

WBE: JAVASCRIPT ASYNCHRONES PROGRAMMIEREN

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

1

ÜBERSICHT

- File API
- Reagieren auf Ereignisse
- Modul „events“
- Promises, Async/Await

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

2

ÜBERSICHT

- File API
- Reagieren auf Ereignisse
- Modul „events“
- Promises, Async/Await

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

3

SYNCHRONES LESEN AUS DATEI

```
1 const fs = require('fs')
2 let data = fs.readFileSync('/etc/hosts')
3 console.log(data)
4 /* → <Buffer 23 23 0a 23 20 48 6f 73 74 20 44 61 74 61 62 61 ... */
5
6 data = fs.readFileSync('/etc/hosts', 'utf8')
7 console.log(data)
8 /* →
9 ##
10 # Host Database
11 #
12 ...
13 */
```

Problem?

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

4

EIN-/AUSGABE

Access	Cycles
L1	3 cycles
L2	14 cycles
RAM	250 cycles
DISK	41'000'000 cycles
NETWORK	240'000'000 cycles

Zahlen nicht mehr aktuell (ca. 2010), aber die Größenordnung dürfte in etwa noch stimmen

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

5

FAIL!

SYNCHRONES LESEN AUS DATEI

```
1 data = fs.readFileSync('/etc/hosts', 'utf8')
2 /*
3  wait...
4  wait...
5  wait...
6  wait...
7  wait...
8  wait...
9  wait...
10 */
11 console.log(data)
```

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

6

ASYNCHRONES LESEN AUS DATEI

```
1 const fs = require('fs')
2 fs.readFile('/etc/hosts', "utf8", (err, data) => {
3   if (err) throw err
4   console.log(data)
5 })
6
7 doSomethingElse()
```

WIN ✓

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

7

CALLBACKS

- Ein **Callback** ist eine Funktion, welche als Argument einer anderen Funktion übergeben wird und erst aufgerufen wird, wenn das Ereignis eingetreten ist
- Beispiel: `fs.readFile` mit Callback
- Ursprünglich in JS: Reaktion auf Webseiten-Ereignisse

```
document.getElementById('button').addEventListener('click', () => {
  //item clicked
})
```

Mehr zum Browser-DOM in einer späteren Lektion

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

8

FILE-API

- Datei-Operationen sind in der Regel langsam
- Sie sollten praktisch immer asynchron ausgeführt werden
- Erstes Argument statt Pfad auch: **File Descriptor**
- Methode `open` liefert einen File Descriptor

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

fs.open('test.txt', 'r', (err, fd) => {
  // fd is our file descriptor
})
```

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

9

DATEI-INFORMATIONEN

```
1 const fs = require('fs')
2 fs.stat('test.txt', (err, stats) => {
3   if (err) {
4     console.error(err)
5     return
6   }
7
8   stats.isFile()          /* true */
9   stats.isDirectory()      /* false */
10  stats.isSymbolicLink()  /* false */
11  stats.size              /* 1024000 = ca 1MB */
12 })
```

https://nodejs.org/api/fs.html#fs_class_fs_stats

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

10

DATEIEN UND PFADE

```
1 const path = require('path')
2 const notes = '/users/bkrt/notes.txt'
3
4 path.dirname(notes)           /* /users/bkrt */
5 path.basename(notes)          /* notes.txt */
6 path.extname(notes)          /* .txt */
7 path.basename(notes, path.extname(notes)) /* notes */
```

- Diverse weitere Methoden
- <https://nodejs.org/api/path.html>

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

11

DATEIEN SCHREIBEN

```
1 const fs = require('fs')
2 const content = 'Node was here!'
3
4 fs.writeFile('/Users/bkrt/test.txt', content, (err) => {
5   if (err) {
6     console.error(`Failed to write file: ${err}`)
7     return
8   }
9   /* file written successfully */
10 })
```

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

12

STREAMS

- Grössere Dateien eher mit Streams lesen und schreiben
- Daten werden nach und nach geliefert oder geschrieben
- Lesen: `data`- und `end`-Events

Mehr zum Thema *Streams* in der nächsten Lektion

MEHR ZUM FS-MODUL

Funktion	Bedeutung
<code>fs.chmod</code>	Berechtigungen ändern
<code>fs.chown</code>	Besitzer und Gruppe ändern
<code>fs.copyFile</code>	Datei kopieren
<code>fs.link</code>	Hardlink anlegen
<code>fs.symlink</code>	Symbolic Link anlegen
<code>fs.watchFile</code>	Datei auf Änderungen überwachen

VERZEICHNISSE

- Im `fs`-Modul: auch Funktionen zur Arbeit mit Ordnern
- Die meisten davon gibt es auch in einer synchronen Variante

Funktion	Bedeutung
<code>fs.access</code>	Zugriff auf Datei oder Ordner prüfen
<code>fs.mkdir</code>	Verzeichnis anlegen
<code>fs.readdir</code>	Verzeichnis lesen, liefert Array von Einträgen
<code>fs.rename</code>	Verzeichnis umbenennen
<code>fs.rmdir</code>	Verzeichnis löschen

ÜBERSICHT

- File API
- Reagieren auf Ereignisse
- Modul „events“
- Promises, Async/Await

EIN THREAD

- JavaScript-Code wird in einem Thread abgearbeitet
- Probleme vermieden, die bei paralleler Ausführung auftreten können (gemeinsame Ressourcen, mögliche Blockaden)
- **Vorsicht: Thread darf nicht blockiert werden**
- Browser: Tabs normalerweise mit unabhängigen Event Loops

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

17

BEISPIEL

```
// script.js
setTimeout(() => {
  console.log("fertig :)")
}, 5000)

console.log("starting...")
```

Aufruf:
\$ node script.js
starting...
fertig :)
\$

- Mehr zu `setTimeout()` gleich
- Callback nach Ablauf des Timers aufgerufen
- Ausgabe `fertig :)` erscheint 5 Sekunden nach `starting...`

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

18

ABLAUF

- Script wird ausgeführt
- Funktionsaufrufe → **Call Stack**
- Callbacks asynchroner Operationen in **Event Queue(s)** gelegt
- Wenn Call Stack leer, d.h. (synchrone) Aufrufe abgearbeitet:
 - Übergang in eine so genannte **Event Loop**
 - Callbacks aus Event Queue abgearbeitet
 - Event Queue leer: Programm beendet

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

19

EVENT LOOP

Ein vereinfachtes Modell kann das Verhalten asynchroner Programme in vielen Situationen ganz gut erklären. Es basiert auf diesen Annahmen:

- Es gibt eine Event Queue
- Ablage der Callbacks in der Event Queue basiert auf OS-APIs

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

20

EVENT LOOP: SIMULATOR

```
1. setTimeout(function() {  
2.   console.log("fertig :");  
3. }, 5000);  
4.  
5. $on("button", "click", function() {  
6.   console.log("clicked");  
7. })  
8.  
9. console.log("starting...");  
10 |
```

Call me! Edit Call Stack Web APIs Callback Queue

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

<http://latentflip.com/loupe/>

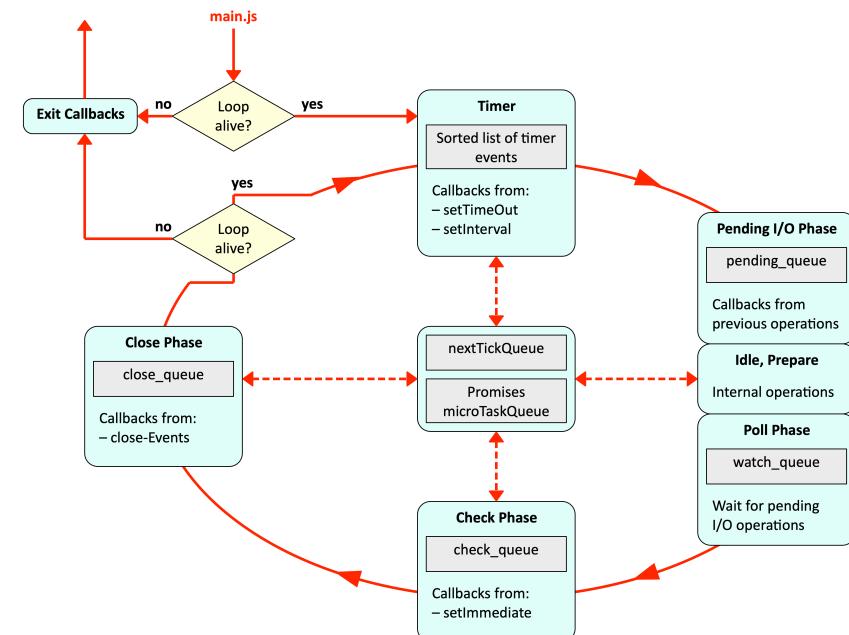
EVENT LOOP: MEHR DETAILS

- Einfaches Modell der Event Loop entspricht nicht der Realität
- Es genügt, um viele, aber nicht alle Situationen zu erklären
- Ein paar Richtigstellungen
 - Event Loop ist nicht Teil der JS-Engine sondern steuert diese
 - Es gibt mehrere Queues
 - Die Event Loop läuft nicht in einem separaten Thread
 - Mit `[setTimeout]` wird keine OS-API beauftragt

WICHTIG

- Event Loop nicht blockieren!
- Grund: blockiert die gesamte Applikation
- Im Browser: blockiert den Browser
- Zu vermeiden also:
 - synchrone Operationen (etwa für Datei- oder Netzwerkzugriff)
 - aufwändige Berechnungen ohne Unterbrechung
 - Endlosschleifen

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)



Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

23

24

EVENT LOOP: ABLAUF

- Script-Aufruf: Event Loop und Datenstrukturen angelegt
- Script mit synchronen Operationen ausgeführt (Call Stack)
- Dabei werden die Listen und Queues ggf. mit Callbacks gefüllt
- Nach Abschluss des Scripts (Call Stack leer):
Eintritt in die Event Loop
- Schleife bis alle Callbacks abgearbeitet

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

25

SETTIMEOUT

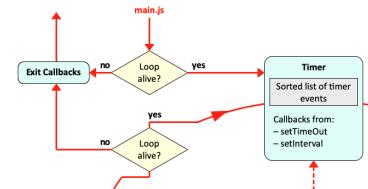
- Mit `setTimeout` kann Code definiert werden, der zu einem späteren Zeitpunkt ausgeführt werden soll
- Eintrag in die Timer-Liste, auch wenn Zeit auf 0 gesetzt wird
- Kann mit `clearTimeout` entfernt werden

```
setTimeout(() => {
  /* runs after 50 milliseconds */
}, 50)
```

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

27

EVENT LOOP: TIMER



- Callbacks für bereits verstrichenen Zeitpunkte abgearbeitet
- Abbruch auch, wenn systemspezifisches Limit erreicht

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

26

SETINTERVAL

- Callback alle n Millisekunden in die Callback Queue eingefügt
- Kann mit `clearInterval` beendet werden

```
const id = setInterval(() => {
  // runs every 2 seconds
}, 2000)

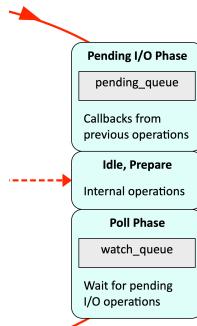
 clearInterval(id)
```

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

28

EVENT LOOP: PENDING I/O, ...

- Von vorhergehenden Durchgängen aufgeschobene Callbacks
- Beispiel: Fehlermeldungen bestimmter TCP-Aufrufe
- Idle, Prepare: interne Aufgaben



Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

29

EVENT LOOP: POLL PHASE

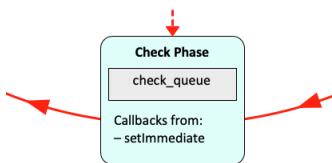
- Abarbeiten der `watch_queue` (auch: `poll_queue`)
- Auf I/O (Verbindungsanfragen etc.) warten
- Wartezeit abhängig von Füllstand der Timer-Liste und der `check_queue` (nächster Punkt in der Loop)
- Abbruch auch, wenn systemspezifisches Limit erreicht

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

30

EVENT LOOP: CHECK PHASE

- Abarbeiten der `check_queue`
- Callbacks von `setImmediate`
- Abbruch auch, wenn systemspezifisches Limit erreicht



Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

31

SETIMMEDIATE

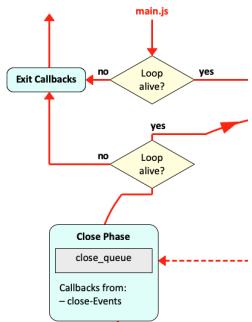
- Für Callbacks, die direkt nach der Poll Phase ausgeführt werden sollen
- Damit: Unterschied zwischen `setImmediate(...)` und `setTimeout(..., 0)`

```
setImmediate(() => {
  console.log('immediate')
})
```

32

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

EVENT LOOP: CLOSE PHASE



- Verarbeiten bestimmter close-Events
- Zum Beispiel:
`socket.on('close', ...)`
- Wenn dann alle Queues und Listen leer sind, wird die Event Loop beendet

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

35

NEXTTICKQUEUE

- Durch die API `process.nextTick` angelegte Callbacks
- `process.nextTick` daher vor `setImmediate` bearbeitet

```
fs.readFile("nexttick.js", () => {
  setTimeout(() => { console.log('timeout'); }, 0)
  setImmediate(() => { console.log('immediate'); })
  process.nextTick(() => { console.log('nexttick'); })
})

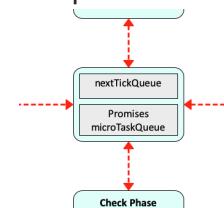
// Output:
// nexttick
// immediate
// timeout
```

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

35

NEXTTICKQUEUE UND PROMISES

- So früh wie möglich abgearbeitet
- Nicht Teil der Event Loop



- Von Node.js nach jeder Operation eingefügt
- Operation hier: JavaScript-Aufruf von C/C++ aus

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

36

PROMISES MICROTASKQUEUE

- Callbacks von erfüllten/abgewiesenen Promises
- Das betrifft die native Promise-API von JavaScript
- Nach den `nextTick`-Callbacks abgearbeitet

```
Promise.resolve().then(() => console.log('promise resolved'))
setImmediate(() => console.log('set immediate'))
process.nextTick(() => console.log('next tick'))
setTimeout(() => console.log('set timeout'), 0)

// next tick
// promise resolved
// set timeout
// set immediate
```

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

36

ÄNDERUNG SEIT NODE.JS 11

- Micro Tasks werden neu auch zwischen den Callbacks der anderen Phasen ausgeführt
- Entspricht dem Verhalten in Browsern

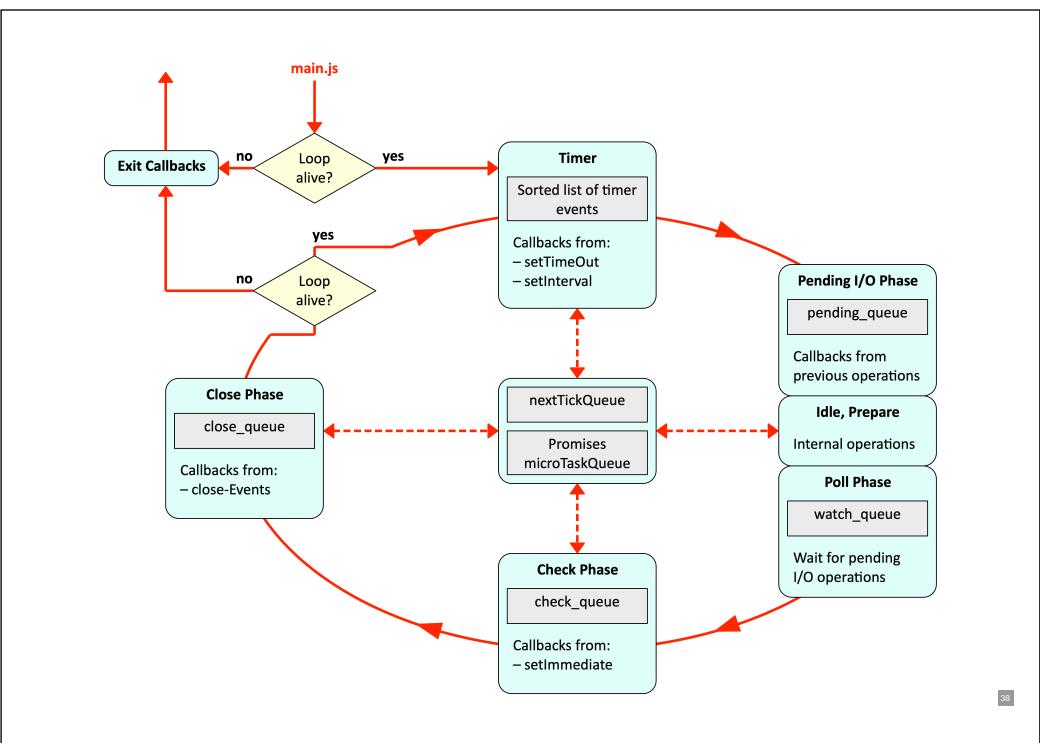
```
setTimeout(() => {
  console.log('timeout1')
  Promise.resolve().then(() => console.log('promise resolve'))
})
setTimeout(() => console.log('timeout2'))
```

Node.js < 11:
timeout1
timeout2
promise resolve

Node.js >= 11 und Browser:
timeout1
promise resolve
timeout2

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

37



38

ÜBERSICHT

- File API
- Reagieren auf Ereignisse
- **Modul „events“**
- Promises, Async/Await

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

39

EVENT Emitter

```
const EventEmitter = require('events')
const door = new EventEmitter()
```

- **EventEmitter** verwaltet Liste von Listenern zu bestimmten Events
- Listener für das Event können hinzugefügt oder entfernt werden
- Event kann ausgelöst werden → Listener werden informiert

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

40

LISTENER HINZUFÜGEN

```
const EventEmitter = require('events')
const door = new EventEmitter()

door.on('open', () => {
  console.log('Door was opened')
})
```

- Fügt Event Listener hinzu
- Alias: `emitter.addListener`

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

41

THIS

```
const myEmitter = new EventEmitter()
myEmitter.on('event', function (a, b) {
  console.log(a, b, this, this === myEmitter)
  // Prints: a b EventEmitter { domain: null, ... } true
})
myEmitter.emit('event', 'a', 'b')
```

- Normale Listener-Funktion: `this` referenziert die EventEmitter-Instanz, an welche der Listener angehängt ist
- Achtung: Dies gilt nicht für Arrow Functions

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

43

EVENT AUSLÖSEN

```
door.on('open', (speed) => {
  console.log(`Door was opened, speed: ${speed || 'unknown'}`)
})

door.emit('open')
door.emit('open', 'slow')
```

- Methode `emit`
- Ruft *synchron* alle Listener auf
- Und zwar in der Reihenfolge wie sie definiert wurden
- Es können Argumente übergeben werden

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

42

EVENTS ASYNCHRON

- Nach Ereignisauslösung (`emit`) werden die Listener ausgeführt
- Listener werden *synchron* aufgerufen
- Und zwar in der Reihenfolge der Registrierung
- Listener können selbst auf *asynchronen* Modus wechseln

```
myEmitter.on('event', (a, b) => {
  setImmediate(() => {
    console.log('this happens asynchronously')
  })
})
```

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

44

ÜBERSICHT

- File API
- Reagieren auf Ereignisse
- Modul „events“
- Promises, Async/Await

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

45

FUNKTION MIT CALLBACKS

- Bekanntes Beispiel
- Asynchrones Lesen mit `fs`-Modul
- Diesmal in Funktion verpackt

```
1 const fs = require('fs')
2
3 function readFileSync (file, success, error) {
4   fs.readFile(file, "utf8", (err, data) => {
5     if (err) error(err)
6     else success(data)
7   })
8 }
9
10 /* Aufruf: */
11 readFileSync(file, okCallback, failCallback)
```

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

47

PROMISES

- Eingeführt mit ES6 (ES2015)
- Vermeiden von verschachtelten Callbacks
- Vereinfacht Fehlerbehandlung

Promise

Platzhalter für einen Wert, der erst später voraussichtlich verfügbar sein wird

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

46

FUNKTION MIT PROMISE

```
1 function readFilePromise (file) {
2   let promise = new Promise(
3     function resolver (resolve, reject) {
4       fs.readFile(file, "utf8", (err, data) => {
5         if (err) reject(err)
6         else resolve(data)
7       })
8     })
9   return promise
10 }
```

- Gibt nun ein Promise-Objekt zurück
- Promise-Konstruktor erhält `resolver`-Funktion

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

48

FUNKTION MIT PROMISE

- Rückgabe einer Promise: potentieller Wert
- Kann später erfüllt oder zurückgewiesen werden
- Aufruf neu:

```
1 readFilePromise('/etc/hosts')
2 .then(console.log)
3 .catch(() => {
4   console.log("Error reading file")
5 })
```

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

49

ÜBUNG: AUSGABE?

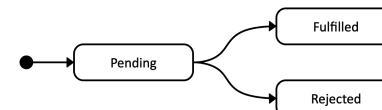
```
1 var promise = new Promise((resolve, reject) => {
2   setTimeout(resolve, 500, 'done')
3   setTimeout(reject, 300, 'failed')
4   /* throw Error('So goes it not :)') */
5 })
6
7 promise.then(function (data) {
8   console.log('success: ' + data)
9 })
10 .catch(function (data) {
11   console.log('fail: ' + data)
12 })
```

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

51

PROMISE-ZUSTÄNDE

- `pending`: Ausgangszustand
- `fulfilled`: erfolgreich abgeschlossen
- `rejected`: ohne Erfolg abgeschlossen



- Nur ein Zustandsübergang möglich
- Zustand in Promise-Objekt gekapselt

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

50

PROMISES

- `then`-Aufruf gibt selbst Promise zurück
- `catch`-Aufruf ebenfalls, per Default erfüllt
- So können diese Aufrufe verkettet werden
- Promise, welche unmittelbar resolved wird:
`Promise.resolve(...)`
- Promise, welche unmittelbar rejected wird:
`Promise.reject(...)`

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

52

ÜBUNG: AUSGABE?

```
1 var promise = new Promise((resolve, reject) => {
2   throw Error('fail')
3   resolve()
4 })
5
6 promise
7   .then (() => console.log('step1'))
8   .then (() => { throw Error('fail') })
9   .then (() => console.log('step2'))
10  .catch(() => console.log('catch1'))
11  .then (() => console.log('step3'))
12  .catch(() => console.log('catch2'))
13  .then (() => console.log('step4'))
```

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

53

PROMISES VERKNÜPFEN

- `Promise.all()`
 - Erhält Array von Promises
 - Erfüllt mit Array der Resultate, wenn alle erfüllt sind
 - Zurückgewiesen sobald eine Promise zurückgewiesen wird
- `Promise.race()`
 - Erhält Array von Promises
 - Erfüllt sobald eine davon erfüllt ist
 - Zurückgewiesen sobald eine davon zurückgewiesen wird

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

54

ASYNC / AWAIT

- Asynchrone Funktionen
- Grundlage: Promise API
- Eingeführt mit ES8 (ES2017)
- Grund: Einsatz von Promises immer noch kompliziert
- Nun: asynchroner Code ähnlich synchronem Code aufgebaut

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

55

ASYNC/AWAIT: BEISPIEL 1

```
1 /* Bekanntes Beispiel */
2 const readHosts = () => {
3   readFilePromise('/etc/hosts')
4     .then(console.log)
5     .catch(() => {
6       console.log("Error reading file")
7     })
8 }
```

```
1 /* Mit async/await */
2 const readHosts = async () => {
3   try {
4     console.log(await readFilePromise('/etc/hosts'))
5   }
6   catch (err) {
7     console.log("Error reading file")
8   }
9 }
```

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

56

ASYNC/AWAIT: BEISPIEL 2

```
1 function resolveAfter2Seconds (x) {  
2   return new Promise(resolve => {  
3     setTimeout(() => {  
4       resolve(x)  
5     }, 2000)  
6   })  
7 }  
8  
9 async function add1(x) {  
10  var a = resolveAfter2Seconds(20)  
11  var b = resolveAfter2Seconds(30)  
12  return x + await a + await b  
13 }  
14  
15 add1(10).then(console.log)
```

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

57

PROMISE API VON FS

- Ab Node.js 10
- Bisher: Callback oder Promise selber bauen
- Nun: viele fs-Methoden mit Promise-Rückgabe

```
1 const {readFile} = require("fs").promises  
2  
3 readFile("file.txt", "utf8")  
4   .then(text => console.log("The file contains:", text))
```

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

58

QUELLEN

- Marijn Haverbeke: Eloquent JavaScript, 3rd Edition
<https://eloquentjavascript.net/>
- Ältere Slides aus WEB2 und WEB3
- Dokumentationen, u.a. zu Node.js

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

59

LEESTOFF

Geeignet zur Ergänzung und Vertiefung

- Einzelne Abschnitte in Kapitel 11 von:
Marijn Haverbeke: Eloquent JavaScript, 3rd Edition
<https://eloquentjavascript.net/>

Copyright by Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)

60