Reactive Programming Deep Dive

Agenda

Required JavaScript Language Features

Introduction to Reactive Extensions for Java Script (RxJS)

Base Operators

Debugging RxJS &Error Handling

Combining & Transforming

Action Streams

Async Pipe

Consuming Flex Layout Responsive API

Custom Operators

Language Features

Imperative vs Functional Programming

IMPERATIV

- Not Pure
- Mutates State / Globals
- No Return Value

```
var name = "Sandra";

function greet() {
  name += ", how are you today?";
  console.log(name);
}

greet();
greet();
```

FUNCTIONAL

- Pure
- Does not mutate State / Globals
- Input / Output

```
function greet(name) {
  return `${name}, how are you today`;
}

console.log(greet("Sandra"));
console.log(greet("Heinz"));
```

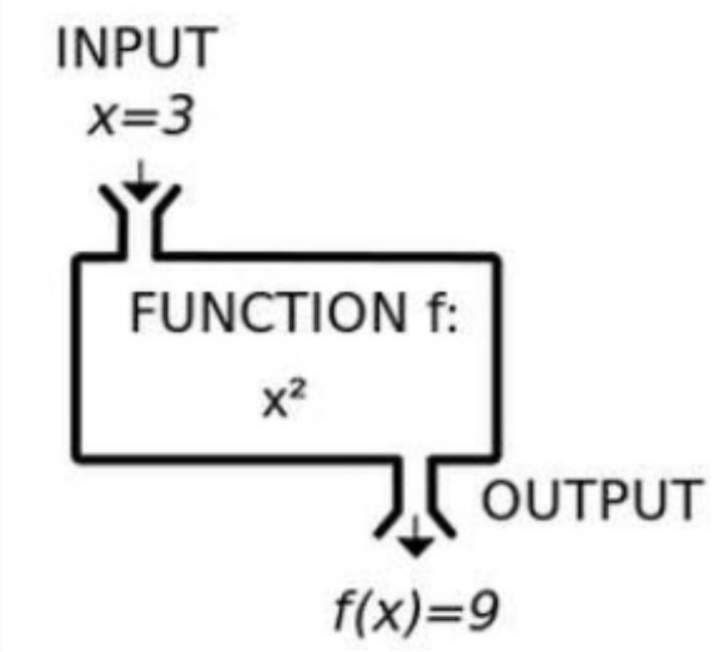
Pure Functions

Follows a simple Paradigma

- Same Input always produces the same Output

Does not look / mutate Global Objects

- No side effects by / to global objects



High Order Functions

Functions that have Functions as Input (Params) and | or Output

Very likely you are using this Pattern already

- Callbacks
- Closures
- Array Util Functions
  - Filter
  - Map
  - Reduce

```
const vehicles = [
  { make: "Honda", model: "CR-V", type: "suv", price: 24045 },
  { make: "Honda", model: "Accord", type: "sedan", price: 22455 },
  { make: "Mazda", model: "Mazda 6", type: "sedan", price: 24195 },
  { make: "Mazda", model: "CX-9", type: "suv", price: 31520 },
];

const averageSUVPrice = vehicles
  .filter(v => v.type === "suv")
  .map(v => v.price)
  .reduce((sum, price, i, array) => sum + price / array.length, 0);

console.log(averageSUVPrice);
```

Immutability

Immutable Objects (State) should not be changed

Instead they should be replaced by new ones

Angular Change Detection has two options

- Always
- OnPush

To make sure that Change Detection always works with OnPush you will have to ensure Immutability by Cloning Objects

Cloning Options

When cloning Objects we have two choices

- Shallow Clone
- Deep Clone

Cloning Options:

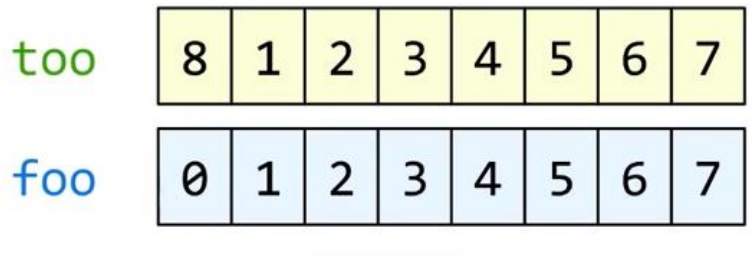
Spread Operator: ...	Shallow Clone
Object.Assign	Shallow Clone
Lodash: .clone & .cloneDeep	Deep Clone

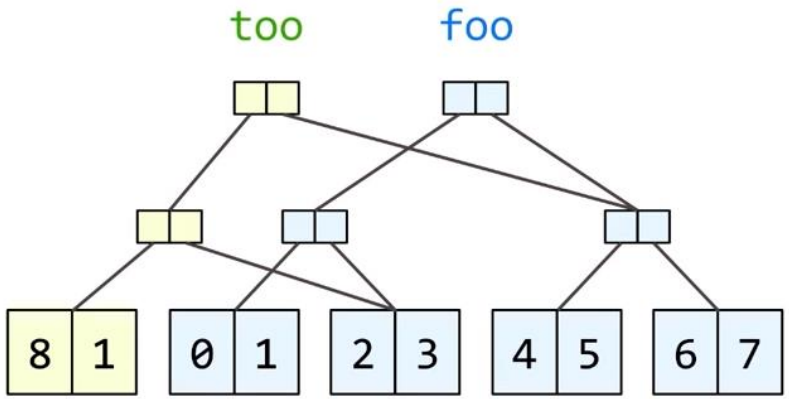
Persistent Data Structures

Copying large (nested) data structures can create great overhead

Solution: Libs that provide Persistent Data Structures like:

- Immutable.js
- Mori





Reactive Extensions for JavaScript

What is RxJS?

API for async programming with Observable Streams

An implementation of the Observable Pattern

Provides a collection of Operators to manipulate response (Observables)

Operator documented @ https://rxjs-dev.firebaseapp.com/api

Can replace:

- Callbacks
- Promises
- async / await

Why RxJS

Angular uses RxJS for

- HttpClient
  - Returns Observables by default
- Routing
  - paramMap instead of Snapshots
  - Router State in ngrx
- Reactive Forms
- Advanced State Management
- Security

Observable

Observables are Streams, that are created by:

- A one-time response (of http operations)
- A Sequence of items (using WebSockets, or Streams)
- [DOM] Events, triggered by Code or User input
- A response of an action in a Reactive Form
- Manually
  - To handle something that is not an Observable in a Reactive manner: Promise, Callback

Observer

Subscribes to an Observable (… observes Observable)

Reacts to whatever item or sequence of items the Observable emits:

- Handle Data: next()
- Handle Error: error()
- Handle Completion: complete()

Might use Operators to manipulate the Observable

- ie. Filter

```
let url = 'https://jsonplaceholder.typicode.com/todos';
from($.ajax(url)).subscribe(
  data => console.log('data from jquery', data),
  err => console.log('err:', err),
  () => console.log('complete')
);
```

Creating Observables

Observables can be created using:

- new Observable(subscriber=>{...})
- of() Creation function -> creates an Observable Sequence
- from() Operator -> creates an Observable from Array, Promise, ...
- fromEvent() – Operator -> creates an Observable from an event

```
this.result$ = from([2, 5, 9, 12, 22]);
```

vs.

```
this.result$ = of([2, 5, 9, 12, 22]);
```

```
usePromiseToObs() {
  let url = 'https://jsonplaceholder.typicode.com/todos';
  from($.ajax(url)).subscribe(data => console.log('data from jquery', data));
}
```

```
getGeolocation$(): Observable<Position> {
  return new Observable(observer => {
    navigator.geolocation.getCurrentPosition(
      (pos: Position) => {
        observer.next(pos);
        observer.complete();
      },
      (err: PositionError) => {
        observer.error(err);
      }
    );
  });
}
```

Subject Types

RxJs knows 4 Subject Types

- Subject
- Behaviour Subject
  - Re-Emits last value for Late Subscribers
- Replay Subject
  - Re-Emits values for Late Subscribers
- Async Subject
  - Only the last value of the Observable execution is sent to its subscribers
  - Only when the execution completes

Subscription

An object that represents a disposable resource, usually the execution of an Observable

Is created using Observable.subscribe()

- Without at least one Subscriber the Observable is not executed!

Has one important method: unsubscribe()

- Unproper Subscription management can lead to memory leaks

Unsubscribing can be done:

- manually – whenever you want to end the subscription
- Using ngOnDestroy

```
ngOnDestroy() {
  this.mouseSubs.unsubscribe();
  console.log("Mouse Subscription unsubscribed");
}
```

Unsubscribing

Infinite Observables have to be unsubscribed to prevent Memory Leaks

Common Patterns available:

- complete() emitted by the Observable – No unsubscribe needed
- unsubscribe() in ngOnDestroy()
- Subsink from Warden Bell

Don‘t unsubscribe:

- httpClient
- Async Pipe
- @HostListener

```
export class MouseDomObservablesComponent implements OnInit, OnDestroy {
  constructor(private ms: MovieService) {}

  mouseSubs: Subscription;

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

Operators

Operators allow us to deal with / manipulate Observables

Operators can be chained using pipe()

Can be grouped into Operators that:

- Create Observables (Create, From, ...)
- Filter Observables (Filter, Take, Distinct, ...)
- Error Handling & Utitlity (Catch, Retry, Subscribe, ...)
- Transform Observables (Map, GroupBy, ...)
- Combine (And / Then / When, ...)

Base Operators

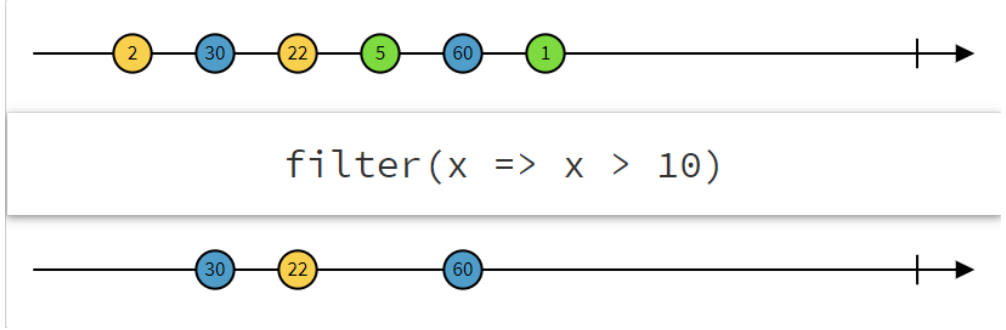
Marble Diagrams

Visualize a graphical representation of observables

Useful to help to understand what Observable Operators do

In the Sample below the filter Operator

- takes in an Input Steam
- evaluates each item in the stream
- if it satisfies the condition they are emitted in the Output Stream



Operators vs JS Methods

Many similarities between Operators and JS Methods

- Map,
- Filter,
- Reduce, ...

Main difference: Operators act on Observables and also return Observables

.pipe() & .tap()

.pipe() is used to combine | pipe several other Operatores which are seperated by commas

.tap() let you do actions (side effects) without modifying the output stream.

- Logging
- Debugging
- ...

```
usePipeMapAndTap() {
  //RxJS 6 pattern
  // tap() is the RxJS replacement for do() to ensure ES compatibility
  this.vs
    .getVouchers()
    .pipe(
      tap(data => console.log("logged by tap(): ", data)),
      map(vs => vs.map(this.setLabel))
    )
    .subscribe(data => this.log("use pipe(), map() & tap()", data));
}
```

.map()

Transform the items emitted by an Observable by applying a function to each item

Often used when working with Http to extract Data from Response

Not to be confused with the array. map() function

```
useMap() {
  this.vs
    .getVouchers()
    .pipe(
      //Obs Operator map()
      map(vs => {
        //ES6 array.map()
        return vs.map(v => ({
          ...v,
          Label: `${v.Text} costs € ${v.Amount}`
        }));
      })
    )
    .subscribe(data => this.log("use map() - RxJS 5 pattern", data));
}
```

find() & filter()

Emit only those items from an Observable that pass a predicate test

.find() returns the first match and then stops

.filter() returns all matching items

```
//JavaScript Array.find
useFindArr() {
  this.vouchers$
    .pipe(map(v => v.find((v: Voucher) => v.ID == 3)))
    .subscribe(data => this.log('getByID - using find()', data));
}

//RxJs find Operator
useFind() {
  this.vouchers$
    .pipe(
      flatMap((vouchers: Voucher[]) => vouchers),
      find(v => v.ID == 3)
    )
    .subscribe(data => this.log('getByID - using find()', data));
}
```

Debugging RxJS

Debugging RxJS

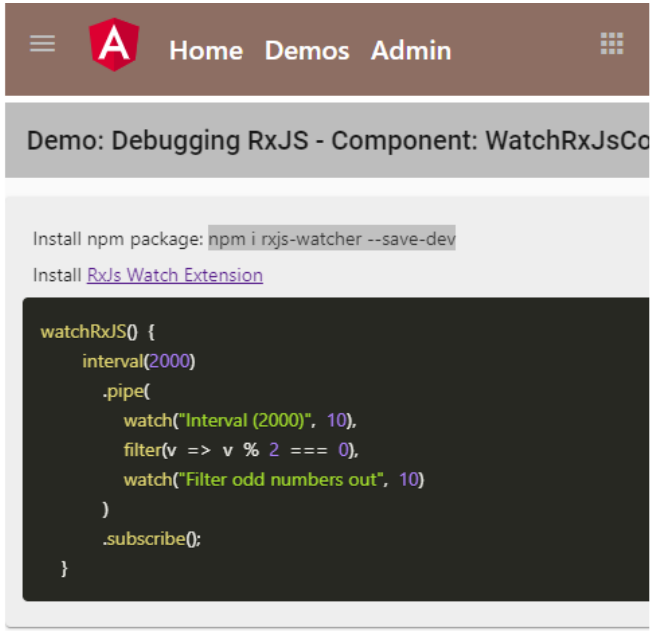
For debugging RxJS one of the following approaches can be used:

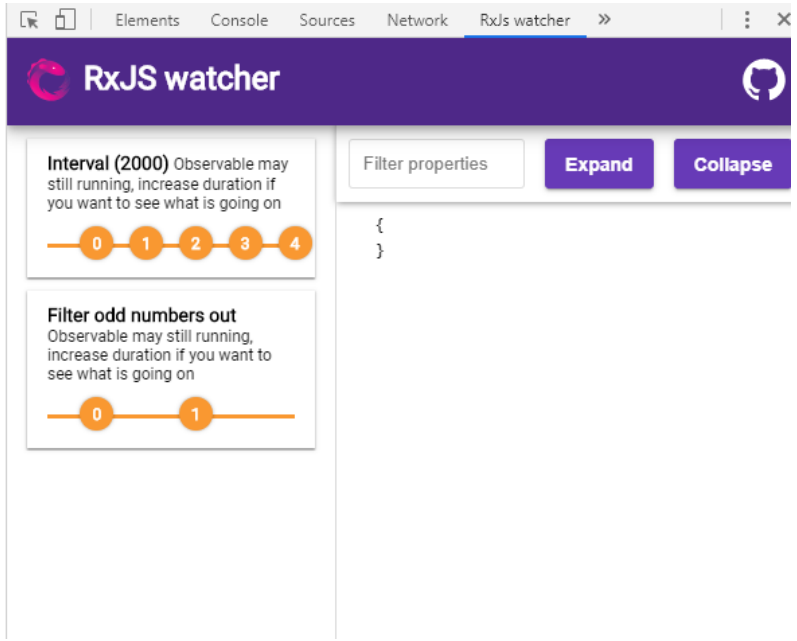
- Use the tap(operator)
- Draw Marble Diagrams
- Use rxjs-watcher

Use Marble Testing as Alternative

- Jasmine-Marbles

  (covered later)





RxJs Watcher

Simple devtools extension to visualize Rxjs observables

- npm i rxjs-watcher --save-dev

Extensions for Chrome, Firefox

Operator takes 3 arguments:

- marbleName: string (label to show above marble)
- duration: number (duration in seconds)
- selector?: function

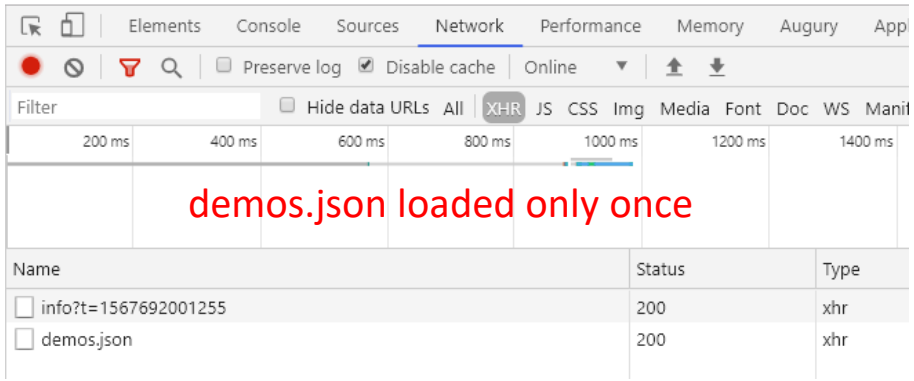
Caching

RxJS allows Caching of Observables

- An earlier result is re-emitted on a later request

Not every request is suitable for caching

Consider Cache Invalidation when using Caching




```
export class DemoService {
  constructor(private httpClient: HttpClient) {}

  getItems(): Observable<DemoItem[]> {
    return this.httpClient.get<DemoItem[]>("/assets/demos.json").pipe(
      tap(data => console.log("loading demos", data)),
      shareReplay(1)
    );
  }
}
```

Error Handling

Streams and Errors

The lifecycle of a Stream ends its liefecycle with any error and Completes

or

When the Streams throws an error -> the stream will not emit any other value

Error Handling can result in

- Throwing the actual Err
- Handling & Replacing the Err by some fallback value or graceful exit
- Retrying

catchError | thowError

Observables have a try-catch-finally like construction to handle errors:

- catchError
- throwError
- finalize
- EMPTY
  - Returns an Observable that emits no values

```
errHandling() {
  this.vs
    .getVouchers()
    .pipe(
      tap(data => console.log("logged by tap(): ", data)),
      map(vs => vs.map(this.setLabel)),
      catchError(err => {
        return throwError("Err happened while processing vouchers");
      }),
      finalize(() => console.log("finalizing ..."))
    )
    .subscribe(data => this.log("errHandling", data));
}
```

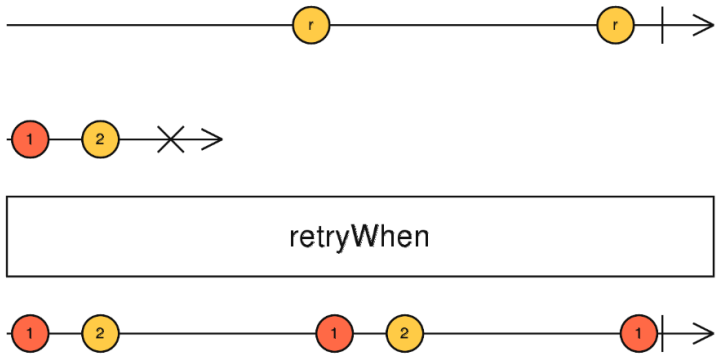
shareReplay & retryWhen

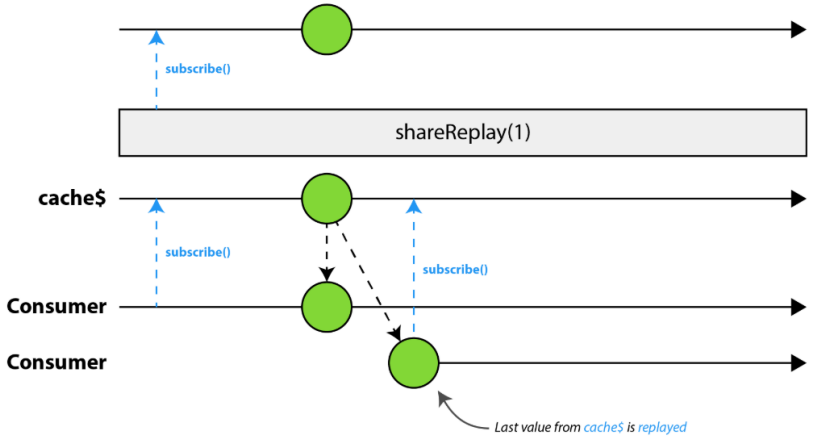
shareReplay

- Share source and replay specified number of emissions on subscription

retryWhen

- Retry an observable sequence on error based on custom criteria



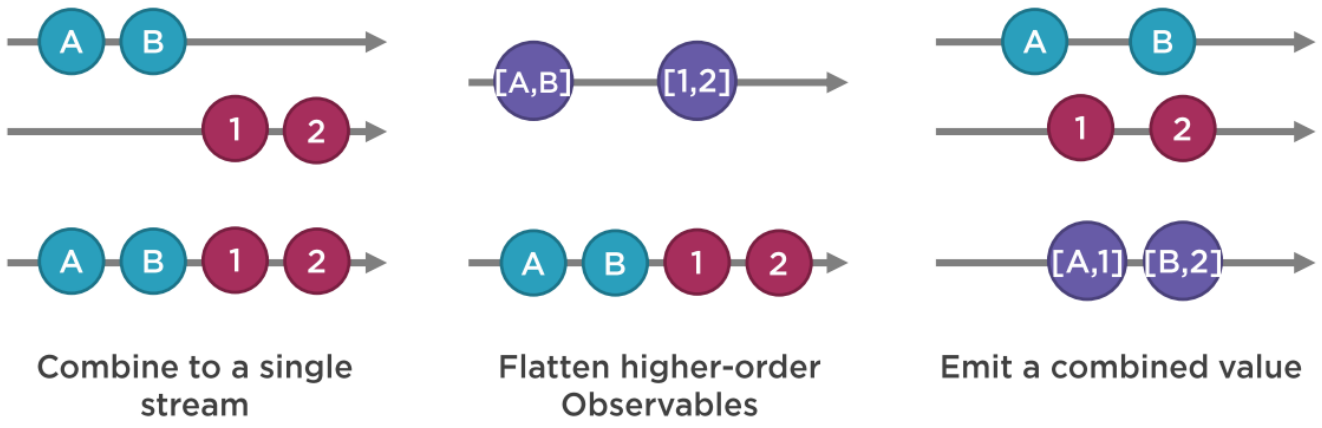


Combination

Combination

Combination is the process when dealing with multible Streams

Types of Combination:

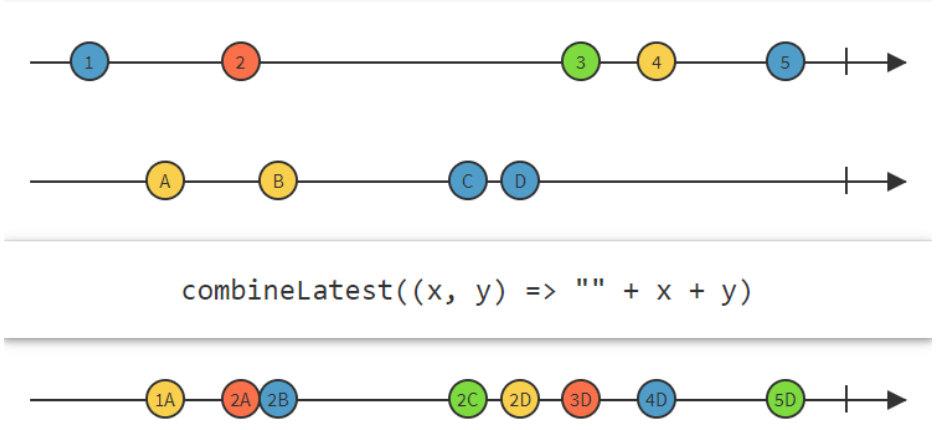


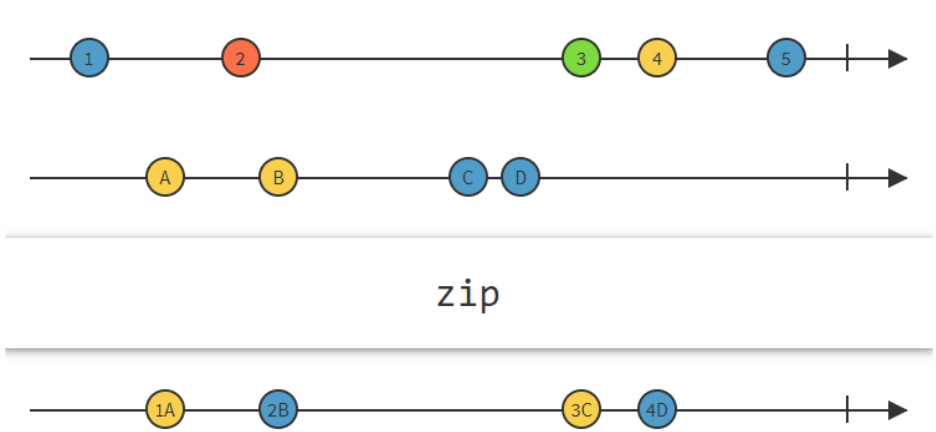
combineLatest vs zip

Creates an Observable whose values are the latest values from each Input Observable

combineLatest emits when ANY of the SOURCE emits a value

ZIP emits when BOTH of the SOURCES emit a value



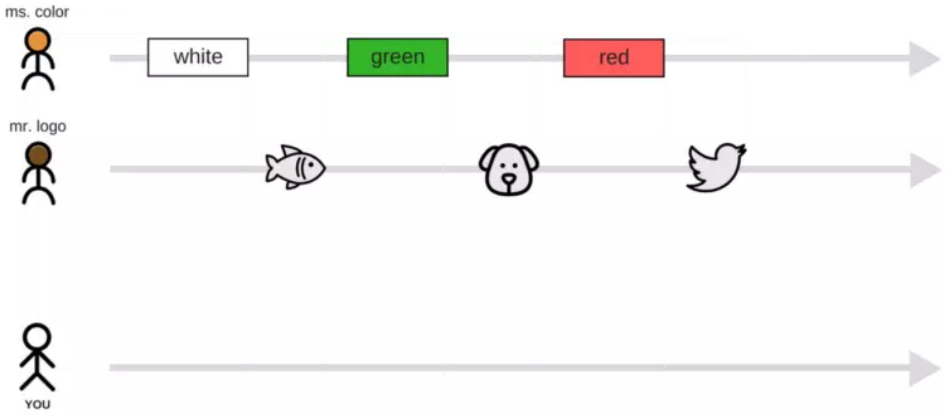


forkJoin

Only commit once when all Inputs complete

Can be used to synchronize (="compile") Inputs

If one Input does not complete - NOTHING happens

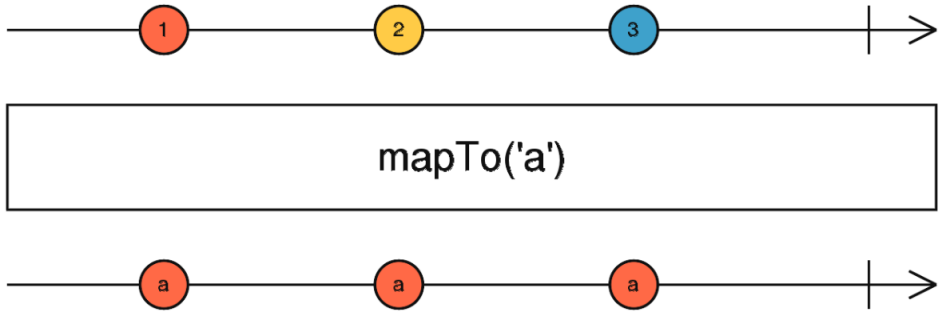


Transformation

mapTo()

Emits the given constant value on the output Observable every time the source Observable emits a value.

Can be used like - if this happens -> do this ...




```
useMapTo() {
  const clicks = fromEvent(document, "click");
  clicks.pipe(mapTo("You clicked the button")).subscribe(console.log);
}
```

Chaining Request

When having multiple Request you can use one of the higher-order Mapping Operators

- mergeMap: Runs request in parallel. Does not guarantee order -> use for put, delete

  mergeMap has an alias called flatMap
- exhaustMap: Ignores all request until current request completes -> use for login request
- switchMap: Cancels the current subscription / request
  - -> use for ie search
- concatMap: Runs requests in order. Waits for finish before performing next

Action Streams

Subscribing to DOM Events

The Observable Pattern can also be used to subscribe to DOM Events like

- Mouse Events
- Button Events
- Change of URL, QueryParams
- ...

- Full list of DOM events:
  - https://www.w3schools.com/jsref/dom_obj_event.asp

Using DOM Events

Subscriptions to Mouse Events are created using:

- fromEvent()
- map() ... reshapes the stream

```
let pad = document.querySelector(".signPad");
let mouse = fromEvent(pad, "mousemove").pipe(
  map((evt: MouseEvent) => {
    return { X: evt.clientX, Y: evt.clientY };
  })
);
```

Another usefull Event is „key up“

- Let‘s us react to user input
- Can be debounced() later on using an operator

```
attachInputDOMEvt() {
  fromEvent(this.inputRef.nativeElement, "keyup").subscribe(val => {
    console.log("Val received from Evt:", val);
  });
}
```

Using Reactive Forms

Reactive FormControls offer some Observables you can subscribe to:

- valueChanges()
- statusChanges() -> valid

Reactive Forms do NOT need a Form Tag!

```
fcFoodName = new FormControl();
foodForm: FormGroup = this.fb.group({ fcFoodName: this.fcFoodName });
status: any;

ngOnInit() {
    this.fcFoodName.statusChanges.subscribe((s) => {
        this.status = s;
    });
}
```

Data- vs Action-Streams

A Data Stream typically is a result of a Data Request

- -> Contains Data

An Action Stream typically is a result of an Interaction in the User Interface

- -> ie. a value of a DropDown is selected

Most of the time you want to respond of a Change in either one

Operators you can use:

- CombineLatest

Async Pipe

Async Pipe

Subscribes to the Observable when component is initalized

Automatically unsubscribes when component is destroyed

Support Aliasing (... as XY ...)

Can be used:

- To switch pattern from subscribe() to Declarative Pattern
- Optimize Change detection

```
//Declarative Approach using async pipe
tasks$: Observable<Task[]>;

getDataStream() {
  this.tasks$ = this.ts.getTasks();
}
```

```
<mat-card-content>
  <div *ngFor="let t of tasks$ | async">{{ t.id }} - {{ t.title }}</div>
</mat-card-content>
```

Declarative Pattern

Is the Combination of using:

- RxJS, and
- Async Pipe

instead of subscribing to Observables

Benefits:

Less Code

Easy ChangeDetection using OnPush

Much cooler ☺

```
export class AsyncPipeComponent implements OnInit {
  constructor(private ts: TaskService) {}

  tasks$: Observable<Task[]> = this.ts.getTasks();
  completed$: Observable<Task> = this.tasks$.pipe(
    flatMap((tasks: Task[]) => tasks),
    filter(t => t.completed)
  );
```

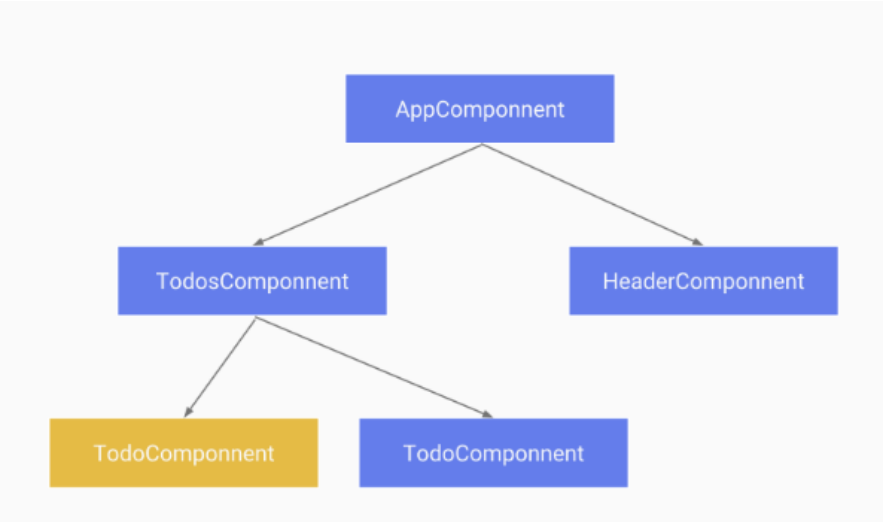
Change Detection

Change Detection is the process where Angular checks it Component Tree to evaluate the Componentes that are effected by Data Changes

Possible Values:

- ChangeDetectionStrategy.Default
- ChangeDetectionStrategy.OnPush

```
@Component({
  selector: "app-persons-list",
  templateUrl: "./persons-list.component.html",
  styleUrls: ["./persons-list.component.scss"],
  changeDetection: ChangeDetectionStrategy.OnPush
})
```



Custom Operators

Custom Operators

An operator is just a pure function that takes the source Observable as it's input and returns an Observable as its output:

- const myOperator = () => (sourceObservable) => new Observable()

Implementation

Create a function that takes an Observable as Input

Return a new Observable

Handle Errors and Complete

```
import { Observable } from "rxjs";
export function filterEven(source: Observable<any>): Observable<any> {
  return new Observable(observer => {
    source.subscribe(
      (val: number) => {
        if (val % 2 === 0) {
          observer.next(val);
        }
      },
      err => observer.error(err),
      () => observer.complete()
    );
  });
}
```

```
simpleFilter() {
  let numbers$ = from([1, 4, 6, 7, 9, 11]).pipe(n => filterEven(n));
  numbers$.subscribe(n => console.log(n));
}
```

Combined Operators

Custom Operators can also be:

- Combinations of other Operators
- Encapsulation of
  - Complex Logic
  - Error Handling
  - Logging

```
export function logError() {
  return catchError(err => {
    console.log("Error", err);
    return EMPTY;
  });
}

export function getFromApi(http: HttpClient, url: string) {
  return http.get(url).pipe(logError());
}
```

Flex Layout Responsive API

Responsive Flexbox API

Provides and API based on Observables to detect screen changes

Documented @ https://github.com/angular/flex-layout/wiki/API-Documentation

```
export class FlexLayoutApiComponent implements OnInit {
  constructor(private obsMedia: MediaObserver) {
    this.subscribeIsPhone();
  }
```

```
subscribeIsPhone() {
  this.watcher = this.obsMedia.media$.subscribe((change: MediaChange) => {
    switch (change.mqAlias) {
      case "xs":
        this.isPhone = true;
        this.isTablet = false;
        break;
      case "sm":
        this.isPhone = false;
        this.isTablet = true;
        break;
      default:
        this.isPhone = false;
        this.isTablet = false;
        break;
    }
```

Orientation Change & Resize

Orientation Change & Resize can be detected using:

- window.addEventListener("orientationchange“, callback)
- window.addEventListener("resize“, callback)

Or using Media Queries:

- @media screen and (orientation:portrait)
- @media screen and (orientation:landscape)

This is an experimental technology
Check the Browser compatibility table carefully before using this in production.

The `orientation` read-only property of the `Screen` interface returns the current orientation of the screen.

Syntax

```
var orientation = window.screen.orientation;
```