# Reactive programming With observable streams (RxJS)

by/for h0mmies

# Reactive programming

Reactive programming is programming with asynchronous data streams.

# Asynchronous data streams

- Asynchronous: In JavaScript means we can call a function and register a callback to be notified when results are available, so we can continue with execution and avoid the Web Page from being unresponsive. This is used for AJAX calls, DOM events, Promises, WebWorkers and WebSockets.
- Data: Raw information in the form of JavaScript data types as: Number, String, Objects (Arrays, Sets, Maps).
- Streams: Sequence of ongoing events ordered in time.

# Why to use RxJS

- Unifies both the world of Promises, callbacks as well as evented data such as AJAX calls, DOM events, Web Sockets etc.
- Easy to handle / combine actions ( streams of data) caused by triggering multiple events, in a composable / functional way.

# Observables, Observers, Subscription, Operators...????

Observable streams = Observer pattern + support sequences of data and/or events and adds operators that allow you to compose sequences together.

# OBSERVABLE

An Observable is just a function...with a few special characteristics => it takes in an "**Observer**".

The Observable sends notifications.

# OBSERVER

An Observer can be an Object with ***next***, ***error*** and ***complete*** methods on it and it can be passed to ***observable.subscribe(observer)*** method.

Observable will call the Observer's ***next(value)*** method to provide data.

The Observer receives notifications.

# EXAMPLE

```
const node =
document.querySelector('input[type=text]');
const input$ = Rx.Observable.fromEvent(node,
'input');

input$.subscribe({
  next: event => console.log(`You just typed
${event.target.value}!`),
  error: err => console.log(`Oops...
${err}`),
  complete: () => console.log(`Complete!`),
});
```

1. It takes an *<input type="text">* element and passes it into *Rx.Observable.fromEvent()*, which returns us an Observable of our input's Event object.
2. When the input's event listener fires, the Observable passes the value to the observer.

Note: *Rx.Observable.fromEvent(node, 'input')* also called "producer" and contains source of values, perhaps from a click or input event in the DOM or from HTTP calls.

# SUBSCRIBE

- The “listening” to the stream is called subscribing.
- Calling subscribe method of Observable we can listen to stream.
- To invoke the Observable, we need to subscribe to it.

# OPERATOR

A pure function which takes one Observable as input and generates another Observable as output.

Main categories: creation, transformation, filtering, combination, multicasting, error handling.

Think Operators as a Lodash functions.

# Example (custom operator)

```
function multiplyByTen(input) {
  return Rx.Observable.create((observer) => {
    input.subscribe({
      next: (v) => observer.next(10 * v),
      error: (err) => observer.error(err),
      complete: () => observer.complete()
    });
  });
}

var input = Rx.Observable.from([1, 2, 3, 4]);
var output = multiplyByTen(input);
output.subscribe(x => console.log(x));
// Result:
// 10 20 30 40
```

# Most commonly used operators

- ***forkJoin***: is the Promise.all() of Rx. After all Observables complete, it gives all the values at once (use it for running Observables in parallel).
- ***switchMap***: when the source-Observable emits, cancel any previous subscriptions of the inner-Observable.
- ***mergeMap (or flatMap)***: when the inner Observable emits, let me know by merging the value to the outer Observable.
- ***debounceTime***: Emits a value from the source Observable only after a particular time span has passed without another source emission.
- ***tap***: Tap operator is more of a utility operator which can be used to perform transparent actions such as logging.

# SUBJECT

- A Subject can be subscribed to, just like an Observable.
- A Subject can subscribe to other Observables acting like Observer.
- Subjects, unlike Observables, **share** their work with all subscribers (multicasting).

Notes:

1. By calling *subject.next()*, which cannot be done in a normal observable, Gives The benefit to emit new values to subscribers, without having to rely on some source data.
2. Subjects will make sure each subscription gets the exact same value as the Observable execution is shared among the subscribers.

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

const subA = subject.subscribe( val =>
print(`Sub A: ${val}`) );
const subB = subject.subscribe( val =>
print(`Sub B: ${val}`) );

subject.next('Hello');

setTimeout(() => {
    subject.next('World');
}, 1000)

// Sub A: Hello
// Sub B: Hello
// Sub A: World
// Sub B: World
```

# Observables vs promises

- An observables can emit **any** number of values.
- Observables are **lazy**: the subscriber function is only called when a client subscribes to the observable.
- Observables are **cancellable**: After subscribing to an observable with subscribe, you can cancel this subscription at any time by calling the **unsubscribe** method of the **Subscription** object returned by subscribe. In this case, the handler function that you passed to subscribe won't be called anymore.

- A promise can only emit a **single** value. After that it is in the fulfilled state and can only be used to query this value, but not to calculate and emit new values anymore.
- Promises are **eager**: the executor function is called as soon as the promise is created.
- Promises are **not cancellable**: Once you "subscribed" to a promise with then, then the handler function that you pass to then will be called, no matter what. You can't tell a promise to cancel calling the result handler function once the promise execution has been started.

# Sources

- RxJS: Observables, Observers and Operators Introduction
- RxJS: In 5 minutes!
- RxJS Quick Start With 20 Practical Examples
- RxJS Quick Start with Practical Examples
- Understanding RxJS map, mergeMap, switchMap and concatMap
- JavaScript Promises vs. RxJS Observables