




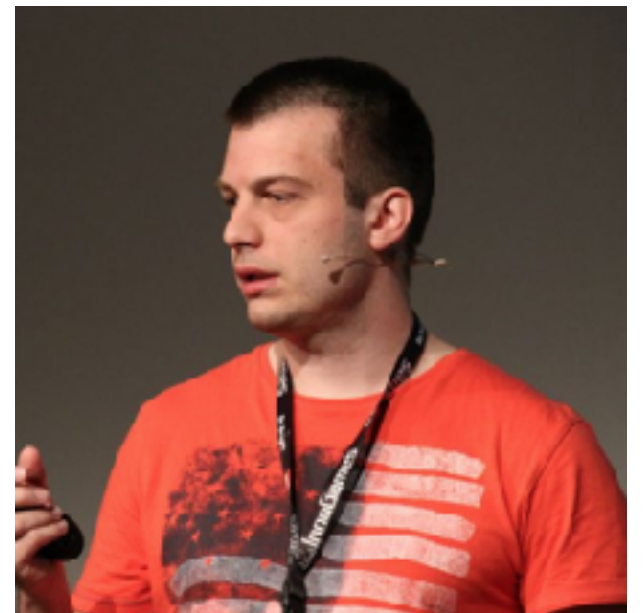
# ReactiveX

# Reactive Programming

mit RxJS

# Wer bin ich?

- Sebastian Springer
- <http://github.com/sspringer82>
-  @basti\_springer
- Entwickler
- JavaScript... schon etwas länger



# Agenda

- Reactive Programming
- RxJS-Grundlagen
- Operatoren
- RxDOM
- RxJS und Angular
- ...und ein bisschen Node

<https://github.com/sspringer82/reactiveProgramming>

Asynchronität

# Asynchronität

- Callbacks
- Promises
- Async
- Streams

# Callbacks

```
function async(cb) {  
  setTimeout(() => {  
    cb('Hello World');  
  }, 1000);  
}
```

```
async((value) => {  
  console.log('Value is: ', value);  
});
```



# Promises

```
function async(cb) {  
  return new Promise((resolve, reject) => {  
    setTimeout(() => {  
      resolve('Hello World');  
    }, 1000);  
  });  
}
```

```
async().then((value) => {  
  console.log('Value is: ', value);  
});
```

# Async

```
function async(cb) {  
  return new Promise((resolve, reject) => {  
    setTimeout(() => {  
      resolve('Hello World');  
    }, 1000);  
  });  
}
```

```
(async () => {  
  const value = await async();  
  console.log('Value is: ', value);  
})();
```

# Streams

```
Rx.Observable  
  .of( 'Hello World' )  
  .delay(1000)  
  .subscribe((value) => {  
    console.log( 'Value is: ', value);  
  });
```

# Reactive Programming

# Reactive Programming

**Asynchrone** Programmierung mit **Datenströmen** und einem sehr mächtigen **Werkzeugkasten**

Eine kurze Erklärung

# Arrays

Arrays kennt ihr alle und hoffentlich auch einige Array-Methoden

[8, 15, 6, 19, 2, 4, 13]

# Arrays

[8, 15, 6, 19, 2, 4, 13]

Wir sind jetzt nur an den **geraden Zahlen** interessiert.  
Diese sollen aber **aufsteigend sortiert** werden.

Das Array soll **immutable** sein.



# Immutable Data?

- Einfachere Applikationen
- Kein defensives kopieren (Backups)
- Change Detection ist wesentlich einfacher.

# Arrays

```
let output = input  
  .filter(item => item % 2 === 0)  
  .sort((a, b) => a > b); // [ 2, 4, 6, 8 ]
```

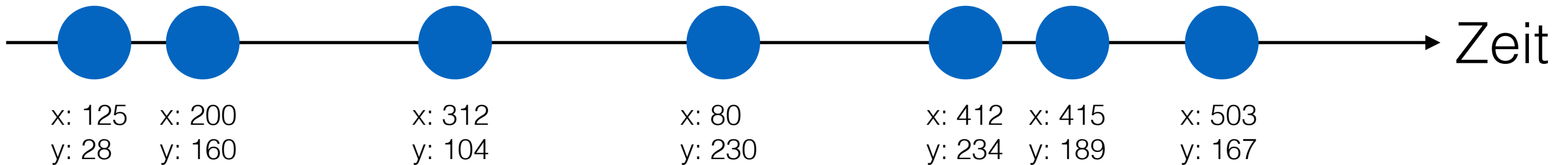
# Events

**Events** können ähnlich behandelt werden wie **Array-Elemente**. Auf diese Event-Elemente können dann auch verschiedene **Operationen** wie **Filterung** oder **Mapping** ausgeführt werden.

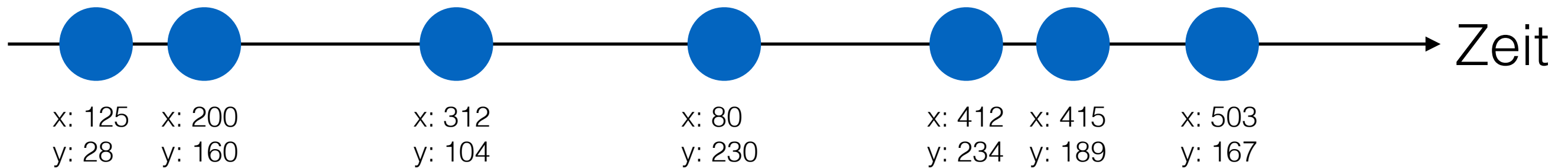
Events sollten ebenfalls nicht durch die Applikationslogik modifiziert werden - sie sind also auch **immutable**.

# Events

z.B. Click Events im Browser



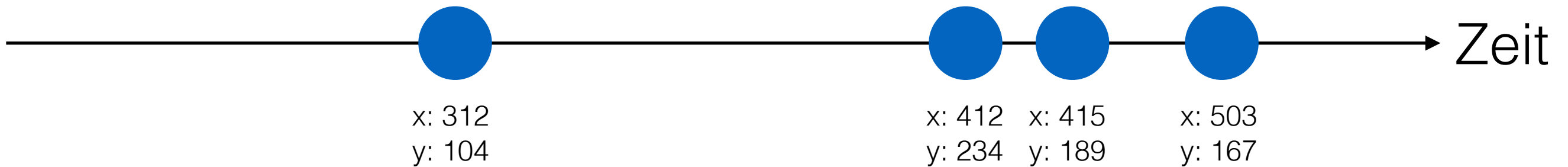
# Events



Hier interessieren wir uns nur für Events mit einem x-Wert von über 250

# Events

```
events.filter(e => e.x > 250)
```



Also sind **Eventstreams** auch nichts weiter als **asynchrone immutable Arrays**.

# Aufwärmübung

Berechnet die Summe aller geraden Zahlen von 1 bis 100.



```
const arr = [];  
for (var i = 1; i <= 100; i++) {  
    arr.push(i);  
}  
const result = arr.filter(v => v % 2 === 0)  
    .reduce((prev, curr) => prev + curr, 0);  
console.log('arr: ', result);
```



# ReactiveX

# Reactive Extensions

ReactiveX is a library for composing **asynchronous** and **event-based** programs by using **observable** sequences.

<http://reactivex.io/intro.html>

Ein Projekt, das aktiv von **Microsoft** Open Technologies und einer Open Source **Community** entwickelt wird.

# Die Kernkomponenten

- Observable
- Operator
- Observer

# Funktionale Programmierung

# Funktionale Programmierung

Die Applikation besteht aus **Funktionsblöcken**.  
**Seiteneffekte** von Funktionen werden **vermieden**.

# Was bringt es?

Einheitliches Interface für verschiedene Datenstrukturen wie  
zum Beispiel

- click-Events
- Server-Responses
- Arrays

# Installation

```
npm install rxjs
```



# Verwendung

Globale Variable

```
<script src="rx.min.js"></script>  
const observable = Rx.Observable.of(1, 2, 3);
```

ES6

```
import { Observable } from 'rx/Observable';  
import 'rxjs/add/observable/of';  
const observable = Observable.of(1, 2, 3);
```

CommonJS

```
const Obs = require('rxjs/Observable').Observable;  
require('rxjs/add/observable/of');  
const observable = Obs.of(1, 2, 3);
```

# RxJS 4 vs 5

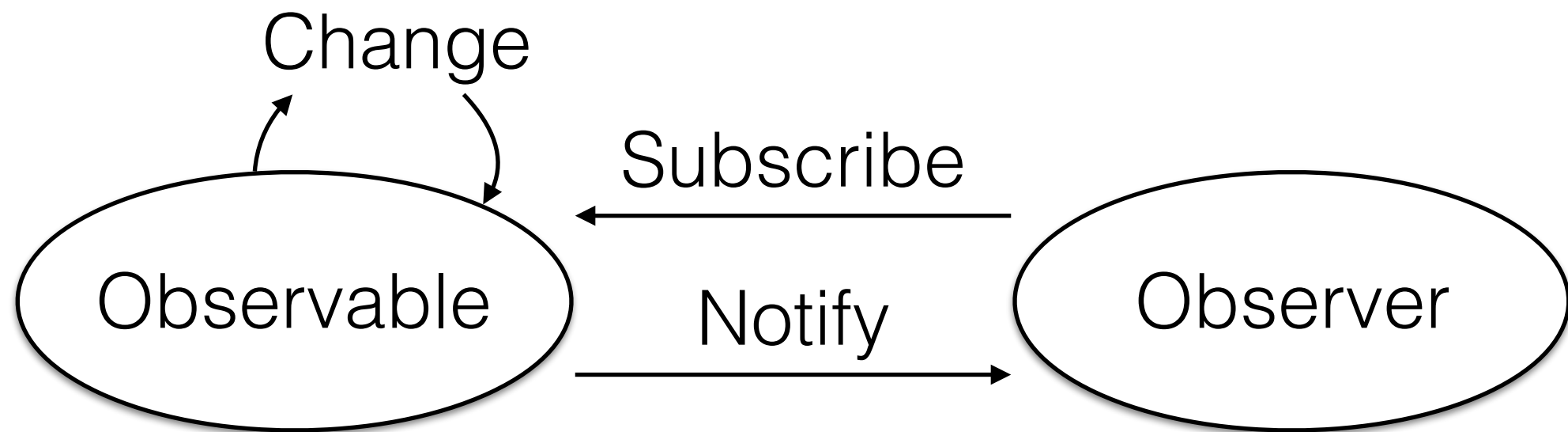
Mit der neuen Version von RxJS hat sich einiges verändert.  
Einige Operatoren wurden umbenannt oder entfernt.

<https://github.com/ReactiveX/rxjs/blob/master/MIGRATION.md>

Der Grund für die massiven Änderungen:

- Performancesteigerung
- Besseres Debugging
- Dem ES7 Observable-Standard entsprechen

# Observer & Observables



# Observable

Die **Datenquelle**. Ein Observable **emitted** ein oder mehrere **Elemente**. Ein **Observer subscribed** sich auf ein Observable und **konsumiert** die Elemente.  
Die Kommunikation erfolgt asynchron.

# Observable

Selbst definierte Observables werden mit der **create**-Methode erzeugt. In der **Callback**-Funktion hat man die vollständige Kontrolle.

```
public static create(onSubscription: function(observer: Observer): TeardownLogic): Observable
```

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

# Observable

```
const observable = Rx.Observable.create((observer)
=> {
  observer.next(1);
  observer.next(2);
  observer.next(3);
  setTimeout(() => {
    observer.next(4);
    observer.complete();
  }, 1000);
});
```

# Observer

Ein Observer subscribed sich auf ein Observable

```
let subscription = observable.subscribe(  
  function onNext(next) {  
    console.log('Next: ', next);  
  }, function onError(err) {  
    console.log('Error', err);  
  }, function onCompleted() {  
    console.log('Completed');  
  }  
);
```



# Aufgabe 1

Erzeugt ein Observable, das im Abstand von 0,5 Sekunden die Zeichenketten "Hello" "World" "how" "are" "you" emittiert.

Subscribed euch auf dieses Observable und gebt die Zeichenketten aus.

# Hot vs. Cold

Hot Observables emittieren Elemente, egal ob ein Observer verbunden ist oder nicht.

Beispiel: `fromEvent`

Cold Observables emittieren Element, sobald sich ein Observer auf sie subscribed.

Beispiel: `range`

# Operatoren

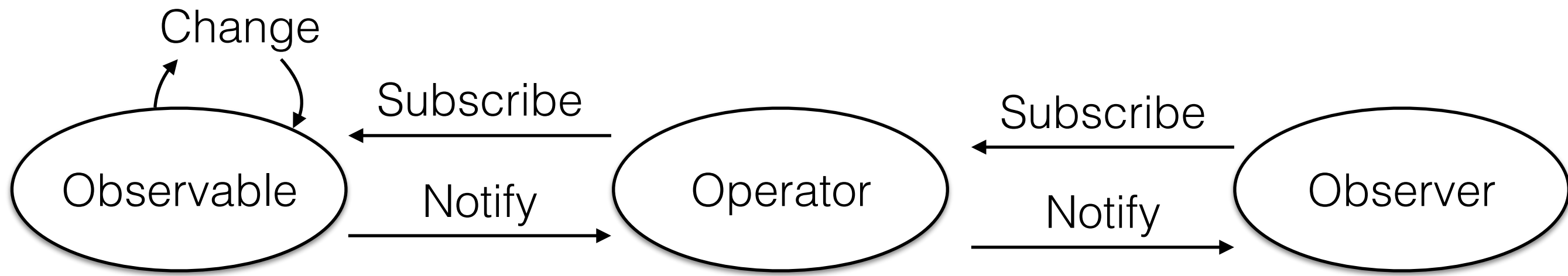
# Operatoren

Eine gute Übersicht und umfangreiche Dokumentation mit Beispielen:

[http://reactivex.io/rxjs/class/es6/  
Observable.js~Observable.html](http://reactivex.io/rxjs/class/es6/Observable.js~Observable.html)

# Operatoren

Operatoren sind die Werkzeugkiste von RxJS. Operatoren erfüllen verschiedene Zwecke: Sie können verwendet werden, um Observables zu erzeugen, Datenströme zu Filtern oder Fehler in der Applikation zu behandeln.



# Operatoren

```
Rx.Observable.range(1, 10)  
  .filter(x => x % 2 === 0)  
  .subscribe(x => console.log(x));
```

# Kategorien

<https://github.com/ReactiveX/rxjs/blob/master/doc/operators.md>

- Erstellende Operatoren
- Transformierende Operatoren
- Filter Operatoren
- Kombinierende Operatoren
- Multicasting Operatoren
- Operatoren zur Fehlerbehandlung
- Hilfsoperatoren
- Boolean Operatoren
- Mathematische Operatoren



# Erstellende Operatoren

# Observable

Es gibt verschiedene Möglichkeiten  
Observables zu erzeugen:

- create: Mit einer Callback-Funktion
- from: Aus einem Array, Promise oder Iterable
- of: Aus mehreren Werten
- defer: Mit einer Factory-Funktion
- generate: Erstellt eine Sequenz (wie für eine Schleife)
- range: Erzeugt eine Spanne von Ganzzahlen
- using: Sequenz die auf einem Ressourcenobjekt basiert
- interval: Emittet in regelmäßigen Abständen

# Konvertieren

Existierende Datenstrukturen wie Arrays, Promises oder Events können in ein Observable umgewandelt werden.  
So ist es relativ einfach Bibliotheken in eine RxJS-Applikation zu integrieren.

- fromPromise
- fromEvent

# Transformierende Operatoren

# Transformierende Operatoren

Hier findet man die Klassiker, die aus einer gegebenen Eingabe eine Ausgabe erzeugen.

- **bufferTime**: Sammelt Werte und gibt sie nach einer bestimmten Zeit als Array aus
- **scan**: Reduce over time
- **map**: Verändert die eingehenden Werte
- **concatMap, mergeMap, switchMap**: Mappen ein Observable auf ein inneres Observable

# map?!

<https://tolikcode.github.io/post/rxjsMap/>

# Aufgabe

Aufgabe 2: Emittet alle 500ms einen Wert aus, der um 1 höher ist als der vorherige. Transformiert den Wert, sodass er mit sich selbst multipliziert wird. Gebt das Ergebnis aus.

```
Rx.Observable.interval  
    .map
```

Aufgabe 3: Emittet nacheinander je ein Objekt mit der Eigenschaft "name", "age", "address" und fügt diese zu einem Objekt zusammen.

```
Rx.Observable.of  
    .scan
```

<https://github.com/sspringer82/rp>

# Filter Operatoren



# Filter Operatoren

Einschränken des Event-Stroms. Nach Werten, zeitlich oder nach Anzahl der Elemente.

# Filter Operatoren

- **debounceTime**: Verwirft alle Werte, die weniger als die angegebene Zeit zwischen den Ausgaben liegen.
- **distinctUntilChanged**: Gibt erst wieder aus, wenn sich der Wert geändert hat
- **filter**: Gibt nur Werte aus, die dem Filterkriterium entsprechen
- **take**: Gibt nur eine bestimmte Anzahl von Werten aus
- **takeUntil**: Akzeptiert Werte bis ein bestimmtes Ereignis auftritt

# Aufgabe 4

Schreibt ein Observable, das ein zufälliges Zeichen ausgibt, sorgt dafür, dass nur Vokale ausgegeben werden. Sorgt dafür, dass nur insgesamt 10 Zeichen ausgegeben werden. Kombiniert die Zeichen zu einer Zeichenkette und gibt diese aus.

# Kombinierende Operatoren

# Kombinierende Operatoren

Kombinieren mehrere Observables zu einem Observable

# Kombinierende Operatoren

- **combineLatest**: Sobald ein Observable emittet, werden alle zuletzt emitteten Werte ausgegeben
- **concat**: Subscribt sich der Reihe nach auf Observables, sobald das vorherige beendet ist.
- **merge**: Kombiniert mehrere Observables in eines
- **startWith**: Einen initialen Wert vorgeben
- **withLatestFrom**: Kombiniert zum aktuellen Observable den letzten Wert eines anderen Observables

# Aufgabe 5

Schreibt ein Observable, das in zufälligen Zeitabständen Meldungen emittiert. Erzeugt ein zweites Observable, das jede Sekunde die aktuelle Zeit und Datum emittiert und kombiniert beide so, dass zu jeder Meldung die passende Zeit angezeigt wird.

create

```
const timer = Rx.Observable.timer(0, 1000).map(() => new Date().toString());
```

withLatestFrom

# Fehlerbehandlung



# Fehlerbehandlung

Ungefangene Fehler führen zum Abbruch der Applikation.

Ein Fehler kann z.B. mit der error-Methode erzeugt werden.

```
Rx.Observable.create(observer => {  
    observer.error('Whooo!');  
});
```

Die Abarbeitung des Observables wird abgebrochen.

Die einfachste Variante mit Fehlern umzugehen, ist die Verwendung des Error-Callbacks in der subscribe-Methode.

# Fehlerbehandlung

Operatoren können bei der Fehlerbehandlung helfen:

- **catch**: Erzeugt neues Observable im Fehlerfall
- **retry**: erneut versuchen

# Aufgabe 6

Schreibt ein Observable, das jede Sekunde einen Wert ausgibt, alle 5 Sekunden soll ein Fehler geworfen werden.

Behandelt den Fehler:

- Fehler auffangen und Beenden
- Erneut versuchen

`mergeMap`

`Rx.Observable.throw`

`Rx.Observable.of`

Subject

# Subject

Subjects sind ein Spezialfall von Observables. Mehrere Observer können sich auf ein Subject subscriben.

Ein Subject kann auch von außen getriggert werden.

# Subject

```
const observable = Rx.Observable.create(observer => {  
  observer.next(Math.random());  
});
```

```
const subject = new Rx.Subject();
```

```
subject.subscribe(v => console.log('A: ', v));  
subject.subscribe(v => console.log('B: ', v));
```

```
observable.subscribe(subject);
```

# Subject vs. Observable

Subscribed man sich mehrfach auf ein Observable, hat jeder Subscriber seine eigene Instanz. Bei einem Subject teilen sich alle Subscriber die gleiche Instanz.

```
const observable = Rx.Observable.create(observer => {  
  observer.next(Math.random());  
});
```

```
observable.subscribe(v => console.log('A: ', v));  
observable.subscribe(v => console.log('B: ', v));
```

VS

```
const observable = Rx.Observable.create(observer => {  
  observer.next(Math.random());  
});
```

```
const subject = new Rx.Subject();
```

```
subject.subscribe(v => console.log('A: ', v));  
subject.subscribe(v => console.log('B: ', v));
```

```
observable.subscribe(subject);
```



# Aufgabe 7

Schreibt ein Observable, das jede Sekunde einen zufälligen Wert zwischen 1 und 256 ausgibt. Subscribt euch mit zwei Observern darauf. Einer soll den Zahlenwert, einer den ASCII-Charakter ausgeben.

```
const observable = Rx.Observable.create(observer => {  
  observer.next(Math.random());  
});
```

```
const subject = new Rx.Subject();
```

```
subject.subscribe(v => console.log('A: ', v));  
subject.subscribe(v => console.log('B: ', v));
```

```
observable.subscribe(subject);
```

# Scheduler

# Scheduler

Ein Scheduler kontrolliert wann eine Subscription startet und wann die Benachrichtigungen stattfinden. Ein Scheduler besteht aus drei Komponenten:

- Datenstruktur
- Ausführungskontext
- Virtuelle Uhr

RxDOM

# RxDOM

Mit RxDOM werden DOM-Schnittstellen reaktiv.

- Event Binding
- Ajax
- Web Sockets
- Web Workers
- SSE
- Geolocation

# Installation

```
npm install rx-dom
```

rx-dom ist lediglich eine Zusatzbibliothek und benötigt eine Installation von rxjs (lite)

# Aufgabe

Eingaben eines Input-Feldes entgegennehmen und in einen Ausgabecontainer schreiben.

Nur wenn der Text länger als 2 Zeichen ist. Änderungen nur alle 500ms und nur wenn es Änderungen gibt.

```
Rx.DOM.keyup(input)
    .pluck('target', 'value')
    .filter(text => text.length > 2)
    .debounce(500)
    .distinctUntilChanged()
    .subscribe(input => {
        output.innerText = input;
    });
```



# Aufgabe

Erzeugt einen WebWorker, der jede Sekunde ein Objekt mit Wetter-Messwerten erzeugt.

Zeigt nur die Temperaturwerte an, dieser Wert soll alle 10 Sekunden aktualisiert werden.

Angular ♥ RxJS

# Angular ♥ RxJS

RxJS ist eine Abhängigkeit von Angular, die bei der Initialisierung eines Projekts automatisch mit installiert wird.

Teile des Frameworks selbst benutzen RxJS z.B. der  
EventEmitter

Module wie http und forms nutzen ebenfalls RxJS.

# Einstieg

RxJS kann überall in einer Angular-Applikation eingesetzt werden (Components, Services, etc.)

# RxJS in Components

Aufgabe: Als Ausgabe soll das aktuelle Datum und die aktuelle Uhrzeit angezeigt werden. Die Anzeige soll jede Sekunde aktualisiert werden.

Benötigte Operatoren:

- interval
- timestamp
- map

# Component

im Wurzelverzeichnis: `ng g component time`

in der `app.component.html`: `<app-time></app-time>`

Auf der Konsole: `ng serve`

In den Browser: `localhost:4200`

# Forms

# forms

Bei den reactive Forms von Angular kommt ebenfalls RxJS  
zum Einsatz.  
Vor allem bei der Validierung.



httpClient

# httpClient

Das http-Modul von Angular basiert auf RxJS. Eine Server-Response ist ein Observable, auf das man sich subscriben kann.

# Reactive Stores

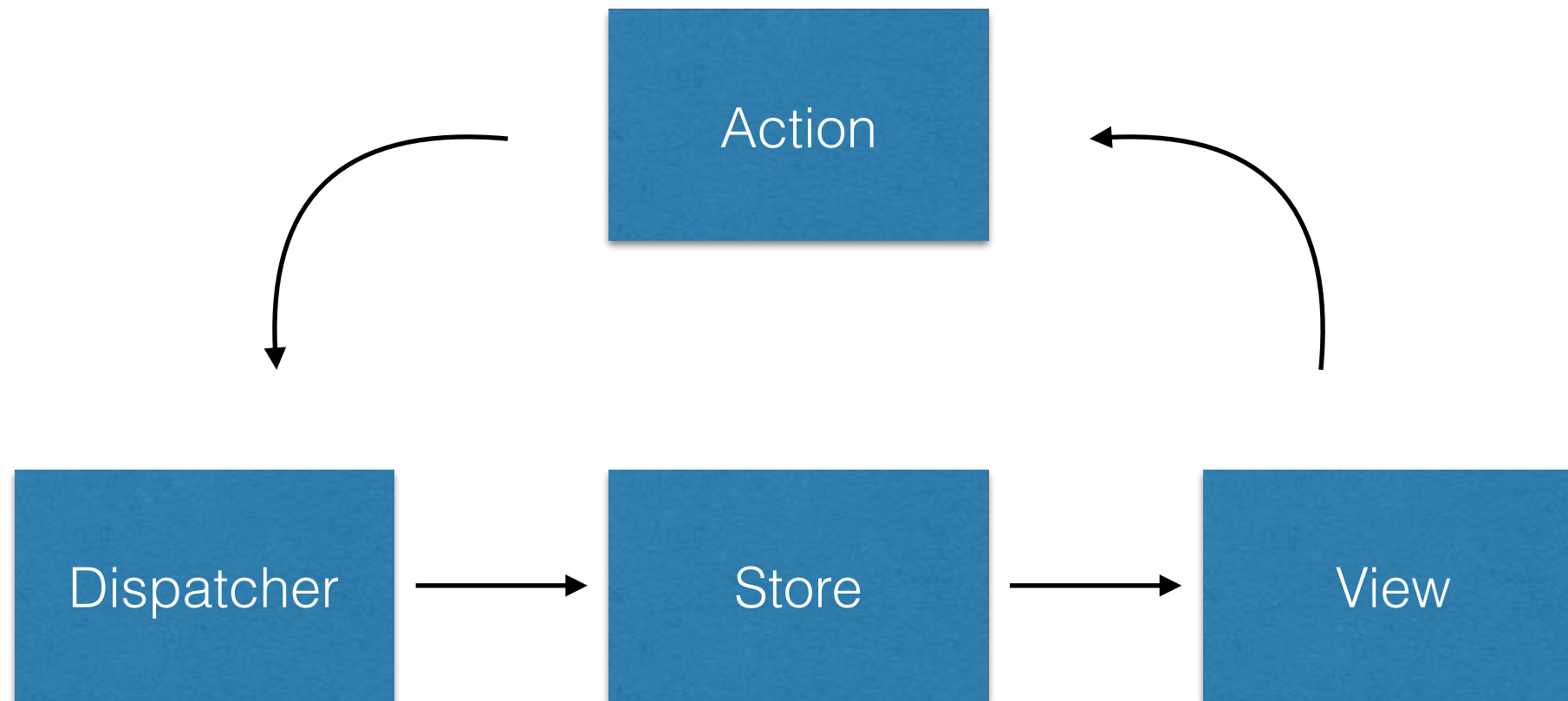
# Reactive Stores

Ein Reactive Store kapselt die Datenhaltung in einer internen Datenstruktur. Der Store wird nicht direkt sondern über eine definierte API modifiziert. Meist ist der Store mit dem HTTP-Service verbunden.

Die API verfügt meist über Methoden für CRUD-Operationen.

# Redux Store

# Redux Store



Zentraler gerichteter Datenfluß in der gesamten Applikation.

# Wann brauche ich das?

- Mehrfache Anzeige von Informationen
- Die Informationen können an verschiedenen Stellen modifiziert werden (User, Server)

# Implementierung

```
npm install @ngrx/core @ngrx/store --save
```



```

@Component({
  selector: 'counter',
  template: `
    <div class="content">
      <button (click)="increment()">+</button>
      <button (click)="decrement()">-</button>
      <h3>{{counter$ | async}}</h3>
    </div>
  `,
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class Counter {
  counter$: Observable<number>;

  constructor(
    private store : Store<number>
  ){
    this.counter$ = this.store.select('counter')
  }

  increment(){
    this.store.dispatch({type: 'INCREMENT'});
  }

  decrement(){
    this.store.dispatch({type: 'DECREMENT'});
  }
}

```

Node.js

# Node.js

An vielen Stellen in einer Node.js-Applikation kann RxJS verwendet werden.

z.B. beim Http-Server

# Node.js

Verwendung:

```
const Rx = require('rxjs/Rx');
```

# Node.js

```
const Rx = require('rxjs');
const http = require('http');

const subject = new Rx.Subject();
const observable = subject.asObservable();

const server = http.createServer(function (request,
response)
{
    subject.next({ request, response });
});
server.listen(8080);

observable.subscribe(({request, response}) => {
    response.end('Hello World');
});
```

# Aufgabe

Ein Benutzer hat die Möglichkeit über die URL seinen Namen anzugeben (<http://localhost:8080/name/basti>). Extrahiert diesen Namen aus der Request-URL und zeigt ihn in der Response an.



# Fragen?





# KONTAKT

Sebastian Springer  
sebastian.springer@maibornwolff.de

MaibornWolff GmbH  
Theresienhöhe 13  
80339 München

@basti\_springer

<https://github.com/sspringer82>