

From zero to hero with the Reactive Extensions for JavaScript

Maurice de Beijer

@mauricedb



Who am I?

- Maurice de Beijer
- The Problem Solver
- Microsoft Azure MVP
- Freelance developer/instructor
- Twitter: @mauricedb and @React_Tutorial
- Web: <http://www.TheProblemSolver.nl>
- E-mail: maurice.de.beijer@gmail.com



Gift This Course



Wishlist

Master RxJS 6 Without Breaking A Sweat

Learn how to solve common programming problems using RxJS

HIGHEST RATED ★★★★★ 4.7 (11 ratings) 730 students enrolled

Created by Maurice de Beijer Last updated 12/2018 English English [Auto-generated]



Preview this course

What you'll learn

- ✓ After this course you will be able to see where using RxJS makes sense.
- ✓ You will be able to solve common programming problems using RxJS.

Requirements

- Basic understanding of JavaScript is required.
- A PC with Node, NPM, a modern browser like Chrome or FireFox and a code editor you like is required.
- Any previous knowledge of RxJS is not required.

€10.99 ~~€99.99~~ 89% off

5 hours left at this price!

Add to cart

Buy now

30-Day Money-Back Guarantee

This course includes

- 2.5 hours on-demand video
- Full lifetime access

Topics

- What is RxJS?
- Why use it?
- How to create observables.
- Using operators with observables.



RxJS

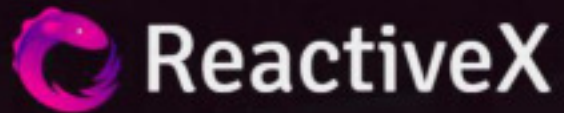
Reactive Extensions Library for JavaScript

[GET STARTED](#)

[API DOCS](#)

REACTIVE EXTENSIONS LIBRARY FOR JAVASCRIPT

RxJS is a library for reactive programming using Observables, to make it easier to compose asynchronous or callback-based code. This project is a rewrite of Reactive-Extensions/RxJS with better performance, better modularity,



An API for asynchronous programming
with observable streams

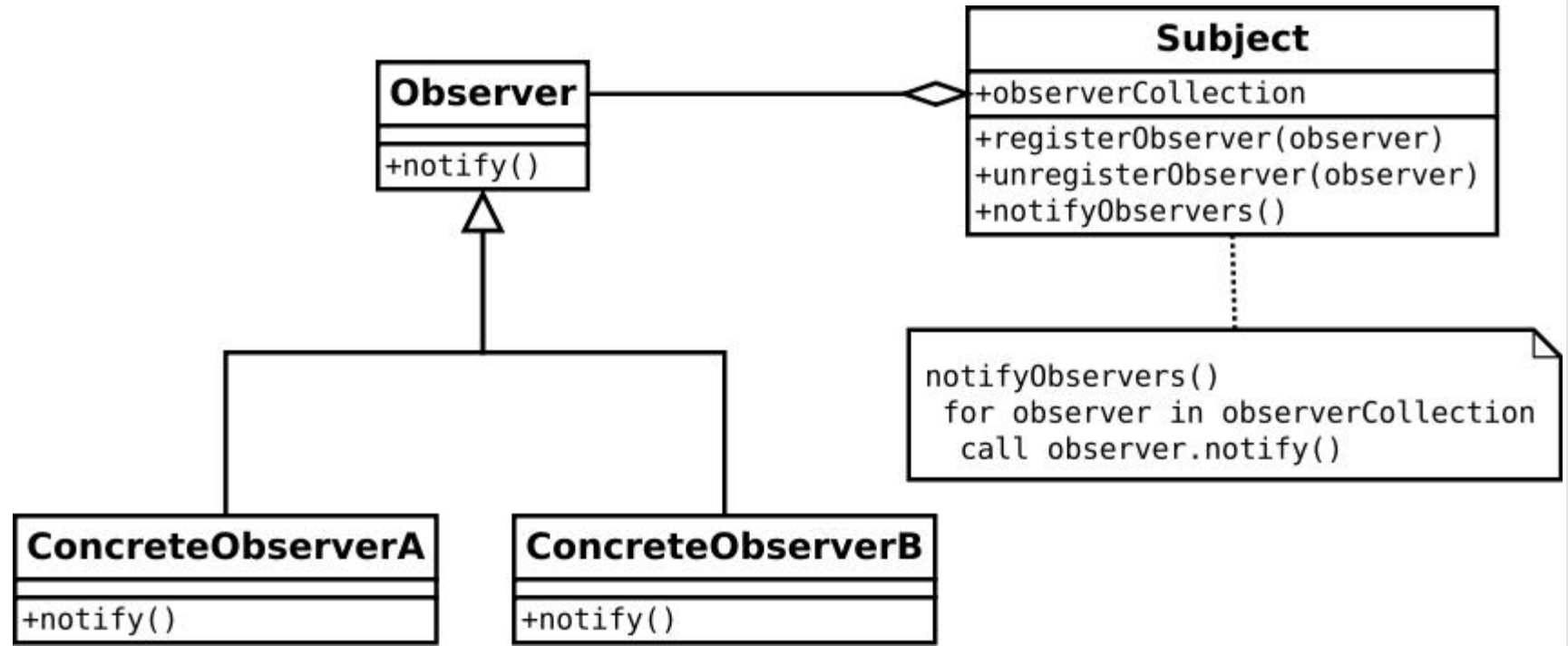
Choose your platform



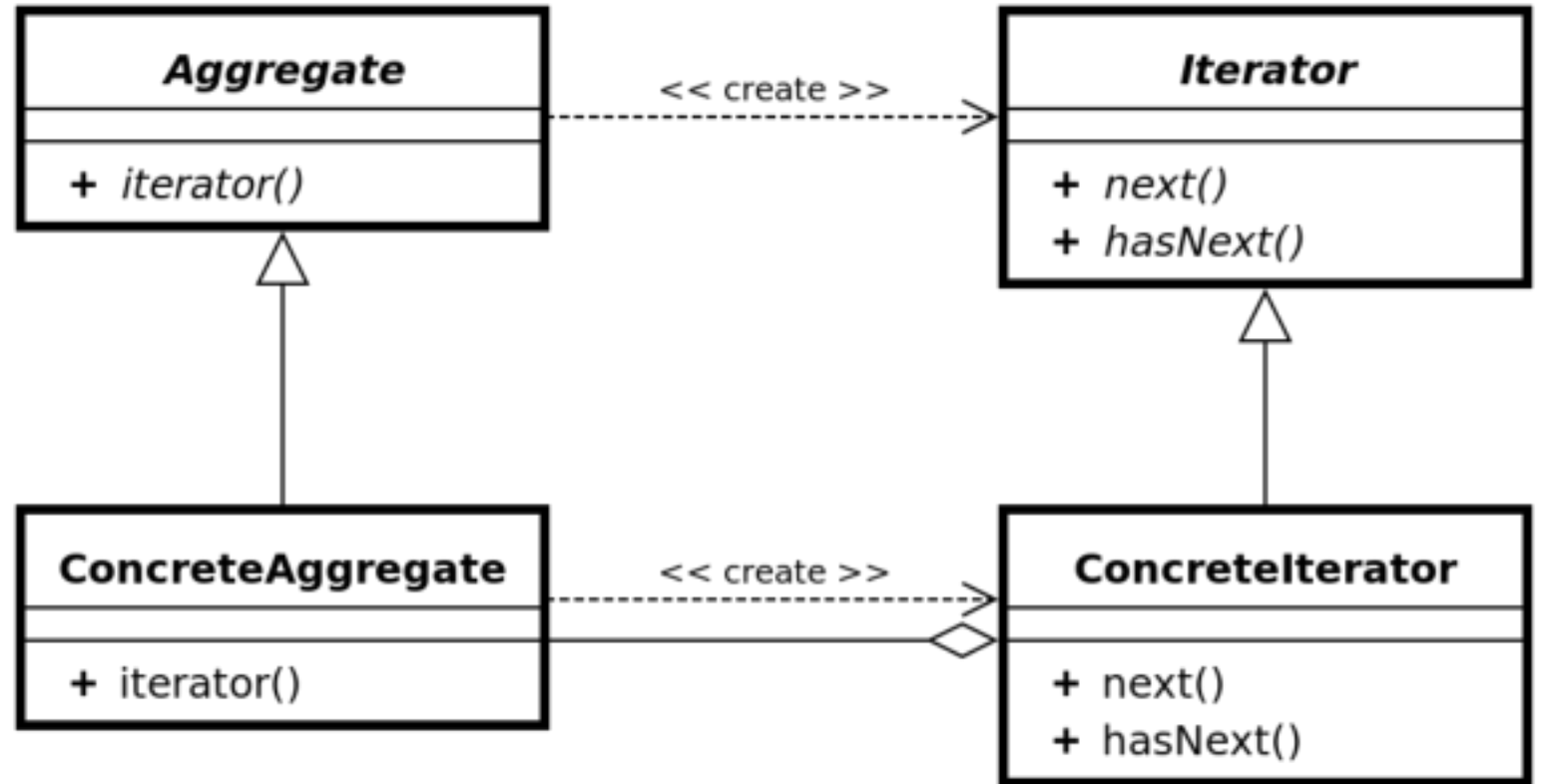
The Observer pattern done right

ReactiveX is a combination of the best ideas from
the **Observer** pattern, the **Iterator** pattern, and functional programming

Observer pattern



Iterator pattern



Why?

- Reactive programming.
 - Programming with asynchronous data streams.
- Most actions are not standalone occurrences.
 - Example: A mouse click triggers an Ajax request which triggers a UI update.
- RxJS composes these streams in a functional style.



Filtering data

With array functions

```
const numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

btnArray.addEventListener('click', function() {
  const data = numbers.map(n => ({ x: n })).filter(obj => obj.x < 7);
  result.textContent = JSON.stringify(data);
});
```

With RxJS

```
const numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

fromEvent(btnArray, 'click')
  .pipe(
    switchMap(() =>
      from(numbers).pipe(
        map(n => ({ x: n })),
        filter(obj => obj.x < 7),
        scan((prev, cur) => prev.concat(cur), [])
      )
    )
  )
  .subscribe(data => (result.textContent = JSON.stringify(data)));
```



Fetching data

With promises

```
const url =
  'http://api.icndb.com/jokes/random/?limitTo=[nerdy]&escape=javascript';

btnAjax.addEventListener('click', () => {
  fetch(url)
    .then(rsp => rsp.json())
    .then(data => ({ x: data.value.joke.length }))
    .then(obj => {
      if (obj.x < 75) {
        result.textContent = JSON.stringify(obj);
      }
    });
});
```

With RxJS

```
const url =
  'http://api.icndb.com/jokes/random/?limitTo=[nerdy]&escape=javascript';

fromEvent(btnAjax, 'click')
  .pipe(
    switchMap(() =>
      ajax.getJSON(url).pipe(
        map(data => data.value.joke.length),
        map(n => ({ x: n })),
        filter(obj => obj.x < 75),
        scan((prev, cur) => prev.concat(cur), [])
      )
    )
  )
  .subscribe(data => (result.textContent = JSON.stringify(data)));
```



Asynchronous data

With
imperative
code

```
btnInterval.addEventListener('click', () => {  
  let number = 0;  
  let numbers = [];  
  const handle = setInterval(() => {  
    if (number < 7) {  
      const obj = { x: number };  
      numbers.push(obj);  
      number++;  
    } else {  
      clearInterval(handle);  
    }  
    result.textContent = JSON.stringify(numbers);  
  }, 1000);  
});
```

With RxJS

```
fromEvent(btnInterval, 'click')
  .pipe(
    switchMap(() =>
      interval(1000).pipe(
        map(n => ({ x: n })),
        filter(obj => obj.x < 7),
        scan((prev, cur) => prev.concat(cur), [])
      )
    )
  )
  .subscribe(data => (result.textContent = JSON.stringify(data)));
```

The RxJS Observable

- An Observable is the object that emits a stream of event.
 - The observer is the code that subscribes to the event stream.

A simple clock

```
import { Observable } from 'rxjs'

const timer$ = Observable.create(subscriber => {
  setInterval(() => subscriber.next(new Date().toLocaleTimeString()), 1000)
})

timer$.subscribe(e => console.log(e))
```

Unsubscribing



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

const timer$ = Observable.create(subscriber => {
  const handle = setInterval(
    () => subscriber.next(new Date().toLocaleTimeString()),
    1000
  );

  return () => clearInterval(handle);
});

const subscription = timer$.subscribe(e => console.log(e));
setTimeout(() => subscription.unsubscribe(), 5000);
```

Creating observables



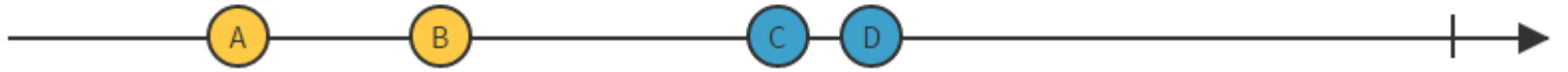
```
import { interval } from 'rxjs';  
  
const timer$ = interval(1000);  
  
const subscription = timer$.subscribe(e => console.log(e));  
setTimeout(() => subscription.unsubscribe(), 5000);
```

RxJS operators

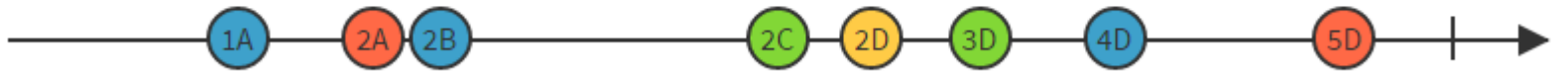
- Operators are used to operate on the event stream between the source and the subscriber.
- There are many operators for all sorts of purposes:
 - Transforming
 - Filtering
 - Combining
 - Error handling
 - Aggregate
 - ...

RxMarbles

Interactive diagrams of
Rx Observables



```
combineLatest((x, y) => "" + x + y)
```



Events

```
import { fromEvent } from 'rxjs';  
  
fromEvent(document.getElementById('btnStart'), 'click').subscribe(  
  e => console.log(e)  
);
```

Ajax

```
import { fromEvent } from 'rxjs';  
import { ajax } from 'rxjs/ajax';  
import { switchMap } from 'rxjs/operators';  
  
fromEvent(document.getElementById('btnStart'), 'click')  
  .pipe(switchMap(() => ajax.get('http://TheProblemSolver.nl')))  
  .subscribe(e => console.log(e));
```

Retry failed requests

```
import { fromEvent } from 'rxjs';
import { ajax } from 'rxjs/ajax';
import { switchMap, retry, map } from 'rxjs/operators';

fromEvent(document.getElementById('btnStart'), 'click')
  .pipe(
    switchMap(() =>
      ajax.get('http://TheProblemSolver.nl/not-found.json')
        .pipe(retry(5))
    ),
    map(rsp => rsp.response)
  )
  .subscribe(console.log);
```

Retry with backing off

```
import { fromEvent, timer } from 'rxjs';
import { ajax } from 'rxjs/ajax';
import {
  switchMap,
  retryWhen,
  map,
  scan,
  delayWhen,
  take
} from 'rxjs/operators';

fromEvent(document.getElementById('btnStart'), 'click')
  .pipe(
    switchMap(() =>
      ajax.get('http://TheProblemSolver.nl/not-found.json').pipe(
        retryWhen(error$ =>
          error$.pipe(
            map(() => 1000),
            scan((p, c) => p + c),
            delayWhen(wait => timer(wait)),
            take(5)
          )
        )
      )
    ),
    map(rsp => rsp.response)
  )
  .subscribe(e => console.log(e));
```

Combining streams

- Streams can be combined in many ways:
 - Switching
 - Combine
 - Merging
 - Zip
 - ...

Merge Example

```
import { fromEvent, merge } from 'rxjs';
import { map, scan, filter } from 'rxjs/operators';


const add$ = fromEvent(document.getElementById('add'), 'click').pipe(
  map(() => 1)
);

const subtract$ = fromEvent(document.getElementById('subtract'), 'click').pipe(
  map(() => -1)
);

merge(add$, subtract$)
  .pipe(
    scan((previous, current) => previous + current),
    filter(value => value >= 0)
  )
  .subscribe(e => console.log(e));
```

Conclusion

- Reactive programming is very powerful.
- Compose multiple asynchronous data streams.
- Transform streams using operators as needed.
- Retry failures.
- Cancel subscriptions as needed.

Thank You! 

Maurice de Beijer - @mauricedb