Chris Mair

Webanwendungen mit Node.js, Express und Websockets

Ein Webmasters Press Lernbuch

Version 2.2.4 vom 10.02.2022

Autorisiertes Curriculum für das Webmasters Europe Ausbildungs- und Zertifizierungsprogramm

— www.webmasters-europe.org —

Inhaltsverzeichnis

Vorwort

Wenn Sie diese Zeilen lesen, haben Sie den Einstieg in die Programmierung mit *JavaScript* hinter sich.

Sie manipulieren HTML wie ein Magier und der Webbrowser macht genau das, was Sie ihm sagen! Auch *Node.js* ist kein Fremdwort für Sie und den Sprung auf die Serverseite haben Sie auch schon gewagt — z.B. mit der Class »Einführung in Node.js« (*C. Mair 2017*).

Vielleicht sind Sie geradezu euphorisch und stellen sich schon vor, wie Sie den nächsten Hit im WWW landen?

Leider gibt es da noch einige Dinge zu entknoten:

- Wie ist eine Webanwendung eigentlich aufgebaut?
- Was hat es mit diesem AJAX auf sich?
- Was macht eine Single-Page-Application besonders und was ist REST?
- Was sind *WebSockets*?

Diese Class möchte Sie an die Hand nehmen und die Stufen vom frisch gekürten *JavaScript*-Programmierer zum Frontend-Entwickler so flach wie möglich halten. Die Class will auf keinen Fall ein Nachschlagewerk zu den verwendeten Technologien sein, sondern vielmehr ein Rundgang. Es soll das Bild von *JavaScript* abrunden, um Sie als Frontend-Programmierer bereit für die große weite Welt zu machen.

Viel Spaß beim weiteren Entknoten!

Chris Mair

1 Versprochen ist versprochen und wird nicht gebrochen

In dieser Lektion lernen Sie

- was Promises sind.
- wie man damit das tiefe Verschachteln von Callbacks vermeidet und übersichtlichen Code schreibt.

1.1 The Pyramid of Doom

Stellen Sie sich vor, Sie bekommen die Aufgabe, einen Newsticker zu programmieren. Der Newsticker soll eine Serie von Strings anzeigen, und zwar mit jeweils einer Pause von einer Sekunde zwischen jedem neuen String. Als Test-Case möchten Sie zunächst einfach einen Satz, Wort für Wort, in die Console ausgeben.

Für diesen Zweck bietet sich die *JavaScript*-Funktion `setTimeout(…)` an: Sie ruft ihr erstes Argument (eine ***Callback***-Funktion) asynchron auf, und zwar mit einer Verzögerung, die durch ihr zweites Argument definiert ist (ein Wert in Millisekunden).

Folgender Code gibt den bekannten Gruß um eine Sekunde verzögert aus:

```
setTimeout(() => console.log('Hello World'), 1000)
```

Codebeispiel 1

Um die Aufgabe mit dem Satz zu lösen, ergibt sich eine klassische ***Verschachtelung*** von Callbacks:

```
'use strict'

// print a phrase, a word at a time:
// the pyramid of doom appears

setTimeout(() => {
    console.log('The')

    setTimeout(() => {
        console.log('pyramid')

        setTimeout(() => {
            console.log('of')

            setTimeout(() => {
                console.log('doom')

                setTimeout(() => {
                    console.log('keeps')

                    setTimeout(() => {
                        console.log('growing.')
                    }, 1000)
                }, 1000)
```

```
25              }, 1000)
26          }, 1000)
27      }, 1000)
28  }, 1000)
```

Codebeispiel 2 *01/examples/pyramid_of_doom.js*

Dieser Code ist aus mehrerlei Hinsicht nicht ideal. Die Verschachtelung und damit die ***Einrücktiefe*** wird zu groß. Der Code sieht aus wie eine auf die Seite gestellte Pyramide. Man spricht von der »***Pyramid of Doom***« (engl. für »Pyramide des Untergangs«).

Man kann das in diesem speziellen Fall besser coden, z.B. mit einer Funktion, die das n-te Wort ausgibt und sich per `setTimeout(…)` mit `n + 1` als Parameter aufruft, solange noch Worte vorhanden sind (siehe dazu Übung 1).

Grundsätzlich kommt man aber um das (tiefe) Verschachteln von Callbacks nicht so ohne weiteres herum. Der Code kommt dann quasi aus der ***Callback-Hölle*** und wird schnell unleserlich.

Ein zusätzliches Problem stellt die ***Fehlerbehandlung*** dar. In Codebeispiel 2 ist eine solche nicht nötig. Was aber, wenn nicht sechs Worte geloggt werden sollen, sondern sechs Dateien eingelesen werden sollen?

```
const data = fs.readFile('data.txt', 'UTF8', (error, data) => {
    /* … */
})
```

Codebeispiel 3

Nach dem Einlesen einer Datei muss jeweils `error` überprüft werden. Das Programm soll nur dann fortgesetzt werden, wenn alle Dateien ohne Fehler eingelesen wurden. Beim Versuch diese Aufgabe zu lösen werden Sie feststellen, dass Sie entweder wieder eine mächtige Code-Pyramide oder eine nicht ganz banale Hilfsfunktion schreiben müssen. Viel bequemer wäre es, Code zu schreiben, der unmittelbar die folgenden Schritte ausführt:

- Lies die Dateien x, y, z, usw. ein.
- Wenn alle eingelesen sind, tue dies.
- Wenn das Einlesen an irgendeiner Stelle fehl schlägt, tue jenes.

Dies und einiges mehr schaffen die mit ES2015 eingeführten Promises!

Übung 1: Die Pyramide abbrechen

Schreiben Sie Codebeispiel 2 so um, dass keine Pyramide entsteht. Benutzen Sie dazu, wie im Text angedeutet, eine Funktion, die das n-te Wort ausgibt und sich per `setTimeout(…)` mit `n + 1` als Parameter aufruft, solange noch Worte im Satz vorhanden sind.

1.2 Das Versprechen

Ein ***Promise*** ist ein *JavaScript*-Sprachbestandteil, der diese Probleme umgeht. Hier ist ein Promise:

```
const doWork = (resolve, reject) => {
    let ergebnis = 21 * 2
    resolve(ergebnis)
}

const p1 = new Promise(doWork)
```

Codebeispiel 5

Die Berechnung `21 * 2` stellt hier natürlich nur ein konstruiertes Minimalbeispiel dar.

Mit `new Promise()` wird ein Promise-Objekt `p1` erstellt. Als Argument erhält der Konstruktor einen ***Handler***: hier `doWork(…)`.

Der Funktion `doWork(…)` werden zwei Argumente übergeben: `resolve` und `reject`.

Jetzt wird's haarig: `resolve` und `reject` sind selbst wiederum Funktionen. Normalerweise erledigt `doWork(…)` irgendwelche (asynchrone) Aufgaben, die entweder mit dem Aufruf vom `resolve(…)` oder dem Aufruf von `reject(…)` enden.

Der Zweck von `resolve(…)` ist es, ein Ergebnis abzuliefern und der Zweck von `reject(…)` ist es, einen Grund zu nennen, warum das Ergebnis nicht geliefert werden kann.

Nomenklatur

Ein neu erzeugtes Promise, dessen Handler bisher weder `resolve(…)` noch `reject(…)` aufgerufen hat, ist im Zustand **anstehend** (engl. ***pending***). Hat der Handler `resolve(…)` aufgerufen — und damit die erfolgreiche Abarbeitung der Operation mitgeteilt, ist das Promise im Zustand **erfüllt** (engl. ***fulfilled***). Falls der Handler `reject(…)` aufgerufen hat, weil ein Fehler aufgetreten ist, so ist das Promise im Zustand **verworfen** (engl. ***rejected***).

Der Trick dabei: Der Aufruf von `new Promise()` blockiert nie. Auch nicht, wenn `doWork(…)` sich erst später, z.B. in einem Callback, entscheidet, `reject(…)` oder `resolve(…)` aufzurufen.

Solch ein Promise-Objekt ist also in gewisser Weise ein Platzhalter für ein zukünftiges Ergebnis — sozusagen ein ***Versprechen***, das Ergebnis noch nachzuliefern. Daher der Name *Promise* (engl. für Versprechen).

Soweit so gut. Wie kommt man in obigem Beispiel nun vom Promise (`p1`) auf das Ergebnis (`42`)? Die Promise-API hält dazu die Methode `then(…)` bereit:

```
p1.then(
    result => console.log('OK: ' + result),
    reason => console.log('KO: ' + reason),
)
```

Codebeispiel 6

Auch `then(…)` erwartet zwei Funktionen als Argumente, wovon nur eine aufgerufen wird: entweder die erste mit dem `resolve(…)`-Ergebnis oder die zweite mit dem `reject(…)`-Fehler.

Der Code bisher gibt also einfach

```
OK: 42
```

Codebeispiel 7

aus.

Normalerweise kümmert sich eine Funktion wie `doWork(…)` um asynchrone Aufgaben. Hier ist ein besseres Beispiel, denn das Ergebnis wird erst nach einer Sekunde ausgegeben:

```
const p2 = new Promise(resolve => setTimeout(() => resolve(42), 1000))

p2.then(result => console.log(result))
```

Codebeispiel 8

Um Überraschungen zu vermeiden wird der `then(…)`-Handler immer asynchron aufgerufen, selbst wenn das Promise unmittelbar erfüllt wird. Folgender Code:

```
const p3 = new Promise(resolve => resolve(42))
console.log('Step 1')
p3.then(result => console.log('OK: ' + result))
console.log('Step 2')
```

Codebeispiel 9

gibt aus:

```
Step 1
Step 2
OK: 42
```

Codebeispiel 10

Soweit betrachtet sind Promises nur eine interessante und etwas umständliche Weise, Code asynchron aufzurufen. Doch es geht noch weiter!

Die Methode `then(…)` gibt stets wiederum ein neues Promise zurück, das aus dem Rückgabewert des `then(…)`-Handlers erzeugt wird. Der Rückgabewert kann explizit eine Promise sein oder auch ein Wert jedes anderen Typs, der dann einfach implizit in ein Promise verpackt und `resolve(…)` übergeben wird.

Im Klartext heißt das, dass `then(…)`-Ketten möglich werden (engl. ***chaining***), die der Reihe nach asynchrone Aufgaben erledigen! Hier ist ein Beispiel:

```
const wait = () => new Promise(resolve => setTimeout(resolve, 1000))

wait()
    .then(() => {
        console.log('The')
        return wait()
    })
    .then(() => {
        console.log('pyramid')
        return wait()
    })
// etc...
```

Codebeispiel 11

Wir sind auf dem besten Wege, die »Pyramid of Doom« abzubauen!

Das folgende Codebeispiel definiert eine Funktion `printDelay(…)`, die ein Promise liefert, das den gegebenen Text verzögert ausgibt:

```
'use strict'

// print a phrase, a word at a time:
// use promises to avoid the pyramid of doom

const printDelay = (time, str) =>
    new Promise(resolve =>
        setTimeout(() => {
            console.log(str)
            resolve()
        }, time),
    )

printDelay(1000, 'The')
```

```
15      .then(() => printDelay(1000, 'pyramid'))
16      .then(() => printDelay(1000, 'of'))
17      .then(() => printDelay(1000, 'doom'))
18      .then(() => printDelay(1000, 'is'))
19      .then(() => printDelay(1000, 'defeated.'))
```

Codebeispiel 12 *01/examples/pyramid_avoided.js*

Die Funktion `printDelay(…)` in Zeile 6 gibt ein Promise zurück, dessen Aufgabe es ist, per `setTimeout(…)` den String `str` nach `time` Millisekunden auszugeben.

In Zeile 10 wird `printDelay(…)` aufgerufen, um nach 1000 Millisekunden »The« auszugeben. Sobald die Promise erfüllt ist, wird der `then(…)`-Handler ausgeführt und gibt eine weitere Promise zurück. Die folgende `then(…)`-Kette gibt somit alle Worte zeitverzögert aus. Das Ergebnis entspricht Codebeispiel 2, die Pyramide ist aber verschwunden!

1.3 Es geht auch mal daneben

Erinnern Sie sich an folgende Sätze aus Abschnitt 1.2:

Der Zweck von `resolve(…)` ist, ein Ergebnis abzuliefern. Der Zweck von `reject(…)` ist, einen Grund zu nennen, warum das Ergebnis nicht geliefert werden kann.

Hier ist eine Variante von `doWork(…)`, die mit einer Wahrscheinlichkeit von 50% kein Ergebnis liefert, sondern `reject(…)` mit dem Argument `"nope"` aufruft: `tryWork(…)`.

```
const tryWork = (resolve, reject) => {
    let ergebnis = 21 * 2
    if (Math.random() < 0.5) {
        resolve(ergebnis)
    } else {
        reject('nope')
    }
}

const p4 = new Promise(tryWork)
```

Codebeispiel 13

So wird `p4` benutzt:

```
p4.then(
    result => console.log('OK: ' + result),
    reason => console.log('KO: ' + reason),
)
```

Codebeispiel 14

Wenn Sie diesen Code mehrmals ausführen, wird das Promise `p4` manchmal erfüllt und manchmal verworfen (etwas blumig formuliert: » Das Versprechen, das Ergebnis (42) zu liefern, wurde manchmal gebrochen). Nach zehn Aufrufen erhalte ich z.B.:

Abb. 1 *Die Frage nach dem Sinn des Lebens, dem Universum und dem ganzen Rest — die Berechnung gelingt nicht immer*

Interessant wird es, sobald Chaining ins Spiel kommt:

```
new Promise(tryWork)
    .then(() => new Promise(tryWork))
    .then(() => new Promise(tryWork))
    .then(
        result => console.log('OK: ' + result),
        reason => console.log('KO: ' + reason),
    )
```

Codebeispiel 15

In diesem Beispiel wird `tryWork(…)` dreimal aufgerufen, oder präziser: **bis zu** dreimal! Jeder einzelne Aufruf wird nämlich mit 50%-iger Wahrscheinlichkeit verworfen. Allerdings besitzt nur das dritte `then(…)` in der Kette einen `reject(…)`-Handler. Es gilt das wichtige Prinzip, dass das erste verworfene Promise die Kette unterbricht und die Code-Ausführung direkt zum nächsten `reject(…)`-Handler springt. Es ergeben sich folgende vier Möglichkeiten:

1. Promise	2. Promise	3. Promise	Ausgabe
liefert 42	liefert 42	liefert 42	'OK: 42'
liefert 42	liefert 42	wird verworfen	'KO: nope'
liefert 42	wird verworfen	wird nicht ausgeführt	'KO: nope'
wird verworfen	wird nicht ausgeführt	wird nicht ausgeführt	'KO: nope'

Wenn Sie den Code wiederholt ausführen, sehen Sie, dass nur noch ungefähr einer von acht Aufrufen (entspricht einer Wahrscheinlichkeit von 12.5%) das Ergebnis ausgibt. Die Wahrscheinlichkeit, dass dieses Ereignis eintritt, setzt sich nämlich aus den Einzelwahrscheinlichkeiten zusammen:

```
0.5 * 0.5 * 0.5 = 0.125
```

Codebeispiel 16

In diesem Zusammenhang noch ein Detail, wie Sie die Lesbarkeit des Codes verbessern. Die Promise-API bietet eine bequeme `catch(…)`-Methode:

```
catch( reason => … )
```

Codebeispiel 17

Dies ist equivalent zu:

```
then( undefined, reason => … )
```

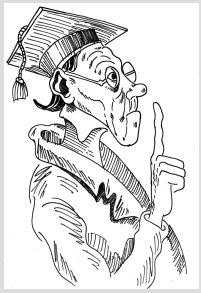
Codebeispiel 18

Damit lässt sich eleganter ausdrücken, dass man eine Sequenz von Aufgaben erledigen will und alle möglicherweise auftretenden Fehler an einer zentralen Stelle »auffangen« (engl. to catch) möchte:

```
new Promise(tryWork)
    .then(() => new Promise(tryWork))
    .then(() => new Promise(tryWork))
    .then(result => console.log('OK: ' + result))
    .catch(reason => console.log('KO: ' + reason))
```

Codebeispiel 19

Da `catch(…)` equivalent zu `then(…)` mit nur dem zweiten Argument ist, kann auch `catch(…)` seinerseits wieder Promises zurückgeben. Gemischte `then(…)`-`catch(…)`-`then(…)`-`catch(…)`-Ketten sind möglich. Typischerweise finden Sie in Programmen das `catch(…)` aber am Ende als Abschluss der Kette.



Keine Vorstellung ohne Sicherheitsnetz

Mit verworfenen Promises will ein Programmierer ausdrücken, dass eine Fehlersituation aufgetreten ist. Gute Programme fangen Fehler selbstverständlich ab und reagieren darauf. Ein Programm sollte niemals in die Situation kommen, dass ein Promise erstellt wird, für die kein `reject(…)`-Handler zuständig ist!

Node.js unterstützt diese gute Programmiertechnik seit Version 6.6 mit einer Warnung. Wird ein Promise verworfen und es ist kein `reject(…)`-Handler vorhanden, wird folgende Warnung in die Console ausgegeben:

```
(node:2619)
UnhandledPromiseRejectionWarning:
Unhandled promise rejection
(rejection id: 2): nope
(node:2619) [DEP0018]
DeprecationWarning: Unhandled
promise rejections are
deprecated. In the future,
promise rejections that are
not handled will terminate the
Node.js process with a
non-zero exit code.
```

Codebeispiel 20

Die Warnung ist deutlich: Zukünftige *Node.js*-Versionen führen an dieser Stelle zu einem sofortigen Programmabbruch!

In anderen Sprachen ist das schon lange üblich. In Java z. B. wird Code gar nicht erst übersetzt, geschweige denn ausgeführt, wenn ein entsprechendes `catch(…)`-Konstrukt fehlt.

An dieser Stelle wissen Sie fast alles zum Thema Promises und Fehlerbehandlung, was für den Anfang nötig ist. Fast. Eine offene Frage bleibt: »Was genau ist eigentlich das Argument von `reject(…)`?« Ein Programm möchte damit einen Fehler melden oder zumindest ausdrücken, das ein bestimmtes Ergebnis nicht geliefert werden kann. Bisher haben unsere Beispiele mit

```
reject('nope')
```

Codebeispiel 21

aber lediglich einen String übergeben. Jede Information zum Fehler muss also erstmal im String kodiert sein — das ist nicht sehr praktisch. An dem `"nope"` sehen wir z.B. nicht, wo die Kette unterbrochen wurde. *JavaScript* stellt ein eigenes Objekt für genau diesen Zweck zur Verfügung: das `Error`-Objekt.

Die endgültige Version unserer Berechnung sehen Sie in Codebeispiel 22.

```
1  'use strict'
2
3  const tryWork = (resolve, reject) => {
4      let ergebnis = 21 * 2
5      if (Math.random() < 0.5) {
6          resolve(ergebnis)
7      } else {
8          reject(new Error('nope'))
9      }
10 }
11
12 new Promise(tryWork)
13     .then(() => new Promise(tryWork))
14     .then(() => new Promise(tryWork))
15     .then(result => console.log('OK: ' + result))
16     .catch(err => {
17         console.log('KO: ' + err.message)
18         console.log(err.stack)
19     })
```

Codebeispiel 22 *01/examples/stack_trace.js*

Neu ist, dass `reject(…)` nun ein `Error`-Objekt übergeben bekommt. Im `catch(…)`-Handler geben wir zwei Eigenschaften aus: die Fehlermeldung `err.message` und den sogenannten ***Stack-Trace*** `err.stack`. Dieser zeigt an, wo im Code ein Fehler aufgetreten ist und die Abfolge der letzten Aufrufe, die zu diesem Punkt geführt haben. Im vorliegenden Beispiel ist die Ausgabe entweder `'OK: 42'` oder:

```
KO: nope
Error: nope
    at tryWork (/Users/chris/01/examples/stack_trace.js:8:16)
    at new Promise (<anonymous>)
    at Object.<anonymous> (/Users/chris/01/examples/stack_trace.js:12:1)
    at Module._compile (module.js:643:30)
    at Object.Module._extensions..js (module.js:654:10)
    at Module.load (module.js:556:32)
    at tryModuleLoad (module.js:499:12)
    at Function.Module._load (module.js:491:3)
    at Function.Module.runMain (module.js:684:10)
    at startup (bootstrap_node.js:187:16)
```

Codebeispiel 23

Die dritte "at"-Zeile variiert: Je nachdem, welche der drei Promises verworfen wurde, ist an dieser Stelle eine der folgenden Zeilen möglich:

```
at Object.<anonymous> (/Users/chris/01/examples/stack_trace.js:12:1)
at Object.<anonymous> (/Users/chris/01/examples/stack_trace.js:13:18)
at Object.<anonymous> (/Users/chris/01/examples/stack_trace.js:14:18)
```

Codebeispiel 24

Die Zahlen am Ende weisen auf die genaue Zeile und Spalte hin, wo das verworfene Promise erzeugt wurde. Mögliche Zeilen sind 12, 13 oder 14. Die Spalten sind 1 oder 18, je nach Code-Einrückung.

Jede der drei Promises ruft schlussendlich `trywork(…)` auf, wo in Zeile 8 und Spalte 16 das `Error`-Objekt erstellt wurde.

Sie kennen Stack-Traces sicher schon, weil *Node.js* selbst welche ausgibt, wenn Sie einen Fehler im Code haben, der zu einem Abbruch führt. Nun wissen Sie, wie Sie Ihre eigenen Stack-Traces ausgeben können. Stack-Traces sind eine wichtige Debug-Hilfe!

1.4 Jetzt alle zusammen

Die Promise-API stellt noch die nützliche und wichtige Methode `Promise.all(…)` zur Verfügung. Diese Funktion erhält als Parameter ein Array von Promises und gibt ein Promise zurück. Dieses Rückgabe-Promise ist erfüllt, sobald **alle** Eingangs-Promises erfüllt sind, oder wird verworfen, sobald das **erste** Eingangs-Promise verworfen wurde.

Greifen wir nochmals auf `printDelay(…)` aus Codebeispiel 12 zurück:

```
const printDelay = (time, str) =>
    new Promise(resolve =>
        setTimeout(() => {
            console.log(str)
            resolve()
        }, time),
    )
```

Codebeispiel 25

Mit `Promise.all(…)` lassen sich alle sechs Promises gleichzeitig auflösen:

```
Promise.all([
    printDelay(1000, 'The'),
    printDelay(1000, 'pyramid'),
    printDelay(1000, 'of'),
    printDelay(1000, 'doom'),
    printDelay(1000, 'is'),
    printDelay(1000, 'defeated.'),
]).then(() => console.log('OK'))
```

Codebeispiel 26

Der Effekt ist, dass nach einer Sekunde Pause alle sechs Worte des Satzes sofort ausgegeben werden, unmittelbar gefolgt von `OK`.

Sobald aber auch nur ein Promise (egal welches) verworfen wird, springt die Ausführung des Codes direkt zum `catch(…)`-Handler:

```
Promise.all([
    printDelay(1000, 'The'),
    printDelay(1000, 'pyramid'),
    printDelay(1000, 'of'),
    new Promise((resolve, reject) => reject()),
    printDelay(1000, 'doom'),
    printDelay(1000, 'is'),
    printDelay(1000, 'defeated.'),
])
    .then(() => console.log('OK'))
    .catch(() => console.log('KO'))
```

Codebeispiel 27

Dieses Codebeispiel gibt unmittelbar `KO` aus. Nach einer Sekunde folgen die Worte des Satzes, aber kein `OK`!

Wie steht es nun mit dem Ergebnis, das an den `then(…)`-Händler übergeben wird? Im Falle von `Promise.all(…)` gibt es mehrere Ergebnisse. Genau so, wie die Eingangs-Promises als Array übergeben wurden, kommen die Ergebnisse als Array zurück — und zwar in der gleichen Reihenfolge!

Folgender Code:

```
Promise.all([
    new Promise((resolve, reject) => resolve('one')),
    new Promise((resolve, reject) => resolve('two')),
    new Promise((resolve, reject) => resolve('three')),
]).then(results => results.forEach(result => console.log(result)))
// => one↲ two↲ three↲
```

Codebeispiel 28

gibt garantiert `one`, `two` und `three` in dieser Reihenfolge aus.

Das ist auch der Fall, wenn eines der Promises mehr Zeit bis zum Aufruf von `resolve(…)` benötigt! Die Reihenfolge entspricht immer der Reihenfolge der Promises im Eingangs-Array. Folgender Code wartet eine halbe Sekunde und gibt anschließend wiederum `one`, `two` und `three` aus:

```
Promise.all([
    new Promise((resolve, reject) => resolve('one')),
    new Promise((resolve, reject) => setTimeout(() => resolve('two'), 500)),
    new Promise((resolve, reject) => resolve('three')),
]).then(results => results.forEach(result => console.log(result)))
// => one↲ two↲ three↲
```

Codebeispiel 29

1.5 Die Konsumgesellschaft

In der Praxis erstellen Sie eher selten eigene Promises. Viel häufiger »konsumieren« Sie sie! In der *JavaScript*-Welt findet gegenwärtig ein Umdenken statt: Viele *JavaScript*-Entwickler stellen ihre Schnittstellen zu asynchronen Funktionen von Callbacks auf Promises um.

Ein Aufruf mit Callback:

```
doSomething(argument, (error, result) => {
    if (error) {
        console.log('KO')
    } else {
        console.log('OK: ' + result)
    }
})
```

Codebeispiel 30

wird in der Promise-Variante zu:

```
doSomething(argument)
    .then(result => console.log('OK: ' + result))
    .catch('KO')
```

Codebeispiel 31

In den folgenden Sektionen lernen Sie eine sehr wichtige Schnittstelle im DOM kennen, die bereits auf Promises setzt. Viele traditionellere Schnittstellen verwenden aber noch Callbacks. Aber auch in solchen Fällen sind Promises nützlich! Ein konkretes Beispiel ist das Einlesen einer Datei mit der Methode `readFile(…)` aus dem `fs`-Modul:

```
'use strict'

const fs = require('fs')

fs.readFile('hello.txt', 'UTF8', (error, content) => {
    if (error) {
        console.log('could not read file')
    } else {
        console.log(content)
    }
})
```

Codebeispiel 32 *01/examples/read_file_callback.js*

Der Aufruf von `readFile(…)` lässt sich prima in eine Funktion kapseln, die ein Promise zurückgibt:

```
'use strict'

const fs = require('fs')

let getFileContent = name =>
    new Promise((resolve, reject) =>
        fs.readFile(name, 'UTF8', (error, content) => {
            if (error) {
                reject('could not read file')
            } else {
                resolve(content)
            }
        }),
    )

getFileContent('hello.txt')
    .then(content => console.log(content))
    .catch(error => console.log(error))
```

Codebeispiel 33 *01/examples/read_file_promise.js*

Es ist offensichtlich, dass die Funktion `getFileContent(…)` wesentlich bequemer ist, um z.B. wie in Abschnitt 1.1 angedeutet, sechs Dateien einzulesen und sich um die Fehlerbehandlung zu kümmern! Der entsprechende Code wird sehr übersichtlich:

```
Promise.all([
    getFileContent('hello1.txt'),
    getFileContent('hello2.txt'),
    getFileContent('hello3.txt'),
    getFileContent('hello4.txt'),
    getFileContent('hello5.txt'),
    getFileContent('hello6.txt'),
])
    .then(contents => contents.forEach(content => console.log(content)))
    .catch(error => console.log(error))
```

Codebeispiel 34

Eine Funktion wie `getFileContent(…)` aus Codebeispiel 33 nennt man eine Wrapper-Funktion (von »to wrap« — engl. für »einwickeln«).

Es ist natürlich aufwändig, für jede asynchrone Funktion (wie `fs.readFile(…)`) eine eigene Wrapper-Funktion zu definieren.

Node.js bringt ab Version 8.0.0 eine allgemeine Wrapper-Funktion mit: `util.promisify(…)`. Die Beschreibung dazu finden Sie in der *Node.js*-API-Dokumentation[1].

1. https://nodejs.org/api/util.html#util_util_promisify_original

Die Funktion `util.promisify(…)` bekommt als Argument eine beliebige asynchrone Funktion, die per Konvention als letztes Argument ein Callback mit den Parametern `(error, result)` akzeptiert. Der Rückgabewert ist die fertige Wrapper-Funktion.

Mit Hilfe dieser Funktion vereinfacht sich Codebeispiel 33 zu:

```
1  'use strict'
2
3  const fs = require('fs')
4  const util = require('util')
5
6  const getFileContent = util.promisify(fs.readFile)
7
8  getFileContent('hello.txt', 'UTF8')
9      .then(content => console.log(content))
10     .catch(error => console.log('could not read file ' + error.path))
```

Codebeispiel 35 *01/examples/read_file_promise_improved.js*

1.6 Übungen

Übung 2: Sechs Dateien einlesen

Erstellen Sie die Dateien `hello1.txt` … `hello6.txt` und lesen Sie sie, wie soeben beschrieben, mit `getFileContent(…)` aus Codebeispiel 33 ein.

Was passiert, wenn eine Datei fehlt? Überprüfen Sie die Fehlerbehandlung. Erweitern Sie die Fehlermeldung so, dass ausgegeben wird, **welche** Dateien nicht gelesen werden konnten!

Übung 3: Das Telefonbuch des Internets

Vielleicht haben Sie einmal die Webseite `https://www.webmasters-press.de/` besucht? Dann hat Ihr Computer die so genannte ***IP-Adresse*** des Servers (`www.webmasters-press.de`) herausgefunden.

Ähnlich wie ein Telefonbuch ordnet das Domain Name System (***DNS***) jedem ***Server*** eine eindeutige IP-Adresse zu, unter der er über das Internet erreichbar ist.

Die *Node.js*-API bietet eine Funktion, um Namen im DNS nachzuschlagen: `dns.lookup(…)`[2].

So geht's:

```
const dns = require('dns')
const IP_V = 4 // we use IP protocol version 4

dns.lookup('www.1006.org', IP_V, (error, address) => {
    if (error) {
        console.log('error: could not lookup host')
    } else {
        console.log('IP address = ' + address)
    }
})
```

2. https://nodejs.org/api/dns.html#dns_dns_lookup_hostname_options_callback

Wenn ich dieses Programm ausführe (der Rechner muss dazu mit dem Internet verbunden sein), erhalte ich die Ausgabe:

```
IP address = 54.93.168.85
```

Am Callback erkennen Sie, dass `dns.lookup(…)` genau wie `fs.readFile(…)` eine asynchrone Funktion ist. Das ist auch gut so, schließlich könnte eine Suche im DNS je nach Internetverbindung länger dauern oder fehlschlagen.

Die Übung besteht nun darin, obiges Beispiel so umzuschreiben, dass der Aufruf von `dns.lookup(…)` in eine Wrapper-Funkion verpackt wird. Verwenden Sie dazu `util.promisify(…)`!

1.7 Testen Sie Ihr Wissen!

1. Um mehrere Promises zusammen aufzulösen, benutzt man:

   Bitte ankreuzen:

   - ☐ die Methode `Promise.all()`
   - ☐ die Methode `Promise.forEach()`
   - ☐ die Methode `Promise.join()`
   - ☐ die Methode `Promise.resolve()`

2. Ein Promise, dessen Handler die `resolve()`-Methode aufgerufen hat, ist im Zustand:

   Bitte ankreuzen:

   - ☐ ready
   - ☐ done
   - ☐ satisfied
   - ☐ fulfilled
   - ☐ returned

3. Welchen Typ hat der Rückgabewert der Promise-Methode `then`?

   Bitte ankreuzen:

   - ☐ ein Boolean
   - ☐ den Wert `undefined`
   - ☐ eine Funktion
   - ☐ ein Promise-Objekt

4. Welche Anweisung ruft die Funktion `f` nach 100 Millisekunden auf?

   Bitte ankreuzen:

   - ☐ `setTimeout(f, 100)`
   - ☐ `setTimeout(() => f(), 100)`
   - ☐ `new Promise(() => f(), 100)`
   - ☐ `new Promise(100, f)`

Index

T

V

W

X

Z