# ReactiveX mit RxJS


## Roman Roelofsen - W11K GmbH / theCodeCampus

Twitter & GitHub: **romanroe**

# Über mich

- Alpha Geek, Entwickler, Trainer
- W11K GmbH - *The Web Engineers*
  - Individualsoftware
- theCodeCampus
  - Schulungsanbieter Angular & TypeScript

# Reaktive Programmierung

*... a programming paradigm oriented around* ***data flows*** *and the* ***propagation of change****."*

## Im Kleinen

- Strom von Daten: Liste
- Änderungen verfolgen: Events (Mouse-Clicks, ...)

## Im Großen

- Strom von Daten: Web-Sockets
- Änderungen verfolgen: Message Bus

# Iterator + Callback

## Iterator

- `java.util.Iterator`
- Synchron, Pull
- Keine Fehler-Konzept

## Callback

- `java.util.function.Funtion<T, R>`
- Asynchron, Push
- Kein standardisiertes Fehler-Konzept

# ReactiveX

- Iterator + Callback + Fehlerbehandlung
- RxJS
- **http://reactivex.io**
- Implementierungen für
  - Java, JavaScript/TypeScript, .NET, Scala, Clojure, Swift, etc.

- Observable
  - Liefert Daten

| | Single return value | Multiple return values |
|---|---|---|
| Pull/Synchronous/Interactive | Object | Iterables(Array \| Set \| Map) |
| Push/Asynchronous/Reactive | ~~Promise~~ Future | Observable |

- Observer
  - Bekommt Daten
- Subject
  - Observable *und* Observer

# Woher kommen Observables?

- Server-Aufruf
- WebSocket-Anfrage
- Benutzereingaben
- (System-)Events

# Operatoren

- Methoden am Observable
  - `map/filter/...`
- Kombination von Observables
  - `flatMap/withLatestFrom/...`
- Operatoren erzeugen immer neue `Observable`s

# Demo

# API - cold/synchron

```
const observable: Observable<number> = Observable.create(e => {
        e.next(1);
        e.next(2);
        e.complete();
});
```

```
observable.subscribe(next => {
    console.log(next);
}, (err) => {
    console.log("error", err);
}, () => {
    console.log("done");
});
```

# Fehlerbehandung

# Observer

```
Observable<Integer> observable = Observable.create(e -> {
        e.next(1);
        e.next(2);
        e.error("error");
        e.next(3); // wird nicht "gesendet"
});
```

- Stream termininert bei einem Fehler!

# Subject

- Observable und Observer
- Multiplexer
- Puffer

- Nützlich, wenn Datenquelle nicht verschachtelt werden kann

```
Observable.create(e => {
        ...
        ...
});
```

# Subject

```
const s1 = new Subject();
s1.next(1);
s1.next(2);
s1.subscribe(i => console.log(i));
```

# Ausgabe

```

```

# ReplaySubject

```
const s2 = new ReplaySubject(1);
s2.next(1);
s2.next(2);
s2.subscribe(i => console.log(i));
```

# Ausgabe

```
2
```

# BehaviorSubject

```
const s3 = new BehaviorSubject(0);
s3.subscribe(i => console.log(i));
s3.next(1);
s3.next(2);
```

# Ausgabe

```
0
1
2
```

# Fazit

- Lernkurve:
  - kurz flach
  - dann lange steil
  - dann wieder flach
- ReactiveX macht komplexe Datenflüsse "einfach"
- Keine komplexe Datenflüsse? Dann Finger weg...
- Kein echtes Projekt zum Lernen nehmen, niemals!
- **http://rxmarbles.com**

Roman Roelofsen

@romanroe