Gemeente Haarlem

# *Angular - Maatwerk*

Peter Kassenaar
info@kassenaar.com

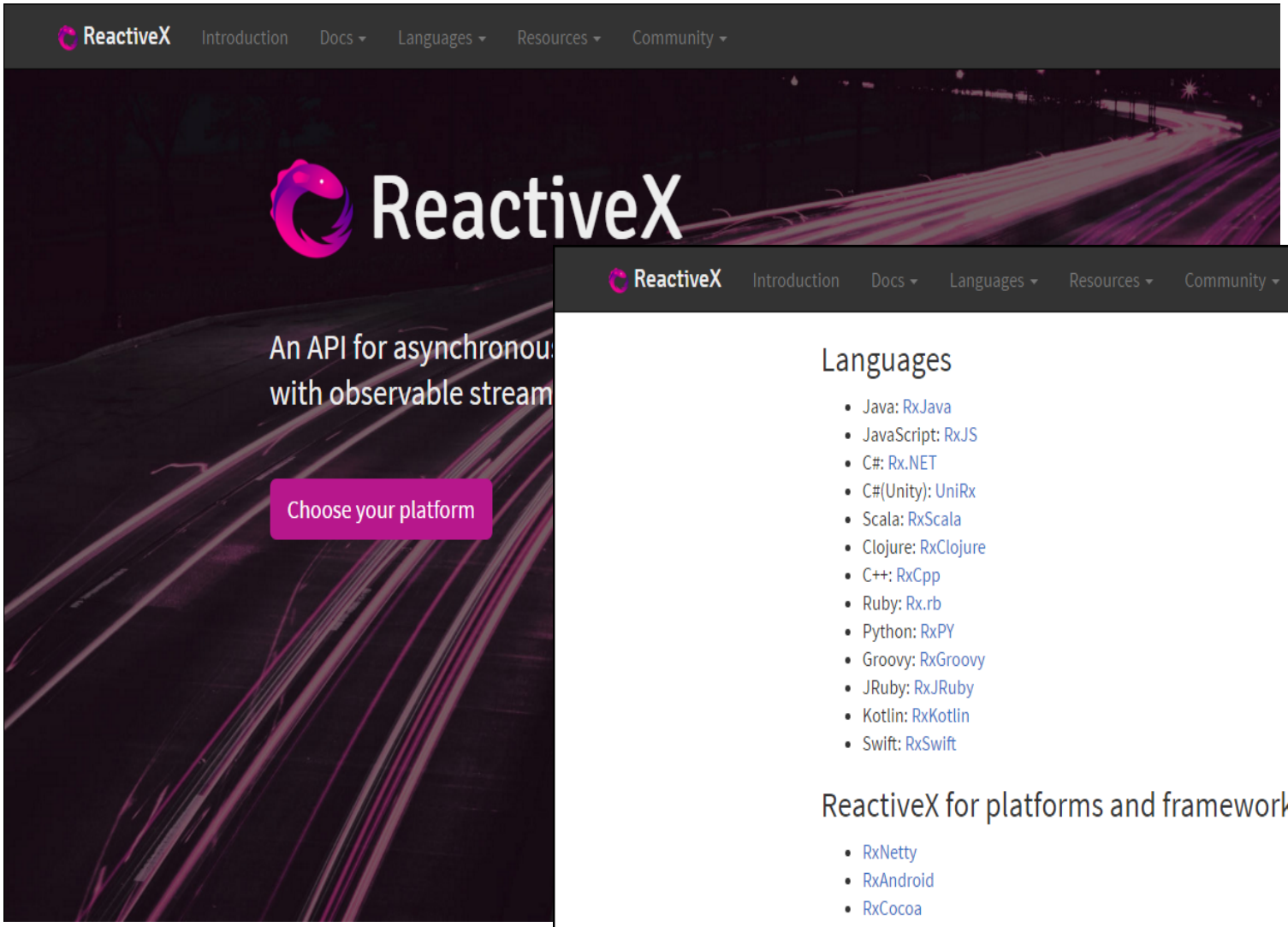
# Async services met RxJS / Observables

Reactive programming with asynchronous streams

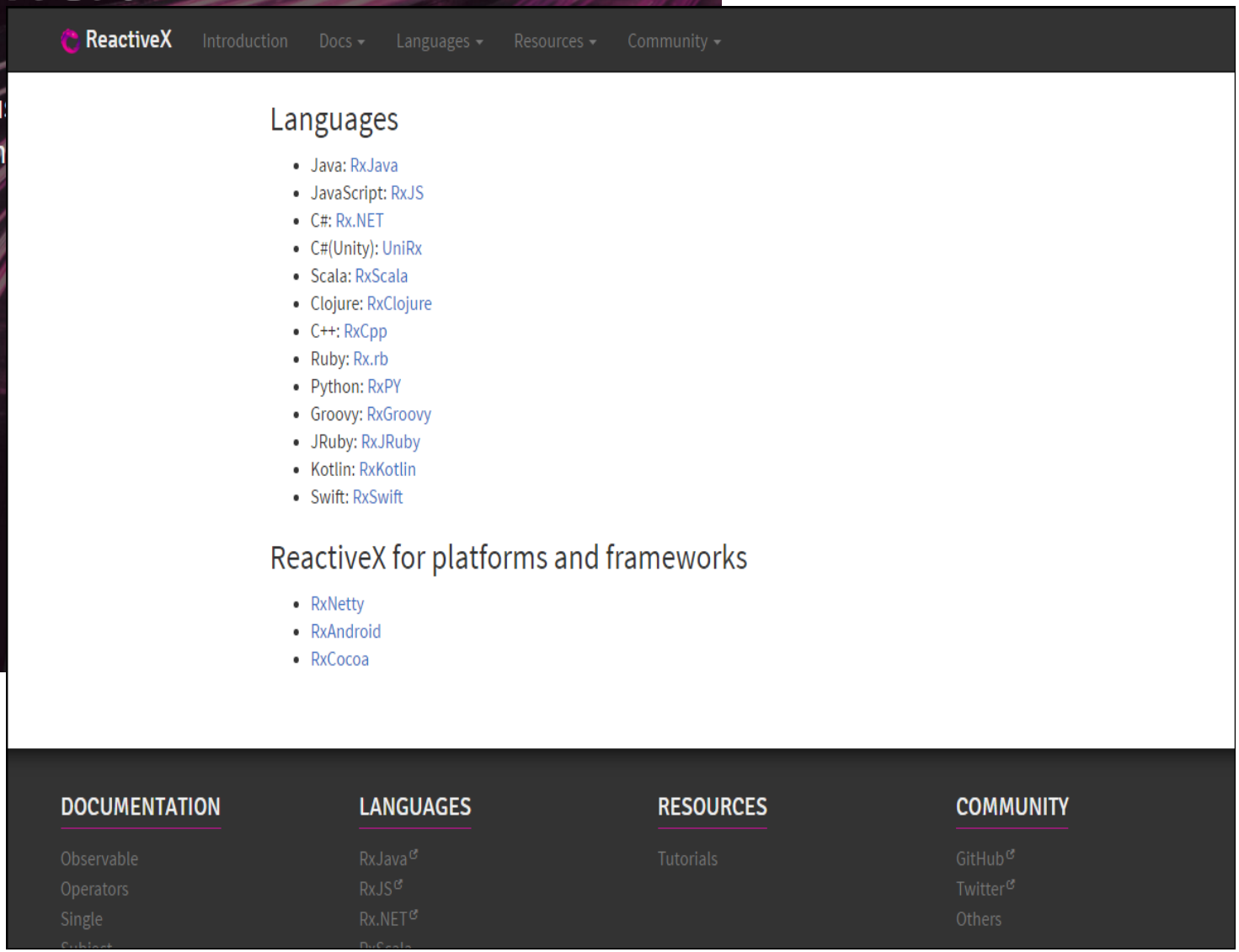
# Async Services

- Fetching static data: *synchronous* action
- Working via `HttpClient`: *asynchronous* action
- Angular 1 and others: `Promises`
- Angular : `Observables`

Also Angular : ReactiveX library RxJS




http://reactivex.io/

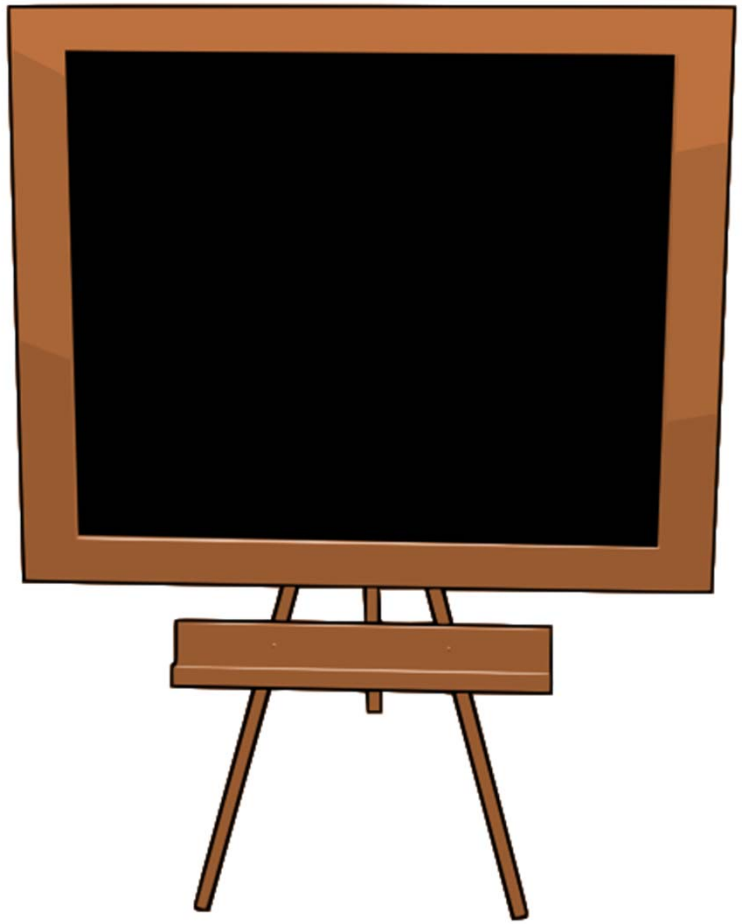


# The observable design pattern

*“Understanding the*

*observable stream”*

# Explanation



# Why Observables?

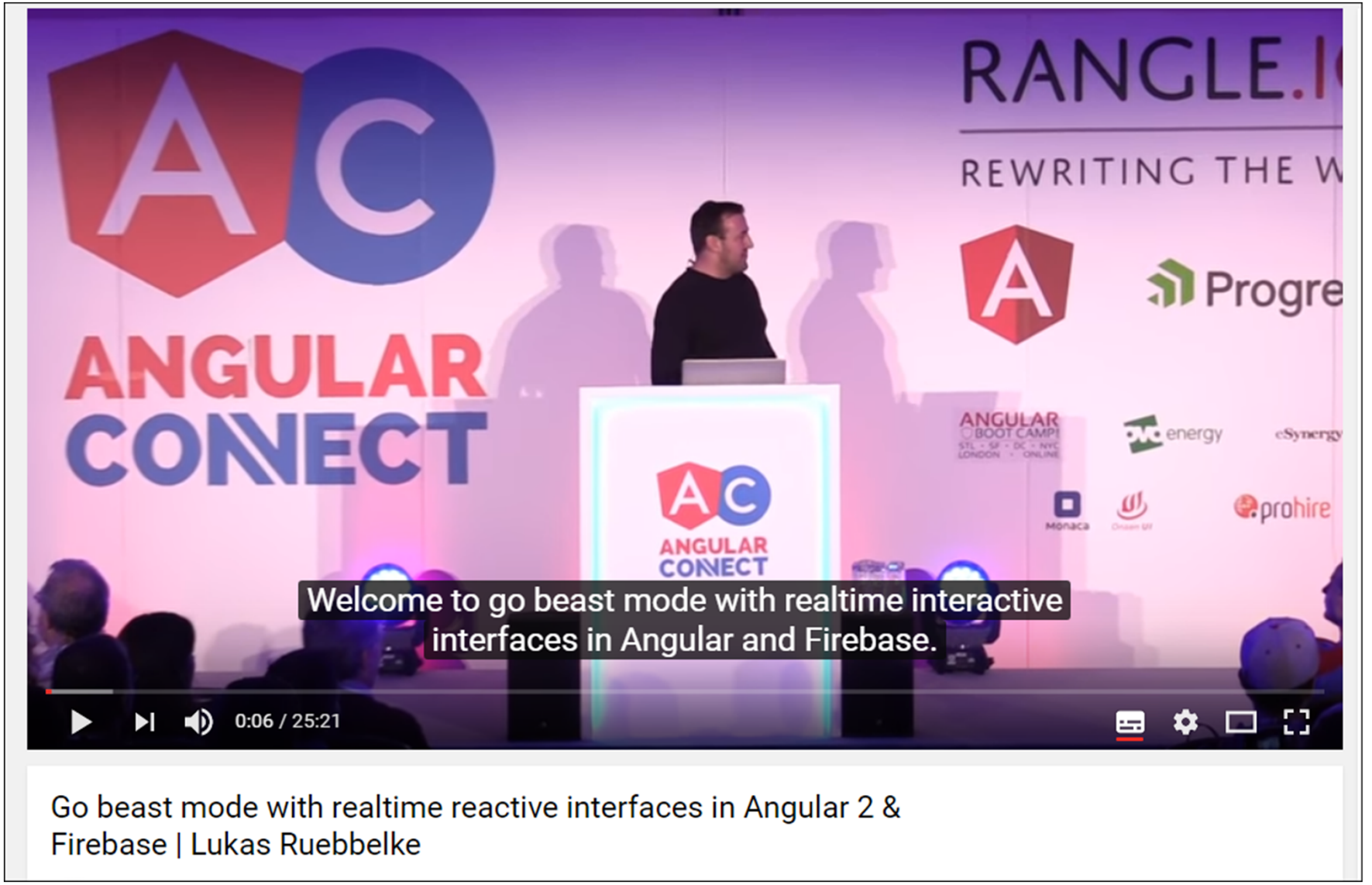
*We can do much more with observables than with promises.*

*With observables, we have a whole bunch of operators to pull from, which let us customize our streams in nearly any way we want.*

https://auth0.com/blog/2015/10/15/angular-2-series-part-3-using-http/

# Observables and RxJs

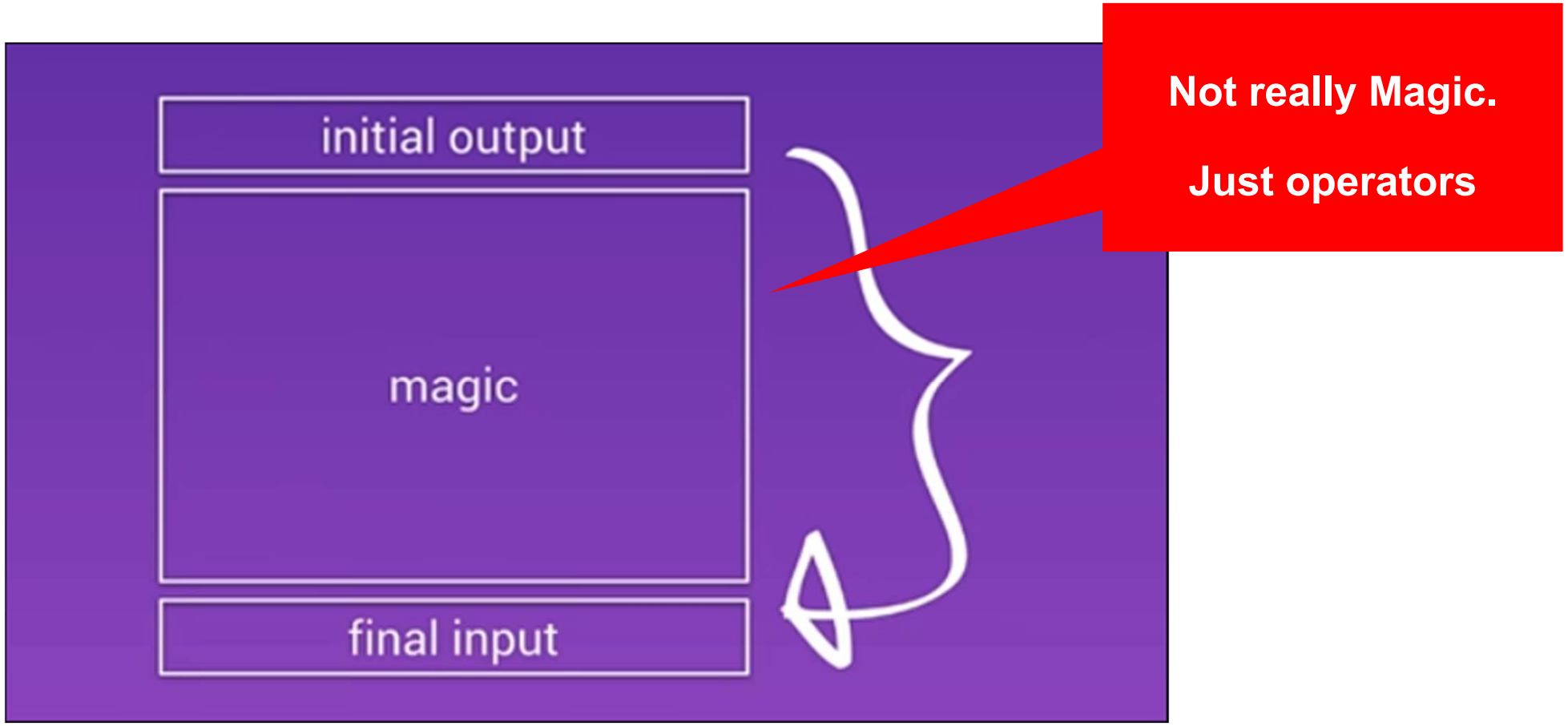
- “Reactive Programming”
  - *“Reactive programming is programming with asynchronous data streams.”*
  - https://gist.github.com/staltz/868e7e9bc2a7b8c1f754
- Observables have a lot of extra options, as opposed to Promises
  - Mapping
  - Filtering
  - Combining
  - Cancel
  - Retry
  - …
- Consequence: no more `.success()`, `.error()` and `.then()` chaining!




https://www.youtube.com/watch?v=5CTL7aqSvJU

https://youtu.be/5CTL7aqSvJU?t=4m31s

# "The observable sandwich"



# Most common usage of observables: http

Modern applications are moduleless

provide the imported `httpClient()`

```
// app.config.ts
import …;
import {provideHttpClient} from '@angular/common/http';

export const appConfig: ApplicationConfig = {
  providers: [
    provideHttpClient(),
    …
  ]
};
```

# In classical applications

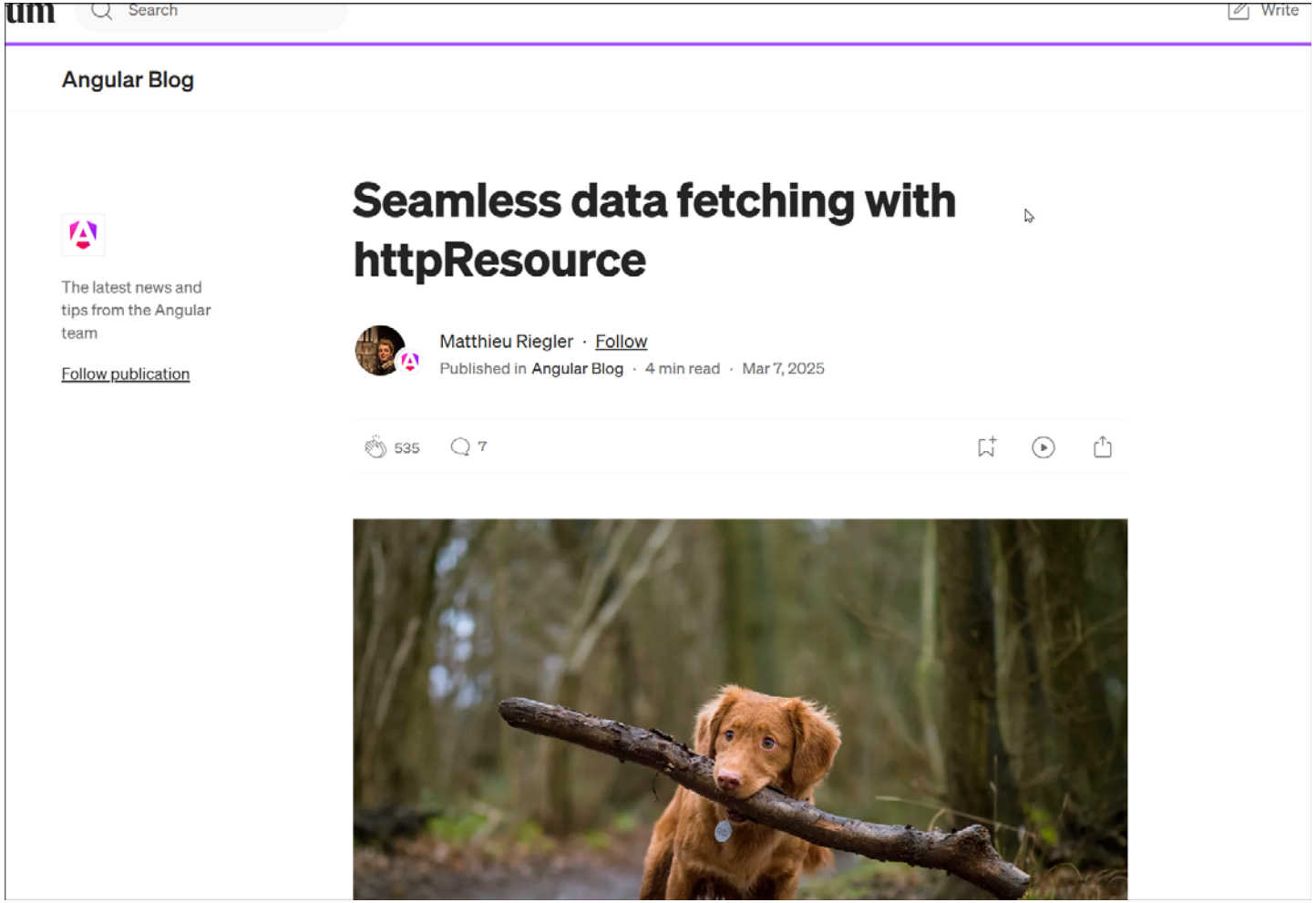
When (still) using ngModules,

import `httpClientModule()`

```
// Angular Modules
import …
import {HttpClientModule} from "@angular/common/http";

// Module declaration
@NgModule({
    imports      : [BrowserModule, HttpClientModule],
    declarations: [AppComponent],
    bootstrap    : [AppComponent],
})
export class AppModule {
}
```

# In even newer applications (v19.2+)

- Angular httpResource available.
- https://blog.angular.dev/seamless-data-fetching-with-httpresource-



# Classical: inject http in Component / service

- Create an `http` variable, of type `HttpClient`
- Use Constructor Injection, or `inject()` function

```
export class AppComponent {
  …
  constructor(private http: HttpClient) {}

}
```

```
export class AppComponent {
  // using inject()
  private http = inject (HttpClient);
}
```

https://angular.dev/guide/http

# Making an http call, for instance

```
this.http.get<City[]>('assets/data/cities.json')
      .pipe(
         delay(…),
         map(…)
      )
     .subscribe((result:City[]) => {
       //... Do something
   });
```

Initial Output

Optioneel:

operator(s)

Final Input

# Using the async pipe

Automagically `.subscribe()` and `.unsubscribe()`

# Async Pipe

- On `.subscribe()`, you actually need to `.unsubscribe()` to avoid memory leaks
  - Best practice
  - Not *really* necessary on HTTP-requests, but you do in other subscriptions.
- No more manually `.subscribe()` and `.unsubscribe()`:
  - Use Angular async pipe

# Where to change / update your code:

Component:

```
1. cities$: Observable<City[]>; // Now: Observable to Type

…
2. ngOnInit() {
   // Call service, returns an Observable
   this.cities$ = this.cityService.getCities()
}
```

View:

```
3. <li *ngFor="let city of cities$ | async”>
```

# OR: using the modern @for syntax with async

For instance, retrieving users instead of cities:

```
users$: Observable<any[]> | undefined;
ngOnInit(){
    …
  this.users$ = this.http.get<any[]>
      ('https://jsonplaceholder.typicode.com/users’);
}
```

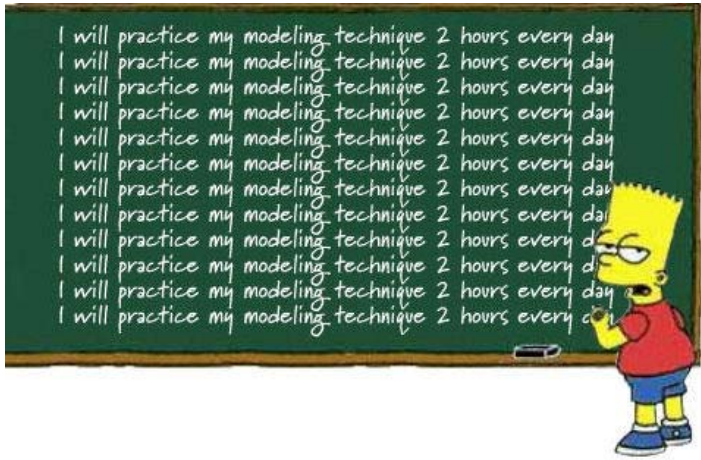
```
@for (user of users$ | async; track user.id) {
  <div>
    {{ user.name }}
  </div>
}
```

# Example API's

- https://pokeapi.co/ - Pokemon API
- http://openweathermap.org/API (weather forecast)
- https://jsonplaceholder.typicode.com/ random users, posts, photos
- http://ergast.com/mrd/ - Ergast Motor (F1) API
- http://www.omdbapi.com/ - Open Movie Database
- http://swapi.dev/ - Star Wars API
- See also `JavaScript APIs.txt` with other API's.

# Workshop

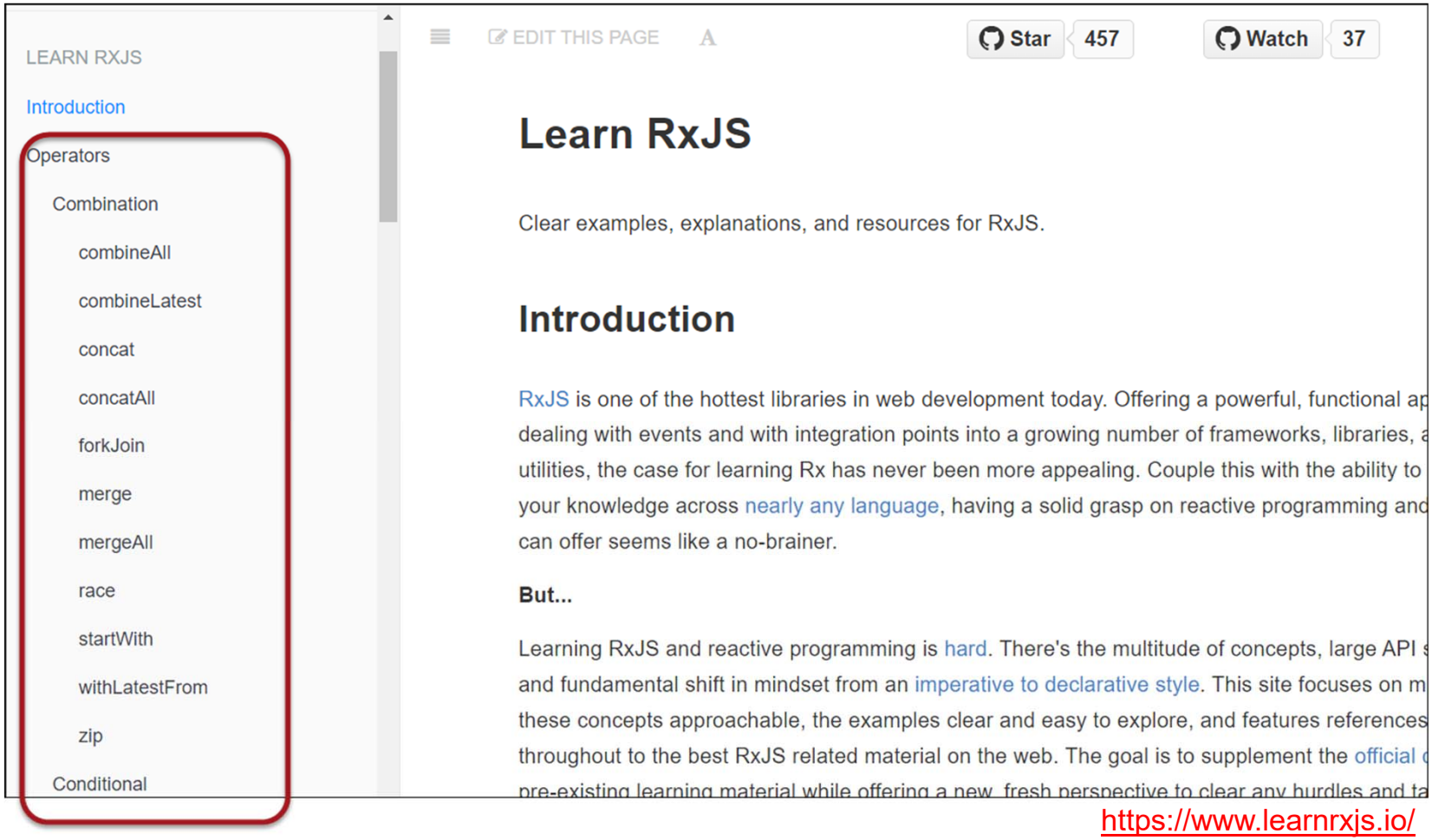
- Pick one of your own projects, or see for instance:
  - `../210-services-live`
- Create a small application using one of the API's in the file `JavaScript API's.txt`, using RxJS-calls, for example
  - Pokemon API
  - Kenteken API
  - OpenWeatherMap API
  - ...



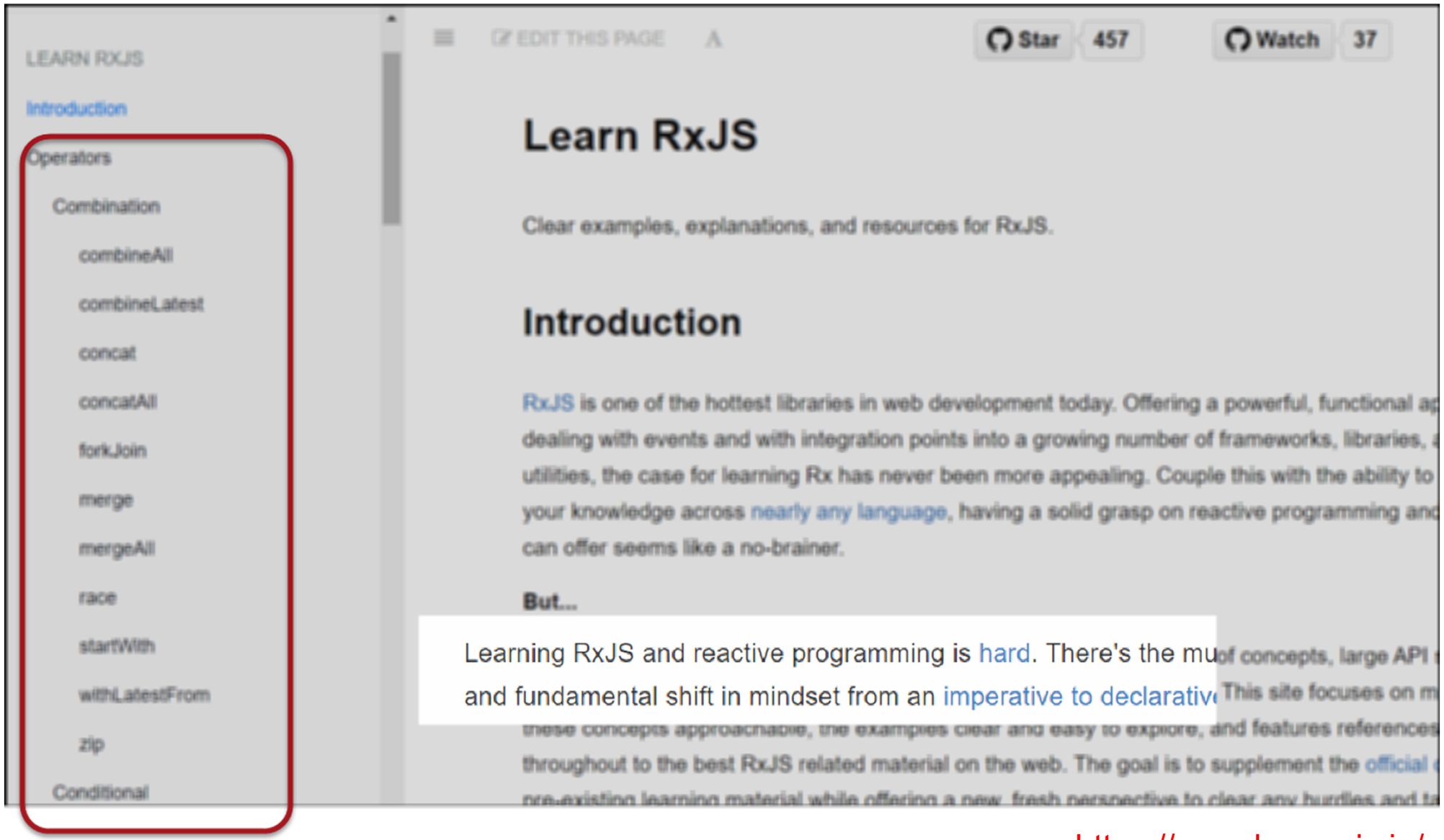
# More on operators

Transforming your stream in the `.pipe()` function

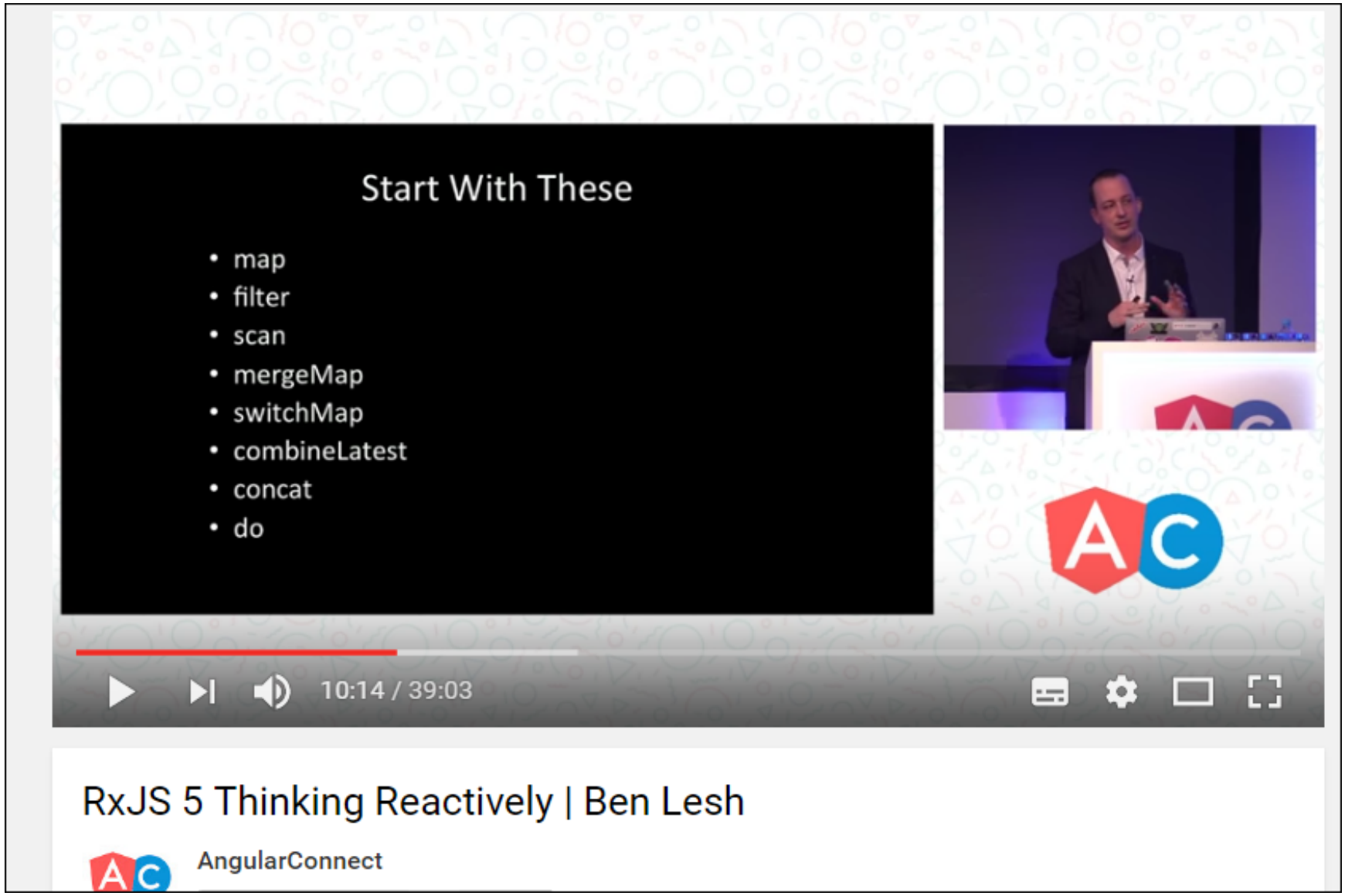
# The Problem with RxJS Operators...




https://www.learnrxjs.io/




https://www.learnrxjs.io/

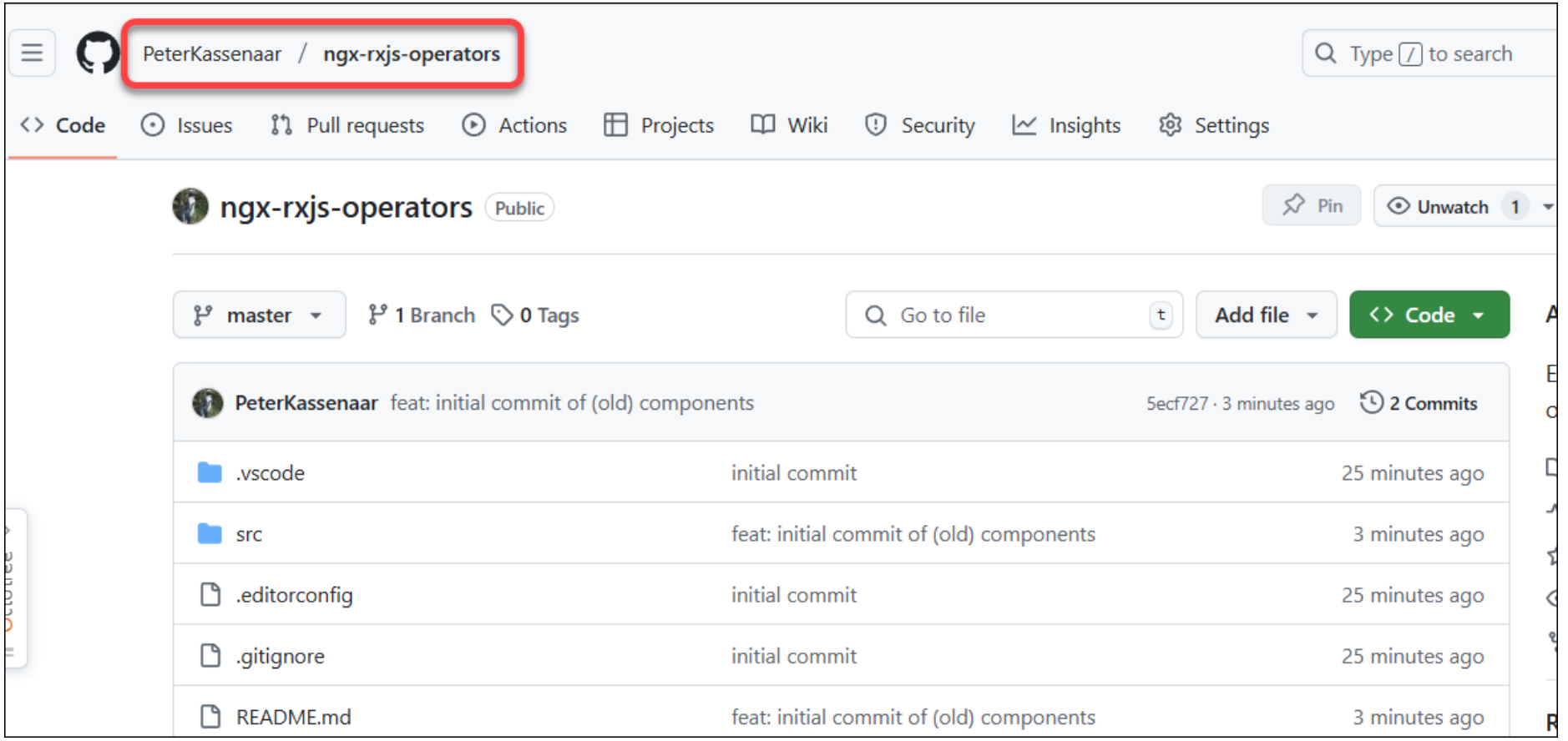



https://www.youtube.com/watch?v=3LKMwkuK0ZE

*"Don't try to "Rx everything". Start with the operators you know, and go imperatively from there. There's nothing wrong with that."*

**- Ben Lesh**

# Example code




github.com/PeterKassenaar/ngx-rxjs-operators

# This part of the presentation - 2 sections

1. Basic Streams
    - Create a stream, or multiple streams from scratch, based on a UI-element
    - Subscribe to that stream(s) and do something
2. Operators

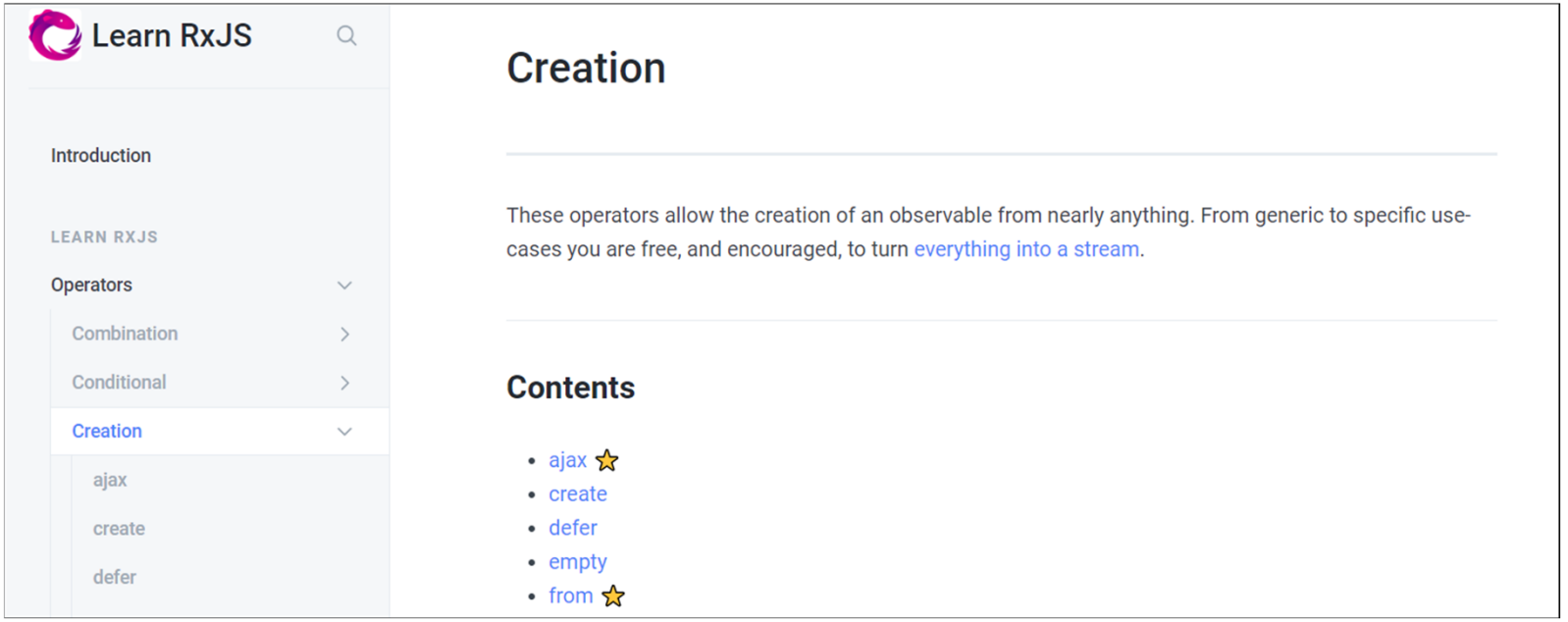
    Demo the purpose and inner workings of some often used operators

# Basic streams

Creating observables from scratch

# Turn something into an observable

- Basic creation operators
  - `create ()` – create an observable manually
  - `from()` – turn an array, promise or iterable into an observable
  - `fromEvent()` – turn an event into an observable
  - `of()` – turn a variable into an observable, emit it's value(s) and complete
- https://www.learnrxjs.io/learn-rxjs/operators/creation

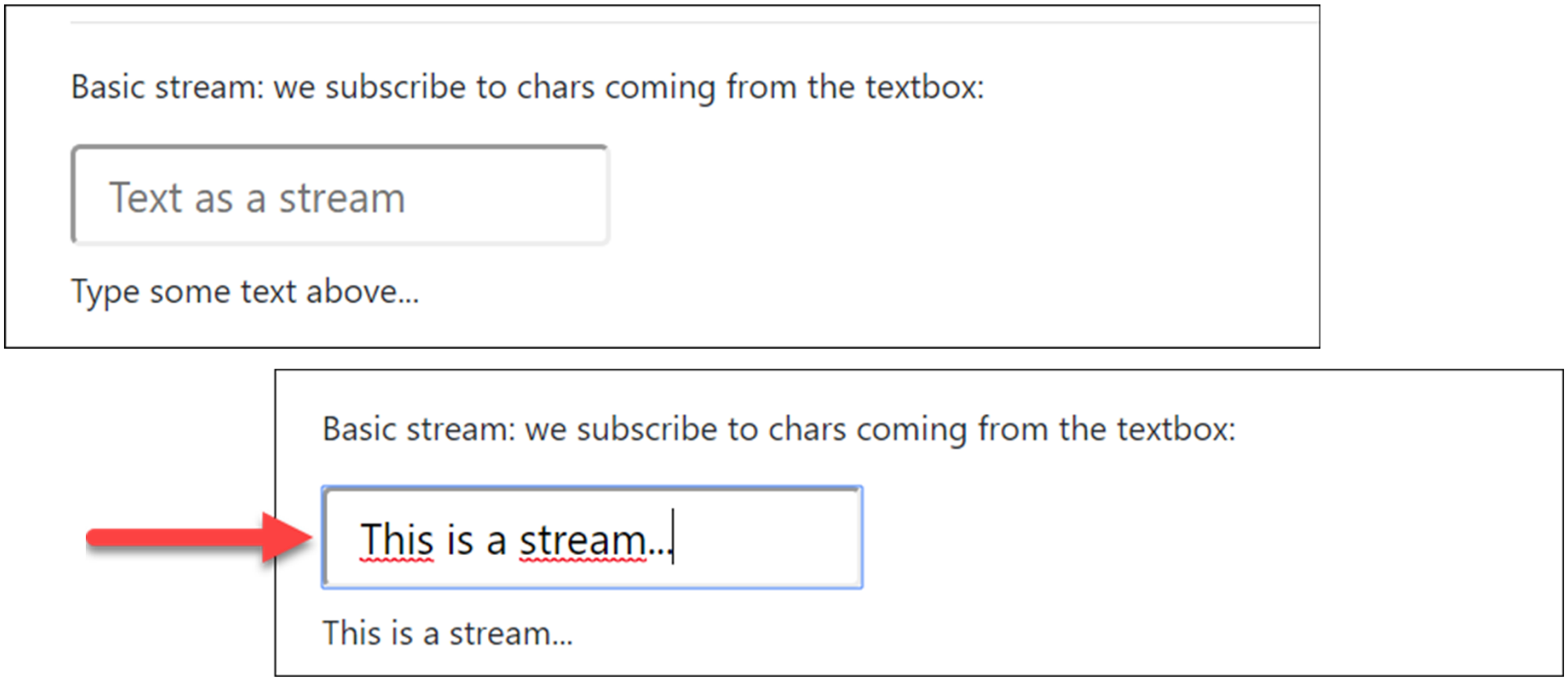


# Creating an observable from a textbox

```
<input class="form-control-lg" type="text" #text1
          id="text1" placeholder="Text as a stream">
<p>{{ textStream1 }}</p>
```

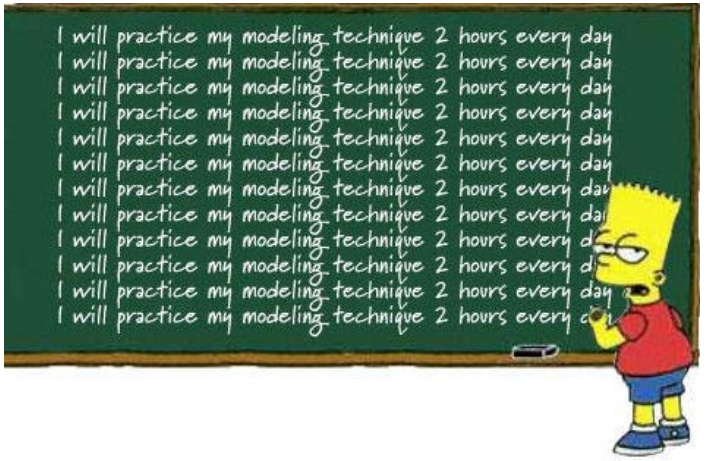
```
textStream1 = 'Type some text above...';
@ViewChild('text1', {static: true}) text1; // two parameters for @ViewChild()

// Suggestion: break the onInit() up in smaller functions
ngOnInit(): void {
  this.onTextStream1();
}

onTextStream1() {
  fromEvent(this.text1.nativeElement, 'keyup')
    .subscribe((event: any) => this.textStream1 = event.target.value);
}
```

# Result


Basic stream: we subscribe to chars coming from the textbox:
Text as a stream
Type some text above...
Basic stream: we subscribe to chars coming from the textbox:
This is a stream...
This is a stream...

# Workshop

- Create a textbox.
  - Add items that are typed in, to a Todo-list
  - Use an observable to subscribe to keyup-events.
- Capture key events, test if the key pressed is the Enter-key
  - If it's Enter, Add the current value of the textbox to a static array `todoItems[]`
- Example `../basic-stream/basic-stream.component.ts`
  - Start from scratch or expand this component



# Using the pipe()

- We can transform the stream by adding operators to the pipeline
- Inside the pipeline we calculate new values and bind them to the UI, using generic attribute binding […]

```
<button #btnRight class="btn btn-secondary">Right</button>
Mario is moved by observable streams from the button click.
<div>
  <img id="mario" #mario
       [style.left]="position.x + 'px'"
       [style.right]="position.y + 'px'"
       src="../assets/img/mario-1.png" alt="Mario">
</div>
```

# Inside our class

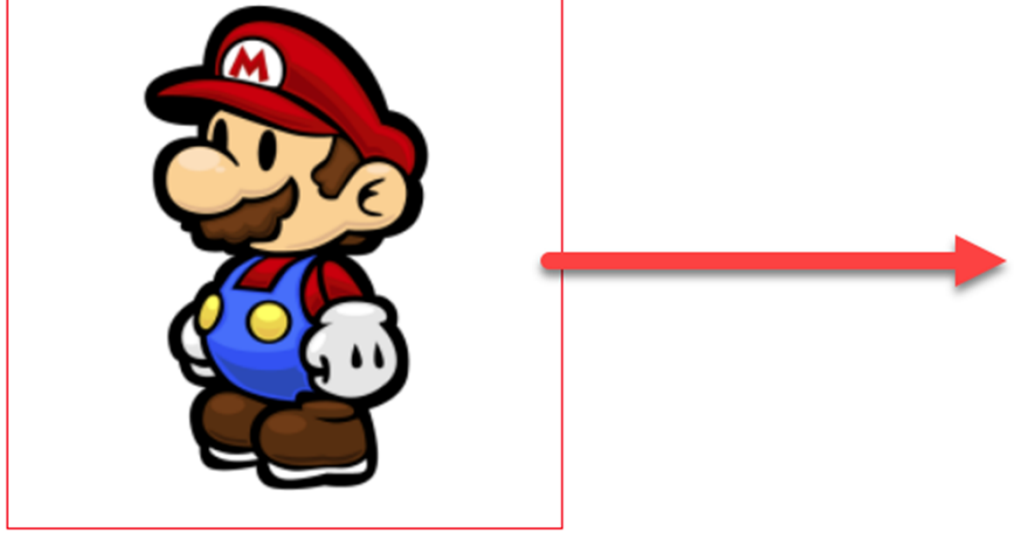
```
// mario
position: any;
@ViewChild('btnRight', {static: true}) btnRight;
```

```
fromEvent(this.btnRight.nativeElement, 'click')
  .pipe(
    map(event => 10), // 1. map the event to a useful value, in this case 10px
    startWith({x: 100, y: 100}), // 2. start with an object of { 100, 100}.
    scan((acc: any, current: number) => { // 3, use the scan operator as reducer function.
      return {
        x: acc.x + current,
        y: acc.y
      };
    })
  )
  .subscribe(result => {
    this.position = result;
  });
```

The result of the previous operator is the input of the next operator in the pipeline

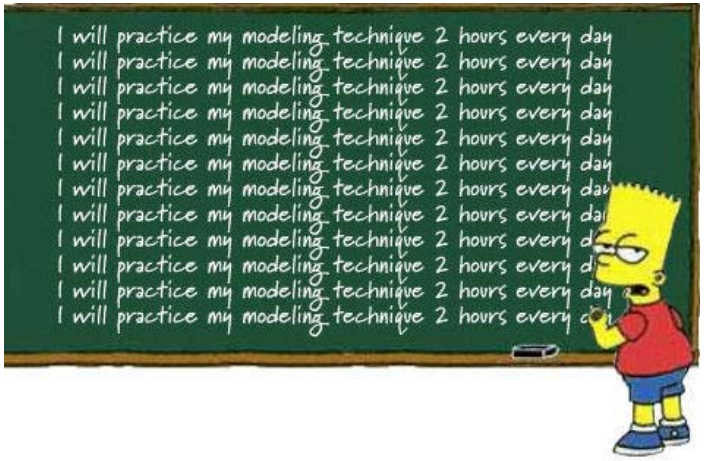
# Result

Right Mario is moved by observable streams from the button click.



# Workshop

- Create  button that moves Mario to the left.
- Example `../basic-stream/basic-stream.component.ts`
  - Start from scratch or expand this component

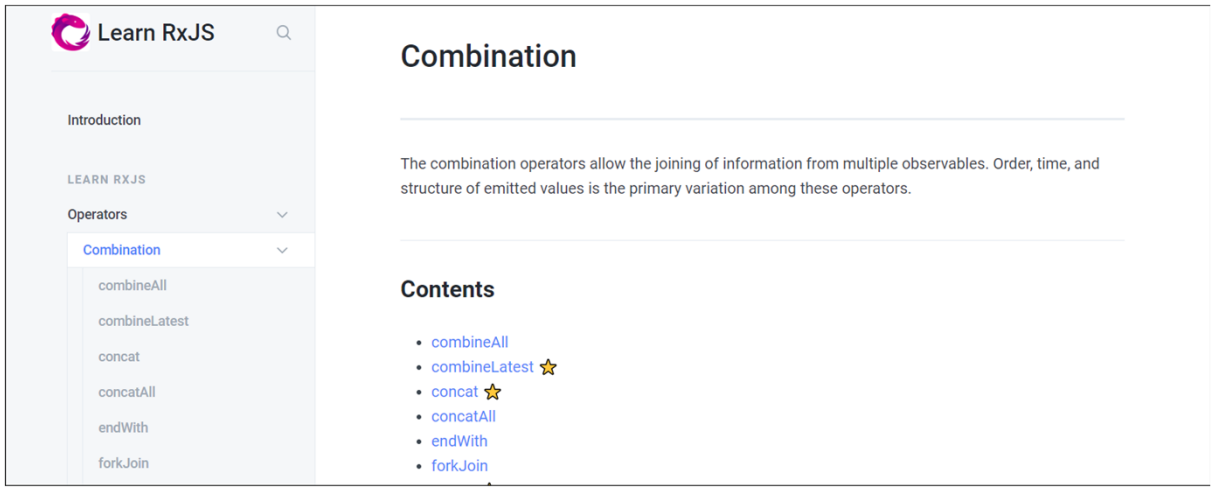


# Combining streams

There are **a *lot*** of options to combine multiple streams into one stream

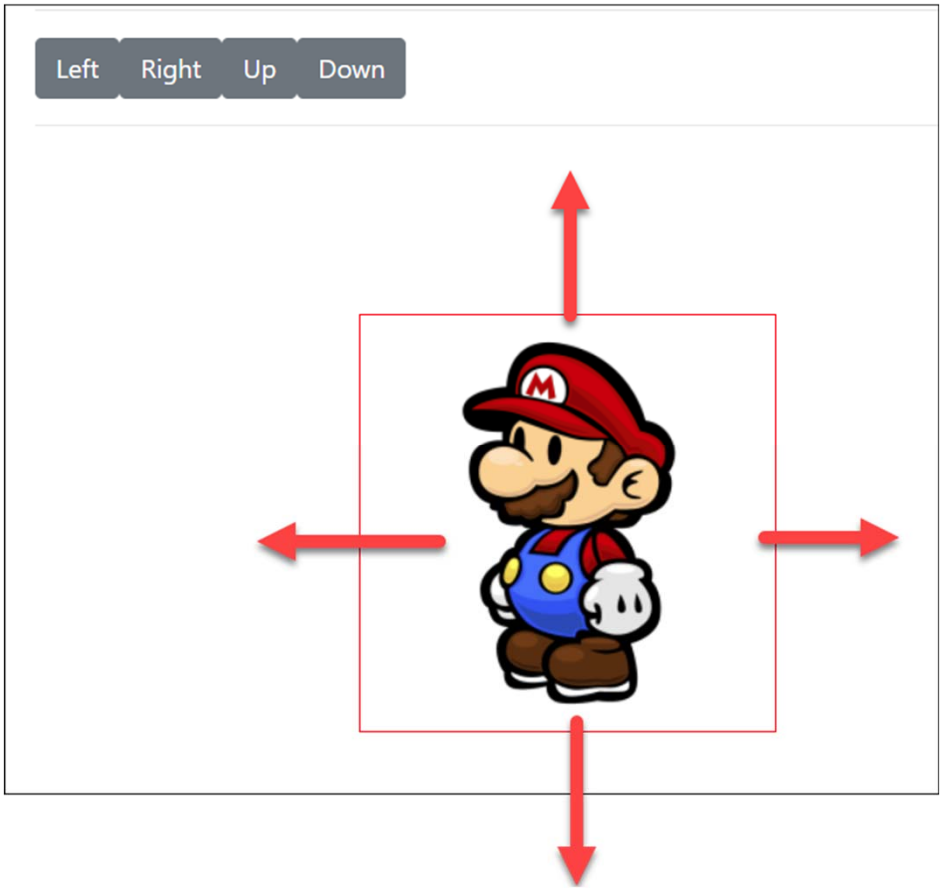
# Options for combining streams

- `combineLatest()` - When any observable emits a value, emit the last emitted value from each
- `concat()` - Subscribe to observables in order as previous completes
- `forkJoin()` - When all observables complete, emit the last emitted value from each
- `merge()` - Turn multiple observables into a single observable
- `startWith()` - Emit given value first
- https://www.learnrxjs.io/learn-rxjs/operators/combination



# Use case: move Mario

- Create four buttons to move Mario in different directions
  - all combined in one stream and one subscription



# Template

```
<button #btnLeft> Left </button>
<button #btnRight> Right </button>
<button #btnUp> Up </button>
<button #btnDown> Down </button>
<hr>

<img id="mario" #mario
     [style.left]="position.x + 'px'"
     [style.top]="position.y + 'px'"
     src="../assets/img/mario-1.png" alt="Mario">
```

```
position: any;
@ViewChild('btnLeft', {static: true}) btnLeft;
@ViewChild('btnRight', {static: true}) btnRight;
@ViewChild('btnUp', {static: true}) btnUp;
@ViewChild('btnDown', {static: true}) btnDown;

ngOnInit(): void {
  // Create multiple streams
  const right$ = fromEvent(this.btnRight.nativeElement, 'click')
    .pipe(
      map(event => {
        return {direction: 'horizontal', value: 10};
      }) // 10 px to the right
    );
  const left$ = fromEvent(this.btnLeft.nativeElement, 'click')
    .pipe(
      map(event => {
        return {direction: 'horizontal', value: -10};
      }) // -10 px to the left
    );
  …
  // combine our streams
  merge(right$, left$, up$, down$)
   .pipe(
    startWith({x: 200, y: 100}),
    scan((acc: any, current: any) => {
    return {
        x: acc.x + (current.direction === 'horizontal' ? current.value : 0),
        y: acc.y + (current.direction === 'vertical' ? current.value : 0)
      };
    })
  ).subscribe(result => {
    this.position = result;
  });
```

Create 4 different streams

Merge the streams

One subscriber

# Sequencing streams

Use streams to start other streams so you can use the combined behavior

# Use case: drag & drop

- We want to move Mario around with the mouse: drag-n-drop
- We compose different streams for that:
  - `down$`, when the mouse is pressed
  - `move$`, when the mouse is moved, return the current mouse position
  - `up$`, when the mouse is released
- Again, we have only one (1) subscription
  - Subscribe to the `down$`. This is the initiator
  - After the initial event, switch to the `move$`: using the `switchMap()` operator
  - But only until the `up$` event occurs!
  - Then complete the observable an release Mario
- This translates to the following code:

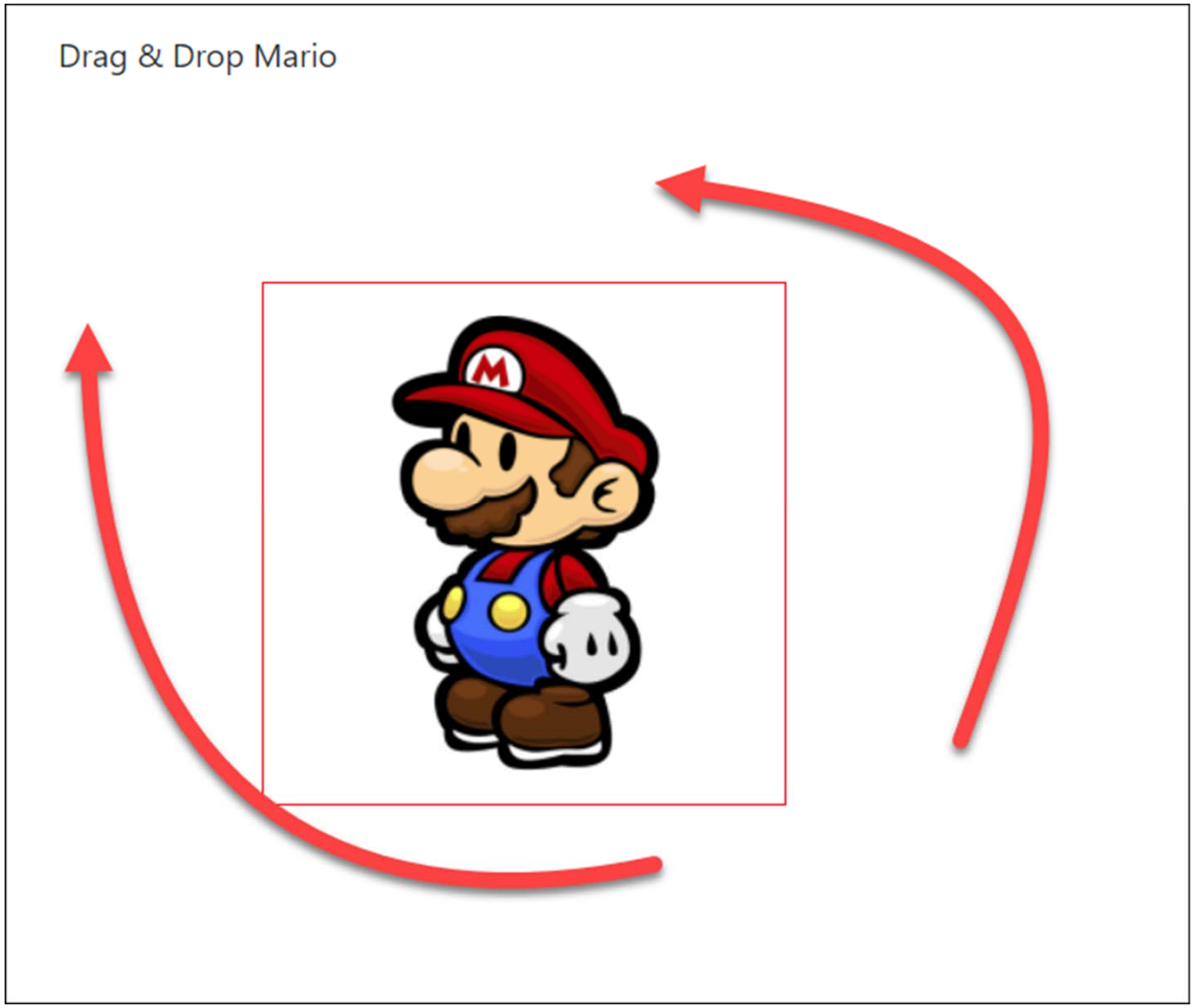
```
// correction factor for image and current page. Your mileage may vary!
  const OFFSET_X = 180;
  const OFFSET_Y = 280;

  // 1. With Drag and drop, first you capture the mousedown event
  const down$ = fromEvent(document, 'mousedown');

  // 2. What to do when the mouse moves
  const move$ = fromEvent(document, 'mousemove')
    .pipe(
      map((event: any) => {
        return {x: event.pageX - OFFSET_X, y: event.pageY - OFFSET_Y}; // OFFSET as correction factor
      })
    );

  // 3. Capture the mouseup event
  const up$ = fromEvent(document, 'mouseup');

  // 4. extend the down$ stream and subscribe
  down$
    .pipe(
      // IF the down$-event happens, we ar no longer interested in it. Instead,
      // we switch the focus to the move$ event.
      // BUT: we are only interested in the move until the mouse is released again.
      // So we add *another* pipe and use the takeUntil() operator.
      switchMap(event => move$.pipe(
        takeUntil(up$)
      )),
      startWith(this.position)
    )
    .subscribe(result => this.position = result);
}
```

**Compose streams**

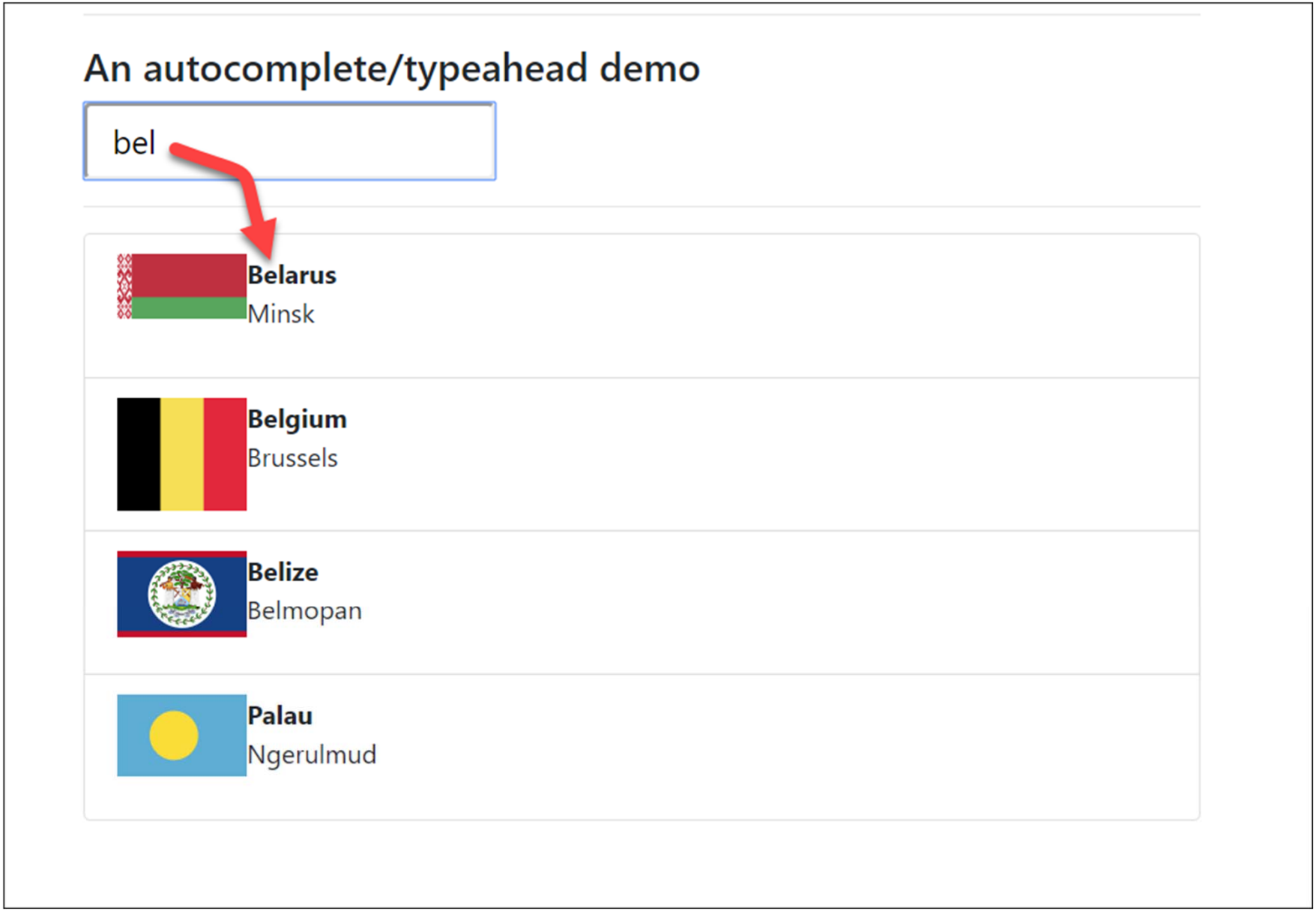
**Switch execution to another observable**

**Subscribe and update position**

# Result



# An autocomplete / typeahead demo




`typeahead-stream.component.html|ts`

# Template

As per usual – little HTML

```
<h4>An autocomplete/typeahead demo</h4>
<!--Get the keyword, the class is subscribed to this inputbox-->
<input type="text" placeholder="Country name..."
       #typeahead class="form-control-lg">
<hr/>
<!--Results-->
<ul class="list-group">
  <li class="list-group-item" *ngFor="let country of countries$ | async ">
    <img src="{{ country.flag }}" alt="" width="80" style="float: left">
    <p>
      <strong>{{ country.name}}</strong><br>
      {{ country.capital }}
    </p>
  </li>
</ul>
```

This time: using the async pipe

# Code

```
interface ICountry {
  name: string;
  capital: string;
  flag: string;
}

const errorCountry: ICountry = {
  name: 'Error',
  capital: 'Not found',
  flag: ''
};
```

Creating interface and simple error message

```
@ViewChild('typeahead', {static: true}) typeahead;
countries$: Observable<ICountry[]>;

constructor(private http: HttpClient) {
}
```

Inside the component class

```
ngOnInit(): void {
  this.countries$ = fromEvent(this.typeahead.nativeElement, 'keyup')
    .pipe(
      filter((e: any) => e.target.value.length >= 2),
      debounceTime(400),
      map((e: any) => e.target.value),
      distinctUntilChanged(),
      switchMap(keyword => this.getCountries(keyword))
    );
}
```

Main method. See example code for comments

# Fetching the countries

```
getCountries(keyword): Observable<ICountry[]> {
  // 7. Create the actual http-call.
   return this.http
    .get<ICountry[]>(`https://restcountries.eu/rest/v2/name/${keyword}?fields=name;capital;flag`)
    .pipe(
      // 8. catch http-errors and return a 'not found' country
      catchError(err => {
        console.log(err);
        return of([errorCountry]);
      })
    );
}
```

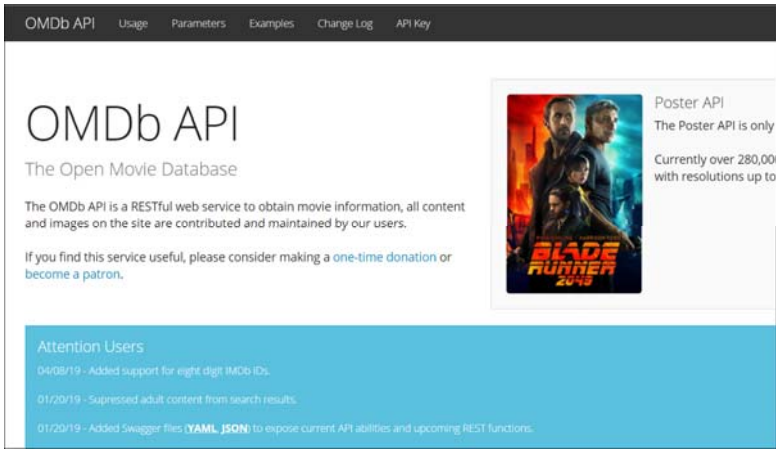
**Documentation:**

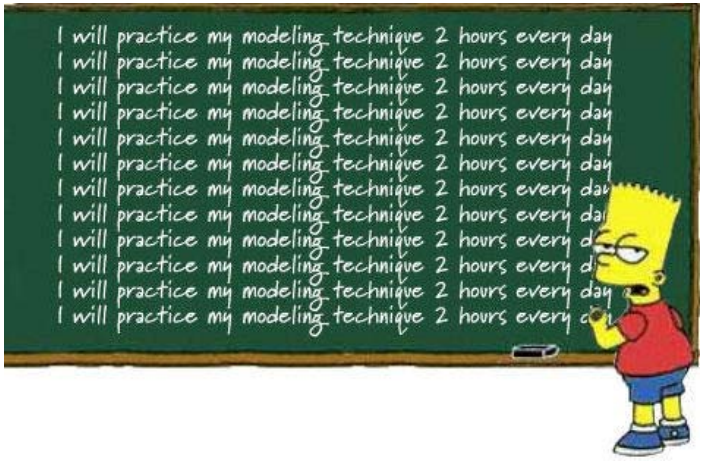
- filter: https://www.learnrxjs.io/learn-rxjs/operators/filtering/filter
- debounceTime: https://www.learnrxjs.io/learn-rxjs/operators/filtering/debouncetime
- distinctUntilChanged: https://www.learnrxjs.io/learn-rxjs/operators/filtering/distinctuntilchanged
- switchMap: https://www.learnrxjs.io/learn-rxjs/operators/transformation/switchmap
- catchError: https://www.learnrxjs.io/learn-rxjs/operators/error_handling/catch

# Workshop



- Search movies in the Open Movie Database.
  - `https://www.omdbapi.com/?apikey=f1f56c8e&s=[keyword]`
  - Create an AutoComplete inputfield for movie titles.
  - For instance, if you type ‘ava…’ it should show movies like `Avatar`, and more
- Example `../basic-stream/typeahead.component.ts`
  - Start from scratch or expand this component

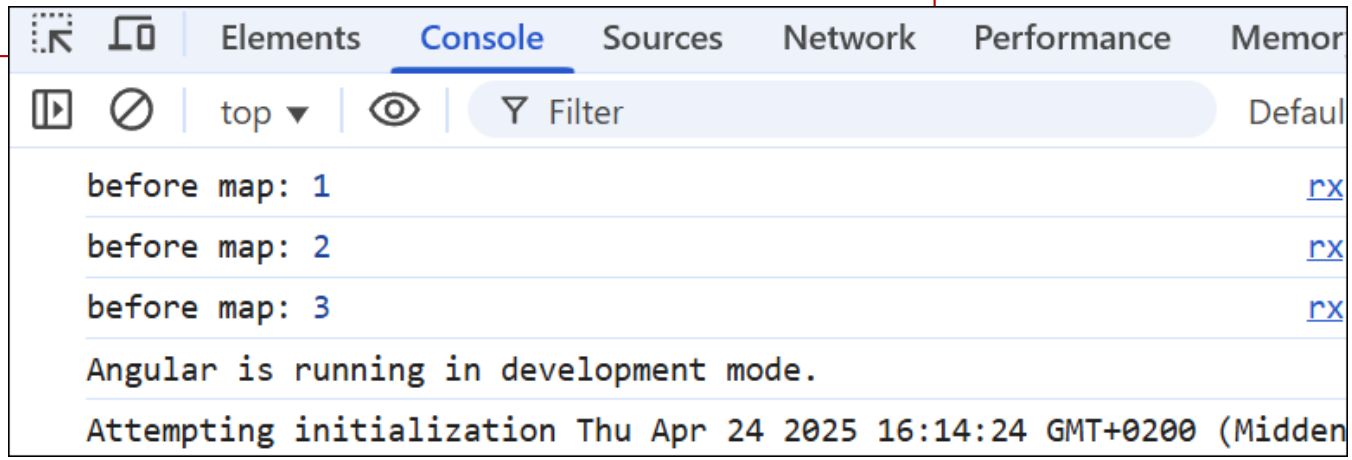


# Summary on some operators : .tap()

- Side effects without modifying the stream. Great for logging, debugging. NOT for production!

```
// 1. The Tap operator is used for debugging purposes.
of(1, 2, 3).pipe(
  tap(x => console.log('before map:', x)),
  map(x => x * 2)
).subscribe();
```




https://rxjs.dev/api/operators/tap

# .map()

- Transforms values in the stream (1:1 mapping).
- *“Map takes a value and transforms it to another value. The transformed value is returned”*

```
// 2. The Map operator is used to transform the data.
of(1, 2, 3).pipe(
  map(x => x * 10)
).subscribe(console.log); // 10, 20, 30
```

```
10                                                    rxjs-ope
20                                                    rxjs-ope
30                                                    rxjs-ope
Angular is running in development mode.
Attempting initialization Thu Apr 24 2025 16:16:59 GMT+0200        cont
(Midden-Europese zomertijd)
```

https://rxjs.dev/api/operators/map

# .debounce() (and debounceTime())

- Waits for silence; emits only after a pause
- Note: since we're using an <input> element here, we also need to wait for it to initialize in the ngAfterViewInit() lifecycle hook!

```
ngAfterViewInit() {
  if(this.inputRef) {
    // 3. The DebounceTime operator is used to delay the emission of an item.
    fromEvent(this.inputRef?.nativeElement, 'input').pipe(
      debounceTime(300),
      map((e: any) => e.target.value)
    ).subscribe(console.log);
  }
}
```

```
Attempting initialization Thu Apr 24 2025 16:27:21 GMT+0200 (Midden- conte
Europese zomertijd)
te
tes
test
Unchecked runtime.lastError: A listener indicated an asynchronous respons
```

https://rxjs.dev/api/operators/debounceTime

# .switchMap()

- Cancels previous inner observable on new emission. Ideal for typeaheads or HTTP calls.
- Dummy Code!:

```
fromEvent(this.inputRef.nativeElement, 'input').pipe(
  map((e: any) => e.target.value),
  switchMap(query => this.http.get(`/search?q=${query}`))
).subscribe(console.log);
```

- Here, the outer observable cancels the previous `http.get()` call when a new value is received, so ‘old’ results from the API are not shown in the console.

https://rxjs.dev/api/operators/switchMap

# .forkJoin()

- Waits for all observables to complete, then emits once with all last values.
- NOTE: Dummy code!

```
forkJoin([
  this.http.get('/user'),
  this.http.get('/settings')
]).subscribe(([user, settings]) => {
  console.log(user, settings);
});
```

https://rxjs.dev/api/index/function/forkJoin

# mergeMap()

- Flattens inner observables concurrently.
- Use when order/cancellation doesn't matter.
- NOTE: Dummy code:

```
from([1, 2, 3]).pipe(
  mergeMap(id => this.http.get(`/item/${id}`))
).subscribe(console.log);
```

- This would show `1 + result http-call1`, `2 + result http-call2` etc. in the console.

https://rxjs.dev/api/operators/mergeMap