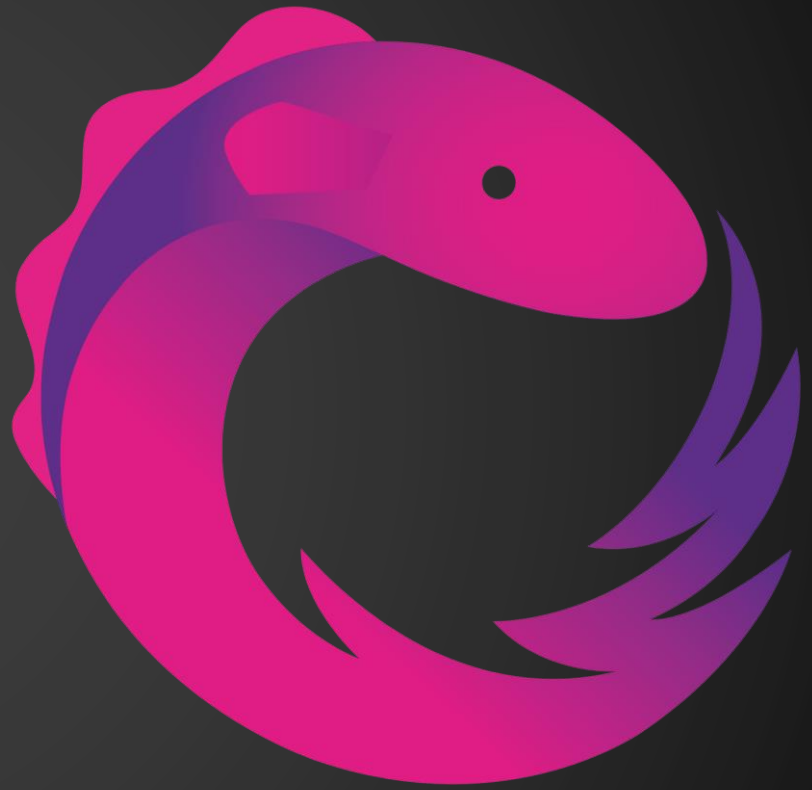
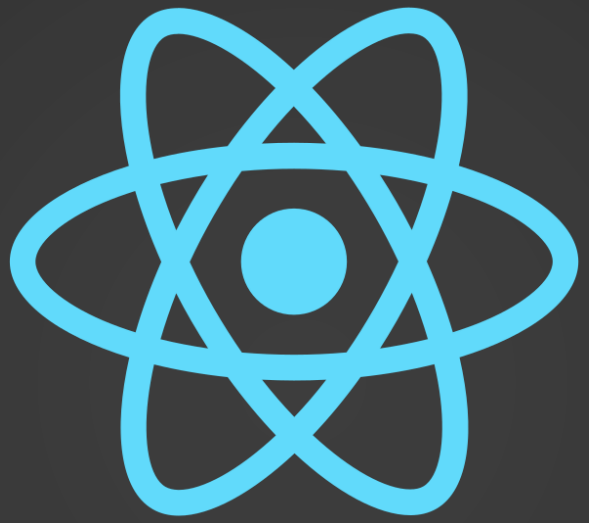
Andreas Roth

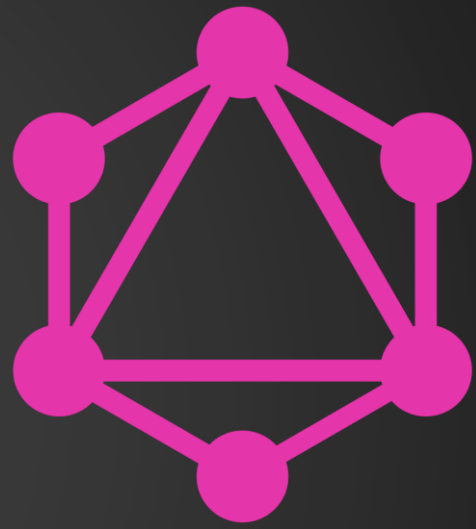
Denken in Streams
RxJS effektiv nutzen

andreas.roth@esveo.com
academy.esveo.com



Wer bin ich?



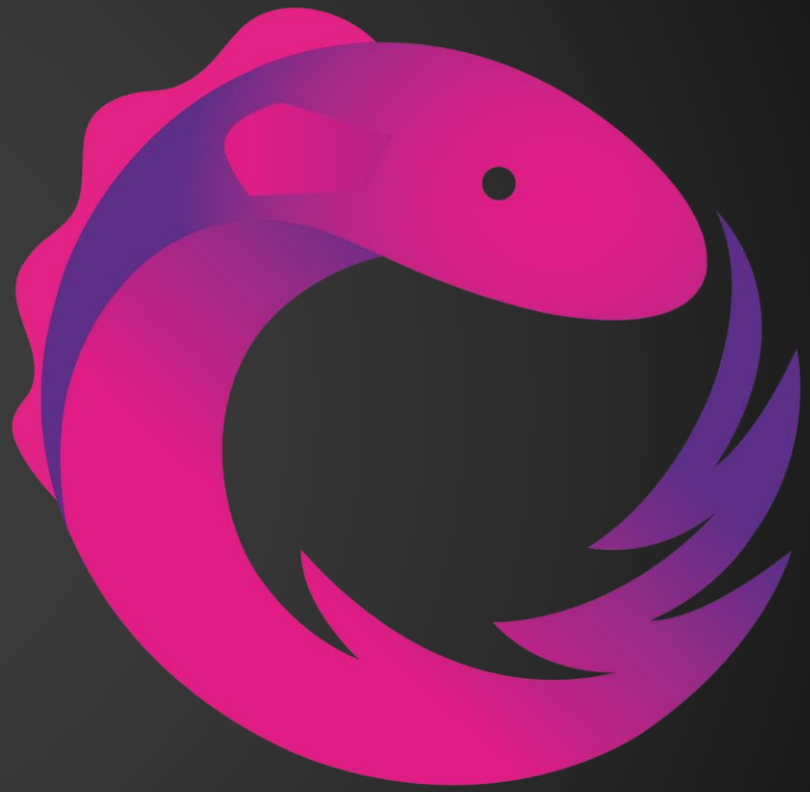


Wer seid ihr?

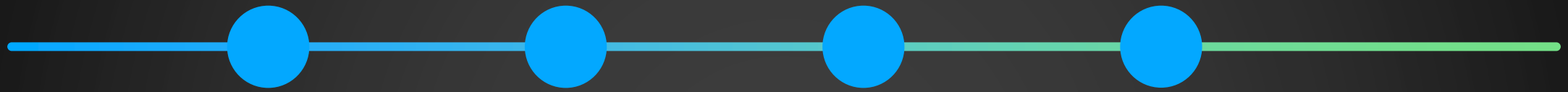
Projekt einrichten

- `npm install -g @angular/cli`
- `ng new rx-workshop`
- `cd rx-workshop`
- `ng serve --hmr=false`

Was ist RxJS?



RxJS – Arbeiten mit Streams



Mischung aus Arrays und Promises

- Sammlung mehrerer Werte
- Über die Zeit verteilt
- „Push“-basiert:
  - „~~Gib mir den nächsten Wert~~“
  - „Hier ist der nächste Wert“

RxJS – Arbeiten mit Streams

Observables

- Produzieren Werte oder Fehler über die Zeit
- Man kann sich über neue Werte benachrichtigen lassen
- Lazy: Es passiert nur etwas, wenn jemand zuhört

Observers

- Werden über neue Werte informiert
- Simple: Funktion
- Complex: Objekt mit next, error und complete Funktionen

Subjects

- Gleichzeitig Observable und Observer

Hands-On

Hands-On

```
import { Observable, Observer, of, Subscription } from 'rxjs';

const simpleObservable: Observable<number> = of(1, 2, 3, 4);

const observer: Observer<number> = {
  next(value): void {
    console.log(value);
  },
  complete(): void {
    console.log('complete');
  },
  error(err): void {
    console.error('error');
  },
};
const subscription: Subscription = simpleObservable.subscribe(
  observer
);
subscription.unsubscribe();
```

Hands-On – Vereinfacht

```
import { of } from 'rxjs';

const simpleObservable = of(1, 2, 3, 4);

const subscription = simpleObservable.subscribe((next) =>
  console.log(next)
);

subscription.unsubscribe();
```

Operatoren

- Funktionen, die ein Observable Transformieren
  - Observable als einziges Argument
  - Observable als Rückgabewert
- Lassen sich dadurch schön hintereinander hängen
  - Können in `observable.pipe(…)` übergeben werden
- „Operator factories“
- Dokumentation
  - Reference: https://rxjs-dev.firebaseapp.com/api
  - Visualisierung: https://rxmarbles.com/
  - Entscheidungsbaum: https://rxjs-dev.firebaseapp.com/operator-decision-tree

Operatoren

```
import { of } from 'rxjs';
import { filter, map } from 'rxjs/operators';

const simpleObservable = of(1, 2, 3, 4);

const transformed = simpleObservable.pipe(
  filter((n) => n % 2 === 0),
  map((n) => n * 2)
);

const subscription = transformed.subscribe((next) =>
  console.log(next)
);

subscription.unsubscribe();
```

Erste Übung – Mit Operatoren arbeiten

- Nutze die `range` Funktion, um ein Observable zu erzeugen, welches die Zahlen von 1 bis 100 emitiert
- Nutze `skip`, um die ersten 20 Werte zu verwerfen
- Nutze `map`, um alle Zahlen zu verdoppeln
- Nutze `filter`, um alle Zahlen zu entfernen, die durch 4 teilbar sind
- Nutze `scan`, um aus der Reihe der Zahlen die kummulierte Summe zu bilden
  (aus 1, 2, 3, 4, 5 soll 1, 3, 6, 10, 15 werden)
  Hinweis: https://rxmarbles.com/#scan
- Subscribe auf das Observable und logge jeden emitierten Wert auf die Konsole
- Vergiss nicht zu unsubscriben

Erste Übung – Mit Operatoren arbeiten

```
import { range } from 'rxjs';
import { filter, map, scan, skip } from 'rxjs/operators';

const obs = range(0, 100).pipe(
  skip(20),
  map((x) => x * 2),
  filter((x) => x % 4 !== 0),
  scan((a, b) => a + b)
);

const sub = obs.subscribe((x) => console.log(x));

sub.unsubscribe();
```

Ein Stream kommt selten allein

- Häufige Kombinationen
  - `merge`: Emitiert alle Werte von allen Quell Observables einzeln
    https://rxmarbles.com/#merge
  - `concat`: Emitiert alle Werte des ersten Observables, danach alle des zweiten usw.
    https://rxmarbles.com/#concat
  - `zip`: Kombiniert die Werte der Observables Paarweise
    https://rxmarbles.com/#zip
  - `combineLatest`: Erzeugt bei jedem emit die aktuellsten Werte in einer Liste
    https://rxmarbles.com/#combineLatest
  - `takeUntil`: Subscribed auf das erste Observable bis zum ersten emit vom Zweiten
    https://rxmarbles.com/#takeUntil
- Erzeugen von Observables mit Timern: `interval` & `timer` (`timer` häufig mit `first`)

Zweite Übung – Observables kombinieren

- Nutze `interval` und erzeuge ein Observable a, welches alle 2 Sekunden emitiert
- Erzeuge ein weiteres Observable b, welches alle 5 Sekunden emitiert (`timer`), wobei du alle emitierten Zahlen umwandelst mit der Funktion `n => String.fromCharCode(n + 65)`
- Erzeuge ein weiteres Observable `timeout`, welches nach 20 Sekunden genau einen Wert erzeugt (`timer` + `first`)
- Erzeuge ein `output` Observable, indem du `a` und `b` mit `zip` kombinierst
- Das `output` Observable soll nur solange Werte emitieren, bis das `timeout` observable einen Wert emitiert (`takeUntil`)
- Subscribe auf das `output` Observable und logge jeden emitierten Wert auf die Konsole
- Wie verhält sich Output, wenn wir `zip` mit `combineLatest` bzw. mit `merge` austauschen? Welches Verhalten würdest du bei `concat` erwarten? Achte bei den Vergleichen vor allem auf den zeitlichen Abstand zwischen den Ergebnissen.

Zweite Übung – Observables kombinieren

```
import { interval, timer, zip } from 'rxjs';
import { first, map, takeUntil } from 'rxjs/operators';

const a = interval(2000);
const b = interval(5000).pipe(
  map((n) => String.fromCharCode(n + 65))
);
const timeout = timer(20000).pipe(first());

const output = zip(a, b).pipe(takeUntil(timeout));
// const output = combineLatest([a, b]).pipe(takeUntil(timeout));
// const output = merge(a, b).pipe(takeUntil(timeout));

output.subscribe((x) => console.log(x));
```

Zweite Übung – Besonderheiten

- `combineLatest` und `zip` müssen warten, bis alle Quellen einen Wert erzeugt haben
- In unserer Situation brauchen wir kein `unsubscribe`, `takeUntil` kümmert sich darum
- In der Praxis gibt es quasi keine Situationen, wo eine Subscription nie beenden wollen!
  → In 95% aller Fälle brauchen wir ein `unsubscribe/takeUntil`

Erzeugen von Streams

- In den seltensten Fällen erzeugen wir ein Observable per Hand
- `timer` oder `interval` für zeitbasierte Observables
- `fromEvent` für `EventEmitter`
  z.B. `fromEvent(buttonElement, 'click')`
- `from` für alle Iterables (Listen, Sets etc.) oder Promises
- Für neue Quellen `new Subject()`
- Für die Ausnahme und für das Verständnis: `new Observable(…)`

Manuelle Observables

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

const manualObservable = new Observable((subscriber) => {
  subscriber.next(1);
  subscriber.next(2);
  subscriber.next(3);
  subscriber.complete();

  return () => {
    console.log('unsub!');
  };
});

manualObservable.subscribe(console.log);
```

Dritte Übung – new Observable

- Nutze den Observable constructor, um ein Observable zu definieren, welches:
  - mit setInterval (MDN Dokumentation) eine aufsteigende Zahlenfolge mit je einer Sekunde Abstand emitiert
  - und nach dem 10. Wert das Observable abschließt (complete)
- Vergiss nicht, das Interval mit clearInterval „aufzuräumen“
- Erzeuge in neues Observable, indem du die Werte des manuellen Observables alle quadrierst. Logge zudem jede der Rechenoperationen auf die Konsole, damit wir einen Einblick in die Anzahl der berechneten Werte erhalten.
- Was erwartest du was passiert, wenn niemand auf das Observable subscribed?
- Was erwartest du was passiert, wenn jemand subscribed und direkt wieder unsubscribed?
- Was erwartest du was passiert, wenn 3 mal subscribe aufgerufen wird?

Dritte Übung – new Observable

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

const manualObservable = new Observable<number>((subscriber) => {
  let i = 0;
  function cleanup(): void {
    clearInterval(interval);
    subscriber.complete();
  }
  const interval = setInterval(() => {
    subscriber.next(i++);
    if (i === 9) {
      cleanup();
    }
  }, 1000);
  return cleanup;
});
const squared = manualObservable.pipe(
  map((n) => {
    console.log(`${n} * ${n} is ${n * n}`);
    return n * n;
  })
);
squared.subscribe(console.log);
```

Dritte Übung – Besonderheiten

- Cold Observables
  - Nichts passiert, solange niemand subscribed
  - Jeder Subscriber bekommt seine eigene Pipeline
- Hot Observables
  - Es gibt eine Quelle, die immer Werte produziert (egal ob subscriber da ist oder nicht)
  - Jeder subscriber bekommt die selben Werte
- → Die allermeisten Observables sind Cold
- Beispiel für Hot Observables: Subjects

Vierte Übung – Vorbereiten von Angular

- Füge das `FormsModule` und das `ReactiveFormsModule` zum `AppModule` hinzu
- Definiere ein neues Feld `inputControl` in der `AppComponent` und belege es direkt mit einem neuen FormControl: `inputControl = new FormControl('');`
- Definiere ein neues Feld `inputValue$` in der AppComponent und instanziere es direkt zu this.inputControl.valueChanges
- In `app.component.html`
  - Entferne den ganzen Inhalt
  - Definiere ein `input` Element und nutze die `[formControl]`-Direktive, um das `input` Element mit `inputControl` zu verknüpfen
  - Füge unter dem `input` ein `p` Element hinzu, welches immer den aktuellen Wert des `inputValue$` observables darstellt (nutze dafür die `async` pipe)

Vierte Übung – AppModule

```
import { NgModule } from '@angular/core';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, FormsModule, ReactiveFormsModule],
  providers: [],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

Vierte Übung – AppComponent

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent {
  inputControl = new FormControl('');

  inputValue$ = this.inputControl.valueChanges;
}
```

Vierte Übung – app.component.html

```
<input [formControl]="inputControl" />
<p>{{ inputValue$ | async }}</p>
```

Vierte Übung – Besonderheiten

- Kein manuelles Subscribe!
  - Jedes manuelle Subscribe muss auch wieder aufgeräumt werden
  - Gefahr das aufräumen zu vergessen
- async pipe subscribed und unsubscribed eigenständig
  - Jede async pipe subscribed
  - Deklaration in `*ngIf` oder `*ngrxLet` (https://ngrx.io/guide/component/let) ziehen
- FormControl ist unsere Datenquelle
- inputValue$ ist nur ein Observable was sich aus der Datenquelle ableitet

Ausflug: HttpClient

- Angular eigene Bibliothek für Netzwerkanfragen
- Basiert auf observables

```
const response: Observable<T> = this.httpClient.get<T>(
  `https://api.github.com/search/users?q=${name}`
);
```

- Cold Observable! Request wird erst abgeschickt, wenn jemand subscribed
- Im AppModule muss zunächst das HttpClientModule hinzugefügt werden
- Anschließend kann eine Instanz von HttpClient in die AppComponent injected werden.

Fünfte Übung

- Füge das `HttpClientModule` den imports des `AppModules` hinzu
- Lass dir in der `AppComponent` eine Instanz von `HttpClient` injecten
- Definiere in der `AppComponent` eine neue Funktion `loadUsersLike(name: string)`, die die GitHub API anspricht, um alle Nutzer mit dem Suchstring im Namen zu laden.
  `https://api.github.com/search/users?q=${name}`
- Wenn die Funktion mit einem String kürzer als 3 Buchstaben aufgerufen wird, soll sie ein Dummy-Observable zurückgeben `of({ items: [] })`
- Rufe diese Funktion im `constructor` der `AppComponent` mit einem Beispielstring auf und logge das Ergebnis auf die Konsole
- **Bonusaufgabe:** Prüfe das Ergebnis auf der Konsole und schreibe in der `app.component.ts` ein interface/type, der das Resultat von der GitHub API definiert und nutze diesen Typ in `loadUsersLike`
- **Bonusaufgabe**: Lagere diese Funktion in einen eigenen Service aus

Fünfte Übung

```
  constructor(private httpClient: HttpClient) {
    this.loadUsersLike('andrewgreen').subscribe(console.log);
  }

  loadUsersLike(name: string): Observable<GitHubResponse> {
    if (name.length < 3) {
      return of({ items: [] });
    }
    const response: Observable<GitHubResponse> = this.httpClient.get<GitHubResponse>(
      `https://api.github.com/search/users?q=${name}`
    );
    return response;
  }
}
type GitHubResponse = {
  items: {
    avatar_url: string;
    login: string;
  }[];
};
```

Transformieren in andere Observables

- Problem: Häufig ist der Output einer Transformation wieder ein Observable.
  → zum Beispiel unser Anwendungsfall: Wir wollen ein Textfeld mit Vorschlägen umsetzen. Jeder neue Wert des Textfeldes (äußeres Observable), soll jetzt in Daten vom Server transformiert werden (inneres Observable).
  - `concatMap`: https://rxmarbles.com/#concatMap, warte auf completion des inneren Observables, bevor das nächste Observable erzeugt wird. Z.B.: sequentielles Speichern
  - `exhaustMap`: Gleiches wie `concatMap` außer, dass Werte im äußeren Observable ignoriert werden, solange das innere Observable noch läuft: Klick auf speichern Button
  - `switchMap`: https://rxmarbles.com/#switchMap, brich das innere Observable ab, wenn der nächste äußere Wert kommt. Z.B.: Vorschläge in einem Autocomplete
  - `mergeMap`: https://rxmarbles.com/#mergeMap, starte das nächste innere Observable einfach gleichzeitig und kombiniere die Outputs von allen inneren, die noch laufen. z.B. Crawling

Sechste Übung

- Definiere ein neues Feld auf AppComponent: `suggestions$`
- Nutze den `debounceTime` Operator (Dokumentation), um zu warten, bis der Nutzer mit Tippen fertig ist (Verzögerung von 250ms)
- Nutze `distinctUntilChanged`, (Dokumentation) um zu verhindern, dass zwei mal hintereinander der gleiche Suchstring abgeschickt wird
- Für jeden neuen Eingabe-Wert, soll `loadUsersLike` genutzt werden, um für die Eingabe die richtigen Vorschläge zu laden. Wähle für diesen Usecase den richtigen Operator.
- Transformiere das Resultat noch so, dass in `suggestions$` ein `Observable<GithubUser[]>` resultiert.
- Nutze die `async` pipe und `*ngFor`, um alle Vorschläge in der UI darzustellen.

Feinschliff

- Zurücksetzen, immer wenn jemand etwas eingibt: Wir mergen in unser `suggests$` observable einfach ein Observable, was für jeden `inputValue` ein leeres Array zurück gibt.
- Bug: Was passiert, wenn es noch einen subscriber gibt auf `suggestions$?` Für jeden Subscriber wird der Request ausgeführt.
  → Lösung: Der `share` Operator: Sorgt dafür, dass die Pipeline für alle folgenden Subscriber geteilt wird. Bei verspäteten subscribern, wurden dadurch evtl. Werte verpasst.
- Fehlerbehandlung:
  - `retry`, um Netzwerkrequest zu wiederholen
  - `catchError`, um Fehler in ein funktionierendes Observable umzuwandeln

Feinschliff

```
suggestions$ = merge(
  this.inputValue$.pipe(
    debounceTime(250),
    distinctUntilChanged(),
    switchMap((search) =>
      this.loadUsersLike(search).pipe(
        catchError((err) => of({ items: [] }))
      )
    ),
    map((response) => response.items)
  ),
  this.inputValue$.pipe(map(() => []))
).pipe(share());
```

Extrameile

- Erwartetes Verhalten:
  - Sobald der Nutzer einen Buchstaben eintippt, sollen die aktuellen Suggestions verschwinden
  - Sobald der eigentliche Netzwerkrequest startet, soll “Loading” dargestellt warden
  - Fehler beim Laden der Daten sollen in der UI dargestellt warden
- Implementierungstipps
  - Zustand explizit aufteilen in idle, success, error und loading
  - KEIN subscribe in der Komponente

Extrameile

- Lösung in `appExtraMile.component.ts`

esveo.com/s/fb-rx

Andreas Roth

andreas.roth@esveo.com
academy.esveo.com