Introduction SPA

Browser-based Applications

Benefits
- Work from anywhere, anytime
- Platform independent, including mobile
- No software update, no application, easy maintenance
- Software can be provided as a service (SaaS - pay as you go)
- Code separation

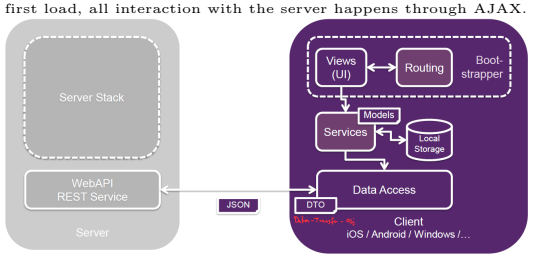
Liabilities
- No data sovereignty (Datenhoheit)
- Limited/restricted hardware access
- SEO - Search engines must execute JavaScript
- More complex deployment strategies

SPA

A website that fits on a single web page with a user experience similar to that of a desktop application. All code is retrieved with a single page load or resources are dynamically loaded. SPAs use AJAX and HTML5 to create responsive Web apps, without constant page reloads.

Architecture

Website interacts with user by rewriting parts of the DOM. After first load, all interaction with the server happens through AJAX.



Bundling

All JS code must be delivered to the client over potentially slow networks. Bundling and minifying the source leads to smaller SPA footprint. Larger SPAs with many modules need a reliable dependency management. Initial Footprint can be reduced by loading dependent modules on-demand.

WebPack as Bundler

Entry: Start, follows the graph of dependencies to know what to bundle.
Output: Tell webpack where to bundle your application.
Loaders: Transforms these files into modules as they are added to your dependency graph.
Plugins: Perform tasks like bundle optimization, asset management and injection of env variables.
Mode: Enable built-in optimization mechanisms.

Routing

- Completely on client-side by JS
- Navigation behaves as usual
- Browser needs to fake the URL to change and store page state
- *window.history.pushState*

Dependency Injection

Benefits
- Reduces coupling between consumer and implementation
- Contracts between classes are based on interfaces
- Supports the open/closed principle
- Allows flexible replacement of an implementation

Decorators

- Provide a way to add annotations / meta-programming syntax
- Can be attached to a class declaration, method, accessor, property or parameter
- Widely used in Angular

React

- Library, kein Framework
- Um User Interfaces zu bauen
- View in MVC
- Minimales Featureset
- Entwickelt von Facebook
- Verwendet für: WhatsApp, Insta, AirBnb, etc.

Prinzipien

- Komplexes Problem aufteilen in einfachere Komponenten
- Für eine bessere: Wiederverwendbarkeit, Erweiterbarkeit, Wartbarkeit, Testbarkeit, Aufgabenverteilung

Entwicklung von UIs

- Beschreibung des UIs
- Event-Handling
- Aktualisieren der Views

Komponenten und Elemente

- Funktionen die HTML zurückgeben
- Beliebige Komposition von React-Elementen und DOM-Elementen

```
function App() {
 return (                          Parameterübergabe an Funktion
   <div>
     <HelloMessage name="HSR" />
     <img src="/logo.png" />
   </div>                          Äquivalent zu Attribut für DOM-Element
 )
}
```

JavaScript XML

React verwendet JSX (blau), eine Erweiterung von JavaScript (gelb). Überall wo JSX verwendet wird, muss *react* importiert werden.

```
const menu = entries.map(entry =>
  <ListItem as="a" to={`/${entry.path}`}>
    <h1>{entry.title.toUpperCase()}</h1>
    <p>{entry.subtitle}</p>
  </ListItem>
)
```

Styles: werden nicht als Strings sondern als Object angegeben.

Conditionals

```
<Container>
 { error &&
   <Message>
     Fehler: {error}
   </Message>
 }
</Container>
```

oder

```
<Container>
 { error
   ? <span>
       Fehler: {error}
     </span>
   : <span>OK!</span>
 }
</Container>
```

Props

Komponenten erhalten alle Parameter/Properties als **props** Objekt.
- *this.props* bei Klassen
- Bei Funktionen als Parameter
- Immer **read-only**

Rendering und Mounting

Mounting: nötig um Komponenten auf Webseite anzuzeigen. *ReactDOM.render*

```
ReactDOM.render(
    <App/>
    document.getElementById('root')
)
```

React State

React-Klassenkomponenten können einen veränderbaren Zustand haben. Der **state** einer Komponente ist immer privat. Ändert der State, wird auch die Komponente aktualisiert.

```
class Counter extends React.Component {
    state = { counter: 0 }
    // ...
}
```

Event Handler

```
const increment = () => {
    this.setState({counter: this.state.counter +
        1})
} // ...
<button onClick={this.increment}>
```

Reconciliation

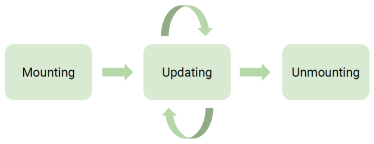
1. React Komponenten werden als virtueller DOM gerendert
2. Wird der **state** geändert, erstellt React einen virtuellen DOM
3. Alter und neuer DOM werden verglichen
4. Erst dann werden geänderte DOM-Knoten im Browser erstellt

Formulare

Input Handling

```
<form onSubmit={this.handleSubmit}>
<input value={this.state.username}
       onChange={this.handleUsernameChange} //...
</form>
handleUsernameChange = (event) => {
    this.setState({username: event.target.value});
};
handleSubmit = (event) => {
    event.preventDefault();
    //...
}
```

Komponenten Lifecycle



Mounting

1. **constructor(props)**
   - State initialisieren, sonst weglassen
2. `static getDerivedStateFromProps(props, state)`
   - Von State abhängige Props initialisieren
3. **render()**
4. **componentDidMount()**
   - DOM ist aufgebaut
   - Guter Punkt um zum Beispiel Async-Daten zu laden
   - setState Aufruf führt zu re-rendering

Updating

1. `static getDerivedStateFromProps(props, state)`
   - Von State abhängige Props aktualisieren
2. `shouldComponentUpdate(nextProps, nextState)`
   - wird false zurückgegeben wird render übersprungen
3. **render()**
4. `getSnapshotBeforeUpdate(prevProps, prevState)`
5. **componentDidUpdate(prevProps, prevState, snaspshot)**
   - Analog zu componentDidMount, DOM ist aktualisiert

Unmounting

1. **componentWillUnmount()**
   - Aufräumen

Error Handling

1. `static getDerivedStateFromError(error)`
   - Error im State abbilden
2. `componentDidCatch(error, info)`
   - Logging
   - Verhindern, dass Error propagiert wird, analog zu catch-Block in try-catch

React Router

- Komponentenbibliothek
- Komponenten anzeigen oder verstecken abhängig von der URL
- Für React Web und React Native

Router Komponenten

```
<Router>
```

Alle Routen müssen Teil des Routers sein, typischerweise nahe der Root-Komponente

```
<Route exact path="/" component={Home} />
```

Home-Komponente wird nur gerendert, wenn der path (exakt) matcht. Mehrere Route Elemente können gleichzeitig aktiv sein.

```
<Link to="/">Home</Link>
```

App-interne Links, welche nicht wie <a >die Seite neu laden.

```
<Redirect to="/somewhere/else">
```

Wird ausgeführt, sobald gerendert.

Hooks

Problem von Lifecycle Methoden Zusammengehörender Code ist auf mehrere Methoden verteilt (Mount/Unmount).
Problem von Klassen-State State ist über verschiedene Methoden verteilt
Fazit:
- Lifecycle und State ohne Klassen machen react verständlicher
- Klassen sind weiterhin unterstützt
- Hooks erlauben, Logik mit Zustand einfacher wiederzuverwenden

State Hook

```
function Counter() {
    const [count, setCount] = useState(0);
    // button => setCount(count + 1)
    return( <p>{count}</p> );
}
```

Mehrere State-Variablen: useState Aufrufe müssen immer in derselben Reihenfolge gemacht werden.

Effect Hook

```
useEffect(() => {
    // Mount stuff
    return () => {
        // Unmount stuff
    }
}, [] /* <= Dependencies */);
```

Flow

- Erweitert JavaScript um Typenannotationen
- Typ-Annotation im Code Typ-Inferenz für lokale Definitionen
- Generics, Maybe-Types, Union and Intersection-Types

TypeScript und React

- Mehr Typensicherheit in React-Komponenten
- Props und State lassen sich typisieren

Vorteil gegenüber Flow:
- Vollwertige Programmiersprache
- Besser unterstützt von Libraries und IDEs
- TypeScript Fehler müssen korrigiert werden

React Context

Ermöglicht es, Props für alle Unterkomponenten zur Verfügung zu stellen. (Theme Variablen)

```
// provider
const c = React.createContext(themes.light);
const theme = useContext(c); // consumer
```

Redux

Library für Statemanagement (Repräsentation / Veränderung / Benachrichtigung). State wird als Tree (immutable) von Objekten dargestellt. Veränderung am Tree führt durch den Reducer zu einem neuen Tree t+1 (funktionale Programmierung). State wird im **Store** verwaltet.

Actions

Benötigt um Stateänderungen zu machen. Wird an den Store gesendet / dispatched. Action ist eine reine Beschreibung der Action.

```
{type: 'TRANSFER', amount: 100 }
```

Reducer

Pure Funktionen, haben keine Seiteneffekte.

```
function balance(state = 0, action) {
    switch (action.type) {
        case 'TRANSFER':
            return (state + action.amount);
        default:
            return state;
    }
}
```

Reducer kombinieren: Jeder Reducer erhält einen Teil des States, für den er zuständig ist. Resultat wird in einem neuen State-Objekt kombiniert.

```
function rootReducer(state = {}, action) {
    return {
        balance: balance(state.balance, action),
        transactions: transactions(state.
            transactions, action)
    }
}
// Hilfsfunktion combineReducers:
const rootReducer = combineReducers({
    balance, transactions
});
```

Store erstellen

```
const store = createStore(rootReducer);
```

Mit dem root-Reducer kann der Store erstellt werden. In Kombination mit React führt das zu einem re-rendering der Komponenten.

React <3 Redux

Redux mit React verbinden:

```
const mapStateToProps = (state) => {
    return {
        transactions: state.transactions
    }
}
const mapDispatchToProps = {
    fetchTransactions
}
export default connect(mapStateToProps,
    mapDispatchToProps)(Component);

// Root Komponente
const store = createStore(
    rootReducer, applyMiddleware(thunkMiddleware));
render(
    <Provider store={store}>
        <App />
    </Provider>
    document.getElementById('root')
)
```

mapStateToProps: erhält State und kann daraus Props ableiten. Die Komponente bekommt auch die *dispatch* Methode des Stores als Prop. Das Resultat von *connect* ist eine React-Komponente die mit dem Store verbunden ist.
Store muss der Root-Komponente mitgegeben werden.
thunkMiddleware: Erlaubt es, anstelle eines Objektes eine Funktion zu dispatchen (benötigt für asynchrone Actions).

Thunk Actions

```
function fetchTransactions(token) {
    return (dispatch, getState) => {
        dispatch({type: "FETCH_TRANSACTIONS_STARTED
            "});
        api.getTransactions(token)
            .then(({result: transactions}) => {
                dispatch({type: "
                    FETCH_TRANSACTIONS_SUCCEEDED",
                    transactions});
            })
    };
```

}

Selectors

Getter bei den Reducern, die einen Subtree des Stores zurückgeben. Wissen über den Aufbau des State-Trees bleibt bei den Reducern.

Firebase

Läuft in der Google Cloud Platform. Hauptfokus von Firebase sind Mobile- und Web-Apps.

Firebase Authentication

Backend Services für Authentifizierung und einfache Userverwaltung. SDKs für diverse Plattformen. Vorgefertigte UI Libraries

Firebase Hosting

Einfaches Hosting für statischen Content.

- Immer per HTTPS ausgeliefert
- Automatisches Caching in CDNs

Dynamischer Content nur über **Cloud Function**, wenn das nicht reicht:

- PaaS: Google App Engine
- Docker: Google Container oder Kubernetes Engine

Serverless Computing

Cloud Provider verwaltet Functions:

- Deployment geschieht on-demand
- Plattform bestimmt die Parallelisierung
- Entwickler hat keine Kontrolle über laufende Instanzen
- Funktionen sind Stateless
- Abgerechnet werden Aufrufe und Laufzeit der Funktion

Limitationen: Ausführungszeit / Memory begrenzt. Teilweise hohe Latenz.

Firebase Cloud Functions

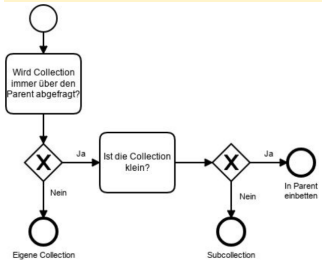
Anwendungszenarien: Code als Reaktion auf einen Event ausführen, Administration (Cron Jobs), REST API für Mobile und SPAs zur Verfügung stellen.

Cloud Firestore

- NoSQL, document-oriented database
- DB besteht aus mehreren Collections mit Documents
- Document ist ein JSON-Objekt
- Document kann Collections beinhalten
- Vergleichbar mit MongoDB
- Stark eingeschränkte Queries (keine Volltextsuche)

```
// Auf Collections / Documents zugreifen
const colRef = db.collection("todos");
const docRef = db.collection("todos").doc("...");
// Dokumente erstellen
db.collection("todos").add({text: "..."});
// Dokument bearbeiten
.doc("...").update({text: "..."});
// Daten Abfragen
db.collection("todos").doc("...").get().then(d => {
    if(!doc.exists) { /* ... */ }
    else { console.log(d.data()); }
}).catch(err => { /* ... */ });
// Daten abfragen mit Filter
db.collection("todos").where("checked", "==", true)
    .orderBy("createdAt").get().then(snapshot => {
// ...
});
```

NoSQL One-To-Many



NoSQL Many-To-Many

- Wie in relationaler Datenbank mit Assoziationstabelle
  - Kein kopieren von Daten
  - Komplexere Abfragen, keine Joins im Firestore
- Oder Daten kopieren und einbetten

Kopieren der Daten: muss kein Nachteil sein. Preisänderung eines Produktes hat keinen Einfluss auf vergangene Bestellungen. Adressänderung eines Kunden verändert keine alten Bestellungen. Kopierte Daten können mittels Trigger und Cloud Function wieder synchronisiert werden.

Angular

Flexible SPA Framework for CRUD applications

- Typescript 4.1 based
- Reduces boilerplate Code
- Dependency Injection Mechanism
- JS-optimized 2-way binding
- Clearly structured, information hiding
- Increases testability / maintainability of client-side code

Architecture

ngModules: Cohesive block of code dedicated to closely related set of capabilities. (*module*) **Directives:** Provides instructions to transform the DOM. (*class*) **Components:** Directive-with-a-template; it controls a section of the view. (*class*) **Templates:** Form of HTML defining how to render the component. (*HTML / CSS*) **Metadata:** Describes a class and defines how to process it. (*decorator*) **Services:** Provides logic of any value, function or feature that the app needs. (*class*)

Angular Modules (ngModule)

Base for Angular modularity system. Every app has at least one Module, the root Module (a.k.a app). Root Module ist launched to bootstrap the app. Modules export features (directives, services) required by other modules.

TypeScript Module vs. ngModule:
ngModule is a logical block of multipe TypeScript modules linked together. The ngModule declaration itself is placed into a TypeScript module. Modules can accommodate submodules. All public TS members are exported as an overall *barrel*

```
@NgModule({
  imports: [
    CommonModule
  ],
  declarations: []
})
export class CoreModule { }
```

NgModule with metadata object whose properties describe the module.

other modules whose exported classes are needed by components in *this* module.

the view classes that belong to this module.

declarations: View Classes that belong to this module (Components, Directives, Pipes).
exports: Subset of declarations that should be visible and usable by other modules.
imports: Specifies the modules which exports/providers should be imported.
providers: Creators of services that this module contributes to the global collection of services (DI Container). They become accessible in all parts of the app.
bootstrap: Main application view, root component. Only the root module should set this property.

Components

Manages the view and binds data from the model. Consists of:

- Controller (App logic), TS Class with *@Component* decorator
- HTML file, visual interface (HTML / template expression)
- (S)CSS file, styles behind HTML

Can be nested, results in Component tree.
Provide **Information Hiding:**

- Each Component declares part of the UI
- Should be implemented as small coherent piece to support:
  - Testability, Maintainability, Reusability

```
import { Component } from '@angular/core';

@Component({
  selector: 'wed-navigation',
  templateUrl: './navigation.component.html',
  styleUrls: ['./navigation.component.css']
})
export class NavigationComponent {

  Logic (TypeScript)
}
```

...
navigation.component.html
.css { }
navigation.component.css

Components must be declared within the containing module so its **selector** is registered for all sub-components of that module. They can be exported, so other modules can import and use then.

Component Lifecycle

Most important events are **ngOnInit** (Creation / Hydration) and **ngOnDestroy** (Destruction / Dehydration).
ngAfter... events are mainly for control developers to handle sub-components and their DOM. To hook into the lifecycle, interfaces of the Angular core can be implemented. Each interface has a single hook method, prefixed with *ng*. (**OnInit** contains method *ngOnInit*).

```
export class CounterComponent implements OnInit, OnDestroy {

  ngOnInit() {
    console.log("OnInit");
  }
  ngOnDestroy() {
    console.log("OnDestroy");
  }
}
```

Content Projection

```
<!-- component usage -->
<section>
  <wed-navigation>
    <h1 wed-title>WED3 Lecture</h1>
    <menu><!-- … --></menu>
  </wed-navigation>
</section>

<!-- resulting DOM -->
<section>
  <wed-navigation>
    <header>
      <h1 wed-title>WED3 Lecture</h1>
    </header>
    <nav>
      <menu><!-- … --></menu>
    </nav>
  </wed-navigation>
</section>
```

```
<header>
  <ng-content select='[wed-title]'>
  </ng-content>
</header>
<nav>
  <ng-content select='menu'>
  </ng-content>
</nav>
```

navigation.component.html

Templates

View in MVC. Written in HTML annotated with Angular **template syntax**:

- HTML5 except script-Tag
- Angular extends the HTML with
  - Interpolation (…)
  - Template Expression/Statements
  - Binding Syntax
  - Directives
  - Template Reference Variables
  - Template Expression Operators

Binding

Two Way Binding / *Banana in a box* [(…)]

```
<input type="text" [(ngModel)]="counter.team">
```

One Way (from View to Model / *Event Binding*) (…)

```
<button (click)="counter.eventHandler($event)">
```

One Way (from Model to View / *Property Binding*) […] or {{ … }}

```
<p>... {{counter.team}} ...</p>
<img [attr.alt]="counter.team" src="team.jpg">
```

```
@Component({…})
export class CounterComponent
{
  public counter: any = {
    get team() { return …; },
    set team(val) { },
    eventHandler: ()=>{ }
  }
}
```

Model Object

Binding targets must be declared as **Inputs or Outputs:** Targets stand on the left side of the binding declaration. e.g. the click / title property: `<wed-navigation (click)="..." [title]="...">`

```
@Component({…})
export class NavigationComponent {
  @Output() click =
          new EventEmitter<any>();
  @Input() title: string;
}
```

Directives

Similar to a component, but without a template. TypeScript class with an *@Directive()* function decorator.

Attribute Directives

Changes the appearance or behaviour of an element, component or another directive. Applied to a host element as an attribute.

NgStyle Directive
Sets the inline styles dynamically, based on the state of the component.

```
<div [style.font-size]="isSpecial ? 'x-large' : 'smaller'">
    <!-- render element -->
</div>
```

NgClass Directive
Bind to the ngClass directive to add or remove several classes simultaneously.

```
<div [class.special]="isSpecial">
    <!-- render element -->
</div>
```

Structural Directives

Responsible for HTML layout. Reshape the DOM's structure by adding, removing or manipulating elements. Applied to a host element as an attribute. Asterisk (*) precedes the directive attribute name.

NgIf Directive
Takes a boolean value and makes an entire chunk of the DOM appear or disappear.

```
<div *ngIf="hasTitle"><!-- shown if title available --></div>
```

NgFor Directive
Represents a way to present a list of items.

```
<li *ngFor="let element of elements"><!-- render element --></li>
```

NgTemplates:

```
<ng-template #toReference><!-- content --></ng-template>
```

Aren't rendered directly. They need a directive or component which takes over this part. Can be referenced by their *id*:

```
<div *ngIf="hasTitle; else toReference"><!-- conditional content --></div>
```

Template Reference Variables

References a DOM element within a template. Can also be a reference to an component or directive. A hash () declares a reference variable.

```
<input placeholder="phone number" #phone>

<!-- lots of other elements -->

<!-- phone refers to the input element; pass its
`value` to an event handler -->
<button (click)="callPhone(phone.value)">Call</button>
```

Services

Provides any value, function or feature. Typical Services: logging service, data service, message bus, tax calculator, etc.
Strongly related to DI: Angular uses DI to provide components with needed services. Therefore, services must be registered within the DI container.

```
@Injectable ({ providedIn: 'root' })
export class CounterService { /* ... */ }
```

providedIn: 'root': The service is registered for the whole application.

Decorator function to mark class as an injectable instance

```
@Injectable({ providedIn: 'root' })
export class CounterService {
  private model: CounterModel
    = new CounterModel();

  public load():CounterModel {…}
  public up():CounterModel {…}
}
```

```
@Component(…)
export class CounterComponent {
  counter?: CounterModel;

  constructor(private counterService:
      CounterService) {
    this.counter = counterService.load();
  } …
}
```

Required services (dependencies) are automatically injected by Angulars injector.

Forms

Angular Forms is an external, optional ngModule called FormsModule. It's a combination of multiple provided services and multiple directives (ngModel, ngForm, ngSubmit).
Template-driven forms: Angular Template syntax with the form-specific directives and techniques. Less code but places validation logic into HTML. (Useful for small forms)
Reactive / model driven forms: Import ReactiveFormsModule. Form is built within the Controller (FormBuilder). Validation logic is also part of the controller (easier to test).

Template-driven

```
<input type="text" class="form-control" id="name"
       required
       [(ngModel)]="model.name" name="name"
       #nameField="ngModel">
<div [hidden]="nameField.valid || nameField.pristine" class="alert alert-danger">
  Name is required
</div>
```

Two-Way-Binding: [(ngModel)] directive to bind values. Reads out the value of the model for the first time. Updates are automatically written back into the bound model.
Validation: Reference the [ngModel] directive and check its valid property.
Submitting the form:

ngForm can also be referenced

- This is useful to bind validation state on the submit button or pass the form to the submit method

```
<form (ngSubmit)="doLogin(sampleForm)" #sampleForm="ngForm">
      <button type="submit"
              class="btn btn-success"
              [disabled]="!sampleForm.form.valid">Submit</button>
</form>
```

```
@Component({ ... })
export class SampleComponent {
  public doLogin(f?: NgForm): boolean {
    if (f?.form.valid) { // store data
    }
    return false; // avoid postback
  }
}
```

Asynchronous Services

Event Emitter example:

```
@Injectable({providedIn: 'root'})
export class SampleService {
  private samples: SampleModel[] = []; // simple cache
  public samplesChanged: EventEmitter<SampleModel[]> =
    new EventEmitter<SampleModel[]>();

  constructor( /* inject data resource service */ ) {
  }

  load(): void {
    /* in real word app, invoke data resource service here */
    this.samples = [ new SampleModel() ];
    this.samplesChanged.emit(this.samples);
  }
}
```

Create emitter instance. The type argument specifies the kind of object to be passed to the subscriber.

Logic to execute when data ready. Emit changed event to notify the registrars (e.g. UI components).

```
export class SampleModel { }
@Component({ ... })
export class SampleComponent implements OnInit, OnDestroy {

  private samples: SampleModel[];
  private samplesSubscription: Subscription;
  constructor(private sampleService: SampleService) {
  }

  ngOnInit() {
    this.samplesSubscription = this.sampleService.samplesChanged.subscribe(
      (data: SampleModel[]) => { this.samples = data; });
  }

  ngOnDestroy() {
    this.sampleSubscription.unsubscribe();
  }
}
```

Subscription is used to unsubscribe the update event when the component is de-hydrated.

Register samplesChanged event on underlying business service when component is hydrated. Subscribe() returns a Subscription which is used for deregistration.

Update procedure; refresh data on the UI level.

Unsubscribe the update event when the component is de-hydrated.

Data Access

HTTP Client API

Implements asynchronisms by using the RxJS library. RxJS is a third-party library that implements the Observable pattern. An Observable can be turned into a promise.
Hot Observables: Sequences of events (mouse moves / stock tickers). Shared amoung all subscribers.
Cold Observables: Start running on subscriptions (such as async web requests). Not shared amoung subscribers. Are automatically closed after Task is finished.

```
var subscription = this.http.get('api/samples').subscribe(
  function (x) { /* onNext -> data received (in x) */ },
  function (e) { /* onError -> the error (e) has been thrown */ },
  function () { /* onCompleted -> the stream is closing down */ }
);
```