eine Funktion fooPromise verwandelt und über ihr Promise-API verwendet.

```
//Example inspired by
//Simpson, Kyle, Async & performance, p. 76ff

//async function with callback API
function foo(x, y, cb) {
    ajax("http://some.url.1/?x=" + x + "&y=" + y, cb);
}

//using foo() with callback API
foo(11, 31, function (err, text) {
    if (err) {
```

31 Siehe z. B. das Paket https://www.npmjs.com/package/es6-promisify oder die Funktionen in der Bluebird Promise-Library http://bluebirdjs.com/docs/api/promisification.html. Eine Suche nach dem Stichwort „promisify“ auf npm liefert aktuell 380 Ergebnisse (https://www.npmjs.com/search?q=promisify).

```
        console.error(err);
    } else {
        console.log(text);
    }
});

//promisification of foo
var fooPromise = Promise.wrap(foo);
//using fooPromise() with promise API
fooPromise(11, 31)
    .then(text => console.log(text))
    .catch(error => console.error(err))
```

Listing A.2: Verwendung des Promisification Wrappers

.

A.2 Error Handling in Promise Chains

Um die vielfältigen Möglichkeiten der struktutrierten Fehlerbehandlung bei der Verwendung von Promises zu verdeutlichen, soll in diesem Abschnitt ein etwas umfangreicheres Beispiel angegeben werden. Dabei werden verschiedenen Methoden zur Fehlerbehandlung gezeigt, die in der Praxis vielfältig kombiniert werden.

Ein Rejection-Handler muss nicht zwangsläufig am Ende einer Promise Chain benutzt werden, sondern kann auch in der Mitte einer Kette installiert werden, um zum Beispiel detailliertere Log-Dateien zu schreiben oder frühzeitig nicht mehr benötigte Resourcen freizugeben.

Dabei ist jedoch zu beachten, dass die vom Rejection-Handler zurückgegebene Promise nicht automatisch den Status *rejected* annimmt. Mit einem in der Mitte der Promise Chain eingehängten Rejection-Handler kann also die weitere Abarbeitung wieder aufgenommen werden. Wenn man die Abarbeitung nach einem Fehler jedoch endgültig abbrechen möchte, so muss man aus dem Rejection-Handler wieder eine *rejected* Promise zurückgeben. Dazu ist es häufig zweckmäßig, den empfangenen Rejection Reason wieder als Exception zu werfen und damit die vom Rejection-Handler zurückgegebene Promise ihrerseits zurück zu weisen.

```
//multiple Rejection Handlers within the Promise chain
//the called functions are promise returning asynch functions


var p = foo();

p.then(result => {
    return bar(result);
}).catch(e => {
    console.log("error in bar: ", e);
    return defaultBarValue; //gently return a default value
}).then(result => {
    return baz(result);
}).catch(e => {
    console.log("error in baz: ", e);
    unBaz(); //do some cleanup for failed baz()
    throw e; //rethrow error to end chain
}).then(result => {
    return booze(result);
}).then(result => {
    console.log("Chain ended with success: ", result)
}).catch(e => {
    //will only get called when baz or booze caused the error
    //failing bar gently returned a default value to continue
```

```
25      console.log("chain got stuck with error:", e);
})
```

Listing A.3: Mehrere Rejection-Handler inmitten einer Promise-Chain

Am Beispiel in Listing A.3 werden die verschiedenen Möglichkeiten verdeutlicht: Für eine normale Ausführung sollen die asynchronen Funktionen bar, baz und booze jeweils mit dem Resultat ihrer Vorgängerfunktion aufgerufen werden. Es wird angenommen, dass bei einem Fehlschlag der Funktion bar mit einem default Wert weiter gearbeitet werden kann. Bei Fehlschlagen von baz jedoch soll die Abarbeitung der Kette abgebrochen werden, jedoch muss in diesem Fall aufgeräumt werden. Fehlschlagen von booze erfordert keine spezielle Fehlerreaktion.

Es wird also hinter bar ein Rejection-Handler in die Promise-Chain eingehängt. Dieser liefert bei Fehlschlag eine *fulfilled* Promise mit default Wert. Der nächste Rejection-Handler hinter baz muss neben dem Aufräumen auch dafür sorgen, dass die Kette nicht weiter abgearbeitet wird. Daher wird der Eingangsfehler aus dem Aufruf von baz in die Rückgabepromise des Rejection-Handlers verpackt. Diese *rejected* Promise wird nun bis zum finalen Rejection-Handler durch die Kette gereicht.

Die geneigte Leserin mag sich selber überlegen, wie die beschriebene Logik mittels konventioneller Callbacks zu formulieren wäre.

A.3 Asynchrone Sequenz mit verschiedenen Parametern

Zuweilen kommt es vor, dass eine bestimmte Sequenz aus asynchronen Aufrufen nacheinander abgearbeitet werden soll. Die auszuführende asynchrone Funktion ist dabei immer die gleiche, allerdings sind nacheinander unterschiedliche Parameter an diese zu übergeben. Dieses Muster kommt relativ häufig beim Schreiben von Tests vor. Die zu testende Funktion soll nacheinander mit verschiedenen Parametern aufgerufen werden. Bei einem auftretenden Fehler soll die gesamte Aufrufkette abgebrochen werden.

Im folgenden Beispiel soll die Funktion getestet werden, die eine Web-API zum Hinzufügen weiterer Teilnehmer zur Verfügung stellt. Je nachdem, welche Vorgeschichte der aktuelle Testfall hat, soll die zu testende Funktion entweder den Status 200 („OK“) oder den Status 409 („Conflict“) zurück melden. Alle Ergebnisse der sequentiellen Aufrufe sollen bei Erfolg in einem Array zur Verfügung stehen.

Es wäre einfach den Testfall manuell als Promise-Chain zu formulieren. Übersichtlicher ist es jedoch, sich eine kleine Hilfsfunktion zu schreiben, welche die Aufrufsequenz übernimmt und die Parameter aus einer generischen Kollektion entnimmt.

Eine solche Hilfsfunktion ist in Listing A.4 dargestellt:

```
//code for sequence() taken from
//Parker, Daniel, JavaScript with promises, p. 33

/**
5 * executes the callback with the arguments given in the
     paramArray;
```

```
 * The callback function shall return a Promise.
 * The callback functions with arguments are executed serially.
 * @param paramArray: array of data to be passed into the
     callback function as parameter
 * @param callback: The function to be executed in series.
     Shall return a promise.
 * @returns {*}: an empty resolved promise in case of success;
     a rejected promise in case of error;
 */
function sequence(paramArray, callback) {
    var resultArray = [];

    function chain(ops, index) {
        if (index == ops.length) return
            Promise.resolve(resultArray);
        return
            Promise.resolve(callback(ops[index])).then(function
            (result) {
            resultArray.push(result);
            return chain(ops, index + 1);
        })
    }

    return chain(paramArray, 0);
}

module.exports = sequence;
```

Listing A.4: Hilfsfunktion sequence zum sequentiellen Aufruf einer asynchronen Funktion mit verschiedenen Parametern. (aus [Parker, 2015, S. 33])

Die interne Hilfsfunktion chain() wird beginnend mit dem ersten Parameter im .then() der vorherigen Promise immer wieder rekursiv aufgerufen, bis das Parameter-Array erschöpft ist. Dann werden die Ergebnisse der einzelnen Aufrufe als Array in einer Promise verpackt zurückgegeben.

Interessant bei diesem Muster ist der rekursive Aufruf der chain()-Funktion: Trotz der Rekursion kann hier kein Stack-Overflow auftreten, da die rekursiven Aufrufe nicht direkt erfolgen, sondern asynchron in die Queue des Laufzeitsystems eingereiht werden. Jeder neue Aufruf startet daher wieder mit einem leeren Stack.

Im folgenden Listing A.5 wird ein Testfall eines Unit-Tests simuliert, der sich die vorgestellte sequence()-Funktion zu nutze macht, um nacheinander mehrere Testfälle auf der Funktion addParticipantPost auszuführen.

```
//example of using the generic sequence to perform module tests
    on
//the promise returning addParticipantPost() function

var sequence = require('./promiseSequence.js');
var utils = require("./promiseSequenceData.js")


//define the parameters to apply in sequence
var ops = [
    {participant: utils.testParticipants[0], returnCode: 200},
    {participant: utils.testParticipants[1], returnCode: 200},
    {participant: utils.testParticipants[1], returnCode: 409},
    {participant: utils.testParticipants[2], returnCode: 200},
    {participant: utils.testParticipants[2], returnCode: 409},
    {participant: utils.testParticipants[3], returnCode: 200},
    {participant: utils.testParticipants[4], returnCode: 200}
];
```

```
//perform the sequence with different parameters
sequence(ops, utils.addParticipantPost)
    .then(function (result) {
        console.log(result);
        console.log("Success!");
        done(); //everything is OK and the test is passed
    })
    .catch(function (err) {
        console.error(err);
        done(err); //an error occured and is propagated to the
            testing framework
    });

//Called with participant: [object Object], returnCode: 200
//Called with participant: [object Object], returnCode: 200
//Called with participant: [object Object], returnCode: 409
//Called with participant: [object Object], returnCode: 200
//Called with participant: [object Object], returnCode: 409
//Called with participant: [object Object], returnCode: 200
//Called with participant: [object Object], returnCode: 200
//[ 200, 200, 409, 200, 409, 200, 200 ]
//Success!
```

Listing A.5: Simulierter Testfall unter Verwendung von sequence.

A.4 Asynchrone Funktionen mit Timeout

Häufig ist es sinnvoll den Aufruf von asynchronen Funktionen mit einem Timeout zu versehen. Dadurch kann garantiert werden, dass nach Ablauf der festgelegten Zeit auch tatsächlich ein Ergebnis vorliegt. Dieses kann dann zwar auch ein Fehler oder ein Default-Wert sein, jedoch ist es garantiert vorhanden und die eigene Applikation kann geeignete Maßnahmen ergreifen.

Mit Promise.race() lässt sich sehr einfach eine Hilfsfunktion erstellen, die einen asynchronen Funktionsaufruf mit einem Timeout versieht. Optional kann ein Default-Wert angegeben werden. Eine Möglichkeit einer solchen Hilfsfunktion ist in Listing A.6 angegeben.

```
module.exports = function timeoutPromise(promise, delay,
    defaultVal) {
    var promiseTimer = function (delay, defaultValue) {
        return new Promise(function (resolve, reject) {
            setTimeout(function () {
                if (defaultValue) resolve(defaultValue);
                else reject(new Error("Promise timed out"))
            }, delay)
        })
    }

    return Promise.race([promise, promiseTimer(delay,
        defaultVal)]);
};
```

Listing A.6: Hilfsfunktion, um einen asynchronen Aufruf mit einem Timeout und einem optionalen Default-Wert zu versehen

Der Funktion wird als erster Parameter die Promise aus einem asynchronen Aufruf übergeben. Im zweiten Parameter wird ein Timeout in Millisekunden spezifiziert. Der dritte Parameter ist optional und gibt einen Default-Wert an. Wenn die übergebene Promise nach der angegebenen Zeit noch nicht *settled* ist, dann wird die resultierende Promise

entweder mit einem timeout-Fehler *rejected* oder aber mit dem angegebenen Default-Wert *fulfilled*. Ein Anwendungsbeispiel ist in Listing A.7 zu sehen.

```
var timeoutPromise = require('./timeoutPromise.js');

//simulated long lasting async function
var asyncDelay = function (delay) {
    return new Promise(function (resolve) {
        setTimeout(function () {
            resolve("Promise resolved after " + delay + " ms.");
        }, delay);
    });
}

var p100 = timeoutPromise(asyncDelay(100), 200);
var p500 = timeoutPromise(asyncDelay(500), 200);
var p1500 = timeoutPromise(asyncDelay(1500), 200, "Default
    Value");

p100.then(console.log, console.error);
p500.then(console.log, console.error);
p1500.then(console.log, console.error);

//Promise resolved after 100 ms.
//Error: Promise timed out
//    at Timeout._onTimeout
    (xxxxx\snippets\timeoutPromise.js:6:29)
//    at ontimeout (timers.js:365:14)
//    at tryOnTimeout (timers.js:237:5)
//    at Timer.listOnTimeout (timers.js:207:5)
//Default Value
```

Listing A.7: Anwendung einer Promise mit Timeout und Default-Wert

Auch in diesem Fall ist es sinnvoll, eine solche Funktion nicht selber zu implementieren sondern geeignete Bibliotheksfunktionen einzubinden. In diesem Fall lohnt sich ein Blick auf https://www.npmjs.com/package/time-limit-promise.

A.5 Pipelining komplexer Abläufe – „functional composition“

Ein bekanntes Muster der Software-Architektur ist das „Pipe-and-Filter“ Muster, bei dem komplexe Aufgaben derart in kleinere Teilaufgaben zerlegt werden, dass diese Teilaufgaben einzeln einfach zu realisieren sind. Auf die Daten werden nacheinander einzelne Filter angewendet und das Ergebnis über eine Pipe an die nächste Filterstufe weiter transportiert.

Ganz ähnlich kann man sich eine Sequenz aus asynchronen und synchronen Funktionen vorstellen, die einen Eingangswert durch mehrere hintereinander angewendete Filterstufen in ein Endergebnis transformiert. Das Ergebnis einer Stufe ist jeweils das Eingangsdatum der nächsten Stufe. Dadurch lassen sich über die Komposition einfacher Funktionen komplexe Abläufe realisieren. Man spricht daher auch von „functional composition“.

Da Funktionen in JavaScript First-Class-Citizens sind, können sie leicht in einem Array gespeichert werden. Im folgenden Listing A.8 wird eine Hilfsfunktion vorgestellt, die in einem Array abgelegte Funktionen nacheinander aufruft und jeweils das Ergebnis der vorhergehenden Funktion als Eingangsparameter für die nachfolgende Funktion verwen-

det. Es wird eine Pipeline aufgebaut, um eine komplexe Funktion aus einfachen Bestandteilen zusammen zu setzen.

```
//example taken from
//http://dapperdeveloper.com/2015/06/09/
//  clean-promise-chains-using-a-pipeline/

module.exports = function(tasks) {
  this.go = function() {
    if (!tasks || tasks.length === 0)
      return Promise.resolve();

    var promise = tasks[0].apply(this, arguments);
        if (!promise.then)
            promise = Promise.resolve(promise);
    for (var i = 1; i < tasks.length; i++)
      promise = promise.then(tasks[i]);
    return promise;
  }
};
```

Listing A.8: Hilfsfunktion zum Pipelining (aus [Harrington, 2015])

Es wird ein Pipeline-Konstruktor exportiert, dem zur Erzeugung eines neuen Objekts ein Array aus Funktionen übergeben wird. Dieses Pipeline-Objekt hat eine Methode `go()`, welche ihre eigenen Parameter an die erste Funktion des Arrays übergibt und darauffolgend das Ergebnis eines jeden Funktionsaufrufs als Eingangsparameter des jeweils nächsten benutzt. Wenn das übergebene Funktionsarray erschöpft ist wird das Endergebnis zurückgegeben. Bei dem Array mit den einzelnen Funktionen ist es egal, ob diese asynchron eine Promise zurückgeben, oder direkt synchron ein Ergebnis. Alle Aufrufe werden über eine Promise-Chain asynchron entkoppelt.

Die Anwendung der vorgestellten Funktion wird im Listing A.9 anhand eines simplen Workflows demonstriert. Es lassen sich natürlich auch leicht sinnvollere Anwendungen dieses Musters finden.

```
//Example inspired and mainly taken from
//http://dapperdeveloper.com/2015/06/09/
//  clean-promise-chains-using-a-pipeline/

var Pipeline = require("./promisePipeline.js")

var tasks = [
    _multiplyBy5,
    _add.bind(null, 50),
    _wait.bind(null, 666),
    _convertToString,
    _addCurrentDate
]

new Pipeline(tasks).go(10)
    .then(function (result) {
        console.log("Success! " + result);
    })
    .catch(function (err) {
        console.error(err);
    });

function _multiplyBy5(number) {
    return number * 5;
}

function _add(summand1, summand2) {
    return summand1 + summand2;
}
```

```
30
function _convertToString(number) {
    return Promise.resolve(number.toString());
}

35 function _addCurrentDate(string) {
    return new Date().toString() + ": " + string;
}

function _wait(time, string) {
40     return new Promise(function (resolve) {
        setTimeout(function () {
            resolve(string);
        }, time);
    });
45 }

//Success! Thu Dec 29 2016 16:32:45 GMT+0100 (Mitteleuropaische
    Zeit): 100
```

Listing A.9: Anwendung einer Funktionspipeline zur Realisierung eines komplexen Workflows

Es soll nicht unerwähnt bleiben, dass es natürlich auch für das hier vorgestellte Muster schon vielfältige Realisierungen als Bibliotheksfunktionen gibt. Aus der Masse sticht hier die Realisierung mittels der Bibliothek „Ramda“ hervor, die sich auf die funktionale Programmierung in JavaScript spezialisiert hat und dadurch häufig sehr elegante Lösungen für ansonsten relativ komplizierte Abläufe bietet. Siehe hierzu: http://ramdajs.com/docs/#pipeP.