

RXJS

Ori Calvo, 2017

oric@trainologic.com

<https://trainologic.com>

RxJS

2

- “RxJS is a library for reactive programming using Observables, to make it easier to compose asynchronous or callback-based code”

<http://reactivex.io/rxjs>

- There is a formal proposal at

<https://github.com/tc39/proposal-observable>

- ▣ Currently at Stage 1

RxJS – Core Ingredients

3

- Observable
- Observer
- Subscription

Observable

4

- Observable is basically a wrapper around a data source/stream

```
const observable = Rx.Observable.create(  
  obs => {  
    obs.next('a value');  
    obs.next('a second value');  
    obs.complete();  
  });
```



Observable which wraps a function

Observer

5

- The observer's role is to execute code when the observable receives a new value

```
let observer = {  
  next: value => {console.log(value)},  
  error: error => {console.log(error)},  
  complete: () => {console.log('completed')}  
};
```

```
Rx.Observable.create(  
  obs => {...}  
) .subscribe(observer);
```

Creating the
observer


Executing the
observer's functions

subscribe

6

- ❑ subscribe calls are not shared among multiple observers
- ❑ Each subscribe invocation causes a new “run”

Invoking subscribe again causes the observable to “run” again



```
const obs = Rx.Observable.create(observer => {  
  observer.next(1);  
  observer.next(2);  
  observer.complete();  
});  
  
obs.subscribe(val => {console.log(val)});  
obs.subscribe(val => {console.log(val)});
```

Subscription

7

- ❑ An observable might never complete
- ❑ In that case the client (observer) must unsubscribe manually
- ❑ Else, its lifetime is bound to the lifetime of the observable → Memory leak

```
let observer = {...};

let subscription = Rx.Observable.create(...).subscribe(observer);

setTimeout(() => {
    subscription.unsubscribe();
}, 2000);
```

Observables are Synchronous

8

- Unlike Promises, observables are synchronous by default

```
const obs = Rx.Observable.create(obs => {  
  obs.next(1);  
  obs.next(2);  
  obs.complete();  
});  
  
console.log("Before");  
  
obs.subscribe(val => {  
  console.log(val);  
});  
  
console.log("After");
```

The output is:
Before
1
2
After

Observables may be Asynchronous

9

```
const obs = Rx.Observable.create(obs => {  
  obs.next(1);  
  
  setTimeout(function () {  
    obs.next(2);  
    obs.complete();  
  }, 0);  
});  
  
console.log("Before");  
obs.subscribe(val => {  
  console.log(val);  
});  
console.log("After");
```

The output is now:

Before

1

After


2

Cold Observable

10

- ❑ This is the default
- ❑ Each subscribe get its own stream of data

Invoking subscribe
again causes the
observable to “run”
again



```
const obs = Rx.Observable.create(observer => {  
  observer.next(1);  
  observer.next(2);  
  observer.complete();  
});  
  
obs.subscribe(val => {console.log(val)});  
obs.subscribe(val => {console.log(val)});
```

Hot Observable

11

- Each subscriber is registered to a live stream of data
- New subscriber sees only new data

```
const producer = new Producer();
producer.run();

const obs = Observable.create(observer => {
  producer.listeners.push(function(val) {
    observer.next(val);
  });
});
```

Subject

12

- ❑ `Observable.subscribe` creates a new stream of data each time is executed
- ❑ `Subject` allows you to share the same stream with different clients
- ❑ A client can subscribe in the middle of a stream
- ❑ Resembles an event emitter

Subject

13

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

subject.subscribe({
  next: value=>{console.log(value)},
  error: error=>{console.log(error)}
});

subject.next('a new data!');
subject.error(error);
```

The example reflect how
subject behaves when
creating and executing

- Executing **subject.next** after **subject.complete** yields nothing since the observable is considered completed

Subject & unsubscribe

14

- Subject is often used as an event emitter
- A client registers
- The subject is never completed → The client is bound to the lifetime of the subject
- It is important to remember to call the **unsubscribe** method once a client is “inactive” or “dead”
 - ▣ Else, you risk a memory leak

BehaviorSubject

15

- Allows you to specify an initial value
- When subscribing, the behavioral subject notifies the observer with last passed value immediately
- Grants the ability to retrieve the last value passed down the subject
- Useful for interaction between a service and a component where any new component must re-render upon subscribing

BehaviorSubject

16

```
let subject = new Rx.BehaviorSubject("a");

subject.subscribe(value => {
  console.log("Subscription 1", value)
});

subject.next("b");

subject.subscribe((value) => {
  console.log("Subscription 2", value)
});

subject.next("c");
subject.next("d");

console.log(subject.getValue());
```

It is mandatory to set
an initial value .

Retrieving the last
value

AsyncSubject

17

- ❑ AsyncSubject emits the data only when the **subject.complete** is invoked
- ❑ Is often used when heavy computations are streaming through the observable
- ❑ It remembers only the final result of the heavy computation

AsyncSubject

18

```
let subject = new Rx.AsyncSubject();

let obs = {
  next: value => {console.log(value)},
  error: error => {console.log(error)},
  complete: () => {console.log('complete')}
};

subject.subscribe(obs);

asyncSubject.next('1');
asyncSubject.next('2');
asyncSubject.complete();
```

next method is
invoked only once
when the subject
completes

Operators

19

- ❑ Operators allows you to create new observable from an existing one
- ❑ Resembles the concept of promise chaining/transformation
- ❑ For example, given an observable that produces the values [1,2,3] you can use the **map** method to create a new observable that produces the values [2,4,6]

Operators

20

- What will be printed during runtime?

```
let observable = Rx.Observable.interval(1000);  
  
observable  
  .throttleTime(2000)  
  .map(x => x*2)  
  .subscribe(value => {console.log(value);});
```

throttleTime emits latest value
when specified duration has
passed

Observable.from

21

- Method that turns an iterable object into an observable

```
const obj = {  
  [Symbol.iterator]: function() {  
    let num = 0;  
    return {  
      next: function() {  
        return {value: num, done: num++===100};  
      }  
    };  
  }  
};
```

Observable.from does not cache values. Once subscribing it iterates through obj and notifies observer of each value

```
const arraySource = Rx.Observable.from([1,2,3,4,5]);  
const subscribe = arraySource.subscribe(val => console.log(val));
```

Observable.of

22

- Turn an amount of values into a sequenced observable

```
const source = Rx.Observable.of(1,2,3,4,5);  
source.subscribe(val => {console.log(val);});
```

filter

23

- A method which is chained to the observable and will filter all data according to the manipulated code

```
let observable = Rx.Observable.interval(1000);  
  
observable.filter(value => {  
    return value % 2 == 0;  
})  
.subscribe(value => {console.log(value)});
```

The example reflects a simple use case for the filter method chained to the observable before the subscription

do

24

□ Perform actions without transformation

```
Rx.Observable.interval(500)
  .do(x => {
    console.log(x);

    return x * 2;
  })
  .subscribe(x => {
    console.log(x);
  });
```

Return value is ignored. Use map for transforming values

debounceTime

25

- “blocks” the stream until inactivity is detected
- Once inactivity detected, it emits latest values

```
Rx.Observable.interval(500)
  .do(x => {
    console.log(x);
  })
  .debounceTime(501)
  .subscribe(x => {
    console.log("never happens");
  });
```

distinctUntilChanged

26

- “blocks” the stream until a new value is detected

```
let input = document.querySelector('input');

Rx.Observable.fromEvent(input, 'input')
  .map(event => event.target.value)
  .debounceTime(2000)
  .distinctUntilChanged()
  .subscribe({
    next: value => {console.log(value);}
  });
```

reduce

27

- Reduces a stream of values into a single value
- Waits for the completion of the source and only then emits the accumulated single value

```
let observable = Rx.Observable.of(1,2,3,4);

observable
  .reduce((total, currentValue) => {
    return total + currentValue;
  }, 0)
  .subscribe(value => {
    console.log(value);
  });
```

Observer will be
invoked only
once and will
emit the value 10

scan

28

- Unlink reduce scan doesn't wait for source completeness
- It emits the accumulated value immediately

```
Rx.Observable.interval(500)
  .reduce((total, currentValue) => {
    return total + currentValue;
  }, 0)
  .subscribe(value => {
    console.log(value);
  });
```

pluck

29

- Returns a “deep” property
- Returns **undefined** if path is broken

```
Rx.Observable.from([  
  {name: "Ori", address: {city: "Rehovot"}},  
  {name: "Roni"},  
])  
.pluck("address", "city")  
.subscribe(x => {  
  console.log(x);  
});
```

“Rehovot”
undefined

concat

30

- Concatenates all observables, but only after the former has been completed

```
const obs1 = Rx.Observable.of(1, 2, 3);  
const obs2 = Rx.Observable.of(4, 5, 6);  
const obs3 = obs1.concat(obs2);  
  
obs3.subscribe(val => console.log(val));
```



Prints: 1,2,3,4,5,6

merge

31

- Creates new observables that emits all sources values
- First come first served ...

```
const obs1 = Rx.Observable.interval(1000).mapTo("1000");  
const obs2 = Rx.Observable.interval(2000).mapTo("2000");  
  
obs1.merge(obs2).subscribe(x => {  
  console.log(x);  
});
```

mapTo method simply set
the emitted value to a
fixed value

partition

32

- Given a criteria it returns two observables
- First observable matches the criteria, the other one doesn't match

```
const [even, odd] =  
Rx.Observable.from([1, 2, 3, 4, 5, 6]).partition(x => x % 2 == 0);  
  
even.subscribe(x => { console.log(x); });  
odd.subscribe(x => { console.log(x); });
```

Example will
result in:
2,4,6,1,3,5

groupBy

33

- Transforms single stream into a stream of groups
- Each group is a stream and has a unique key

1: 1
2: 2
0: 3
1: 4
2: 5
0: 6

```
Rx.Observable.from([1,2,3,4,5,6])  
  .groupBy(num => num%3)  
  .subscribe(group => {  
    group.subscribe(num => {  
      console.log(group.key + ": " + num);  
    });  
  });
```

A group has key and can
be used as observable

zip

34

- Transforms two streams (or more) into one
- The 1st emitted value is an array of the 1st values from the source streams
- The 2nd emitted value is an array of the 2nd values from the source streams
- And so on ...

The new stream
emits value every 3
seconds

The emitted value is
in the form of
[obs1[n], obs2[n]]

```
const obs1 = Rx.Observable.interval(500).map(i => "X" + i);
const obs2 = Rx.Observable.interval(3000).map(i => "Y" + i);

Rx.Observable.zip(obs1, obs2)
  .subscribe(x => {
    console.log(x);
  });
```

flatMap (A.K.A mergeMap)

35

- Transforms a stream of arrays into a stream of single values (A.K.A flattening)

Instead of array you
can use an
observable and the
result will be the
same

```
const obs = Rx.Observable.from([  
  Rx.Observable.from([1,10,100]),  
  Rx.Observable.from([2,20,200]),  
  Rx.Observable.from([3,30,300]),  
]);  
  
obs.flatMap(x=>x).subscribe(x => {  
  console.log(x);  
});
```

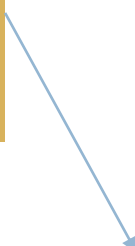
flatMap does not
wait for
completeness of
the sources

switchMap

36

- Switch to a new observable and cancel the previous
- Maintain only one inner subscription

Every 1010 ms
switches to the inner
observable



```
const obs = Observable.interval(1010);  
  
obs.switchMap(()=>Observable.interval(200)).subscribe(x => {  
  console.log(x);  
});
```

Error Handling

37

- An observable can report an error using the **error** method
- Once doing so the stream is considered faulty and future values are not emitted

Value is not
emitted to
observer

```
const obs = Rx.Observable.create(observer => {  
  observer.next(1);  
  observer.next(2);  
  observer.error(new Error("XXX"));  
  observer.next(3);  
  console.log("XXX");  
});
```


error method
does not stop
execution

Throwing Error

38

- You should be careful when throwing error from inside an observable
- Assuming synchronous observable, the **observer.error** will be invoked and the subscribe method will throw

subscribe itself
might throws



```
obs.subscribe({  
  next: x => {  
    console.log(x);  
  },  
  error: err => {  
    console.error(err);  
  }  
});
```

Observable Chain

39

- An error inside source stream makes the “chained” stream to become faulty too

obs1 emits 0,1
and then fails.
obs2 emits 0,2
and then fails
too

```
const obs1 = Rx.Observable.interval(500)
  .take(3)
  .do(x => {
    if (x == 2) {
      throw new Error("XXX");
    }
  });

const obs2 = obs1.map(x => x * 2);

obs2.subscribe(x => {
  console.log(x);
});
```

catch

40

- ❑ Transforms a faulty stream into valid one
- ❑ Must return a new stream instead of the faulty one

Emitted
values are
1, 2, X

```
const obs1 = Rx.Observable.interval(500).take(3)
  .do(x => {
    if (x == 2) {
      throw new Error("XXX");
    }
  });

obs1.catch(err => {
  return ["X"];
}).subscribe(x => {
  console.log(x);
});
```


Ngrx

41

- ❑ RxJS powered state management
- ❑ Inspired by Redux
- ❑ State is a single immutable data structure
- ❑ State can be accessed in an observable fashion
- ❑ Action dispatched to the store are described by an observable too

Getting Started

42

- npm install @ngrx/store
- Define a reducer (just like Redux)
- Imports Ngrx module

```
imports: [  
  BrowserModule,  
  StoreModule.forRoot({ counter: counterReducer })  
]
```

Using the Store

43

- The store is injectable
- Can select from the store

```
export class AppComponent {  
  counter: Observable<number>;  
  
  constructor(private store: Store<AppState>) {  
    this.counter = store.select('counter');  
  }  
  
  inc(){this.store.dispatch({ type: INCREMENT });}  
  
  dec(){this.store.dispatch({ type: DECREMENT });}  
  
  reset(){this.store.dispatch({ type: RESET });}  
}
```

@ngrx/effects

44

- ❑ `npm install @ngrx/effects`
- ❑ Provides API to model actions being dispatched as a single reactive stream
- ❑ Effect listens for an action
- ❑ Initiate an activity (usually HTTP request)
- ❑ Dispatches new actions which
 - ▣ Reduce application state
 - ▣ Initiates a new activity

Actions

45

- An observable that represents all actions being dispatched to the store

```
@Injectable()
export class AuthEffects {
  @Effect()
  login$: Observable<any> = this.actions$.ofType('LOGIN')
    .mergeMap((action: any) =>
      this.http.post('/auth', action.payload)
        .map(data => ({type: 'LOGIN_SUCCESS', payload: data}))
        .catch(() => of({type: 'LOGIN_FAILED'})))
    );

  constructor(private http: HttpClient, private actions$: Actions) {
  }
}
```

Component side

46

```
export class AppComponent {  
  userName: Observable<string>;  
  
  constructor(private store: Store<AppState>) {  
    this.userName = store.pluck('user', 'userName');  
  }  
}
```

```
dec(){  
  this.store.dispatch({ type: DECREMENT });  
}
```

Summary

47

- Reactive programming is fun to write
- Usually not so fun to read
- Using few operators you can implement a complex reactive flow that would take other wise many lines of imperative code