FREE eBook

LEARNING

rxjs

Free unaffiliated eBook created from
Stack Overflow contributors.

#rxjs

Table of Contents

About ... 1

Chapter 1: Getting started with rxjs ... 2

Remarks ... 2

Versions ... 2

Examples ... 2

Installation or Setup ... 2

Using a CDN: ... 2

Using a bundler: ... 3

Chapter 2: Common recipes ... 4

Introduction ... 4

Examples ... 4

Caching HTTP responses ... 4

Discarding slow/outdated rest calls ... 7

Sending multiple parallel HTTP requests ... 8

Sending multiple sequential HTTP requests ... 11

Rate limiting ... 14

Chapter 3: Operator: map / select ... 18

Syntax ... 18

Parameters ... 18

Remarks ... 18

Examples ... 18

Using an element to yield ... 18

Using a transform function ... 18

Using a transform function and the element index ... 19

Chapter 4: Operator: PublishReplay ... 20

Examples ... 20

How does PublishReplay work ... 20

Unexpected emissions when using publishReplay ... 20

Chapter 5: Retry and RetryWhen Operators ... 22

Introduction ... 22

Syntax 22
Examples 22
Retry with backoff, until success or max number of attempts reached 22
Chapter 6: Scheduler 23
Examples 23
Using a TestScheduler to advance time manually 23
Chapter 7: Subject 24
Introduction 24
Examples 24
Subject and its internal state 24
Credits 26

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: rxjs

It is an unofficial and free rxjs ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official rxjs.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with rxjs

Remarks

This section provides an overview of what rxjs is, and why a developer might want to use it.

It should also mention any large subjects within rxjs, and link out to the related topics. Since the Documentation for rxjs is new, you may need to create initial versions of those related topics.

Versions

Version	Release date
RxJS 4	2015-09-25
RxJS 5	2016-12-13
RxJS 5.0.1	2016-12-13
RxJS 5.1.0	2017-02-01

Examples

Installation or Setup

Using a CDN:

```
<!DOCTYPE html>
<head>
  <script src="https://cdn.jsdelivr.net/rxjs/4.1.0/rx.min.js"></script>
</head>
<body>

  <script>
    // `Rx` is available
    var one$ = Rx.Observable.of(1);
    var onesub = one$.subscribe(function (one) {
      console.log(one); // 1
    });
    // alternatively: subscribe(console.log)
  </script>
</body>
</html>
```

CDN if using RxJS 5 (RC):

```
<script src="https://npmcdn.com/@reactivex/rxjs@5.0.0-rc.1/dist/global/Rx.js"></script>
```

Using a bundler:

First install into your project directory with npm:

```
npm install rx
```

Or using RxJS 5 (RC):

```
npm install rxjs
```

Then, in your JavaScript file:

```
var Rx = require('rx');
//var Rx = require('rxjs/Rx'); // v5beta

var one$ = Rx.Observable.of(1);
var onesub = one$.subscribe(function (one) {
  console.log(one); // 1
});
```

If using an es6/2015 compatible bundler:

```
import Rx from 'rx';
//import Rx from 'rxjs/Rx'; // v5beta

const one$ = Rx.Observable.of(1);
const onesub = one$.subscribe(one => console.log(one)); // 1
```

Read Getting started with rxjs online: https://riptutorial.com/rxjs/topic/2361/getting-started-with-rxjs

Chapter 2: Common recipes

Introduction

A collection of common use cases and their implementation in RxJS.

Examples

Caching HTTP responses

A typical use case for RxJS is creating HTTP requests and caching their results for some period of time. Also, we always want to run only one request at a time and share its response.

For example the following code caches **1** item for max. **1000ms**:

```
var updateRequest = Observable.defer(() => makeMockHttpRequest())
    .publishReplay(1, 1000)
    .refCount()
    .take(1);

var counter = 1;
function makeMockHttpRequest() {
    return Observable.of(counter++)
        .delay(100);
}

function requestCachedHttpResult() {
    return updateRequest;
}
```

Function `makeMockHttpRequest()` simulates an HTTP request that arrives with `100ms` delay.

Function `requestCachedHttpResult()` is where we subscribe to get actual or cached response.

With `.publishReplay(1, 1000)` we used RxJS multicasting to internally use `ReplaySubject` and keep `1` item for maximum `1000ms`. Then `refCount()` is used to keep always only one subscription to the `source` which is `Observable.defer()`. This Observable is used to create a new request and increments `counter` to prove that cached values and new subscriptions share the same data.

When we want to get current data we call `requestCachedHttpResult()`. To ensure the Observer will be completed properly after emitting data we used `take(1)` operator.

```
requestCachedHttpResult()
    .subscribe(val => console.log("Response 0:", val));
```

This creates a single request with `mockDataFetch()` and prints to console:

```
1
```

A more complicated example will call multiple requests at different times where we want to test that the mocked HTTP connections and responses are shared.

```
requestCachedHttpResult()
    .subscribe(val => console.log("Response 0:", val));

setTimeout(() => requestCachedHttpResult()
    .subscribe(val => console.log("Response 50:", val))
, 50);

setTimeout(() => requestCachedHttpResult()
    .subscribe(val => console.log("Response 200:", val))
, 200);

setTimeout(() => requestCachedHttpResult()
    .subscribe(val => console.log("Response 1200:", val))
, 1200);

setTimeout(() => requestCachedHttpResult()
    .subscribe(val => console.log("Response 1500:", val))
, 1500);

setTimeout(() => requestCachedHttpResult()
    .subscribe(val => console.log("Response 3500:", val))
, 3500);
```

See live demo: https://jsbin.com/todude/5/edit?js,console

Each request is sent with delay and happen in the following order:

`0` - First request that makes the `refCount()` to subscribe to its `source` which makes the `mockDataFetch()` call. Its response is going to be delayed by `100ms`. At this moment `ConnectableObservable` inside `publishReplay()` operator has **one** Observer.

`50` - Second request subscribes to the `ConnectableObservable` as well. At this moment `ConnectableObservable` inside `publishReplay()` operator has **two** Observer. It doesn't create another request with `makeMockHttpRequest()` because `refCount()` is already subscribed.

`100` - The first response is ready. It's first cached by the `ReplaySubject` and then reemitted to the **two** Observers subscribed to `ConnectableObservable`. Both Observers are completed thanks to `take(1)` and unsubscribed.

`200` - Subscribes to the `ReplaySubject` that immediately emits its cached value which causes `take(1)` to complete the Observer and unsubscribes right away. No HTTP requests are made and no subscription remains.

`1200` - The same as the first event at `0`. At this point the cached value has been discarded because it's older than `1000ms`.

`1500` - Same as the fourth event at `200`.

`3500` - The same as the first event at `1200`.

The output in console is the following:

```
Response 0: 1
Response 50: 1
Response 200: 1
Response 1200: 2
Response 1500: 2
Response 3500: 3
```

In RxJS 5 a similar functionality was covered by `cache()` operator. However, it was removed in `5.0.0-rc.1` due to its limited functionality.

Handling errors

If we want to handle errors produced by the remote service (the `makeMockHttpRequest` function) we need to catch them before they're merged into the main Observable chain because any error received by the `ReplaySubject` inside `publishReplay()` would mark its internal state as `stopped` (Read more here Subject and its internal state) which is definitelly not what we want.

In the following example we're simulating an error when `counter === 2` and catching it with the `catch()` operator. We're using `catch()` to only transform the `error` notification into a regular `next` so we can handle the error in observers:

```
function makeMockHttpRequest() {
    return Observable.of(counter++)
        .delay(100)
        .map(i => {
            if (i === 2) {
                throw new Error('Invalid URL');
            }
            return i;
        })
        .catch(err => Observable.of(err));
}
```

See live demo: https://jsbin.com/kavihu/10/edit?js,console

This will print to console the following output. Notice the errors are received in the `next` handlers:

```
Response 0: 1
Response 50: 1
Response 200: 1
Response 1200: [object Error] { ... }
Response 1500: [object Error] { ... }
Response 3500: 3
```

If we want to handle errors as regular `error` notifications in each observer we have to rethrow them after the `publishReplay()` operator for the reasons explained above.

```
var updateRequest = Observable.defer(() => makeMockHttpRequest())
    .publishReplay(1, 1000)
    .refCount()
    .take(1)
    .map(val => {
        if (val instanceof Error) {
            throw val;
```

```
        }
        return val;
    });
```

See live demo: https://jsbin.com/fabosam/5/edit?js,console (notice that we had to add also error callbacks for each observer).

```
Response 0: 1
Response 50: 1
Response 200: 1
error callback: Error: Invalid URL
error callback: Error: Invalid URL
Response 3500: 3
```

Discarding slow/outdated rest calls

A common usecase is to discard certain rest-calls, that are not needed any more after certain user-inputs. The most prominent example would be, when a user uses some search-function, makes a request, makes another request and for some reason the first request arrives after the second request and the application displays the outdated data of the old request.

This is a perfect usecase for the `switchMap`-operator.

```
searchInput$
    .switchMap(() => makeRestCall());
```

In this case the stream will `switch` to the rest-call, but only until new data in emitted on the `searchInput$`, then the stream inside the `switchMap` is discarded and a new rest-call is made. So a rest-result will only be considered if it finished before the next click.

And here is a fully fledged mocked example:

```
// some initial data-mocking
const Observable = Rx.Observable;
var counter = 1;
function mockDataFetch() {
    return Observable.of(counter++)
        .delay(500);
}

// the recipe

const searchInput$ = new Rx.Subject();
searchInput$
    .do(searchInput => console.log("Searching for " + searchInput))
    .switchMap(searchInput => mockDataFetch()
              .map(result => ({result, searchInput}))
     )
    .do(data => console.log("Received result for " + data.searchInput + ": " + data.result))
    .subscribe();

// simulating search inputs
searchInput$.next("google");
setTimeout(() => searchInput$.next("froogle"), 600);
```

```
setTimeout(() => searchInput$.next("doodle"), 800);
setTimeout(() => searchInput$.next("poodle"), 1000);
setTimeout(() => searchInput$.next("noodle"), 1600);
```

See live demo: https://jsbin.com/suzakahoro/1/edit?js,console

Sending multiple parallel HTTP requests

A very common use-case in web applications is performing multiple asynchronous (eg. HTTP) requests and gathering their results as they arrive or all of them at once (eg. in Angular2 with the HTTP service).

1. Gathering async responses one by one as they arrive

This is typically done with `mergeMap()` operator that takes a projection function that has to return an Observable. Operator `mergeMap()` internally subscribes to each Observable immediately even if the previous Observable hasn't completed yet.

```
function mockHTTPRequest(url) {
    return Observable.of(`Response from ${url}`)
      .delay(Math.random() * 1000);
}

var urls = ['url-1', 'url-2', 'url-3', 'url-4'];

Observable.from(urls)
  .mergeMap(url => mockHTTPRequest(url))
  .subscribe(val => console.log(val));
```

This prints responses to console in different order because of the random delay:

```
Response from url-3
Response from url-4
Response from url-2
Response from url-1
```

See live demo: https://jsbin.com/xaqudan/2/edit?js,console

Each response (item emitted via `next` call) is reemitted by `mergeMap()` immediately.

For our purpose of sending multiple HTTP requests it's useful to mention that `mergeMap()` can take three arguments in total:

1. The projection function that needs to return an Observable.
2. The result selector function that allows us to modify the result before emitting it further.
3. The number of concurrently subscribed Observables.

Controlling the number of parallel requests

With the third argument we can control how many parallel requests we'll handle (assuming that each Observable performing an HTTP request is "cold").

In the following example we'll run only 2 requests at the same time.

```
function mockHTTPRequest(url) {
    return Observable.of(`Response from ${url}`)
      .delay(1000);
}

let urls = ['url-1', 'url-2', 'url-3', 'url-4'];
let start = (new Date()).getTime();

Observable.from(urls)
  .mergeMap(url => mockHTTPRequest(url), undefined, 2)
  .timestamp()
  .map(stamp => [stamp.timestamp - start, stamp.value])
  .subscribe(val => console.log(val));
```

See live demo: https://jsbin.com/sojejal/4/edit?js,console

Notice that the first two requests completed after 1s while the other two after 2s.

```
[1004, "Response from url-1"]
[1010, "Response from url-2"]
[2007, "Response from url-3"]
[2012, "Response from url-4"]
```

Handling errors

If any of the source Observables fail (sends `error` notification) the `mergeMap()` resends the error further as `error`. In case we want each Observable to fail gracefully we need to use for example `catch()` operator.

```
function mockHTTPRequest(url) {
  return Observable.of(`Response from ${url}`)
    .delay(Math.random() * 1000)
    .map(value => {
      if (url === 'url-3') {
        throw new Error(`Error response from ${url}`)
      }
      return value;
    });
}

var urls = ['url-1', 'url-2', 'url-3', 'url-4'];

Observable.from(urls)
  .mergeMap(url => mockHTTPRequest(url).catch(() => Observable.empty()))
  .subscribe(val => console.log(val));
```

The response for `url-3` throws an error that is sent as `error` notification. This is later caught by `catch()` operator and replaced with `Observable.empty()` which is just a `complete` notification. For this reason this response is ignored.

The output for this example is the following:

```
Response from url-4
Response from url-1
Response from url-2
```

See live demo: https://jsbin.com/kuqumud/4/edit?js,console

2. Gathering all async responses at once

Following the preceding examples we could gather all responses with `toArray()` operator.

```
function mockHTTPRequest(url) {
    return Observable.of(`Response from ${url}`)
      .delay(Math.random() * 1000);
}

var urls = ['url-1', 'url-2', 'url-3', 'url-4'];

Observable.from(urls)
  .mergeMap(url => mockHTTPRequest(url))
  .toArray()
  .subscribe(val => console.log(val));
```

However, using `toArray()` operator has an important consequence. Whether the subscriber receives the results isn't only controlled by completing all the HTTP requests but also by completing the source Observable (`Observable.from` in our case). This means that we can't use source Observables that never complete (eg. `Observable.fromEvent`).

Another way to achieve the same result is using `Observable.forkJoin()` which takes as argument an array of Observables that we want to subscribe to and wait until all of them **emit at least one value and complete**.

```
function mockHTTPRequest(url) {
    return Observable.of(`Response from ${url}`)
      .delay(Math.random() * 1000);
}

var urls = ['url-1', 'url-2', 'url-3', 'url-4'];
var observables = urls.map(url => mockHTTPRequest(url));

Observable.forkJoin(observables)
  .subscribe(val => console.log(val));
```

This prints all responses as a single array:

```
["Response from url-1", "Response from url-2", "Response from url-3", "Response from url-4"]
```

See live demo: https://jsbin.com/fomoye/2/edit?js,console

The `Observable.forkJoin()` also takes as an optional argument a result selector function that lets us modify the final result before emitting it further:

```
Observable.forkJoin(observables, (...results) => {
    return results.length;
```

```
  })
  .subscribe(val => console.log(val));
```

This prints to console:

```
4
```

See live demo: https://jsbin.com/muwiqic/1/edit?js,console

Note that the argument for the result selector function are unpacked.

Handling errors

For error handling we can use the same approach as in the preceding example with `catch()` operator.

However, there's one important thing to be aware of. The `forkJoin()` requires every source Observable to emit at least one value. If we used `catch(() => Observable.empty())` like we did before the `forkJoin()` would never emit anything because `Observable.empty()` is just a `complete` notification.
This is why we need to use for example `Observable.of(null)` which is a `null` value followed by `complete` notification.

```
function mockHTTPRequest(url) {
  return Observable.of(`Response from ${url}`)
    .delay(Math.random() * 1000)
    .map(value => {
      if (url === 'url-3') {
        throw new Error(`Error response from ${url}`)
      }
      return value;
    });
}

var urls = ['url-1', 'url-2', 'url-3', 'url-4'];

var observables = urls.map(url => mockHTTPRequest(url).catch(() => Observable.of(null)));

Observable.forkJoin(observables)
  .subscribe(val => console.log(val));
```

See live demo: https://jsbin.com/yidiked/2/edit?js,console

This prints to console:

```
["Response from url-1", "Response from url-2", null, "Response from url-4"]
```

Notice that the error is replaced by `null`. If we used just `Observable.empty()` the `forkJoin()` would never emit anything.

Sending multiple sequential HTTP requests

Making a sequence of HTTP requests has two primary reasons:

- Requests are depending on each other (the result from one requests is required for a consecutive request).
- We want to spread server load into multiple requests.

1. Making multiple dependent requests

This can be performed using the `concatMap()` operator to transform one response to parameters required for the consecutive request.

```
function mockHTTPRequest(url) {
    return Observable.of(`Response from ${url}`)
      .delay(1000);
}

function timestamp() {
  return (new Date()).getTime() - start;
}

var start = (new Date()).getTime();

Observable.of('url-1')
  // first request
  .concatMap(url => {
    console.log(timestamp() + ': Sending request to ' + url);
    return mockHTTPRequest(url);
  })
  .do(response => console.log(timestamp() + ': ' + response))

  // second request
  .concatMap(response => {
    console.log(timestamp() + ': Sending request to ' + response);
    let newUrl = 'url-' + response.length; // create new requests url
    return mockHTTPRequest(newUrl);
  })
  .do(response => console.log(timestamp() + ': ' + response))

  // third request
  .concatMap(response => {
    console.log(timestamp() + ': Sending request to ' + response);
    let newUrl = 'url-' + response.length; // create another requests url
    return mockHTTPRequest(newUrl);
  })
  .subscribe(response => console.log(timestamp() + ': ' + response));
```

Operator `concatMap()` internally subscribes to the Observable returned the from its projection function and waits until it completes while re-emitting all its values.

This example timestamps each request and response:

```
3: Sending request to url-1
1010: Response from url-1
1011: Sending request to Response from url-1
2014: Response from url-19
2015: Sending request to Response from url-19
```

```
3017: Response from url-20
```

See live demo: https://jsbin.com/fewidiv/6/edit?js,console

Handling errors

If any of the HTTP requests fail we obviously don't want to continue because we're not able to construct the following request.

2. Making consecutive requests

In case we're not interested in the response from the previous HTTP request we can take just an array of URLs and execute them one after another.

```
function mockHTTPRequest(url) {
    return Observable.of(`Response from ${url}`)
      .delay(1000);
}

let urls = ['url-1', 'url-2', 'url-3', 'url-4'];
let start = (new Date()).getTime();

Observable.from(urls)
  .concatMap(url => mockHTTPRequest(url))
  .timestamp()
  .map(stamp => [stamp.timestamp - start, stamp.value])
  .subscribe(val => console.log(val));
```

This example prints timestamped responses:

```
[1006, "Response from url-1"]
[2012, "Response from url-2"]
[3014, "Response from url-3"]
[4016, "Response from url-4"]
```

See live demo: https://jsbin.com/kakede/3/edit?js,console

Delaying consecutive calls

We might also want to make a small delay between each request. In such case we need to append `delay()` after each `mockHTTPRequest()` call.

```
Observable.from(urls)
  .concatMap(url => {
    return mockHTTPRequest(url)
      .do(response => console.log(((new Date()).getTime() - start) + ': Sending request to ' +
url))
      .delay(500);
  })
  .timestamp()
  .map(stamp => [stamp.timestamp - start, stamp.value])
  .subscribe(val => console.log(val));
```

This prints to console the following output:

```
2024: Sending request to url-1
[2833, "Response from url-1"]
4569: Sending request to url-2
[5897, "Response from url-2"]
7880: Sending request to url-3
[8674, "Response from url-3"]
9789: Sending request to url-4
[10796, "Response from url-4"]
```

See live demo: https://jsbin.com/kakede/4/edit?js,console

Handling errors

If we simply want to ignore when any of the HTTP requests fail we have to chain `catch()` ofter each `mockHTTPRequest()`.

```
function mockHTTPRequest(url) {
  if (url == 'url-3') {
    return Observable.throw(new Error(`Request ${url} failed.`));
  } else {
    return Observable.of(`Response from ${url}`)
      .delay(1000);
  }
}

let urls = ['url-1', 'url-2', 'url-3', 'url-4'];
let start = (new Date()).getTime();

Observable.from(urls)
  .concatMap(url => mockHTTPRequest(url).catch(obs => Observable.empty()))
  .timestamp()
  .map(stamp => [stamp.timestamp - start, stamp.value])
  .subscribe(val => console.log(val));
```

This simply ignores the `url-3` call:

```
[1004, "Response from url-1"]
[2012, "Response from url-2"]
[3016, "Response from url-4"]
```

See live demo: https://jsbin.com/jowiqo/2/edit?js,console

If we didn't use the `catch()` operator the `url-3` would cause the chain to send an error notification and the last `url-4` wouldn't be executed.

See live demo: https://jsbin.com/docapim/3/edit?js,console

Rate limiting

A common problem with remote services is rate limiting. The remote service allows us to send only a limited number of requests or amount of data per time period.

In RxJS 5 a very similar functionality is provided by the `bufferTime` operator and especially if we leave the second parameter unspecified (it defines how often we want to create a new buffer. If we

leave it undefined/null it'll create a new buffer right after emitting the current one).

A typical usage of `bufferTime` will look like this:

```
bufferTime(1000, null, 5)
```

This will buffer items until one of the two conditions are met. Then it'll emit the buffer and start another one:

- the operator has been collecting items for `1000ms`
- the operator has already collected `5` items

For demonstrational purposes we can create a source Observable that emits very fast so the `bufferTime` will hit the size limit (`5`) and emit more often than once every `1000ms`:

```
const source = Observable.range(1, 25)
  .concatMap(val => Observable.of(val).delay(75));
```

Then we'll chain it with `bufferTime` and `concatMap`. The `concatMap` operator is where we force the `1000ms` delay:

```
const startTime = (new Date()).getTime();

const source = Observable.range(1, 25)
  .concatMap(val => Observable.of(val).delay(75));

source.bufferTime(1000, null, 5)
  .concatMap(buffer => Observable.of(buffer).delay(1000))
  .timestamp()
  .map(obj => {
    obj.timestamp = obj.timestamp - startTime;
    return obj;
  })
  .subscribe(obj => console.log(obj));
```

See live demo: https://jsbin.com/kotibow/3/edit?js,console

We added also `timestamp()` to see the emission times to make sure the delay is really at least `1000ms`.

Note that we didn't have to use `Observable.of(buffer)` at all. We're using it here just to manually check that the number of buffered items is correct.

From the console output we can see that the delay between two emissions is roughly `1000ms`:

```
Timestamp { value: [ 1, 2, 3, 4, 5 ], timestamp: 1475 }
Timestamp { value: [ 6, 7, 8, 9, 10 ], timestamp: 2564 }
Timestamp { value: [ 11, 12, 13, 14, 15 ], timestamp: 3567 }
Timestamp { value: [ 16, 17, 18, 19, 20 ], timestamp: 4572 }
Timestamp { value: [ 21, 22, 23, 24, 25 ], timestamp: 5573 }
Timestamp { value: [], timestamp: 6578 }
```

Now we can also test a situation where the source emits slowly so the `bufferTime` operator is going to hit the max interval condition:

```
const source = Observable.range(1, 25)
  .concatMap(val => Observable.of(val).delay(300));
```

See live demo: https://jsbin.com/tuwowan/2/edit?js,console

Then the output should start after about `2s` because it took `1s` for the `bufferTime` operator to emit and then we added the `1s` delay;

```
Timestamp { value: [ 1, 2, 3 ], timestamp: 2017 }
Timestamp { value: [ 4, 5, 6 ], timestamp: 3079 }
Timestamp { value: [ 7, 8, 9, 10 ], timestamp: 4088 }
Timestamp { value: [ 11, 12, 13 ], timestamp: 5093 }
Timestamp { value: [ 14, 15, 16 ], timestamp: 6094 }
Timestamp { value: [ 17, 18, 19, 20 ], timestamp: 7098 }
Timestamp { value: [ 21, 22, 23 ], timestamp: 8103 }
Timestamp { value: [ 24, 25 ], timestamp: 9104 }
```

If we wanted to use this approach in a real world application we'd put the remote call into the `concatMap` operator. This way we can control whether we want to force the `1s` delay between requests or responses from the remote service.

For example we can force the minimum `1s` delay between requests by using `forkJoin` in the `concatMap` callback:

```
function mockHTTPRequest(buffer) {
  return Observable.of(true).delay(Math.random() * 1500)
}

const startTime = (new Date()).getTime();
const source = Observable.range(1, 25)
  .concatMap(val => Observable.of(val).delay(75));

source.bufferTime(1000, null, 5)
  .concatMap(buffer => Observable.forkJoin(
    mockHTTPRequest(buffer),
    Observable.of(buffer).delay(1000)
  ))
  .timestamp()
  .map(obj => {
    obj.timestamp = obj.timestamp - startTime;
    return obj;
  })
  .subscribe(obj => console.log(obj));
```

See live demo: https://jsbin.com/xijaver/edit?js,console

Thanks to `forkJoin` the `concatMap` needs to wait for both Observables to complete.

On the other hand, if we wanted to force `1s` delay between responses we'd just append the `delay()` operator after `mockHTTPRequest()`:

```
.concatMap(buffer => mockHTTPRequest(buffer).delay(1000))
```

See live demo: https://jsbin.com/munopot/2/edit?js,console

Read Common recipes online: https://riptutorial.com/rxjs/topic/8247/common-recipes

Chapter 3: Operator: map / select

Syntax

- **Rx.Observable.prototype.map(selector, [thisArg])**
- **Rx.Observable.prototype.select(selector, [thisArg])**

Parameters

Parameter, Type	Details
`selector`, `Function` or `Object`	Transform function to apply to each source element or an element to yield. If selector is a function, it is called with the following information: 1. the value of the element, 2. the index of the element, 3. the Observable object being subscribed.
`[thisArg]`, `Any`	Object to use as `this` when executing the predicate.

Remarks

`map` and `select` are aliases.

They produce an observable sequence emitting one element every time the source observable emits an element.

If `selector` is not a function, its value is emitted for each source element.

If `selector` is a function, the emitted element is the result of running `selector` on the source element, and can possibly use its position.

Examples

Using an element to yield

```
const md = Rx.Observable.fromEvent(document, 'mousedown').map(true);
// `md` will emit `true` whenever the mouse is pressed
const mu = Rx.Observable.fromEvent(document, 'mouseup').map(false);
// `mu` will emit `false` whenever the mouse is depressed
```

Using a transform function

```
const source = Rx.Observable.range(1, 3)
  .map(x => x * x);
```

```
const subscription = source.subscribe(
  x => console.log(`Next: ${x}`),
  err => console.log(`Error: ${err}`),
  () => console.log(`Completed`)
);

// => Next: 1
// => Next: 4
// => Next: 9
// => Completed
```

Using a transform function and the element index

```
const source = Rx.Observable.range(1, 3)
  .map((x, idx, obs) => `Element ${x} was at position ${idx}`);

const subscription = source.subscribe(
  x => console.log(`Next: ${x}`),
  err => console.log(`Error: ${err}`),
  () => console.log(`Completed`)
);

// => Next: Element 1 was at position 0
// => Next: Element 2 was at position 1
// => Next: Element 3 was at position 2
// => Completed
```

Read Operator: map / select online: https://riptutorial.com/rxjs/topic/5419/operator--map---select

Chapter 4: Operator: PublishReplay

Examples

How does PublishReplay work

It internally creates a `ReplaySubject` and makes it `multicast` compatible. The minimal replay value of `ReplaySubject` is 1 emission. This results in the following:

- First subscription will trigger the `publishReplay(1)` to internally subscribe to the source stream and pipe all emissions through the `ReplaySubject`, effectively caching the last *n*(=1) emissions
- If a second subscription is started while the source is still active the `multicast()` will connect us to the same `replaySubject` and we will receive all next emissions until the source stream completes.
- If a subscription is started after the source is already completed the replaySubject has cached the last *n* emissions and it will only receive those before completing.

```
const source = Rx.Observable.from([1,2])
  .mergeMap(i => Rx.Observable.of('emission:'+i).delay(i * 100))
  .do(null,null,() => console.log('source stream completed'))
  .publishReplay(1)
  .refCount();

// two subscriptions which are both in time before the stream completes
source.subscribe(val => console.log(`sub1:${val}`), null, () => console.log('sub1
completed'));
source.subscribe(val => console.log(`sub2:${val}`), null, () => console.log('sub2
completed'));

// new subscription after the stream has completed already
setTimeout(() => {
  source.subscribe(val => console.log(`sub_late-to-the-party:${val}`), null, () =>
console.log('sub_late-to-the-party completed'));
}, 500);
```

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/rxjs/5.0.3/Rx.js"></script>
```

Unexpected emissions when using publishReplay

Based on a question. The following snippets Does not cache the expected emission and prevents further calls. Instead it re-subscribes to the *realSource* for every subscription.

```
var state = 5
var realSource = Rx.Observable.create(observer =>  {
  console.log("creating expensive HTTP-based emission");
  observer.next(state++);
//  observer.complete(); //absent on purpose

  return () => {
    console.log('unsubscribing from source')
```

```
  }
});


var source = realSource
.do(null, null, () => console.log('stream completed'))
.publishReplay()
.refCount();

subscription1 = source.subscribe({next: (v) => console.log('observerA: ' + v)});
subscription1.unsubscribe();

subscription2 = source.subscribe(v => console.log('observerB: ' + v));
subscription2.unsubscribe();
```

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/rxjs/5.1.0/Rx.js"></script>
```

When running this snippet we can see clearly that it is not emitting duplicate values for *Observer B* , it is in fact creating new emissions for every subscription. How come?

Every subscription is unsubscribed before the next subscription takes place. This effectively makes the refCount decrease back to zero, no multicasting is being done.

The issue resides in the fact that the `realSource` stream *does not complete*. Because we are not multicasting the next subscriber gets a fresh instance of `realSource` through the ReplaySubject and the new emissions are prepended with the previous already emitted emissions.

Read Operator: PublishReplay online: https://riptutorial.com/rxjs/topic/8987/operator--publishreplay

Chapter 5: Retry and RetryWhen Operators

Introduction

Retry and RetryWhen can be used to attempt to recover Observables that might have errors in their stream.

Syntax

1. .retry(n: number): Observable
    - n: retry will attempt the source Observable this many times.
2. .retryWhen(receives: notificationHandler, the: scheduler): Observable
    - receives: an Observable of notifications which the use can complete or error.
        - If the 'receives' Observable returns cleanly (completes) the source Observable will be reattempted.
        - If the 'receives' Observable returns an error, the source Observable is aborted.
    - scheduler: The source Observable is subscribed to this scheduler.

Examples

Retry with backoff, until success or max number of attempts reached

The following code will attempt to execute `loadFromHttp()` up to 5 times (`maxAttempts`), with each attempt delayed by as many seconds. If `maxAttempts` is surpassed, the Observable gives up.

```
// assume loadFromHttp() returns a Promise, which might fail.
Rx.Observable.from(loadFromHttp())
.retryWhen((attempts) => {
    let maxAttempts = 5;

    Rx.Observable.range(1, maxAttempts+1).zip(attempts, (i, attempt) => [i, attempt])
    .flatMap(([i, attempt]) => {
        if (i <= maxAttempts) {
            console.log(`Retrying in ${i} second(s)`);
            return Rx.Observable.timer(i * 1000);
        } else {
            throw attempt;
        }
    })
})
```

Read Retry and RetryWhen Operators online: https://riptutorial.com/rxjs/topic/9026/retry-and-retrywhen-operators

Chapter 6: Scheduler

Examples

Using a TestScheduler to advance time manually

Most Rx operators take an optional scheduler on which to schedule their future iterations. If not supplied they will use their default configured scheduler. Supplying a scheduler can be useful for testing purposes in which we like to talk about virtual time instead of real time for speed of test execution.

```
const scheduler = new Rx.TestScheduler();
scheduler.stop();
Rx.Observable.interval(100, scheduler)
  .do(i => console.log(i))
  .subscribe();
scheduler.advanceBy(10 * 100);
```

Read Scheduler online: https://riptutorial.com/rxjs/topic/7991/scheduler

Chapter 7: Subject

Introduction

Subjects are classes that behave as Observables and observers at the same time.

http://reactivex.io/documentation/subject.html

Examples

Subject and its internal state

In Rx Subjects have internal states that can control their behavior.

A common use-case form Subject is subscribing it to multiple Observables. The following example creates two different Observables and subscribes a Subject to both of them. Then it tries to print all values that went through:

```
let subject = new Subject();
subject.subscribe(val => console.log(val));

Observable.range(1, 5).subscribe(subject);
Observable.from(['a', 'b', 'c']).subscribe(subject);
```

See live demo: https://jsbin.com/pesumup/2/edit?js,console

This example just prints numbers `1 - 5` and didn't print any of the characters `a`, `b`, `c`.

```
1
2
3
4
5
```

The question is what happened? The problem here is the internal state of the Subject instance when it received the `complete` notification. When a Subject receives an `error` or `complete` notifications it marks itself as stopped and will never emit any other signal.

It needs to be this way because Subjects are basically Observables and Observables can only emit one `complete` or `error` notification at the end of the stream but never both.

The problem with the example above is that the first Observable `Observable.range()` emits also the `complete` notification which is then received by the Subject and therefore it doesn't reemit any value when subscribed to the second Observable.

We can see that the Subject really receives the `complete` notification by setting also the complete callback.

```
subject.subscribe(val => console.log(val), null, () => console.log('complete'));
```

The output is the same just at the end it also prints `complete`.

```
1
2
3
4
5
complete
```

So if we don't want the Subject to receive the `complete` notification, we can just manually send the `next` signals. This means instead of subscribing the Subject directly we'll subscribe a callback that calls the `next()` method on the Subject:

```
Observable.range(1, 5).subscribe(val => subject.next(val));
Observable.from(['a', 'b', 'c']).subscribe(val => subject.next(val));
```

See live demo: https://jsbin.com/funeka/1/edit?js,console

```
1
2
3
4
5
a
b
c
```

Note that this exact same principle applies everywhere where we use Subjects.

For example operators such as `publish()`, `share()` and all their variants that use the same instance of Subject under the hood are affected by this.

Read Subject online: https://riptutorial.com/rxjs/topic/9518/subject

Credits

S. No	Chapters	Contributors
1	Getting started with rxjs	bloodyKnuckles, Community, gsc, martin, Ptival
2	Common recipes	gsc, martin, olsn, vek
3	Operator: map / select	Ptival
4	Operator: PublishReplay	Mark van Straten
5	Retry and RetryWhen Operators	JBCP
6	Scheduler	Mark van Straten
7	Subject	martin