React JS + Redux

July 7, 8, 14, 2018

Recap:

- Chapter 1 was all about setting up, es6 etc.
- Chapter 2 create-react-app, countdown app, react, components,jsx, application state, updating states, props, hooking lifecycles, react bootstrap,
- Chapter 3 reminder app, redux, react-redux, actions and action creators, providing a store, reducers, hooks
- Chapter 4 voting app, designing a react app, prop-functions, binding custom component methods
- Chapter 5 tracker app, framework for developing a react app
- Chapter 6 api endpoints, server
- Chapter 7 redux, containers, components
- Chapter 8 containers,exercise
- Chapter 9 RXJS

Rxjs 6:

RxJS OVERVIEW REFERENCE MIGRATION TEAM

RxJS

Reactive Extensions Library for JavaScript

GET STARTED

REACTIVE EXTENSIONS LIBRARY FOR JAVASCRIPT

RxJS is a library for reactive programming using

Rxjs 6:

- Called Reactive extensions library for JavaScript
- A library for reactive programming using Observables to make it easier to compose asynchronous or callback-based code.
- Rxjs 6 is a rewrite of Reactive-Extentions/RxJS with better performance, better modularity, better debuggable call stacks
- Mostly backward compatible but with some breaking changes that reduce API surface.

To start:

```
# mkdir rxjsapp && cd rxjsapp
# npm init
# npm install rxjs webpack webpack-dev-server typescript ts-loader
# npm install webpack-cli --dev
# npm install --save rxjs-compat
# code .
```

Edit package.json:

```
{
  "name": "rxjsapp",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "webpack-dev-server --mode development"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "rxjs": "^6.2.2",
    "rxjs-compat": "^6.2.2",
    "ts-loader": "^4.4.2",
    "typescript": "^2.9.2",
    "webpack": "^4.16.1",
    "webpack-cli": "^3.1.0",
    "webpack-dev-server": "^3.1.4"
  }
}
```

Webpack.config.js(root):

```
const path = require('path');

module.exports = {
  entry: './src/code.ts',
  devtool: 'inline-source-map',
  module: {
    rules: [
      {
        test: /\.tsx?$/,
        use: 'ts-loader',
        exclude: /node_modules/
      }
    ]
  },
  resolve: {
    extensions: [ '.ts', '.js', '.tsx' ]
  },
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  }
};
```

Tsconfig.json(root):

```
{
    "compilerOptions": {
        "outDir": "./dist/",
        "noImplicitAny": true,
        "module": "es6",
        "moduleResolution": "node",
        "sourceMap": true,
        "target": "es6",
        "typeRoots": [
            "node_modules/@types"
        ],
        "lib": [
            "es2017",
            "dom"
        ]
    }
}
```

Index.html(root):

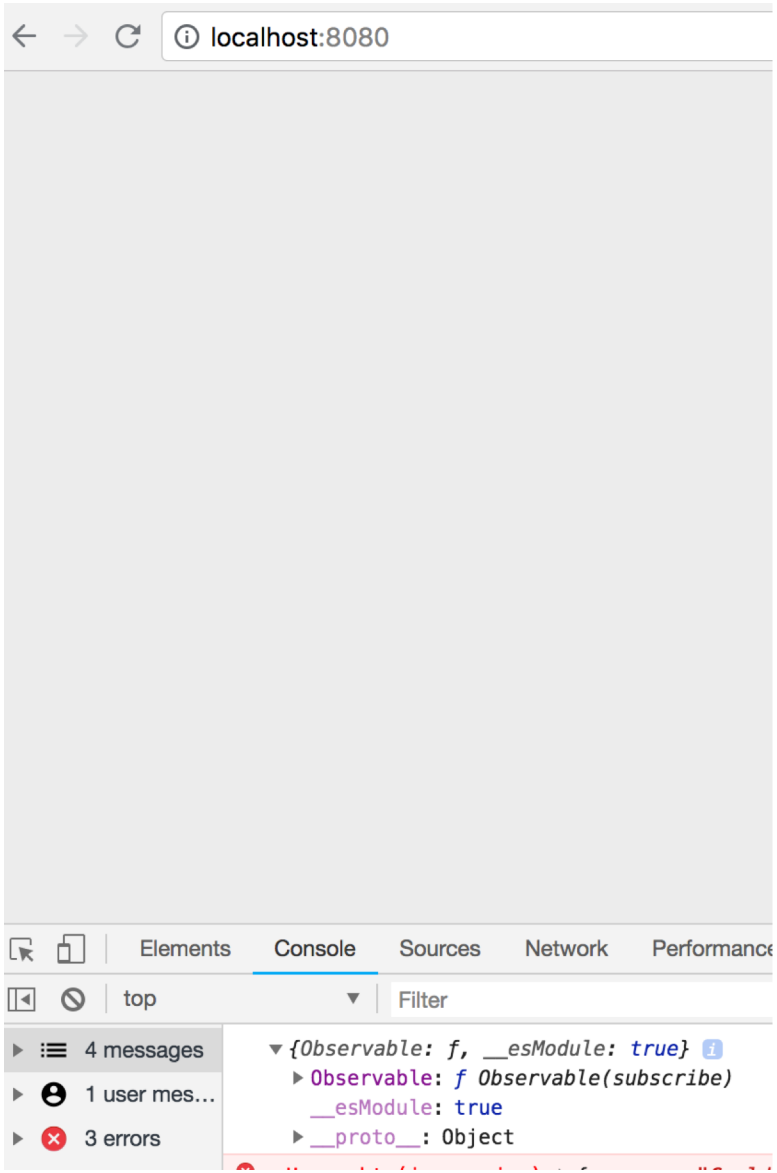
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Learn RxJS with Coursetro</title>

    <style>
        body { font-family: 'Arial'; background: #ececec; }
        ul { list-style-type: none; padding: 20px; }
        li { padding: 20px; background: white; margin-bottom: 5px; }
    </style>
</head>
<body>
    <ul id="output"></ul>

    <script src="/bundle.js"></script>
</body>
</html>
```

Create src/code.ts:

code.ts:

```
import * as Rx from "rxjs/Observable";

console.log(Rx);
```

Page:


localhost:8080
Elements
Console
Sources
Network
Performance
top
Filter
4 messages
1 user mes...
3 errors
{Observable: f, __esModule: true}
Observable: f Observable(subscribe)
__esModule: true
__proto__: Object

Reactive programming:

- The hardest part of the learning journey is **thinking in Reactive**
- It's a lot about letting go of old imperative and stateful habits of typical programming, and forcing your brain to work in a different paradigm

So what is Reactive programming?:

- Wikipedia is too generic and theoretical as usual
- Stackoverflow's canonical answer is obviously not suitable for newcomers
- Reactive Manifesto sounds like the kind of thing you show to your project manager or the businessmen at your company
- Microsoft's Rx terminology "Rx = Observables + LINQ + Schedulers" is so heavy and Microsoftish that most of us are left confused
- Terms like "reactive" and "propagation of change" don't convey anything specifically different to what your typical MV* and favorite language already does
- Let’s cut to the chase!

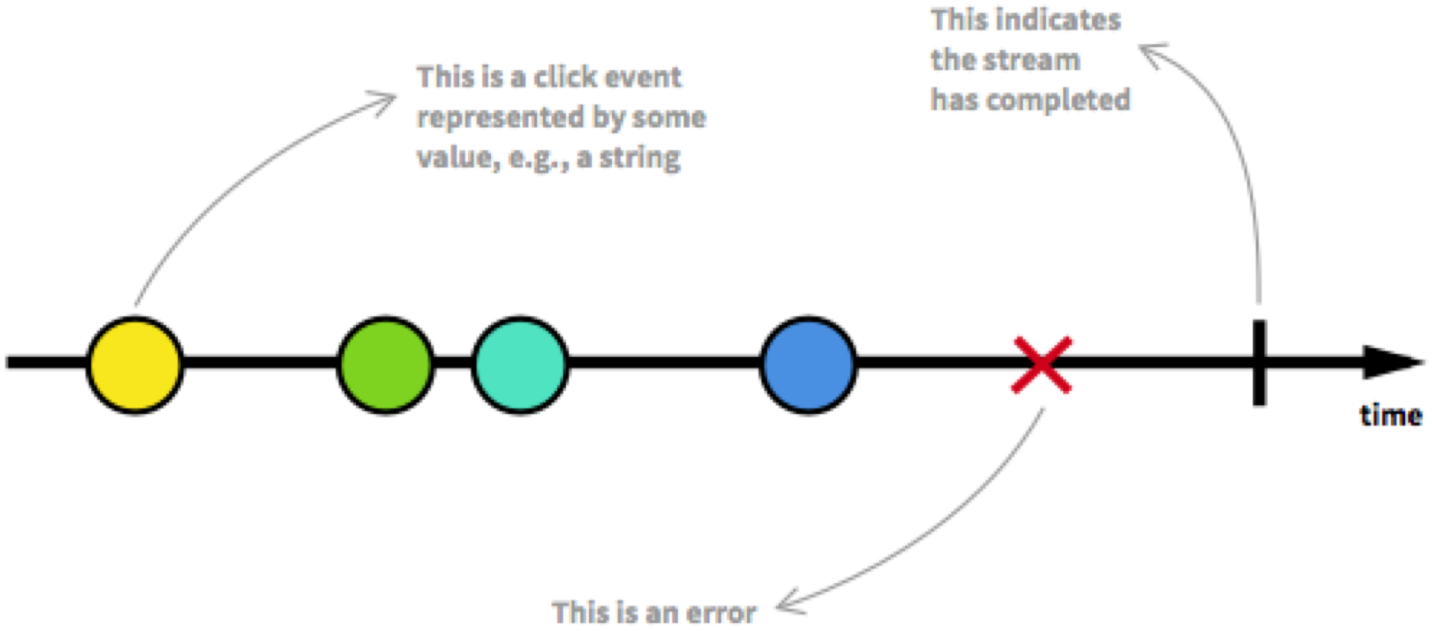
So what is Reactive programming?:

- **Reactive programming is programming with asynchronous data streams.**
- this isn't anything new
- Event buses or your typical click events are really an asynchronous event stream, on which you can observe and do some side effects
- Reactive is that idea on steroids
- You are able to create data streams of anything, not just from click and hover events.
- Streams are cheap and ubiquitous, anything can be a stream: variables, user inputs, properties, caches, data structures, etc
- imagine your Twitter feed would be a data stream in the same fashion that click events are. You can listen to that stream and react accordingly.

So what is Reactive programming?:

- **On top of that, you are given an amazing toolbox of functions to combine, create and filter any of those streams.**
- That's where the "functional" magic kicks in. A stream can be used as an input to another one.
- Even multiple streams can be used as inputs to another stream
- You can *merge* two streams
- You can *filter* a stream to get another one that has only those events you are interested in.
- You can *map* data values from one stream to another new one.
- Streams are so central to Reactive so let's take a careful look at them
- starting with our familiar "clicks on a button" event stream.

Stream


This is a click event represented by some value, e.g., a string
This indicates the stream has completed
time
This is an error

What is a stream?:

- A stream is a sequence of **ongoing events ordered in time**
- It can emit three different things: a value (of some type), an error, or a "completed" signal
- Consider that the "completed" takes place, for instance, when the current window or view containing that button is closed.
- We capture these emitted events only **asynchronously**,
- by defining a function that will execute when a value is emitted, another function when an error is emitted, and another function when 'completed' is emitted
- Sometimes these last two can be omitted and you can just focus on defining the function for values
- The "listening" to the stream is called **subscribing**
- The functions we are defining are observers
- The stream is the subject (or "observable") being observed

What is a Stream?:

- Marbles are the emitted values
- X is an error
- | is the completed signal
- --→ is the timeline

Why should I consider adopting RP?

- Reactive Programming raises the level of abstraction of your code so you can focus on the interdependence of events that define the business logic, rather than having to constantly fiddle with a large amount of implementation details
- The benefit is more evident in modern webapps and mobile apps that are highly interactive with a multitude of UI events related to data events
- 10 years ago, interaction with web pages was basically about submitting a long form to the backend and performing simple rendering to the frontend
- Apps have evolved to be more real-time: modifying a single form field can automatically trigger a save to the backend
- "likes" to some content can be reflected in real time to other connected users, and so forth.

Streams and Observables

- Stream is just a concept of values or events that are emitted over time.
- Samples like bunch of users chatting on a chat room
- Individual filling up a form
- And observable is what facilitates the stream
- It provides you with a function and a means to both emit those values and respond to them
- Empty code.ts

code.ts

```
import { Observable } from 'rxjs/Observable';   49.3K (gzipped: 11.7K)

var observable = Observable.create(function subscribe(observer:any) {
    observer.next('Hey Guys')
});
```

Code.ts:

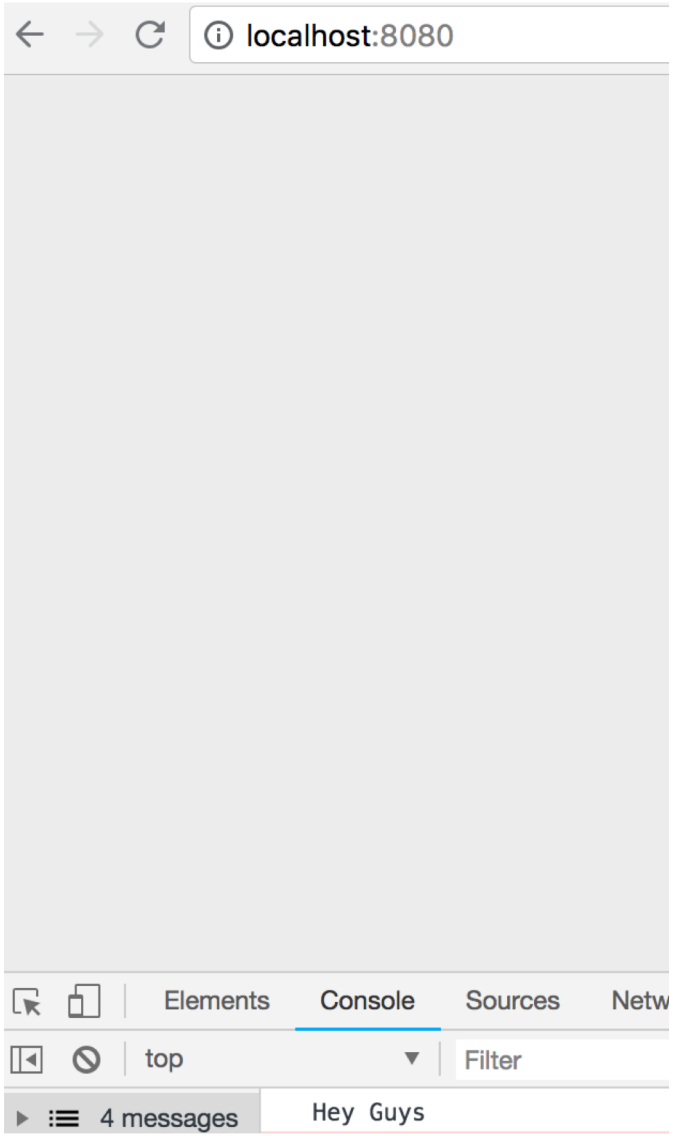
- We cleared code.ts as we don't need to import all the rxjs library
- In this exercise we want to experiment on observables
- So we import { Observable } from "rxjs/Observable";
- Next is we will create an observable
- var observable = Observable.**create**() // only one way of creating an observables there are multiple ways
- Create() method accepts a single argument which is a subscribe function
- We need to pass a subscribe function to create()
- The we pass in that observer inside the subscribe function
- And to emit a value we reference observer.next() and we pass in what ever it is that we want to emit
- By creating this nothing will still happen

code.ts refactor:

```
import { Observable } from 'rxjs/Observable';   49.3K (g

var observable = Observable.create((observer:any) => {
    observer.next('Hey Guys')
});
```

code.ts:

```
import { Observable } from 'rxjs/Observable';   49.3K (

var observable = Observable.create((observer:any) => {
    observer.next('Hey Guys')
});

observable.subscribe((x:any) => console.log(x));
```



Code.ts:

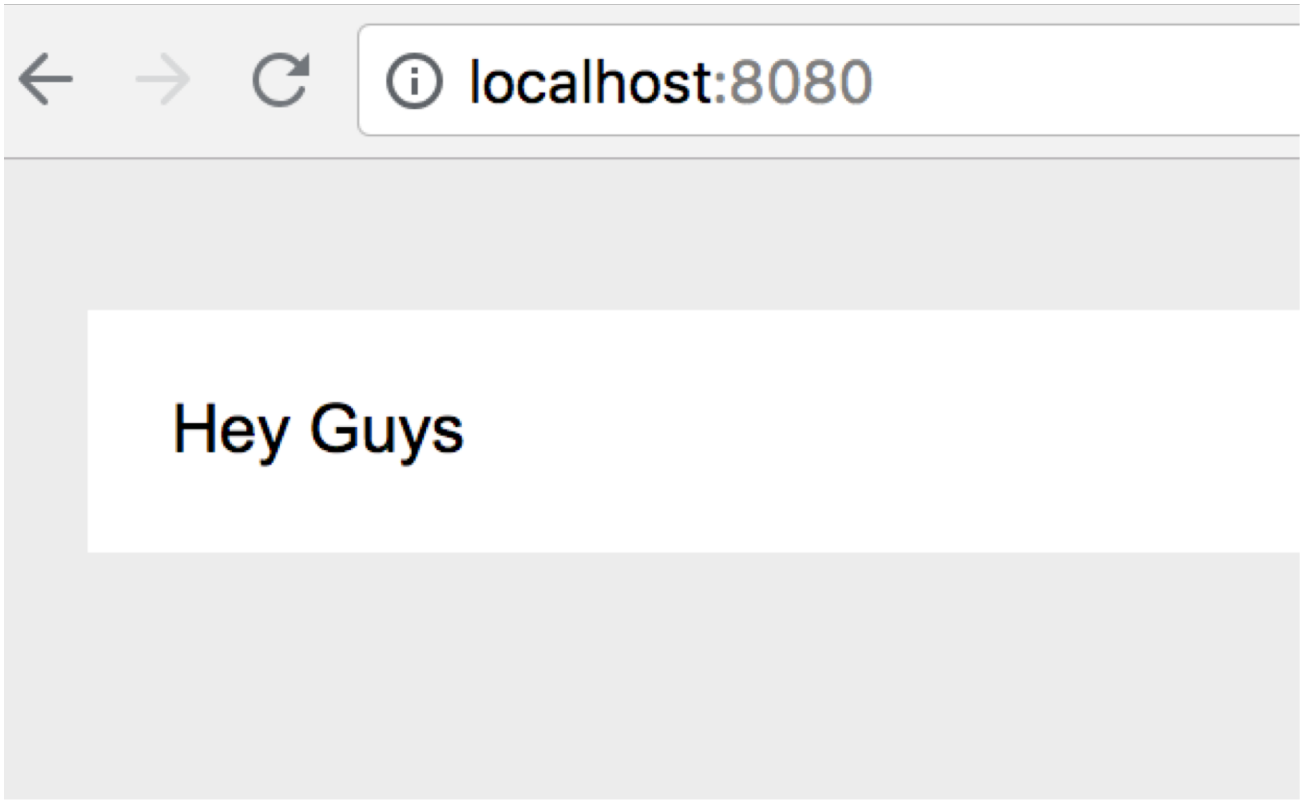
- If we want to grab the value we need to define an observer
- observable.subscribe()
- Then we take which ever value is emitted from next()
- We can reference that of x:any of type any (for ts)
- Using an arrow function
- Console x to log

Presentation fix:

- Instead of us always doing a console.log we will show it in screen
- We will add the addItem function

```
import { Observable } from 'rxjs/Observable';   49.3K (gz

var observable = Observable.create((observer:any) => {
    observer.next('Hey Guys')
});

observable.subscribe((x:any) => addItem(x));

function addItem(val:any) {
    var node = document.createElement("li");
    var textnode = document.createTextNode(val);
    node.appendChild(textnode);
    document.getElementById("output").appendChild(node);
}
```

Page:



Observers:

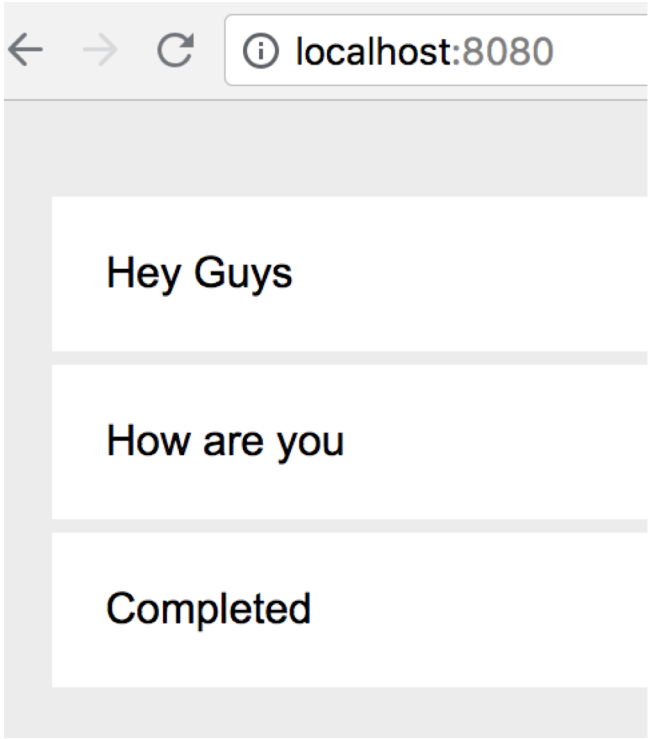
```
observable.subscribe((x:any) => addItem(x));
```

- Everytime we subscribe you are an observer and we create what we call a subscription
- Observers they read values coming from the observable
- An observer is simply a set of callbacks that accept notifications coming from this observer (inside the observable) which include next(), error(), and complete()

```
var observable = Observable.create((observer:any) => {
    observer.next('Hey Guys')
});
```

More complex observables:

```
var observable = Observable.create((observer:any) => {
    try {
    observer.next('Hey Guys')
    observer.next('How are you')
    observer.complete()
    observer.next('this will not work')
    } catch(err) {
        observer.error(err)
    }
});

observable.subscribe(
    (x:any) => addItem(x),
    (error:any) => addItem(error),
    () => addItem('Completed')
);
```



- If we do this what do you think will happen?

Cancelling a subscription:

- When you subscribe to an observable with an observer therby creating a subscription
- We have the ability to cancel that subscription in an event that you no longer need to receive the emitted values from the observer.
- But what is we want just one of our subscribers don't need to receive those emitted values anymore?
- To demonstrate this we would need do some coding.

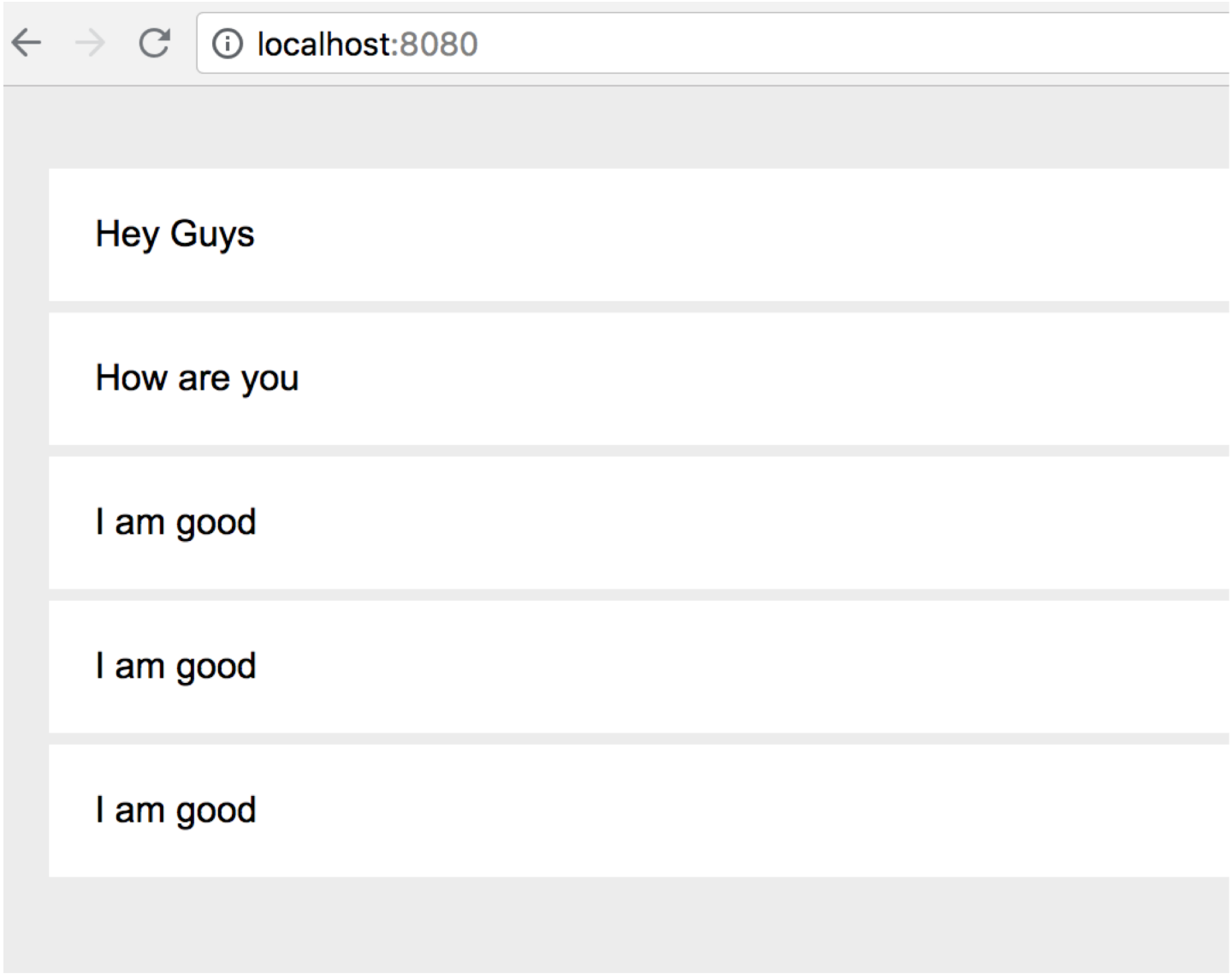
Code.ts:

```
var observable = Observable.create((observer:any) => {
    try {
    observer.next('Hey Guys')
    observer.next('How are you')
    setInterval(() => {
        observer.next('I am good')
    }, 2000)
    } catch(err) {
        observer.error(err)
    }
});

var observer = observable.subscribe(
    (x:any) => addItem(x),
    (error:any) => addItem(error),
    () => addItem('Completed')
);

setTimeout(() => {
    observer.unsubscribe()
}, 6001)
```

Cancelling a subscription:

- We are going to use a set interval function
- This will just repeat whatever is in inside for a certain duration we specify
- Put observer.next('I am good')
- For a duration of every 2 seconds
- Remove the observer.complete and observer.next after it
- Lets give a name(variable) to our observer so we can reference it
- Var observer = observable.subscribe
- Then create a setTimeut function
- Inside it we will call observer.unsubscribe() and at 6001 seconds and 1 miliseconds
- Try it out

Page:


localhost:8080
Hey Guys
How are you
I am good
I am good
I am good

Multiple subscriptions:

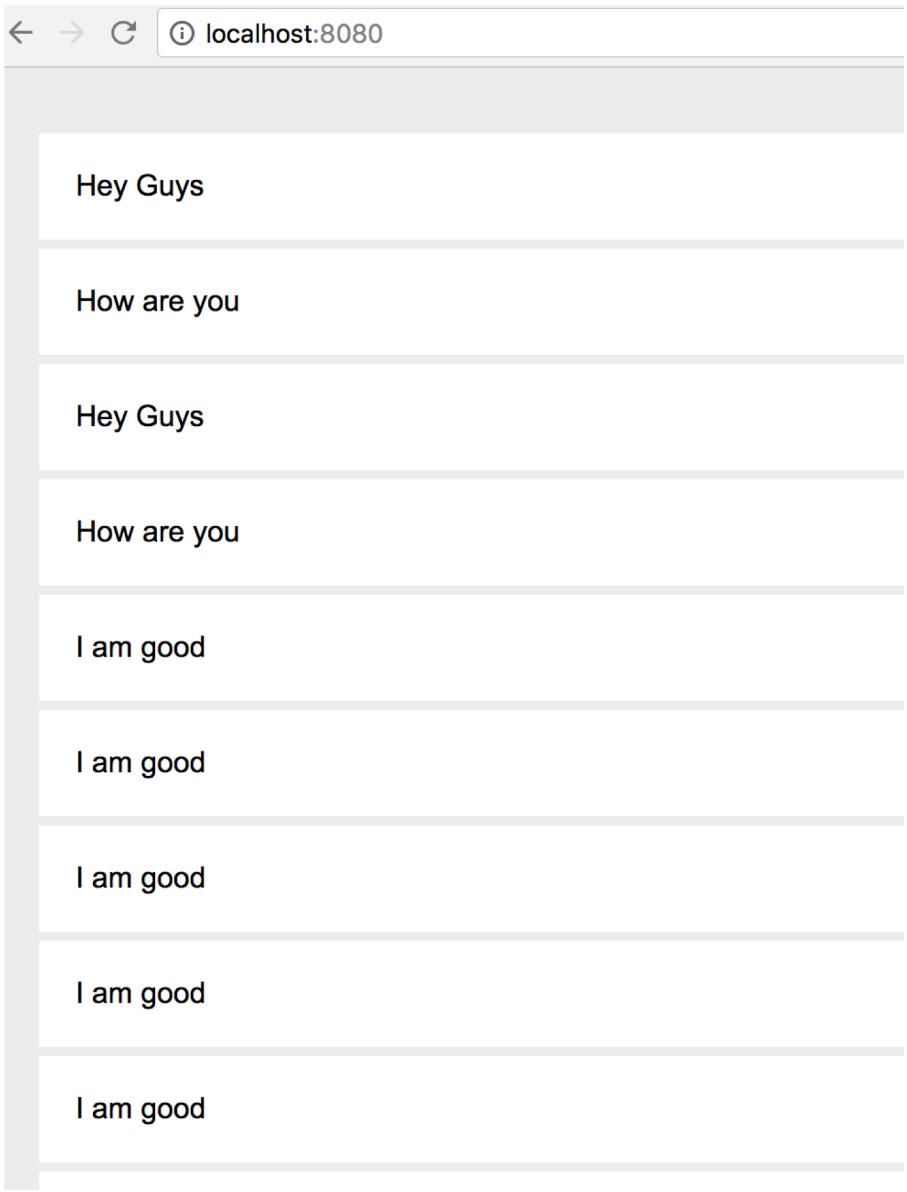
- We are able to create multiple subscriptions on the same observable very easily
- This can be useful if you have different areas of your user interface where you are using the same data but your displaying it in a different way
- So we copy our observer and rename it to observer2
- Let's remove the error and complete callbacks for observer 2
- Save and run and let's check!

Code.ts:

```
var observable = Observable.create((observer:any) => {
    try {…
    } catch(err) {…
    }
});

var observer = observable.subscribe(
    (x:any) => addItem(x),
    (error:any) => addItem(error),
    () => addItem('Completed')
);

var observer2 = observable.subscribe(
    (x:any) => addItem(x)
);

setTimeout(() => {
    observer.unsubscribe()
}, 6001)
```

Phoenix one The Knowledge Institute

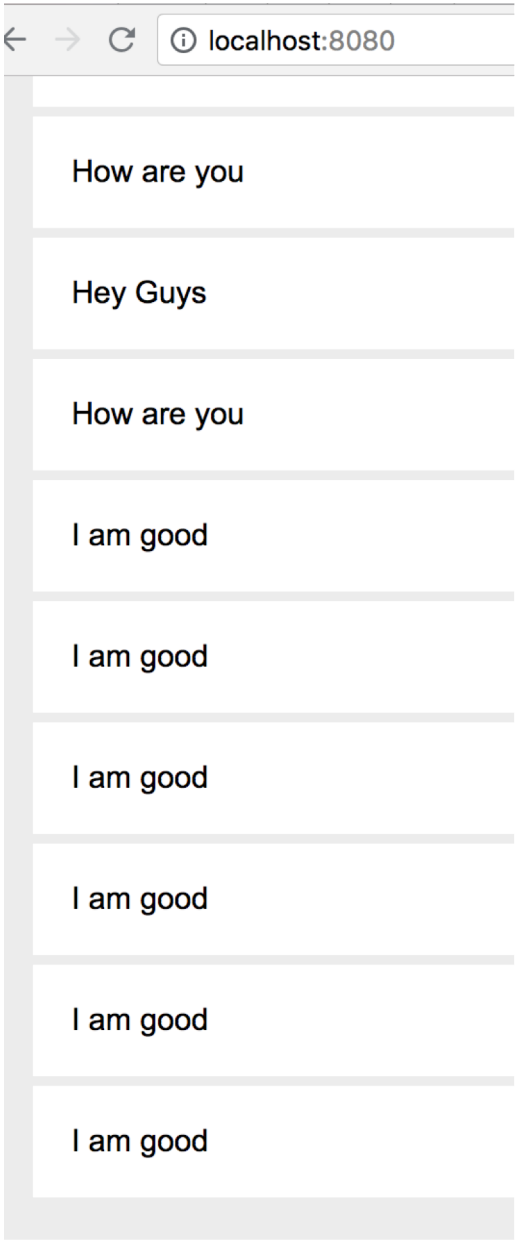
Page:



Child subscriptions:

- What if we wanted to unsubscribe both of our subscriptions if one has been unsubscribed?
- To do that all we have to call is observer.add(observer2)
- Save and check but now they stop at the same time.
- By the way we can also call on observer.remove()

Page:


localhost:8080
How are you
Hey Guys
How are you
I am good
I am good
I am good
I am good
I am good
I am good

Hot vs Cold Observables:

- Now what we need to understand is whether an observable is hot or cold!
- In the case of the cold observable like the type that we are working with

```
var observable = Observable.create((observer:any) => {
    try {
    observer.next('Hey Guys')
    observer.next('How are you')
    setInterval(() => {
        observer.next('I am good')
    }, 2000)
    } catch(err) {
        observer.error(err)
    }
});
```

ACTIVATED

```
var observer = observable.subscribe(
    (x:any) => addItem(x),
    (error:any) => addItem(error),
    () => addItem('Completed')
);
```

Hot vs Cold Observables:

```
var observable = Observable.create((observer:any) => {
    try {
    observer.next('Hey Guys')
    observer.next('How are you')
    setInterval(() => {
        observer.next('I am good')
    }, 2000)
    } catch(err) {
        observer.error(err)
    }
});

var observer = observable.subscribe(
    (x:any) => addItem(x),
    (error:any) => addItem(error),
    () => addItem('Completed')
);

setTimeout(() => {
    var observer2 = observable.subscribe(
        (x:any) => addItem('Subscriber 2:' +x)
    )
}, 1000)
```

Hot vs Cold Observables:

- An observable is hot when the producer is emitting values outside of the observable
- To show a sample of this is import 'rxjs/add/operator/share';
- This is more of a warm observable
- An example of a truly hot observable would be mouse movements for instance that is made by a user

```
var observable = Observable.create((observer:any) => {
    try {
    observer.next('Hey Guys')
    observer.next('How are you')
    setInterval(() => {
        observer.next('I am good')
    }, 2000)
    } catch(err) {
        observer.error(err)
    }
}).share();
```

Code.ts:

```
import { Observable } from 'rxjs/Observable';   49.3K (gzi
import { fromEvent } from 'rxjs/Observable/fromEvent';   4

var observable = fromEvent(document, 'mousemove');

setTimeout(() => {
    var subscription = observable.subscribe(
        (x: any) => addItem(x)
    )
}, 2000);

function addItem(val:any) {
    var node = document.createElement("li");
    var textnode = document.createTextNode(val);
    node.appendChild(textnode);
    document.getElementById("output").appendChild(node);
}
```

Page:

[object MouseEvent]

[object MouseEvent]

[object MouseEvent]

[object MouseEvent]

[object MouseEvent]

[object MouseEvent]

[object MouseEvent]

[object MouseEvent]

Subjects:

- Subjects are just a different type of observables
- It has different capabilities
- A subject in contrast to an observable is simple an observer that is also able to emit values
- Its both an observable and an observer simultaneously
- Unlilke an observable because as an observer that is subscribed to an observable it can only read values emitted from an observable
- So to start please edit code.ts

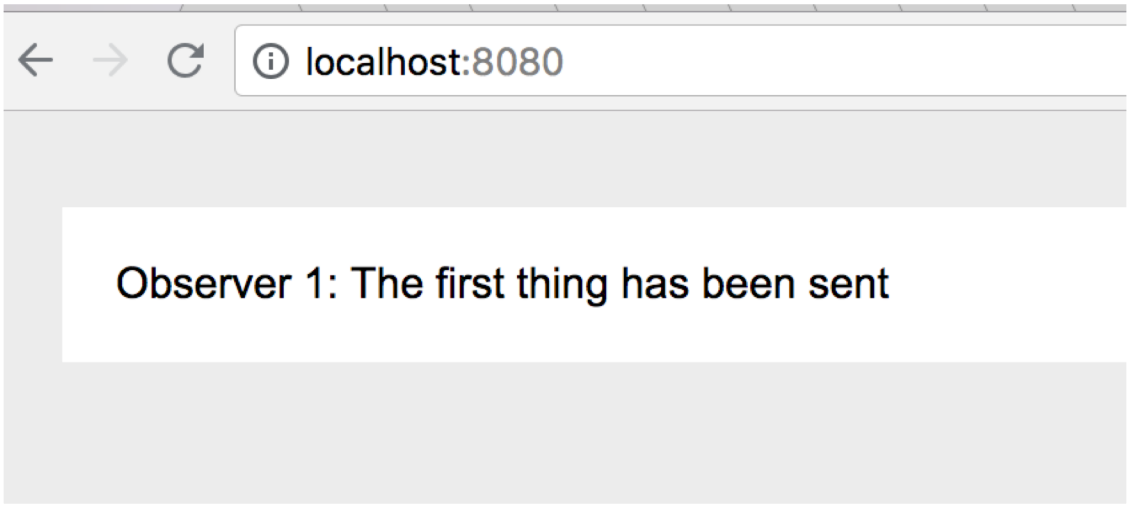
```
import { Subject } from 'rxjs/Subject';   49.3K (gzipped:


function addItem(val:any) {
    var node = document.createElement("li");
    var textnode = document.createTextNode(val);
    node.appendChild(textnode);
    document.getElementById("output").appendChild(node);
}
```

Code.ts:

```
import { Subject } from 'rxjs/Subject';   49.3K (gzipped:

var subject = new Subject()

subject.subscribe(
    data => addItem('Observer 1: ' + data),
    err => addItem(err),
    () => addItem('Observer 1: Completed')
)

subject.next('The first thing has been sent');

function addItem(val:any) {
    var node = document.createElement("li");
    var textnode = document.createTextNode(val);
    node.appendChild(textnode);
    document.getElementById("output").appendChild(node);
}
```

~/react-redux/Sample/rxjsapp/

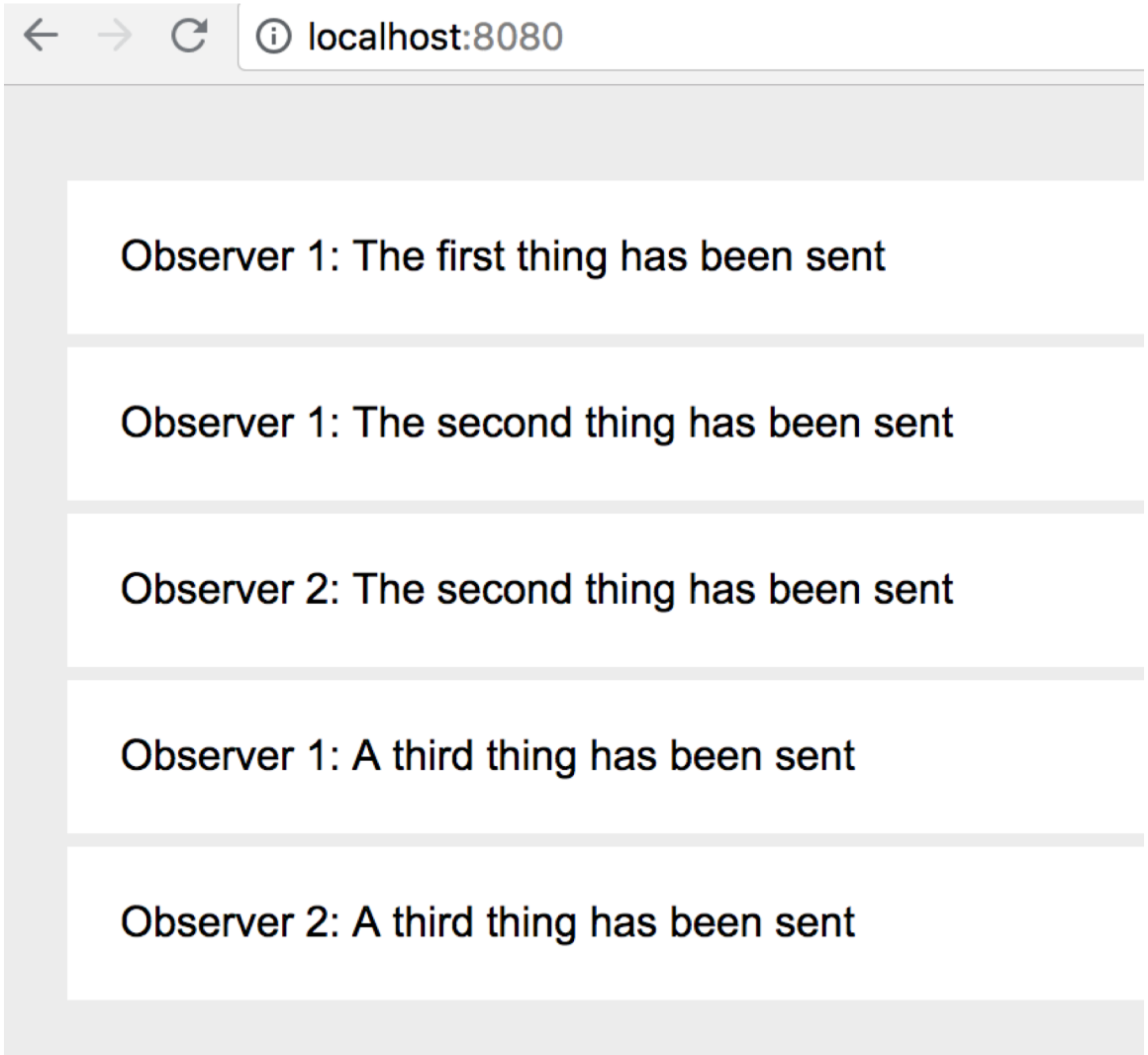
Page:



Code.ts:

```
var subject = new Subject()

subject.subscribe(
    data => addItem('Observer 1: ' + data),
    err => addItem(err),
    () => addItem('Observer 1: Completed')
)

subject.next('The first thing has been sent')

var observer2 = subject.subscribe(
    data => addItem('Observer 2: ' + data)
)

subject.next('The second thing has been sent')
subject.next('A third thing has been sent')
```

Page:


localhost:8080
Observer 1: The first thing has been sent
Observer 1: The second thing has been sent
Observer 2: The second thing has been sent
Observer 1: A third thing has been sent
Observer 2: A third thing has been sent

Code.ts:

```
var subject = new Subject()

subject.subscribe(
    data => addItem('Observer 1: ' + data),
    err => addItem(err),
    () => addItem('Observer 1: Completed')
)

subject.next('The first thing has been sent')

var observer2 = subject.subscribe(
    data => addItem('Observer 2: ' + data)
)

subject.next('The second thing has been sent')
subject.next('A third thing has been sent')

observer2.unsubscribe();

subject.next('A final thing has been sent')
```

Page:

Observer 1: The first thing has been sent

Observer 1: The second thing has been sent

Observer 2: The second thing has been sent

Observer 1: A third thing has been sent

Observer 2: A third thing has been sent

Observer 1: A final thing has been sent

Other Subjects:

- There is 3 different variations of subjects
- First is the behavior subject, they are all very similar but they offer different capabilities.
- A behavior subject is a special type of subject whose only difference is that it will emit the last value upon a new observers subscription
- To test this we edit code.ts

Code.ts:

```
import { BehaviorSubject } from 'rxjs/BehaviorSubject';

var subject = new BehaviorSubject('First')

subject.subscribe(
    data => addItem('Observer 1: ' + data),
    err => addItem(err),
    () => addItem('Observer 1: Completed')
)

subject.next('The first thing has been sent')
subject.next('...Observer 2 is about to subscribe...')

var observer2 = subject.subscribe(
    data => addItem('Observer 2: ' + data)
)
```

Page:

Observer 1: First

Observer 1: The first thing has been sent

Observer 1: ...Observer 2 is about to subscribe...

Observer 2: ...Observer 2 is about to subscribe...

Observer 1: The second thing has been sent

Observer 2: The second thing has been sent

Observer 1: A third thing has been sent

Observer 2: A third thing has been sent

Observer 1: A final thing has been sent

Other Subjects:

- The second type of subject is replay subject
- It is like behavior subject but it allows you to specify a buffer or number of emitted values to dispatch to observers
- If behavior subject only dispatches the last value replay subject allows you dispatch any designated number of value
- Lets edit our code.ts

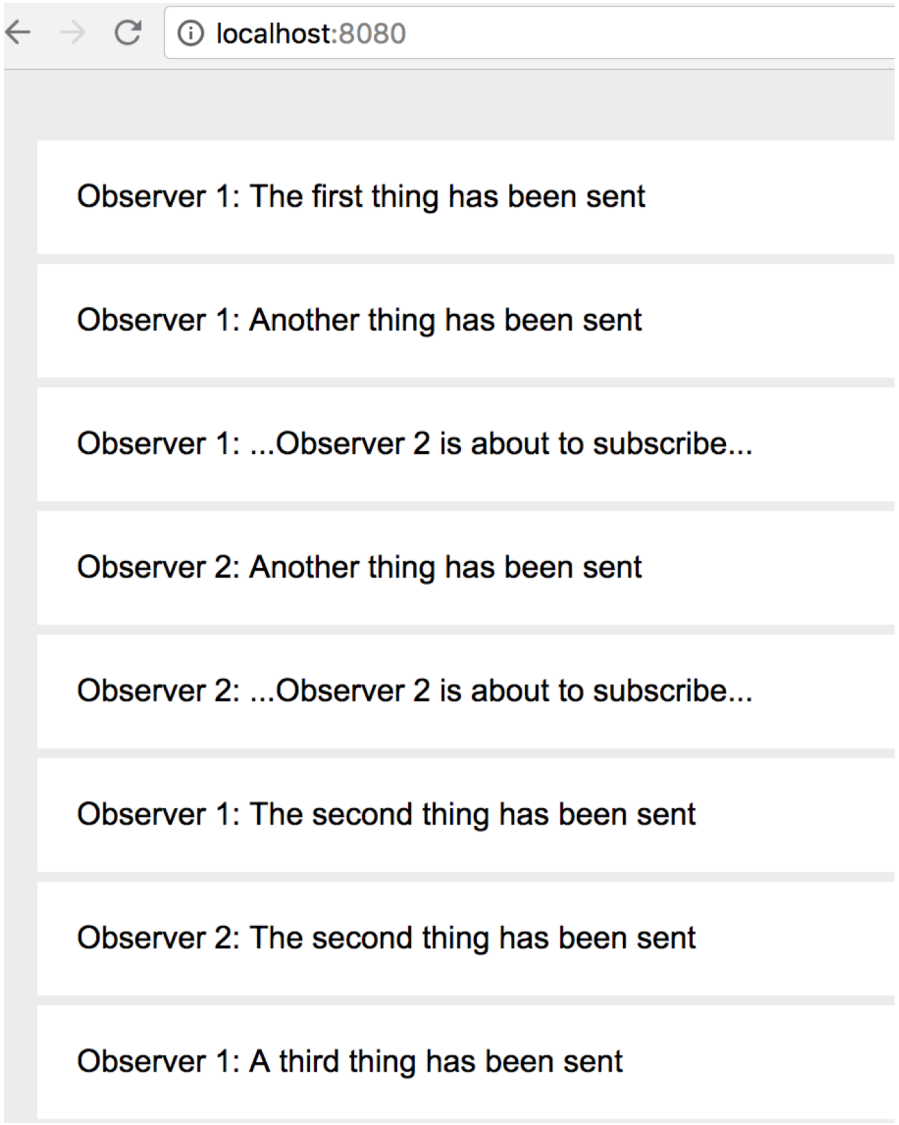
Phoenix one The Knowledge Institute

Code.ts:

```
import { ReplaySubject } from 'rxjs/ReplaySubject';   49

var subject = new ReplaySubject(2)

subject.subscribe(
    data => addItem('Observer 1: ' + data),
    err => addItem(err),
    () => addItem('Observer 1: Completed')
)

subject.next('The first thing has been sent')
subject.next('Another thing has been sent')
subject.next('...Observer 2 is about to subscribe...')

var observer2 = subject.subscribe(
    data => addItem('Observer 2: ' + data)
)
```

Phoenix one The Knowledge Institute

Page:



Replay Subjects:

- Replay Subject also can take another parameter in milliseconds
- Let say we wanted to return 30 events within a 200 millisecond buffer time
- var subject = new **ReplaySubject**(30, 200)

Async Subjects:

- The last type of subject and the most simple to understand
- Async subject only emits the last value and it will only do so once the complete method has been called upon the subject
- Lets edit code.ts

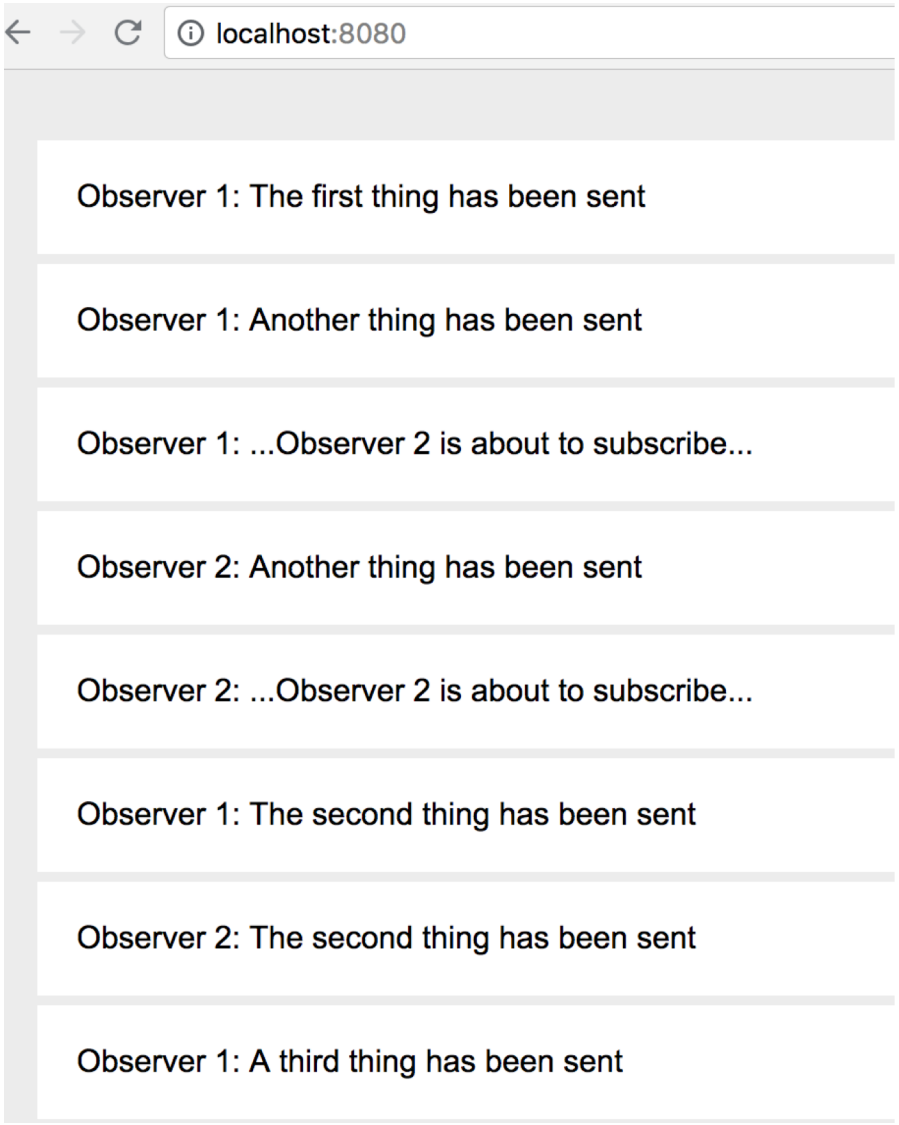
Phoenix one The Knowledge Institute

Code.ts:

```
import { AsyncSubject } from 'rxjs/AsyncSubject';   49.3K

var subject = new AsyncSubject()

subject.subscribe(
    data => addItem('Observer 1: ' + data),
    () => addItem('Observer 1: Completed')
)

var i = 1;
var int = setInterval(() => subject.next(i++), 100);

setTimeout(() => {
    var observer2 = subject.subscribe(
        data => addItem('Observer 2: ' + data)
    )
    subject.complete()
}, 500)


function addItem(val:any) {
    var node = document.createElement("li");
    var textnode = document.createTextNode(val);
    node.appendChild(textnode);
    document.getElementById("output").appendChild(node);
```

Phoenix one The Knowledge Institute

Page:



Operators:

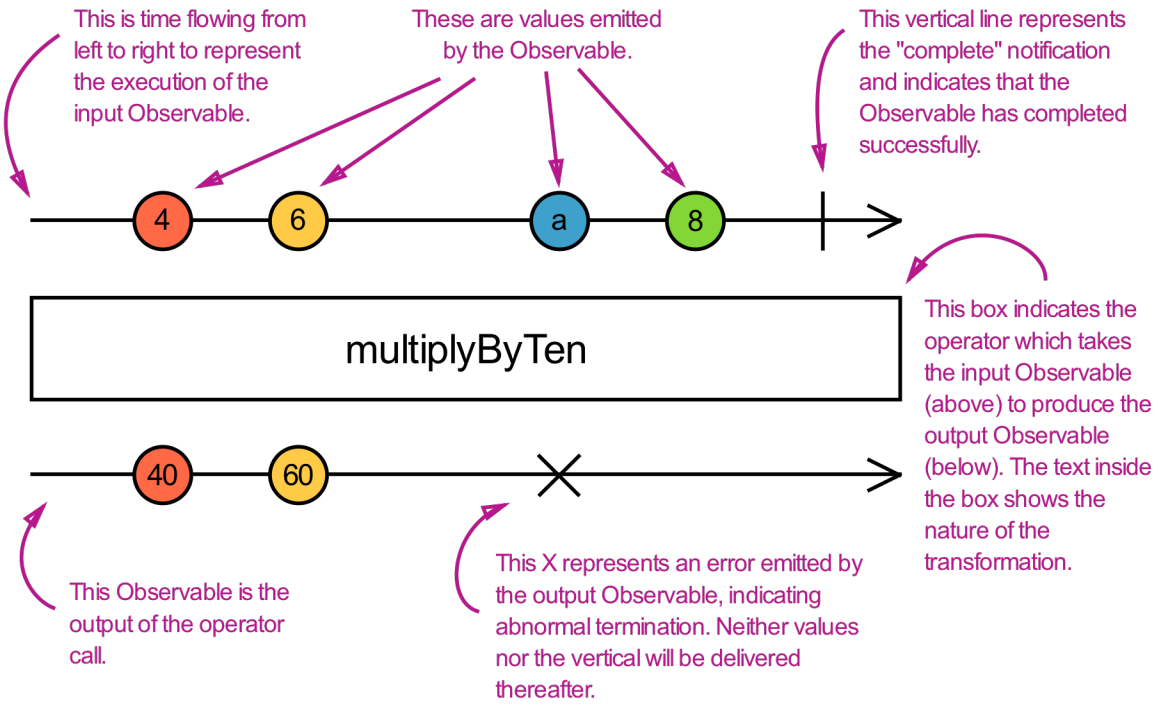
- Rxjs operators are the most used outside observables
- 100+ different operators
- Operators are simply methods that you can use on Observables (and Subjects) that allow you to change the original observable in some manner and return a new observable
- These operators do not change the existing Observable; they simply modify it and return a new one
- Operators are known as *pure functions*, which are functions that do not modify the variables outside of its scope.
- You can also create a sequence of operators that will modify an incoming observable, output a new observable, and so on..

Types of operators:

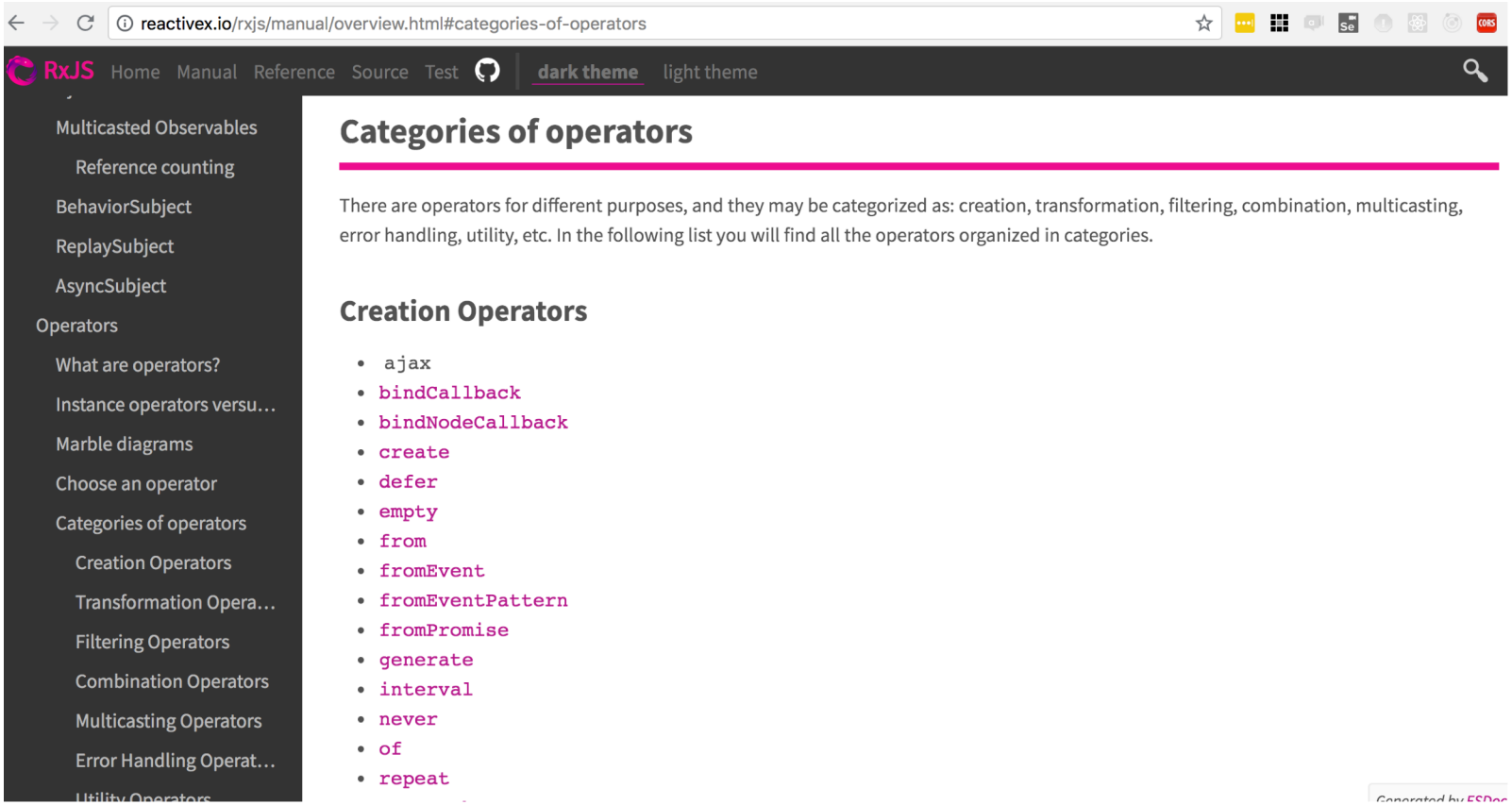
- Static operators - These operators are usually used to create observables. You will find these mainly under the creation operators.
- Instance operators - These are methods on observable *instances*. These account for the majority of RxJS operators that are used.

Understanding Marble Diagrams:

- The official documentation uses marble diagrams to help you understand how a given operator modifies an observable.
- Here's the official explanation of how a marble diagram works:

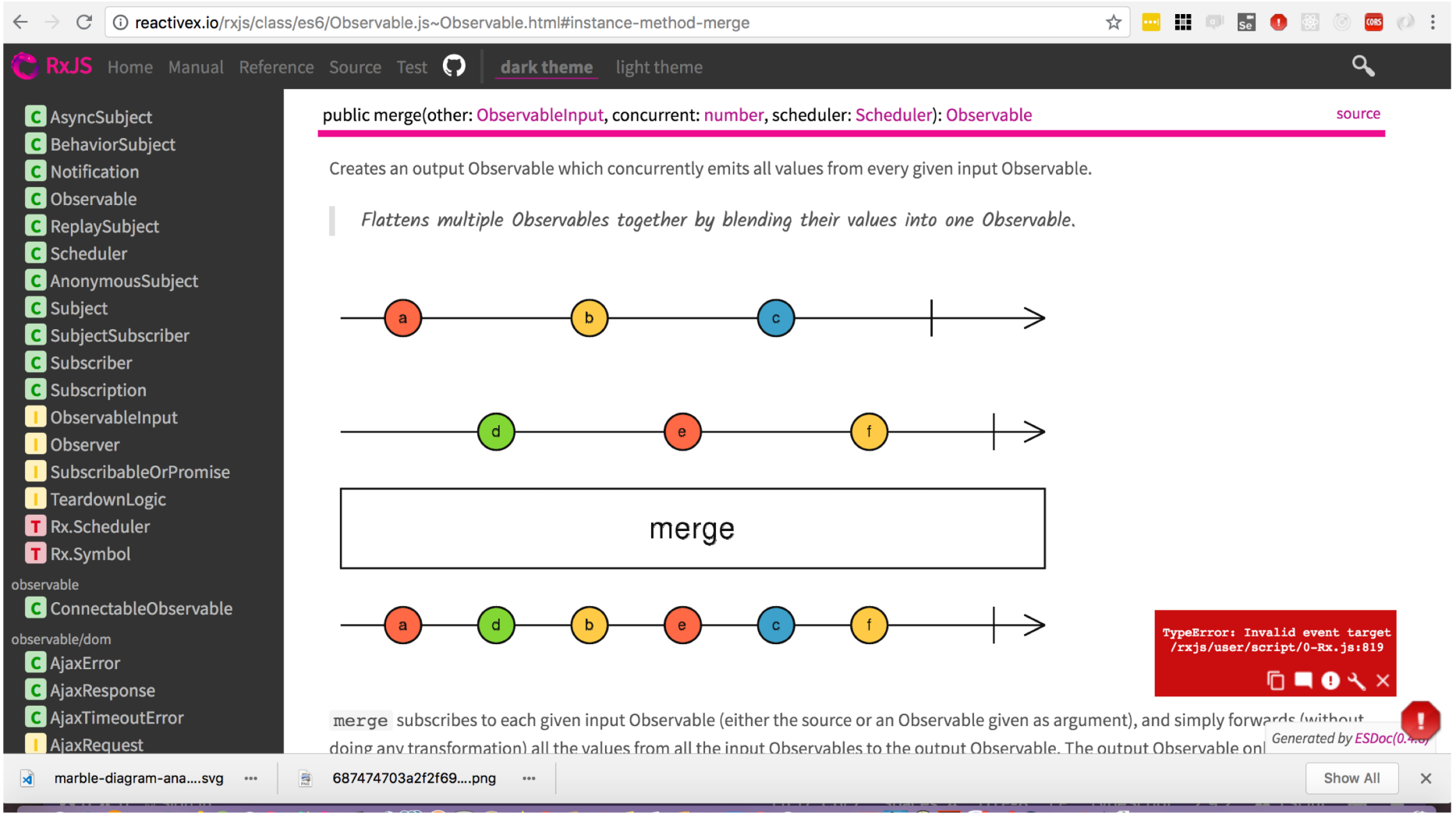


Operators:



http://reactivex.io/rxjs/manual/overview.html#categories-of-operators

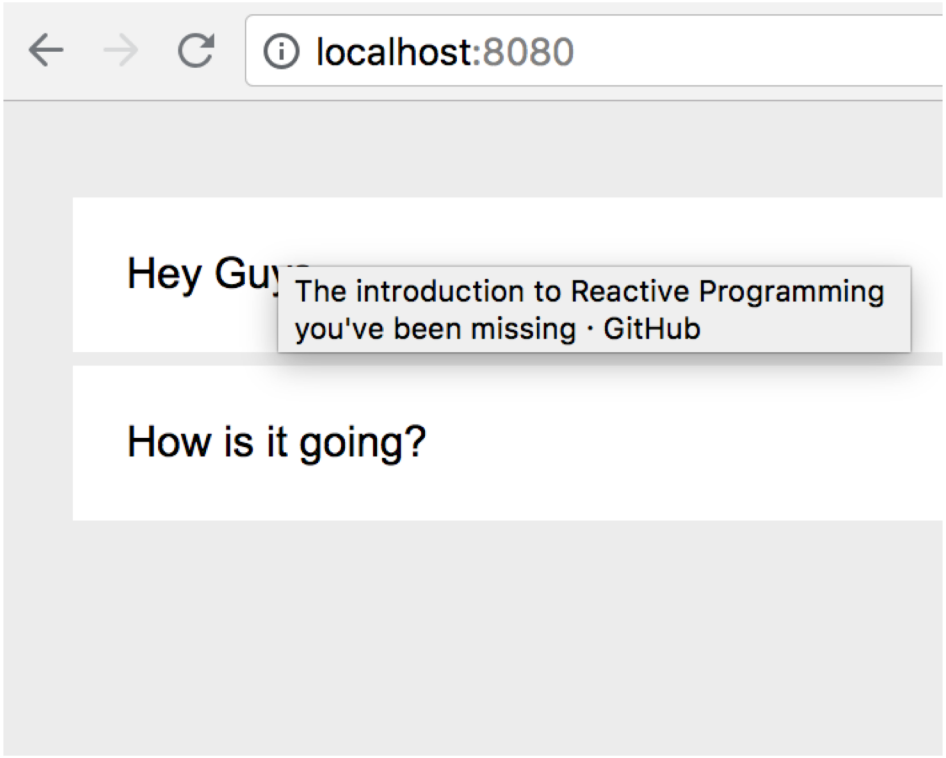
merge:


reactivex.io/rxjs/class/es6/Observable.js~Observable.html#instance-method-merge
RxJS Home Manual Reference Source Test dark theme light theme
AsyncSubject
BehaviorSubject
Notification
Observable
ReplaySubject
Scheduler
AnonymousSubject
Subject
SubjectSubscriber
Subscriber
Subscription
ObservableInput
Observer
SubscribableOrPromise
TeardownLogic
Rx.Scheduler
Rx.Symbol
observable
ConnectableObservable
observable/dom
AjaxError
AjaxResponse
AjaxTimeoutError
AjaxRequest
public merge(other: ObservableInput, concurrent: number, scheduler: Scheduler): Observable
source
Creates an output Observable which concurrently emits all values from every given input Observable.
Flattens multiple Observables together by blending their values into one Observable.
a
b
c
d
e
f
merge
a
d
b
e
c
f
TypeError: Invalid event target
/rxjs/user/script/0-Rx.js:819
merge subscribes to each given input Observable (either the source or an Observable given as argument), and simply forwards (without
doing any transformation) all the values from all the input Observables to the output Observable. The output Observable onl
Generated by ESDoc(0.4.8)
marble-diagram-ana....svg
687474703a2f2f69....png
Show All

Merge in code.ts:

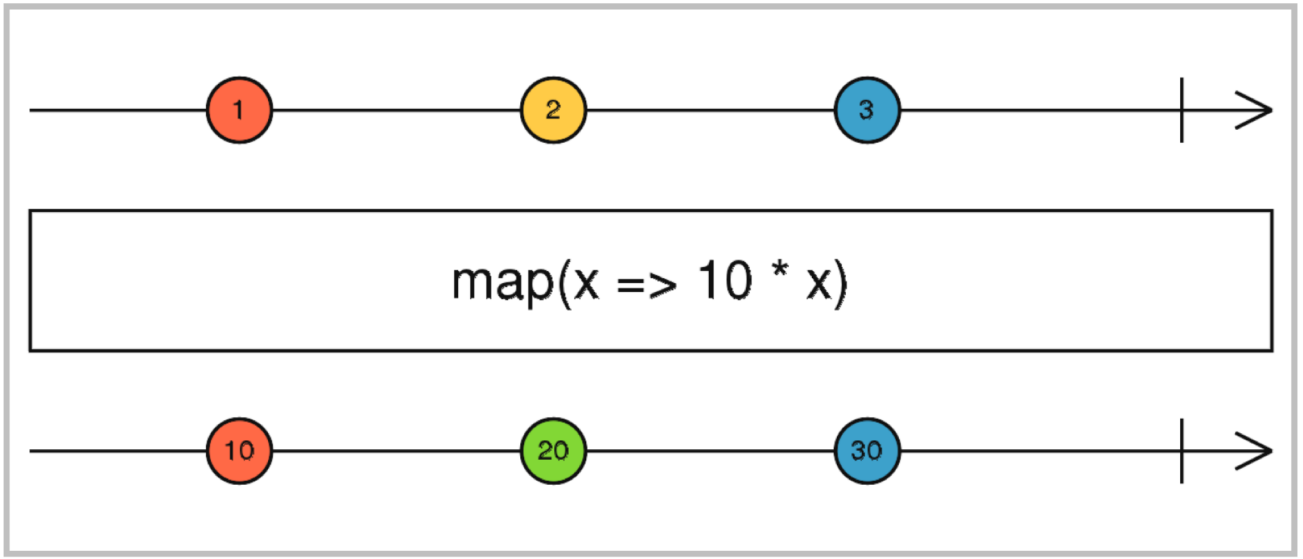
```
import { Observable } from 'rxjs/Observable';   49.3K (gz
import { merge } from 'rxjs/observable/merge';   49.3K (g

var observable = Observable.create((observer:any) => {
    observer.next('Hey Guys')
});

var observable2 = Observable.create((observer:any) => {
    observer.next('How is it going?')
});

var newObs = merge(observable, observable2);

newObs.subscribe((x:any) => addItem(x));

function addItem(val:any) {
    var node = document.createElement("li");
    var textnode = document.createTextNode(val);
    node.appendChild(textnode);
    document.getElementById("output").appendChild(node);
}
```

Page:


localhost:8080
Hey Guy
The introduction to Reactive Programming you've been missing · GitHub
How is it going?

Let us try one more map:

- Map is one operator that you see a lot when working within Angular and API's. It simply allows you to take the input values and make some type of transformation.

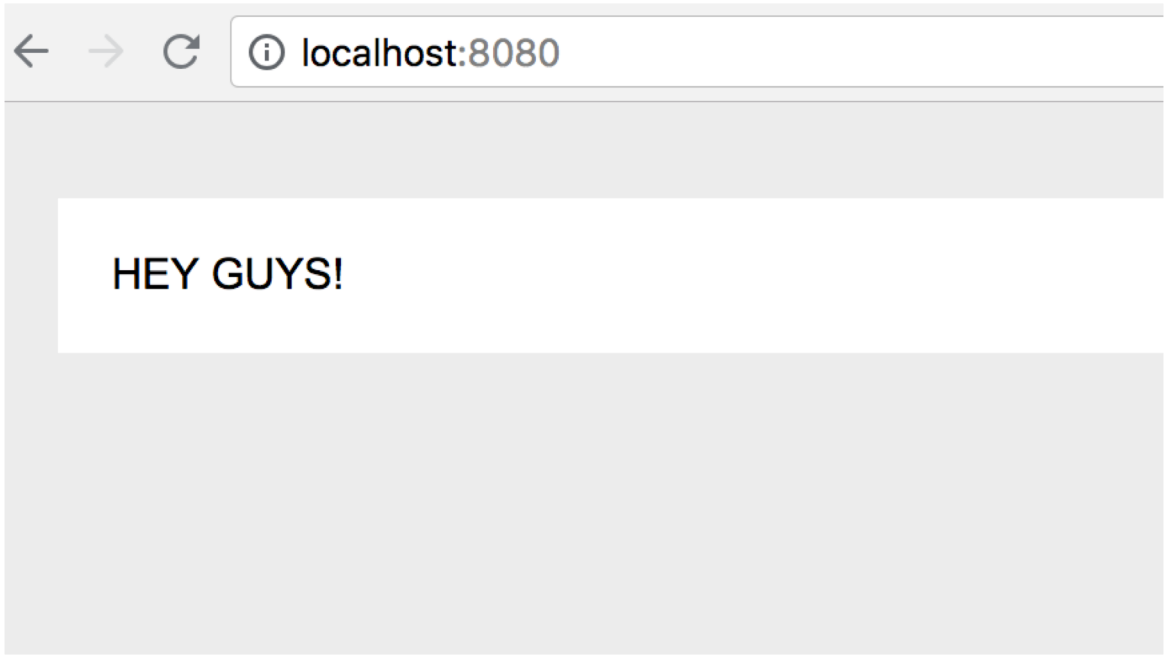


So, if we wanted to multiply numbers, append a string to the end of events, or make all of the results capital, we could use *.map*

map in code.ts:

```
import { Observable } from 'rxjs/Observable';   49.3K (gz
import 'rxjs/add/operator/map';

Observable.create((observer:any) => {
    observer.next('Hey Guys!')
})
.map((val:any) => val.toUpperCase())
.subscribe((x:any) => addItem(x))

function addItem(val:any) {
    var node = document.createElement("li");
    var textnode = document.createTextNode(val);
    node.appendChild(textnode);
    document.getElementById("output").appendChild(node);
}
```

Page:



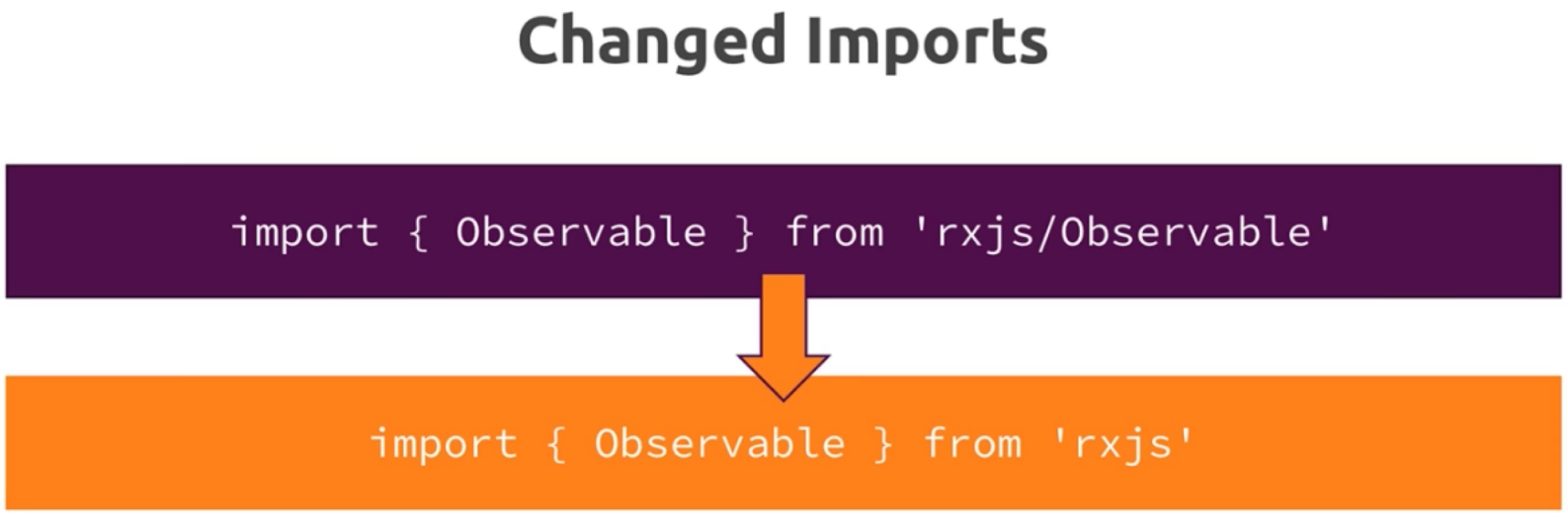
RXJS 6:

- Its is still very new not much documentation
- There are breaking changes
- That is why we had to install rxjs-compat
- It is a quick fix
- Ensures backward compatibility
- Old code should not require any changes

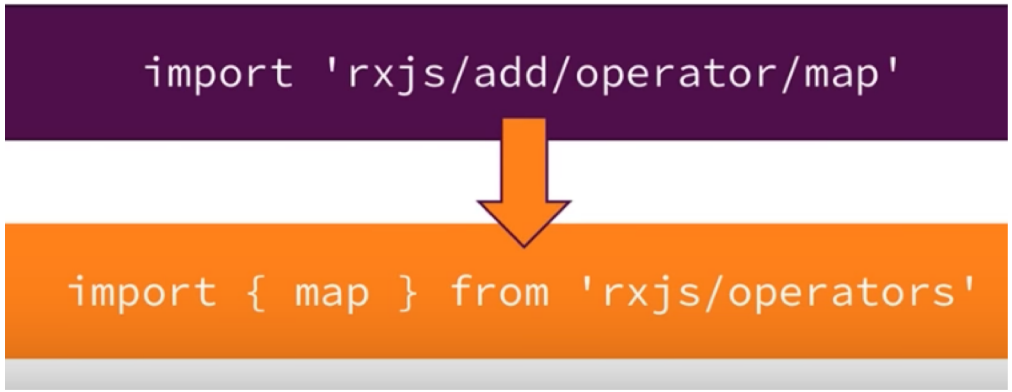
Changes RXJS 6:

- Changed package structure
- Internally it was repackaged to make biundle smaller and more efficient
- To make imports easier
- It updated all import statements and operators
- They also introduced pipes, pipeable operators!
- This allows us to use the operators a little bit different than before to help more on usability that resulted in having some operators renamed

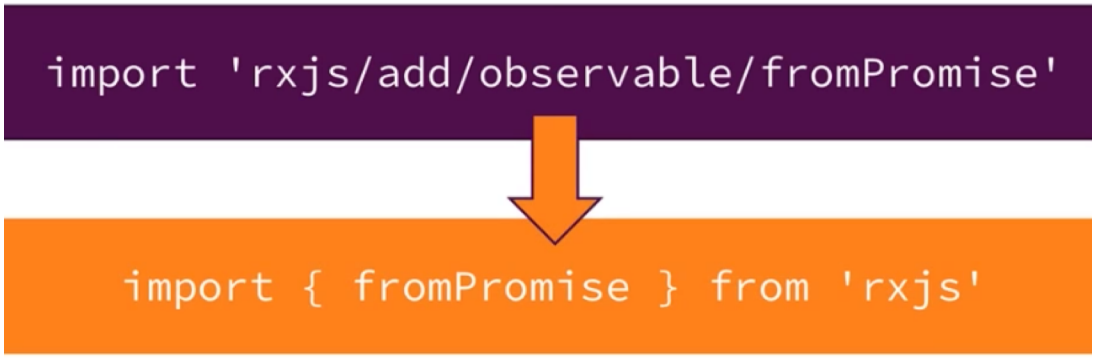
Changed imports:



Changed imports:



Changed imports:

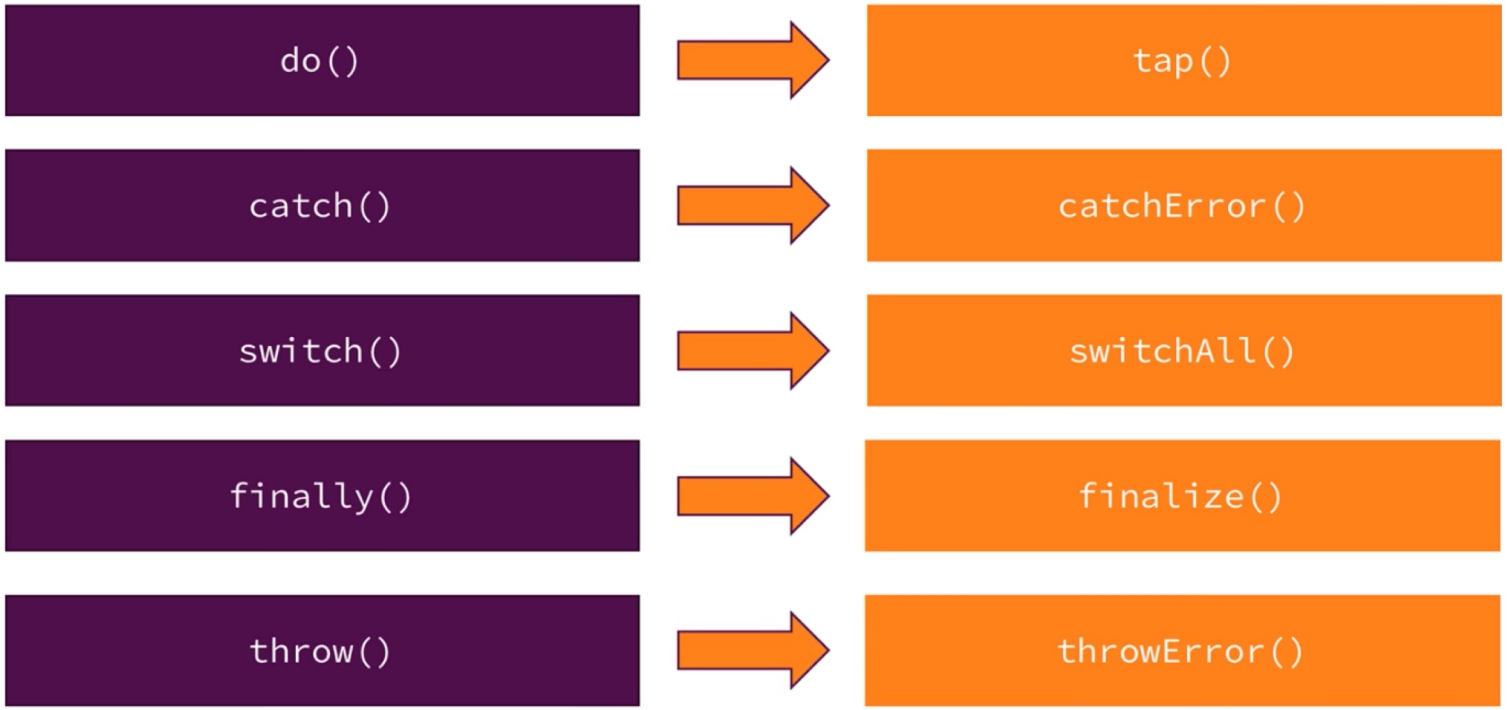


Pipeable Operators:

```
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/throttle';
someObservable
        .map(…)
        .throttle(…)
        .subscribe(…)
```

```
import { map, throttle } from 'rxjs/operators';

someObservable
        .pipe(map(…),
              throttle(…))
        .subscribe(…)
```

Renamed Operators



Epics:

- redux-observable requires an understanding of Observables with RxJS v6
- redux-observable (because of RxJS) truly shines the most for complex async/side effects
- An **Epic** is the core primitive of redux-observable.
- It is a function which takes a stream of actions and returns a stream of actions. **Actions in, actions out**

End of Chapter 9