

5 Fortgeschrittene Themen

5.1 Hashing von Passwörtern mit bcrypt

5.2 Token-basierte Authentifizierung mit JSON Web Token

5.3 Techniken der asynchronen Programmierung

5.4 Bidirektionale Kommunikation mit WebSockets

- Passwörter sollten nie im Klartext abgelegt werden
 - Wird ein Datenbankserver kompromittiert, so kann ein Angreifer die Passwörter direkt auslesen
 - Ein Angreifer kann ein ausgelesenes Passwort oftmals auch für andere Webanwendungen nutzen
 - Viele Benutzer verwenden dasselbe Passwort für unterschiedliche Webanwendungen („Passwort-Recycling“)
 - Dies ist besonders problematisch, wenn zur Benutzererkennung E-Mail-Adressen verwendet werden (statt individuelle Benutzernamen)
- Hohe Sorgfaltspflicht im Umgang mit Passwörtern gilt auch für Anbieter von Webanwendungen mit niedrigem Schutzbedarf

Hashing von Passwörtern

- Beim Hashing eines Passwortes wird dieses mittels einer **kryptografischen Hashfunktion** auf einen sogenannten Hash abgebildet – eine Zeichenfolge fester Länge

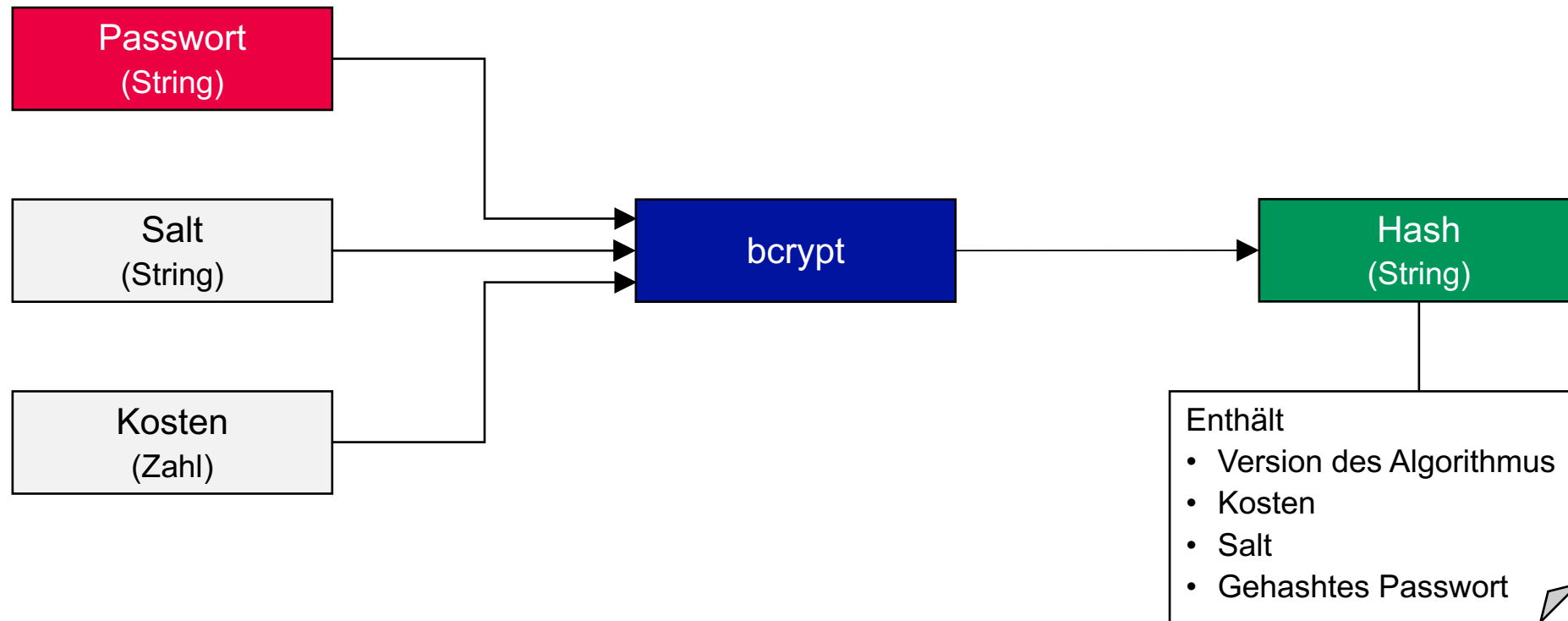


- Die Hashfunktion ist dabei grundsätzlich nicht umkehrbar, sodass für einen Hash das zugehörige Passwort grundsätzlich nicht bestimmt werden kann

Hashing mit bcrypt

- bcrypt ist eine kryptografische Hashfunktion, die speziell für das Hashen und Speichern von Passwörtern entwickelt wurde
- Sie wurde von Niels Provos und David Mazieres auf der USENIX Konferenz 1999 vorgestellt
 - Siehe <https://www.usenix.org/legacy/event/usenix99/provos/provos.pdf>
- Wesentliche Eigenschaften
 - Adaptiv durch **Kostenfaktor**, um sich an verbesserter Hardware anzupassen
 - Verwendet einen 128-bit **Salt**, um Wörterbuch-Attacken zu erschweren
 - Resultierender Hash enthält den Salt, sodass dieser nicht zusätzlich zum Hash gespeichert werden muss

Hashing mit bcrypt

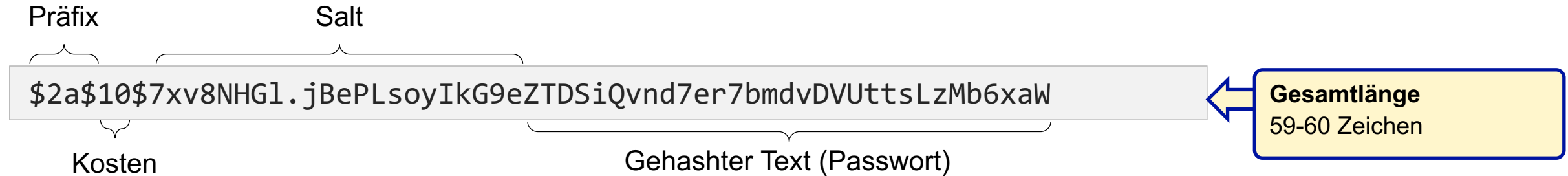


Hashing mit bcrypt

Anatomie eines bcrypt-Hashes

5 Fortgeschrittene Themen

5.1 Hashing von Passwörtern mit bcrypt



| Bestandteil | Erläuterung |
|----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Präfix | Gibt den Algorithmus an, 3 – 4 Zeichen <ul style="list-style-type: none"> Für bcrypt: \$2\$, \$2a\$, \$2x\$, \$2b\$ Andere: \$1\$ für MD5, \$5\$ für SHA-256 und \$6\$ für SHA-512 |
| Kosten | Kostenfaktor für die Berechnung (2^{Kosten} Iterationen) |
| Salt | 128 Bit-Wert in BSD Base64-Kodierung (22 Zeichen je 6 Bit) |
| Gehashter Text | 184 Bit-Wert in BSD Base64-Kodierung (31 Zeichen je 6 Bit) |

bcrypt in Node.js

Installation des Packages

5 Fortgeschrittene Themen

5.1 Hashing von Passwörtern mit bcrypt

- Es gibt mehrere Implementierungen unter Node.js
- Hier: Package bcryptjs (inkl. TypeScript-Definitionsdatei)

```
$ npm i bcryptjs; npm i -D @types/bcryptjs
```

- Vorteile
 - Vollständig in JavaScript implementiert
 - Keine Abhängigkeiten zu anderen Packages
 - Anders als z. B. das Package bcrypt, das unter Windows Probleme bei der Installation bereitet

bcrypt in Node.js

Erstellung eines Hashes (synchron)

5 Fortgeschrittene Themen

5.1 Hashing von Passwörtern mit bcrypt

- Mit expliziter Salt-Generierung (zur Wiederverwendung)

```
import * as bcrypt from 'bcryptjs';

const cost = 10;
const salt = bcrypt.genSaltSync(cost); // prefix + cost + salt
const password = 'secret-password';
const hash = bcrypt.hashSync(password, salt);
```

- Mit impliziter Salt-Generierung

```
import * as bcrypt from 'bcryptjs';

const cost = 10;
const password = 'secret-password';
const hash = bcrypt.hashSync(password, cost);
```


bcrypt in Node.js

Erstellung eines Hashes (asynchron)

5 Fortgeschrittene Themen

5.1 Hashing von Passwörtern mit bcrypt

- Mit expliziter Salt-Generierung (zur Wiederverwendung)

```
import * as bcrypt from 'bcryptjs';

const cost = 10;
bcrypt.genSalt(cost, (err, salt) => {
  bcrypt.hash(password, salt, (err, hash) => { /*...*/ });
});
```

- Mit impliziter Salt-Generierung

```
import * as bcrypt from 'bcryptjs';

const cost = 10;
bcrypt.hash(password, cost, (err, hash) => { /*...*/ });
```

bcrypt in Node.js

Überprüfen



■ Synchroner Variante

```
const input = 'secret-password';  
const isValid = bcrypt.compareSync(input, hash); // true oder false
```

■ Asynchroner Variante

```
const input = 'secret-password';  
bcrypt.compare(input, hash, (err, isValid) => {  
  // ...  
  console.log('is valid: ' + isValid);  
});
```

5 Fortgeschrittene Themen

5.1 Hashing von Passwörtern mit bcrypt

5.2 Token-basierte Authentifizierung mit JSON Web Token

5.3 Techniken der asynchronen Programmierung

5.4 Bidirektionale Kommunikation mit WebSockets

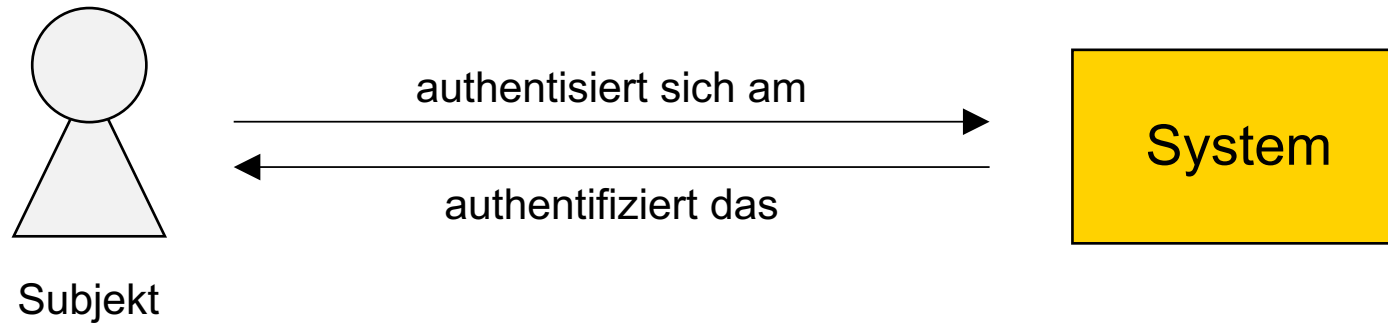
Einleitung

Authentisierung und Authentifizierung

5 Fortgeschrittene Themen

5.2 Token-basierte Authentifizierung mit JSON Web Token

- Die **Authentisierung** ist der Nachweis der Identität eines Subjekts (Benutzer oder System), während die **Authentifizierung** die Verifizierung dieses Nachweises ist (im Englischen heißt beides **Authentication**)



- Frage:** Welche Möglichkeiten der Authentisierung kennen Sie?

Einleitung

Authentisierung und Authentifizierung

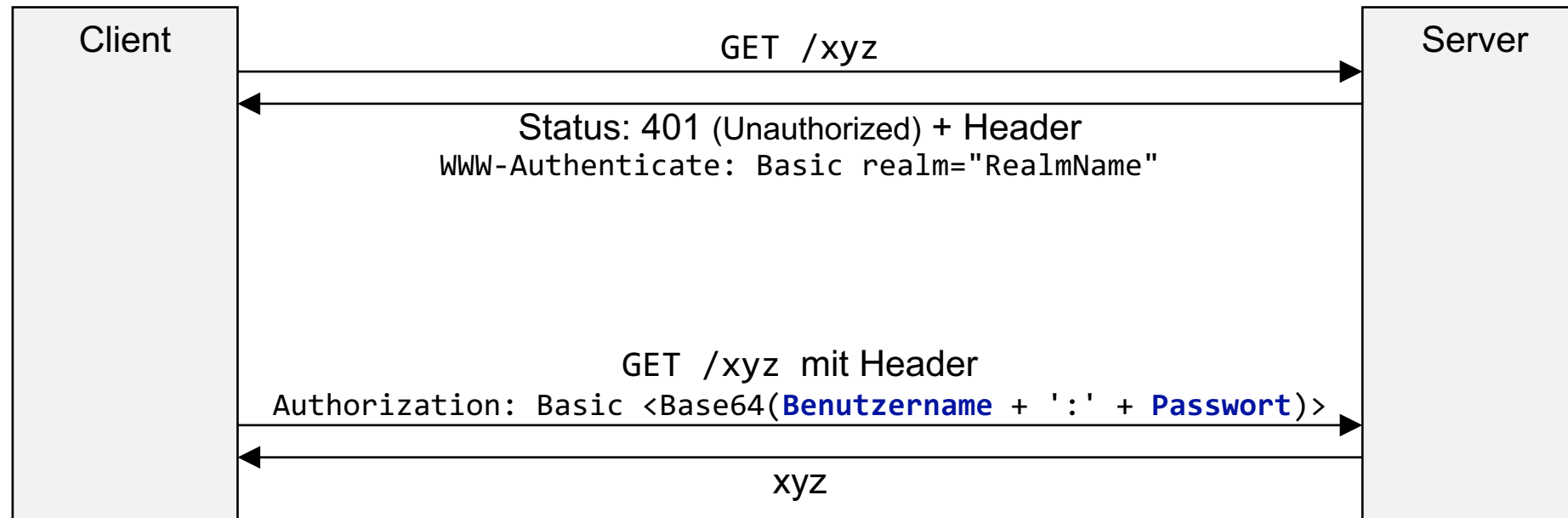
5 Fortgeschrittene Themen

5.2 Token-basierte Authentifizierung mit JSON Web Token

- Für die Entwicklung einer Webanwendung existieren hierzu mehrere Optionen, u. a.
 - Eigenentwicklung
 - Nutzung des Authentifizierungs-Mechanismus anderer Webanwendungen (z. B. Facebook Connect)
 - Nutzung eines allgemeinen Authentifizierungs-Service mit Single Sign-on (Standard: OpenID)
- Klassische Ansätze bei der Eigenentwicklung
 - HTTP Basic Authentication
 - Form-based Authentication

Klassische Ansätze

HTTP Basic Authentication



Klassische Ansätze

HTTP Basic Authentication

Merkmale

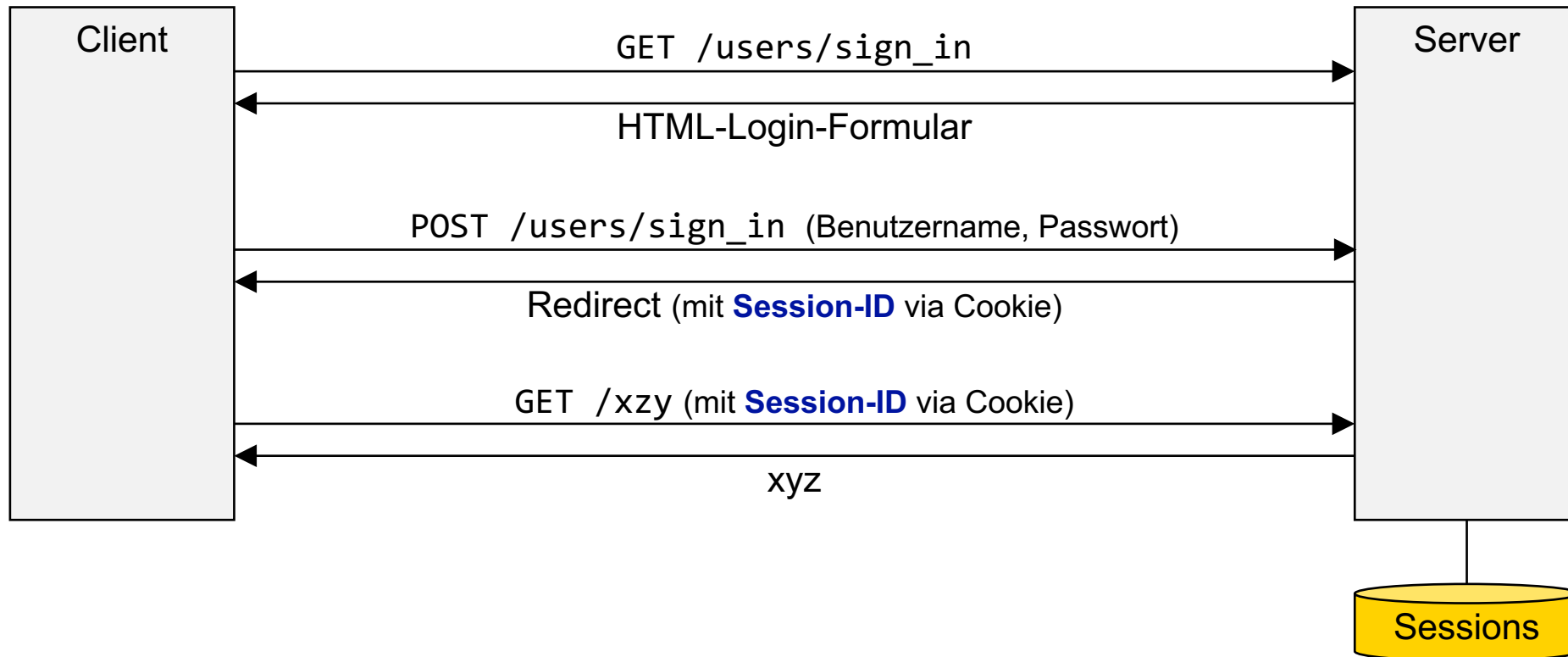
- Ist ein sehr einfach umzusetzendes, standardisiertes Verfahren
RFC 2617 (<https://www.ietf.org/rfc/rfc2617.txt>)
- Benutzername + Passwort werden per Browser-Standardformular erfragt
- Zustandslos, d.h., Authorization-Header muss mit jeder Anfrage erneut geschickt werden
- Keine Verschlüsselung der Daten (→ mit HTTPS kombinieren)
- Kein explizites Abmelden möglich

Klassische Ansätze

Form-based Authentication (klassische Ausgestaltung)

5 Fortgeschrittene Themen

5.2 Token-basierte Authentifizierung mit JSON Web Token



Die Session-ID kann alternativ als Teil der **URL** oder in einem **HTTP-Header** übertragen werden. Letzteres erfordert JavaScript auf Client-Seite.

Diskutieren Sie zu zweit folgende Fragen:

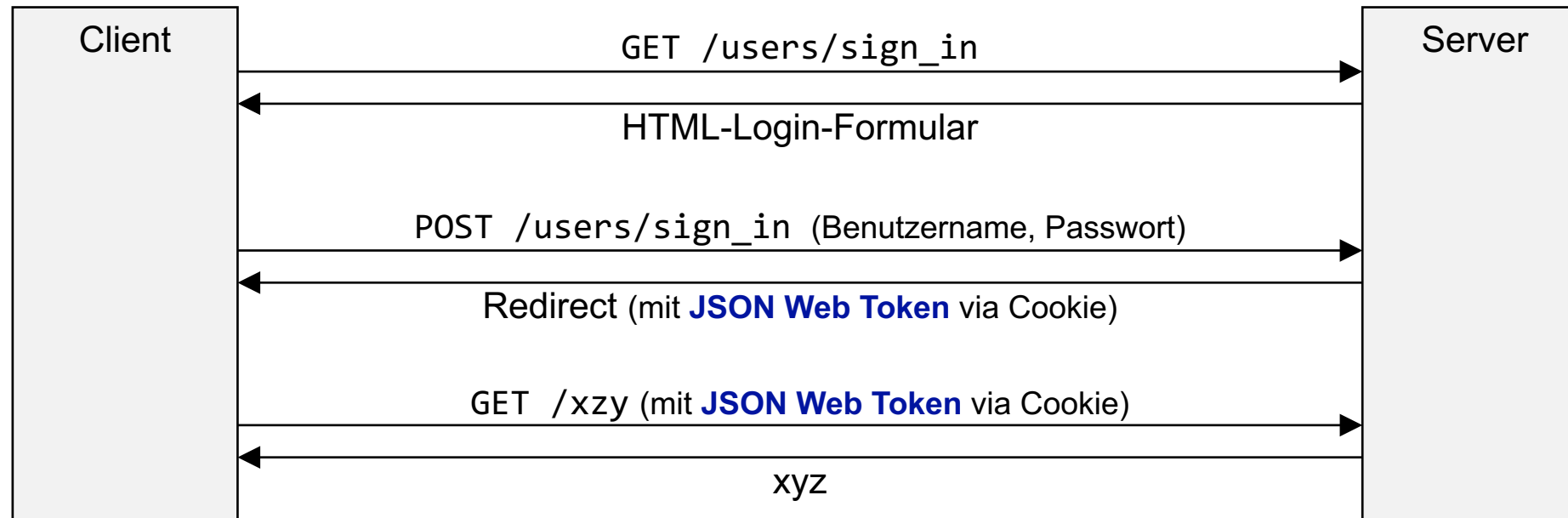
- Gibt es auch Nachteile der Form-based Authentication (klassische Ausgestaltung) im Vergleich zur HTTP Basic Authentication? Wenn ja, welche?
- Gibt es neben Cookies auch weitere Möglichkeiten, die Session-ID zu übertragen? Falls ja, welche Vor- und Nachteile sind damit verbunden?

JSON Web Token

Form-based Authentication (mit JSON Web Token)

5 Fortgeschrittene Themen

5.2 Token-basierte Authentifizierung mit JSON Web Token



Das JSON Web Token kann alternativ als Teil der **URL** oder in einem **HTTP-Header** übertragen werden. Letzteres erfordert JavaScript auf Client-Seite.

*z. B. Authorization: Bearer <token>

JSON Web Token

Definition



- JSON Web Token (JWT*) ist
 - ein Standard für die signierte und/oder verschlüsselte Repräsentation von **Claims** (Aussagen, z. B. über einen angemeldeten Benutzer)
 - ein nach diesem Standard erstelltes Token (String)
- Spezifikation
 - RFC 7519 (<https://tools.ietf.org/html/rfc7519>)

*JWT wird wie das englische Wort *jot* ausgesprochen

JSON Web Token

Wesentliche Merkmale

- Token ist ein URL-sicherer String
- Token ist kompakt
 - für die Nutzung z. B. in Authorization-Headern und URI Anfragestrings gedacht
- Es existieren standardisierte Namen für Claims
 - siehe <http://www.iana.org/assignments/jwt/jwt.xhtml>
- Es existieren Bibliotheken für verschiedene Programmiersprachen
 - C, Go, Haskell, Java, JavaScript, Python, Ruby, Scala, Swift etc.
- Basiert auf den Standards
 - JSON für die Repräsentation der Claims
 - JSON Web Signature (**JWS**, RFC 7515, <https://tools.ietf.org/html/rfc7515>)
 - JSON Web Encryption (**JWE**, RFC 7516, <https://tools.ietf.org/html/rfc7516>)

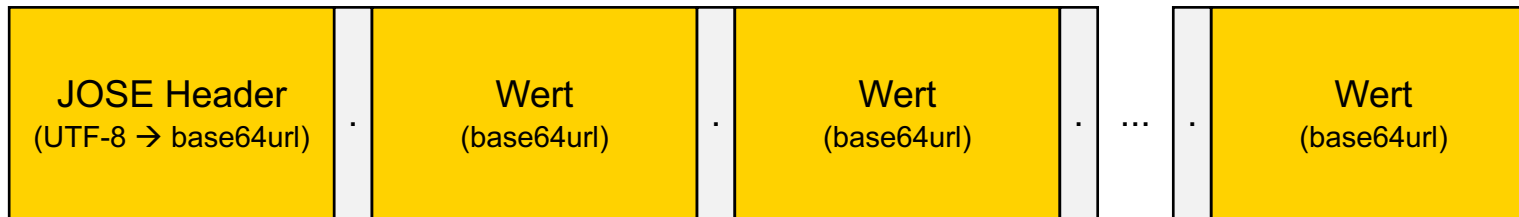
JSON Web Token

Genereller Aufbau

5 Fortgeschrittene Themen

5.2 Token-basierte Authentifizierung mit JSON Web Token

- Ein JWT repräsentiert die Claims in Form eines JSON-Objekts, welches in einer JWS und/oder JWE Struktur kodiert wird
 - Jeweils in der kompakten Serialisierung, nicht JSON-Serialisierung
- Ergebnis ist stets ein **String**: eine Sequenz von **base64url**-kodierten Werten, die durch einen Punkt voneinander getrennt sind

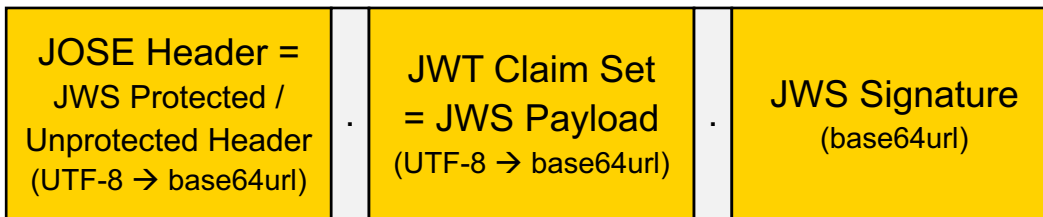


- Der erste Wert ist ein JOSE Header (JSON Object Signing and Encryption)
- Anzahl der Werte hängt von der Struktur ab (JWS und/oder JWE)

JSON Web Token

Strukturspezifischer Aufbau

■ Bei Verwendung einer JWS-Struktur



■ Bei Verwendung einer JWE-Struktur



■ Bei Verwendung einer Verschachtelten Struktur („Nested JWT“)

- Hierbei wird ein JWT in JWS- oder JWE-Struktur als Payload einer JWE- oder als Plaintext für die Erstellung des Ciphertext (=verschlüsselter Text) einer JWS-Struktur verwendet

JSON Web Token

JOSE Header

- Der JOSE Header beschreibt u. a. die Parameter des kryptografischen Algorithmus in Form eines JSON-Objekts
- Die Parameternamen sind standardisiert
- Typische Parameternamen

| Name | Erläuterung | Beispiel |
|------|------------------------------------------------------|----------|
| alg | Name des kryptografischen Algorithmus | "HS256" |
| typ | Media-Typ. Sollte gemäß Spezifikation immer JWT sein | "JWT" |

- Anmerkung
 - Anhand des **alg**-Parameters kann man erkennen, ob es sich um eine JWS oder JWE-Struktur handelt

[*http://www.iana.org/assignments/jose/jose.xhtml](http://www.iana.org/assignments/jose/jose.xhtml)

JSON Web Token

JOSE Header



- Beispiel

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

Signiert mit HMAC SHA-256.

- In UTF-8 und base64url-kodiert

```
eyJhbGciOiJIUzUxMiIsInR5cCI6IkpXVCJ9
```


JSON Web Token

JWT Claim Set



- Das JWT Claims Set ist ein JSON Objekt, dessen Eigenschaften die Claims beschreiben
 - Eigenschaftsname = **Claim Name**
 - Eigenschaftswert = **Claim Value**
- Klassen von JWT Claim Names
 - Standardisierte* Namen (Registered Claim Names)
 - Z. B. "iss" (Issuer), "iat" (Issued At), "exp" (Expiration Time)
 - Öffentliche Namen (Public Claim Names)
 - Frei definierbare, global eindeutige Namen für die öffentliche Nutzung
 - Private Namen (Private Claim Names)
 - Frei definierbare Namen für die private (nicht öffentliche) Nutzung

*<http://www.iana.org/assignments/jwt/jwt.xhtml>

JSON Web Token

JWT Claim Set



■ Beispiel

```
{  
  "iat": 1475232583813,  
  "name": "john doe",  
  "email": "john.doe@example.org",  
  "http://example.org/is_root": true  
}
```

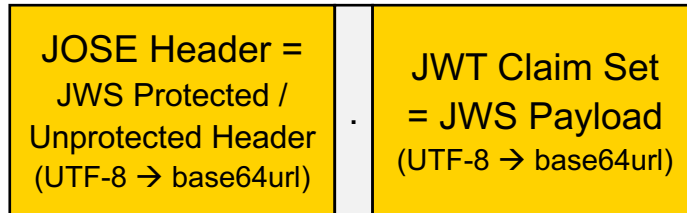
■ In UTF-8 und base64url-kodiert

```
eyJpYXQiOiJlE0NzUyMzI1ODM4MTMsIm5hbWUiOiJqb2huIGRvZSIsImVtYWlsIjoiam9obi5kb  
2VAZXhhbXBsZS5vcmcilLCJodHRwOi8vZXhhbXBsZS5vcmcvaXNfcm9vdCI6dHJ1ZX0
```

JSON Web Token

JWS Signature

- Input für die Signierung
 - Secret oder privater Schlüssel (je nach Algorithmus)
 - JWS Signing Input



- Bei Verwendung von HMAC SHA-256 und dem Secret "mysecret" ergibt sich für das Beispiel

```
YDabDp2jaevfLKDJ2XNJfwtyn3g690maQPCHdM1azY8
```

JSON Web Token

Vollständiges Token



- Für das Beispiel ergibt sich das folgende JSON Web Token

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpYXQiOiJlbnR5bWZlI0DM4MTMsIm5hbWUiOiJqb2huIGRvZSIsImVtYWlsIjoiam9obi5kb2VAZXhhbXBsZS5vcmcilCJodHRwOi8vZXhhbXBsZS5vcmcvaXNfcm9vdCI6dHJ1ZX0.YDabDp2jaevfLKDJ2XNJfwtyn3g690maQPCHdM1azY8
```

JWT in Node.js

Erstellung eines Tokens

5 Fortgeschrittene Themen

5.2 Token-basierte Authentifizierung mit JSON Web Token

- Es gibt mehrere Implementierungen unter Node.js
- Hier: Package jsonwebtoken (inkl. TypeScript-Definitionsdatei)

```
$ npm i jsonwebtoken; npm i -D @types/jsonwebtoken
```

- Vorteil
 - Ausgereifte Implementierung
 - Sehr beliebt (ca 2 Mio Downloads pro Monat)

JWT in Node.js

Erstellung eines Tokens

5 Fortgeschrittene Themen

5.2 Token-basierte Authentifizierung mit JSON Web Token

- Bei Verwendung eines Secrets (hier mit HMAC SHA-256, asynchron)

```
import * as jwt from 'jsonwebtoken';

let claimSet = {
  iat: 1475232583813,
  name: "john doe",
  email: "john.doe@example.org",
  "http://example.org/is_root": true
}

jwt.sign(claimSet, 'mysecret', { algorithm: 'HS256' }, (err, token) => {
  // ...
});
```

JWT in Node.js

Erstellung eines Tokens

5 Fortgeschrittene Themen

5.2 Token-basierte Authentifizierung mit JSON Web Token

- Bei Verwendung eines Schlüssels (hier mit RSA SHA-256, synchron)

```
import * as jwt from 'jsonwebtoken';
import * as fs from 'fs';

let claimSet = {
  iat: 1475232583813,
  name: "john doe",
  email: "john.doe@example.org",
  "http://example.org/is_root": true
}

let cert = fs.readFileSync('private.key');
let token = jwt.sign(claimSet, cert, { algorithm: 'RS256' });
```

JWT in Node.js

Validierung eines Tokens

5 Fortgeschrittene Themen

5.2 Token-basierte Authentifizierung mit JSON Web Token


- Bei Verwendung eines Secrets (hier asynchron)

```
jwt.verify(token, 'mysecret', (err, claimSet) => {  
  // ...  
});
```

- Bei Verwendung eines Schlüssels (hier synchron)

```
const cert = fs.readFileSync('public.pem');    // public key  
try {  
  let claimSet = jwt.verify(token, cert);  
} catch (error) {  
  // ...  
}
```


Überlegen Sie sich, wie Sie eine JWT-basierte Authentifizierung realisieren würden. Leitfragen dabei:

- Wann wird ein Token erstellt und wann wird es verifiziert? 
- Welche Daten soll das Claim Set enthalten?

5 Fortgeschrittene Themen

5.1 Hashing von Passwörtern mit bcrypt

5.2 Token-basierte Authentifizierung mit JSON Web Token

5.3 Techniken der asynchronen Programmierung

5.4 Bidirektionale Kommunikation mit WebSockets

Asynchrone Programmierung

Motivation

5 Fortgeschrittene Themen

5.3 Techniken der asynchronen Programmierung

■ Synchrones Einlesen einer Datei (Node.js)

```
import * as fs from 'fs';
let data: string;
try {
  data = fs.readFileSync('./content.txt', 'utf-8');
} catch (err) {
  console.log(err);
}
if (data) {
  console.log(data);
}
console.log('done!');
```

■ Frage

- Welche Probleme sehen Sie bei einem synchronen Aufruf dieser Art?

- Beim Aufruf einer asynchronen Operation wird dieser eine Funktion (Callback) mitgegeben, die nach Abschluss der Operation aufgerufen wird
- Beispiel

```
import * as fs from 'fs';

fs.readFile('./content.txt', 'utf-8', (err, data) => {
  if (err) {
    console.log(err);
  } else {
    console.log(data);
  }
});

console.log('started');
```

Callbacks

Problem: Callback Hell

```
import * as fs from 'fs';

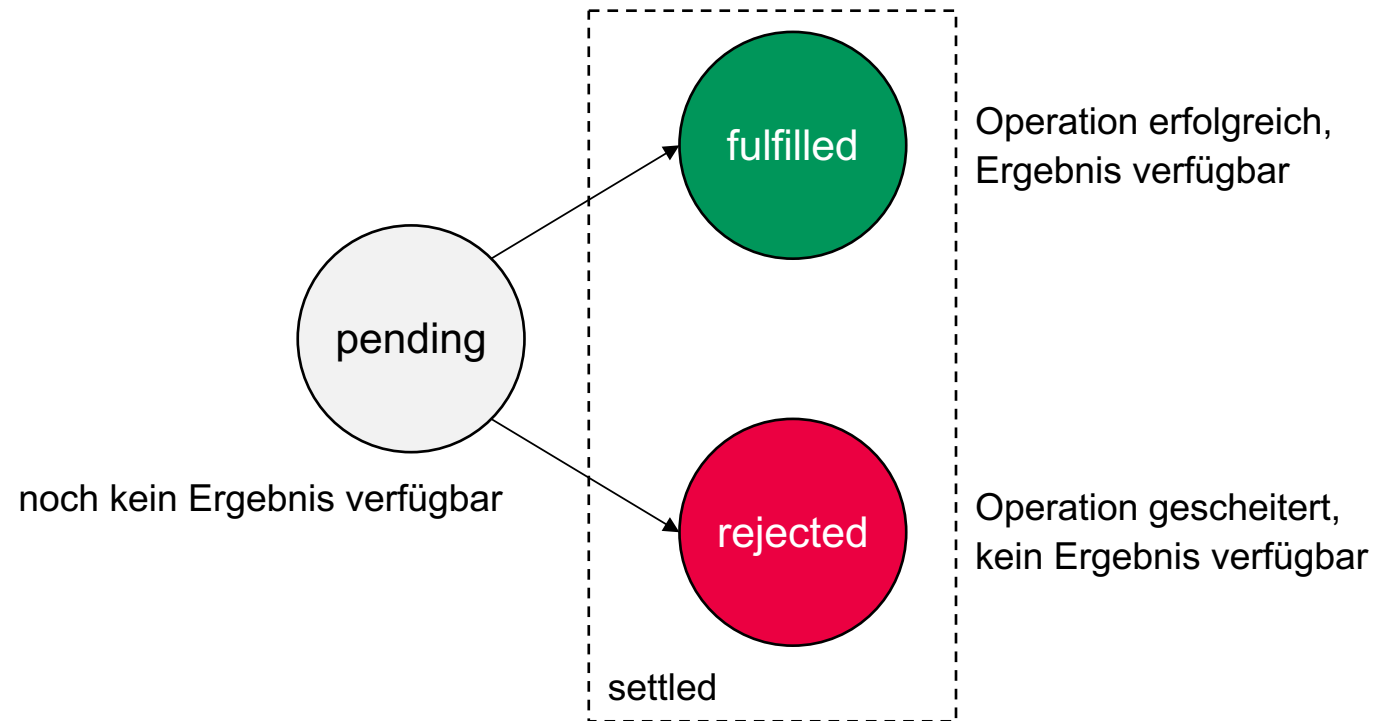
fs.readFile('./content1.txt', 'utf-8', (err, content1) => {
  if (err) return console.log(err);
  fs.readFile('./content2.txt', 'utf-8', (err, content2) => {
    if (err) return console.log(err);
    const content = content1 + '\n' + content2;
    fs.writeFile('./content.txt', content, 'utf-8', (err) => {
      if (err) return console.log(err);
      console.log('done');
    });
  });
});

console.log('started');
```

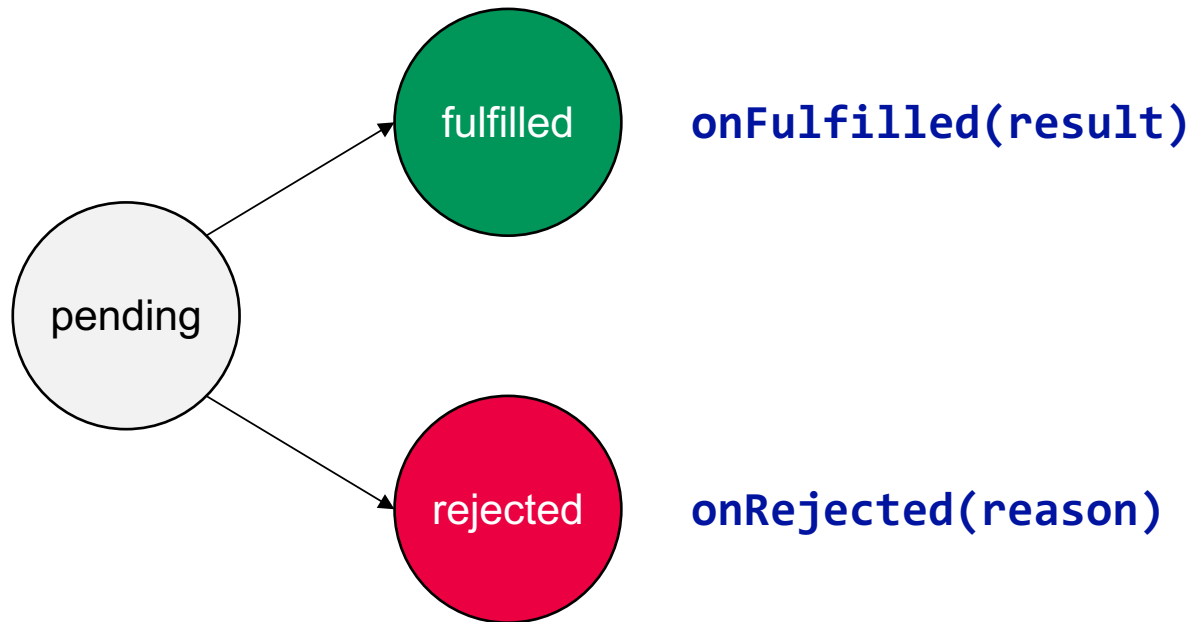
Diskutieren Sie zu zweit folgende Fragen

- Was ist das Problem an der “Callback Hell”?
- Wie könnte man das Problem lösen?

- Ein Promise ist ein Objekt, das das Ergebnis einer (meist asynchronen) Operation repräsentiert. Es kann folgende Zustände haben:



- Ein Promise informiert über eine Zustandsänderung, indem es registrierte Callbacks aufruft



Promises

Erzeugung

5 Fortgeschrittene Themen

5.3 Techniken der asynchronen Programmierung

- Ein Promise wird i.d.R. über den Konstruktor erzeugt, dem eine Funktion übergeben wird, die die Operation implementiert

```
let promise = new Promise<string>(
  (resolve, reject) => {
    // onFulfilled-Callbacks aufrufen
    resolve('Ergebnis');
  }
);
```

} Funktion, die die
Operation implementiert

- Die übergebene Funktion wird **synchron** (sofort) aufgerufen
- Typparameter des Konstruktors = Typ des Ergebnisses der Operation

Promises

Erzeugung

■ Parameter der übergebenen Funktion

| Parameter | Erläuterung |
|-----------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| resolve | Mit Aufruf dieser Funktion wird das Promise in den Zustand fulfilled versetzt. Ferner werden alle onFulfilled-Callbacks informiert. Sie wird mit dem Ergebnis der Operation aufgerufen, z. B. <code>resolve('Ergebnis');</code> |
| reject | Mit Aufruf dieser Funktion wird das Promise in den Zustand rejected versetzt. Ferner werden alle onRejected-Callbacks informiert. Sie wird mit einem Objekt aufgerufen, das den Grund des Fehlschlags darstellt, z. B: <code>reject(new Error('Fehler'));</code> |

■ Anmerkung

- Wird in der übergebenen Funktion ein Error geworfen, dann wechselt das Promise in den Zustand rejected und die onRejected-Callbacks werden aufgerufen

Promises

Erzeugung

- Nutzung für asynchrone Operationen
 - Zu einer asynchronen Funktion, die auf dem Callback-Mechanismus basiert, kann leicht eine Promise-basierte Funktion erzeugt werden
- Beispiel

```
function readFileAsync(filename: string): Promise<string> {  
    return new Promise<string>((resolve, reject) => {  
        fs.readFile(filename, 'utf-8', (error, data) => {  
            if (error) {  
                reject(error);  
            } else {  
                resolve(data);  
            }  
        });  
    });  
}
```

Promises

Erzeugung: Hilfsfunktionen für „triviale“ Promises

5 Fortgeschrittene Themen

5.3 Techniken der asynchronen Programmierung

- Promise erzeugen, das sofort in den Zustand fulfilled wechselt

```
Promise.resolve('Ergebnis');
```

- Promise erzeugen, das sofort in den Zustand rejected wechselt

```
Promise.reject(new Error('Fehler'));
```

Promises

Registrierung von Callbacks

5 Fortgeschrittene Themen

5.3 Techniken der asynchronen Programmierung

- Über die Funktion `then` können die Callbacks registriert werden

```
let p = readFileAsync('content.txt');

p.then(
  result => { // onFulfilled-Callback
    console.log(result);
  },
  error => { // onRejected-Callback
    console.log(error);
  }
);
```

- Die Funktion `then` kann auch mehrmals aufgerufen werden
 - Die Callbacks werden dann in der Reihenfolge ihrer Registrierung aufgerufen

Promises

Registrierung von Callbacks

5 Fortgeschrittene Themen

5.3 Techniken der asynchronen Programmierung

- Über die Funktion `catch` wird nur ein `onRejected`-Callback registriert

```
let p = new Promise<string>((resolve, reject) => {  
    reject(new Error('Fehler'));  
});  
  
p.catch(error => {  
    console.log(error);  
});
```

- `catch` ist eine Abkürzung für `then(undefined, onRejected)`

Promises

Aufruf der Callbacks

5 Fortgeschrittene Themen


5.3 Techniken der asynchronen Programmierung

- Ein Callback wird höchstens einmal aufgerufen
- Ein Callback wird auch dann aufgerufen, wenn er *nach* dem Zustandswechsel registriert wird, jedoch **asynchron**

```
let p = new Promise<string>((resolve, reject) => {  
  console.log('Step 1');  
  resolve('Ergebnis');  
});  
console.log('Step 2');  
p.then(value => {  
  console.log('Step 3');  
});  
console.log('Step 4');
```

Asynchron

D. h., nach der aktuellen
Event-Loop-Iteration



Step 1
Step 2
Step 4
Step 3

Promises

Verkettung von Promises

5 Fortgeschrittene Themen

5.3 Techniken der asynchronen Programmierung

- Die Funktionen `then` und `catch` liefern ein **neues** Promise-Objekt zurück

```
let promise2 = promise1.then(onFulfilled, onRejected);
```

- Erlaubt eine Verkettung

```
readFileAsync('content.txt')  
  .then(result => {  
    console.log(result);  
  })  
  .catch(error => {  
    console.log(error);  
  });
```


Promises

Verkettung von Promises

5 Fortgeschrittene Themen

5.3 Techniken der asynchronen Programmierung

■ Beide Callbacks sind optional

```
let promise2 = promise1.then(undefined, onRejected);
```

- Wenn promise1 in den Zustand fulfilled wechselt, dann wechselt auch promise2 in diesen Zustand, und zwar mit demselben Ergebnis

```
let promise2 = promise1.then(onFulfilled, undefined);
```

- Wenn promise1 in den Zustand rejected wechselt, dann wechselt auch promise2 in diesen Zustand, und zwar mit demselben Grund
- **Nutzen:** Man muss in einer Kette von Promises nur einen onRejected-Callback definieren, der dann für alle Operationen gilt

Promises

Verkettung von Promises

5 Fortgeschrittene Themen

5.3 Techniken der asynchronen Programmierung

- Wird in einem Callback ein Fehler geworfen, dann wechselt das **neue** Promise in den Zustand rejected
 - Gilt für beide Callbacks: onFulfilled und onRejected

```
readFileAsync('content.txt')  
  .then(result => {  
    console.log(result);  
    throw new Error('Oops');  
  })  
  .catch(error => {  
    console.log(error);  
  })
```

Promises

Verkettung von Promises

5 Fortgeschrittene Themen

5.3 Techniken der asynchronen Programmierung

- Wird in einem Callback kein Fehler geworfen und ein „normaler“ Wert zurückgeliefert (ggf. implizit undefined), so wechselt das **neue** Promise in den Zustand fulfilled
 - Das Ergebnis des neuen Promises ist der Rückgabewert des Callbacks

```
readFileAsync('content.txt')  
  .then(result => {  
    return result.length;  
  })  
  .then(result => {  
    console.log(result);  
  })  
  .catch(error => {  
    console.log(error);  
  })
```

Promises

Verkettung von Promises

5 Fortgeschrittene Themen

5.3 Techniken der asynchronen Programmierung

- Liefert ein Callback ein Promise (P_R) zurück, dann gilt:
 - Das neue Promise wechselt erst dann seinen Zustand, wenn P_R seinen Zustand wechselt (auch nachträglich)
 - Es hat dann den gleichen Zustand und das gleiche Ergebnis wie P_R
- Damit können auch asynchrone Operationen in den Callbacks aufgerufen werden, und zwar ohne Verschachtelung von then-Aufrufen

```
asyncFunc1()  
  .then((value1) => {  
    asyncFunc2()  
    .then((value2) => {  
      // ...  
    });  
  });
```



```
asyncFunc1()  
  .then((value1) => {  
    return asyncFunc2();  
  })  
  .then((value2) => {  
    // ...  
  });
```

Promises

Beispiel



```
let content1: string;
readFileAsync('content1.txt')
  .then((result) => {
    content1 = result;
    return readFileAsync('content2.txt');
  })
  .then((result) => {
    const content = content1 + '\n' + result;
    return writeFileAsync('content.txt', content); // analog zu readFileAsync
  })
  .then(() => {
    console.log('done');
  })
  .catch(error => {
    console.log(error);
  });
console.log('started');
```

Diskutieren Sie zu zweit folgende Frage in Bezug auf das vorherige Beispiel

- Ist der hier vorgestellte verkettete Einsatz von Promises eine gute Lösung für das Problem, zwei Dateien zu konkatenieren?

Promises

Zusammenführung von Promises

5 Fortgeschrittene Themen

5.3 Techniken der asynchronen Programmierung

- Über die Funktion `Promise.all` wird ein Promise erzeugt, über das andere Promises zusammengeführt werden können
 - Parameter ist ein iterierbares Objekt, das die Promises liefert (typischerweise ein Array von Promises)
- Das erzeugte Promise wechselt genau dann in den Zustand
 - `fulfilled`, wenn alle Promises in den Zustand `fulfilled` wechseln
 - Ergebnis ist dann ein Array der Ergebnisse der Promises
 - `rejected`, wenn mind. ein Promise in den Zustand `rejected` wechselt
 - Ursache (`reason`) ist die des ersten gescheiterten Promises
- Eignet sich gut zum Zusammenführen parallel auszuführender Operationen

Promises

Zusammenführung von Promises

5 Fortgeschrittene Themen

5.3 Techniken der asynchronen Programmierung

■ Beispiel

Promise

```
.all([readFileAsync('content1.txt'), readFileAsync('content2.txt')])  
.then(([content1, content2]) => {  
  const content = content1 + '\n' + content2;  
  return writeFileAsync('content.txt', content);  
})  
.then(() => {  
  console.log('done');  
})  
.catch(error => {  
  console.log(error);  
});
```


Promises

Hilfsfunktion promisify

5 Fortgeschrittene Themen

5.3 Techniken der asynchronen Programmierung

- Node.js bietet seit Version 8 eine Hilfsfunktion `promisify`, um eine Callback-basierte Funktion in eine Funktion mit einem Promise als Rückgabewert umzuwandeln

```
import * as fs from 'fs';
import * as util from 'util';

const readFileAsync: (filename: string, options: object) =>
Promise<string> = util.promisify(fs.readFile) as any;

readFileAsync('content.txt', { encoding: 'utf8' })
  .then(text => {
    console.log('Inhalt:', text);
  })
  .catch(err => {
    console.log('Fehler:', err);
  });
```

- Async-Funktionen (async functions) wurden von Brian Terlson als neues Feature für ECMAScript 2017 vorgeschlagen
- Dabei markiert das Schlüsselwort **async** eine Funktion als asynchron

| Alternative | Beispiel |
|----------------------|-------------------------------------------------|
| Funktionsdeklaration | <code>async function foo() { };</code> |
| Funktionsausdruck | <code>const foo = async function () { };</code> |
| Arrow-Funktion | <code>const foo = async () => {};</code> |
| Methodendefinition | <code>let obj = { async foo() { } };</code> |

Async-Funktionen

Rückgabewert

5 Fortgeschrittene Themen

5.3 Techniken der asynchronen Programmierung

- Der Rückgabewert einer Async-Funktion ist stets ein **Promise**

- Beispiel: Funktion mit explizitem Rückgabewert

```
async function foo() { return 42; }  
  
let result = foo(); // Typ: Promise<number>  
result.then(value => console.log(value)); // 42
```

- Beispiel: Funktion ohne expliziten Rückgabewert

```
async function foo() { console.log('foo'); }  
  
let result = foo(); // Typ: Promise<void>  
result.then(value => console.log(value)); // undefined
```

Async-Funktionen

Rückgabewert

5 Fortgeschrittene Themen

5.3 Techniken der asynchronen Programmierung

- Wird bei return ein Promise P_R angegeben, so wird dennoch ein **neues** Promise zurückgeliefert

```
let myPromise = Promise.resolve(42);

async function foo() { return myPromise; }

let result = foo(); // Typ: Promise<number>
console.log(result === myPromise); // false
result.then(value => console.log(value)); // 42
```

- Es gilt
 - Das neue Promise wechselt erst dann seinen Zustand, wenn P_R seinen Zustand wechselt (auch nachträglich, wie im obigen Beispiel)
 - Es hat dann den gleichen Zustand und das gleiche Ergebnis wie P_R

Async-Funktionen

Verhalten im Fehlerfall

5 Fortgeschrittene Themen

5.3 Techniken der asynchronen Programmierung

- Wirft eine Async-Funktion einen Fehler, dann geht das zurückgelieferte Promise in den Zustand rejected über

```
async function foo() { throw new Error('ops'); }

let result = foo();
result.catch((err: Error) => {
  console.log('Fehler: ' + err.message); // Fehler: ops
});
```

Async-Funktionen

Operator await

5 Fortgeschrittene Themen

5.3 Techniken der asynchronen Programmierung

- Innerhalb einer Async-Funktion (und nur dort) kann mit dem Operator **await** darauf gewartet werden, dass ein Promise den Zustand settled erreicht

```
function asyncRandom(max: number): Promise<number> {  
  let value = Math.floor(Math.random() * (max + 1));  
  return new Promise(resolve => {  
    setTimeout(() => { resolve(value); }, 500);  
  });  
}  
  
async function foo() {  
  for (let i = 1; i <= 5; i++) {  
    let value = await asyncRandom(10);  
    console.log(value);  
  }  
}  
  
foo();
```

ohne await wäre der Datentyp von value Promise<number>

Async-Funktionen

Operator await

5 Fortgeschrittene Themen

5.3 Techniken der asynchronen Programmierung

- Geht das Promise in den Zustand rejected über, dann wird ein Fehler geworfen

```
function waitForError(): Promise<void> {  
    return new Promise<void>((resolve, reject) => {  
        setTimeout(() => { reject(new Error('rejected')); }, 500);  
    });  
}  
  
async function bar() {  
    console.log('started');  
    try {  
        await waitForError();  
    } catch (error) {  
        console.log('Fehler' + error);  
    }  
}  
bar();
```

Async-Funktionen

Synchroner Start

5 Fortgeschrittene Themen

5.3 Techniken der asynchronen Programmierung

- Eine Async-Funktion wird stets **synchron gestartet** und liefert synchron das Promise-Objekt zurück

```
function wait(millis: number): Promise<void> {  
    return new Promise<void>(resolve => { setTimeout(resolve, millis); });  
}  
  
async function foo() {  
    console.log('Schritt 1');  
    await wait(1000);  
    console.log('Schritt 2');  
}  
  
let result = foo();  
console.log('Schritt 3');
```



Schritt 1
Schritt 3
Schritt 2

Async-Funktionen

Beispiel



```
const readFileAsync: (filename: string, options: object) => Promise<string> =
  util.promisify(fs.readFile) as any;

const writeFileAsync: (filename: string, content: string, options: object) => Promise<void> =
  util.promisify(fs.writeFile) as any;

async function concat(fileA: string, fileB: string, outputFile: string) {
  let [contentA, contentB] = await Promise.all([
    readFileAsync(fileA, { encoding: 'utf8' }),
    readFileAsync(fileB, { encoding: 'utf8' })
  ]);
  let content = contentA + contentB;
  return writeFileAsync(outputFile, content, { encoding: 'utf8' });
}

concat('content1.txt', 'content2.txt', 'content.txt')
  .then(() => { console.log('fertig'); })
  .catch((err: Error) => { console.log('Fehler: ' + err); })
```

- Was gibt der folgende Code aus?

```
function wait(millis: number): Promise<void> {  
    return new Promise<void>(resolve => { setTimeout(resolve, millis); });  
}  
  
async function count(prefix: string, limit: number) {  
    for (let i = 1; i <= limit; i++) {  
        await wait(1000);  
        console.log(prefix + i);  
    }  
}  
  
count('A', 3);  
count('B', 3);
```

Observables

Einführung

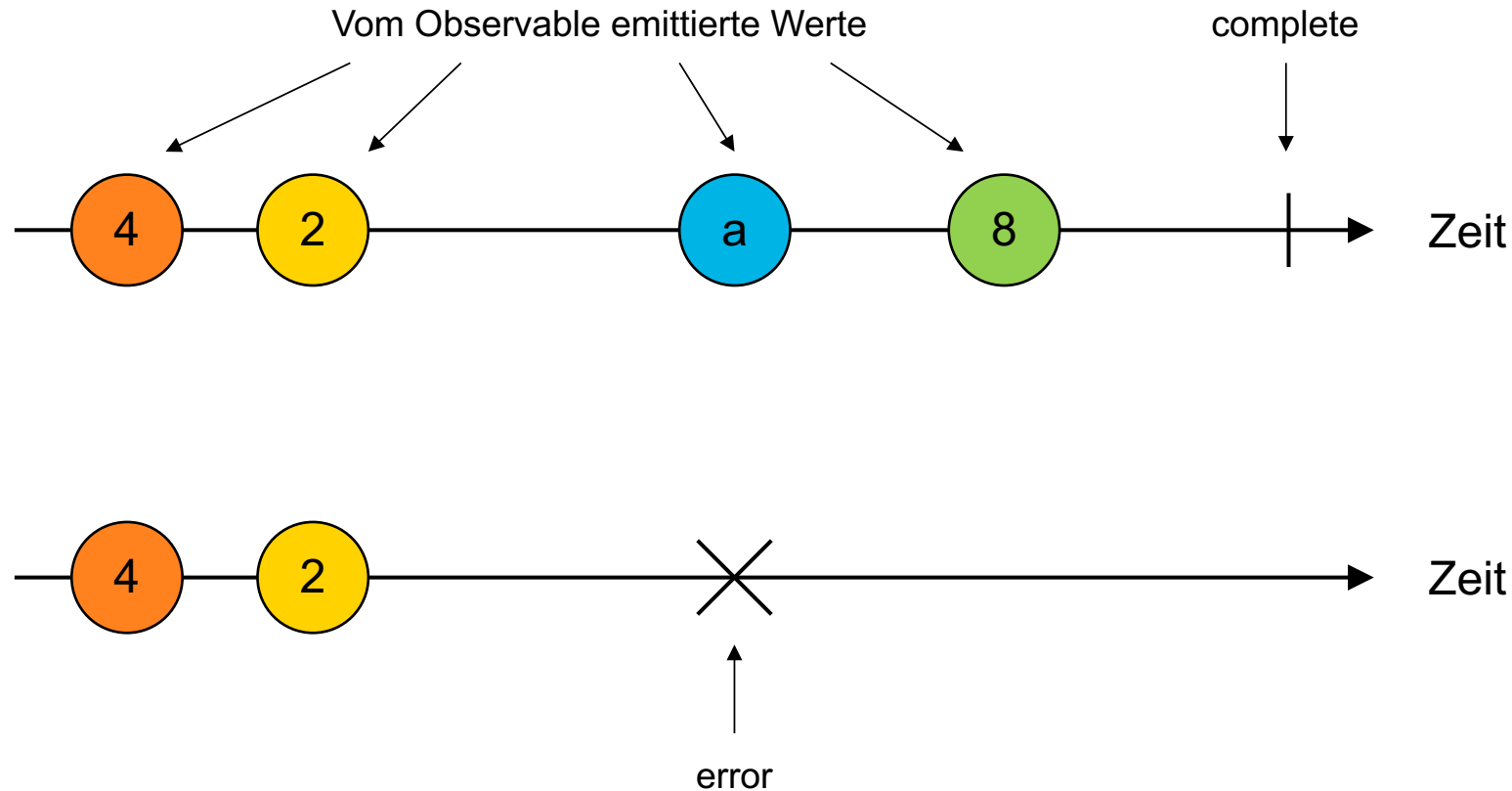
- Ein Observable produziert/emittiert eine Sequenz von Werten
 - Werte können synchron und/oder asynchron emittiert werden
 - Sequenz kann unbegrenzt sein, erfolgreich terminieren (complete) oder aufgrund eines Fehlers abbrechen
 - Werte werden **lazy** emittiert (nur dann, wenn ein Observer vorhanden ist)
- Einordnung

| | Ein Wert | Mehrere Werte |
|------|----------|-------------------|
| Pull | Funktion | Iterator |
| Push | Promise | Observable |

- **Pull**: Consumer bestimmt, wann er die Werte vom Producer erhält
- **Push**: Producer bestimmt, wann er die Werte an den Consumer sendet

Observables

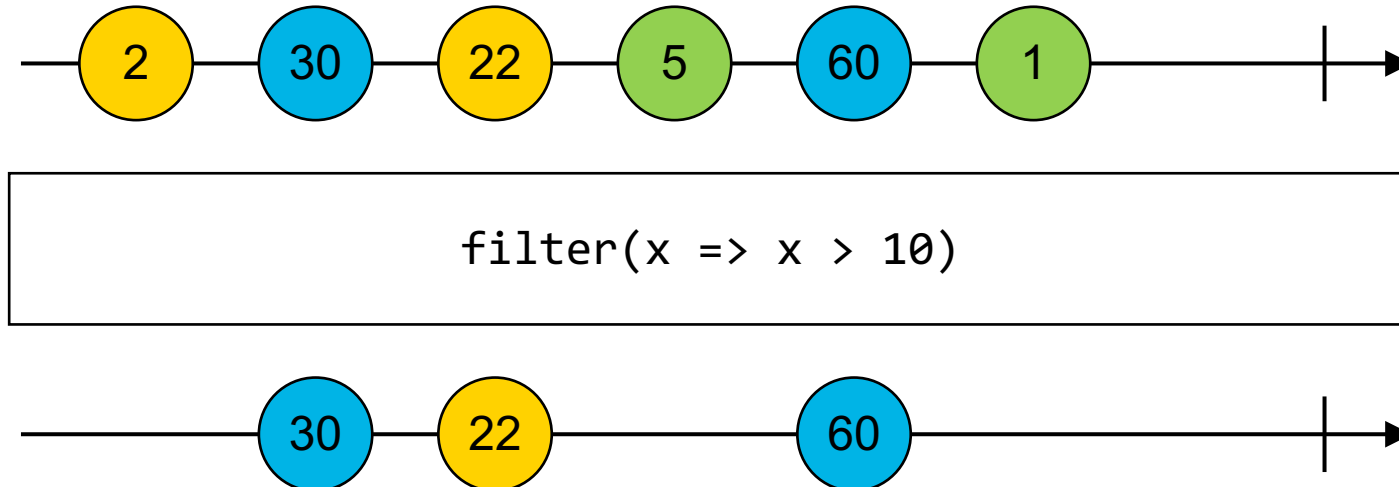
Visualisierung durch Marble-Diagramme



Observables

Operatoren

- Ein Operator erzeugt aus einem oder mehreren Observables ein neues Observable
- Beispiel: filter



Observables

Observer

- Ein Observer konsumiert die Werte, die ein Observable produziert

```
interface Observer<T> {  
    next: (value: T) => void;  
    error: (err: any) => void;  
    complete: () => void;  
}
```

| Funktion | Erläuterung |
|----------|------------------------------------------------------------------------------------------|
| next | Wird für jeden emittierten Wert aufgerufen. Parameter ist der emittierte Wert. |
| error | Wird aufgerufen, wenn die Sequenz mit einem Fehler terminiert. Parameter ist der Fehler. |
| complete | Wird aufgerufen, wenn die Sequenz erfolgreich terminiert. |

Observables

RxJS

- RxJS ist die Standard-Library für Observables in JavaScript
 - <http://reactivex.io/rxjs/>
- Installation als Node.js-Package (TypeScript-Definitionsdateien inkl.)

```
$ npm i rxjs
```

- Anmerkung
 - Es gibt auch Implementierungen für andere Programmiersprachen, wie Java, C#, Scala, C++, Ruby, Python, Swift, PHP
 - <http://reactivex.io>

Observables

RxJS: Erzeugung eines Observables

5 Fortgeschrittene Themen

5.3 Techniken der asynchronen Programmierung

- Beispiel: Observable, das synchron drei Werte emittiert

```
import { Observable } from 'rxjs/Observable';

let observable = new Observable<number>(
  observer => { // subscribe-Funktion
    observer.next(1);
    observer.next(2);
    observer.next(3);
    observer.complete();
  }
);
```

subscribe-Funktion

Wird für **jeden** Observer
ausgeführt!



Observables

RxJS: Erzeugung eines Observables

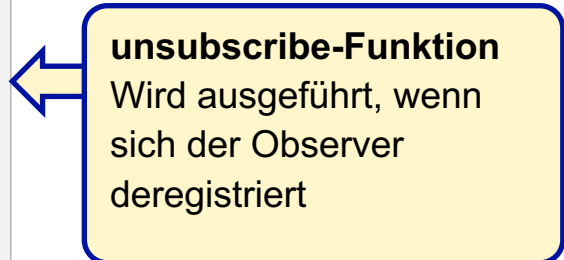
5 Fortgeschrittene Themen

5.3 Techniken der asynchronen Programmierung

- Beispiel: Observable, das asynchron unbegrenzt Werte emittiert

```
import { Observable } from 'rxjs/Observable';

let observable = new Observable<number>(observer => {
  let counter = 0;
  let intervalId = setInterval(() => {
    observer.next(counter++);
  }, 1000);
  return () => { // unsubscribe-Funktion
    clearInterval(intervalId);
  }
});
```



unsubscribe-Funktion
Wird ausgeführt, wenn
sich der Observer
deregistriert

Observables

RxJS: Registrieren eines Observers

5 Fortgeschrittene Themen

5.3 Techniken der asynchronen Programmierung

■ Als Objekt mit 3 Methoden

```
let subscription = observable.subscribe(  
  { // Observer  
    next: value => {  
      console.log('value: ' + value);  
    },  
    error: error => {  
      console.log('error: ' + error);  
    },  
    complete: () => {  
      console.log('complete');  
    }  
  }  
);
```

Observer

Muss mindestens eine
der drei Funktionen
implementieren.

Observables

RxJS: Registrieren eines Observers

5 Fortgeschrittene Themen

5.3 Techniken der asynchronen Programmierung

- Alternativ in Form von 3 Funktionen (alle optional)

```
let subscription = observable.subscribe(  
  value => {  
    console.log('value: ' + value);  
  },  
  error => {  
    console.log('error: ' + error);  
  },  
  () => { console.log('complete'); }  
);
```

- Mit Aufruf von `subscribe` wird (bei beiden Alternativen) die Produktion der Werte angestoßen
 - Bei synchron erzeugten Werten werden die Callbacks synchron aufgerufen, also bevor die Funktion `subscribe` terminiert.

Observables

RxJS: Deregistrieren eines Observers

5 Fortgeschrittene Themen

5.3 Techniken der asynchronen Programmierung

- Erfolgt anhand der Subscription

```
subscription.unsubscribe();
```

- Anmerkungen

- Nach der Deregistrierung produziert das Observable für diesen Observer keine Werte mehr
- War er der letzte registrierte Observer, dann produziert das Observable überhaupt keine Werte mehr

Observables

RxJS: Operatoren

5 Fortgeschrittene Themen

5.3 Techniken der asynchronen Programmierung

- Die Klasse Observables bietet zahlreiche Operatoren
 - Operatoren auf Observables
 - Operatoren zur Erzeugung von Observables
- Beispiel: Operatoren auf Observables

```
import { Observable } from 'rxjs/Observable';
import { map, filter } from 'rxjs/operators';
// ...
let subscription = observable.pipe(
  map(value => (value % 26) + 65),
  filter(value => value % 2 === 1),
  map(value => String.fromCharCode(value))
).subscribe(value => {
  console.log('value: ' + value);
});
```

Observables

RxJS: Operatoren

- Beispiel: Operator `interval` zur Erzeugung einer Zahlensequenz

```
import { merge } from 'rxjs/observable/merge';  
let observable = interval(1000); // jede Sekunde eine Zahl (0, 1, 2, ...)
```

- Beispiel: Operator `merge` zum Vereinigen mehrerer Observables

```
import { interval } from 'rxjs/observable/interval';  
import { merge } from 'rxjs/observable/merge';  
import { map, filter } from 'rxjs/operators';  
  
let o1 = interval(1000).pipe(map(v => String.fromCharCode(65 + v % 26)));  
let o2 = interval(400);  
let merged = merge(o1, o2);  
// jede Sekunde ein Großbuchstabe (A, B, C, ...) und zudem  
// alle 400 ms eine Zahl (0, 1, 2, ...)
```

- Ein Subject ist ein Observable, das ein Multicasting von Werten an mehrere Observer ermöglicht
 - Aus Sicht eines Observers sind normale Observables und Subjects nicht unterscheidbar
 - Die Funktion `subscribe` führt nicht zu einer Ausführung des Observables – es wird lediglich ein Observer als Listener hinzugefügt
- Ein Subject ist gleichzeitig ein Observer
 - Es besitzt die Funktionen `next`, `error` und `complete`
 - Über diese Methoden werden Werte an alle Listener emittiert und diese über einen Fehler bzw. über die erfolgreiche Terminierung informiert

Observables

RxJS: Subjects – Beispiel

5 Fortgeschrittene Themen

5.3 Techniken der asynchronen Programmierung

```
import { Subject } from 'rxjs/Subject';

let subject = new Subject<string>();

subject.subscribe(value => {
  console.log('Observer 1: ' + value);
});

subject.next('foo');

subject.subscribe(value => {
  console.log('Observer 2: ' + value);
});

subject.next('bar');

subject.complete();
```



Observer 1: foo
Observer 1: bar
Observer 2: bar

Observables

RxJS: Subjects



- Da ein Subject auch ein Observer ist, kann jedes Observable über ein Subject in ein Multicasting-Observerable umgewandelt werden

```
import { Subject } from 'rxjs/Subject';
import { interval } from 'rxjs/observable/interval';

let observable = interval(500);

let subject = new Subject<number>();
observable.subscribe(subject);

subject.subscribe(value => console.log('A: ' + value));

setTimeout(() => {
  subject.subscribe(value => console.log('B: ' + value));
}, 2000);
```



```
A: 0
A: 1
A: 2
A: 3
B: 3
A: 4
B: 4
A: 5
B: 5
A: 6
B: 6
A: 7
...
```

Diskutieren Sie zu zweit folgende Fragen

- Was sind Gemeinsamkeiten / Unterschiede von Promises und Observables?
- Lässt sich ein Promise in einen Observable umwandeln?

5 Fortgeschrittene Themen

5.1 Hashing von Passwörtern mit bcrypt

5.2 Token-basierte Authentifizierung mit JSON Web Token

5.3 Techniken der asynchronen Programmierung

5.4 Bidirektionale Kommunikation mit WebSockets

- Das WebSocket-Protokoll ist ein auf TCP basierendes Protokoll für die **bidirektionale** Kommunikation zwischen Client und Server
 - Die bidirektionale Kommunikation erfolgt über **eine** TCP-Verbindung
 - Protokoll: RFC 6455 (<https://tools.ietf.org/html/rfc6455>)
 - URL-Schemata

| Schema | Erläuterung | Beispiel |
|--------|-----------------------------------|------------------------|
| ws | Für unverschlüsselte Verbindungen | ws://example.org:8080 |
| wss | Für verschlüsselte Verbindungen | wss://example.org:8443 |

- Die WebSocket-API ermöglicht Web-Clients die Nutzung des WebSocket-Protokolls
 - <https://www.w3.org/TR/websockets/>

- Verbindung zum Server aufbauen

```
let websocket = new WebSocket('ws://localhost:8080');
```

- Event Handler für eingehende Nachrichten registrieren

```
websocket.onmessage = (event) => {  
  let data = event.data; // ...  
};
```

- Ggf. weitere Event Handler registrieren

| IDL Attribut | Erläuterung |
|--------------|------------------------------------------------------|
| onopen | Wird nach erfolgreichem Verbindungsaufbau aufgerufen |
| onerror | Wird im Fehlerfall aufgerufen |
| onclose | Wird beim Beenden der Verbindung aufgerufen |

WebSockets

WebSocket-API (Client)

5 Fortgeschrittene Themen

5.4 Bidirektionale Kommunikation mit WebSockets

■ Daten versenden

```
websocket.send('Hello');
```

- Parameter kann ein String oder Binärdaten (Blob, ArrayData, ArrayDataView) sein

■ Verbindung schließen

```
websocket.close();
```

WebSockets

Beispiel: Einfacher Chat-Client

5 Fortgeschrittene Themen

5.4 Bidirektionale Kommunikation mit WebSockets

```
<!doctype html>
<html><head><title>Simple Chat</title></head>
<body>
  <input id="message" type="text">
  <button onclick="sendMessage()">Senden</button>
  <div id="messages"></div>
  <script>
    let websocket = new WebSocket('ws://localhost:8080');
    websocket.onmessage = (event) => {
      let element = document.createElement('p');
      element.innerHTML = event.data;
      document.getElementById('messages').appendChild(element);
    };
    function sendMessage() {
      let message = document.getElementById('message').value;
      websocket.send(message);
    }
  </script>
</body>
</html>
```

WebSockets

Server



- Für Node.js eignet sich das Package `ws`

```
$ npm i ws; npm i -D @types/ws
```

- Modul importieren

```
import * as WebSocket from 'ws';
```

- Server erstellen (hier: anhand eines http-Servers)

```
const wss = new WebSocket.Server({ server: httpServer });
```


WebSockets

Server

- Callback für eingehende Verbindungen registrieren

```
wss.on('connection', websocket => {  
  // Callbacks am Socket registrieren  
});
```

- Callback für eingehende Nachrichten registrieren (hier: String-Daten)

```
websocket.on('message', (message: string) => {  
  // ...  
});
```

- Ggf. weitere Callbacks registrieren
 - Weitere Parameter für on sind u. a. 'error' und 'close'
- Daten verschicken (auch Binärdaten möglich)

```
websocket.send('hello');
```

WebSockets

Beispiel: Einfacher Chat-Server

5 Fortgeschrittene Themen

5.4 Bidirektionale Kommunikation mit WebSockets

```
import * as express from 'express';
import * as http from 'http';
import * as WebSocket from 'ws';

const app = express();
app.use(express.static('public'));
const httpServer = http.createServer(app);
httpServer.listen(8080);

const wss = new WebSocket.Server({ server: httpServer });
wss.on('connection', websocket => {
  websocket.on('message', (message: string) => {
    wss.clients.forEach(ws => {
      if (ws !== websocket) { ws.send(message); }
    })
  });
});
```