Power of RxJS when using exponential backoff

Alex Okrushko Follow
Jun 5, 2018 · 5 min read

Most of the modern-day Angular web apps make Ajax requests to the servers. These requests involve multiple network components (such as routers, switches, etc) as well as servers' state and everything has to go just right for them to succeed. However, sometimes it doesn't.

To handle such circumstances web apps typically implement retry logic that retries requests until they go through or until the maximum number of requests is reached. In most cases, simple retries are good enough to achieve the goal, but sometimes more advanced approach is needed.

What's exponential backoff?

Exponential backoff is an algorithm that *uses exponentially longer delays between retries*. In this article, I'll dive deeper into **two custom RxJS operators** (both are part of `backoff-rxjs` package) that use exponential backoff and the use cases they cover:

- `retryBackoff`, operator that retries on errors

- `intervalBackoff`, operator that emits sequential numbers

Exponential function

I've used the term *exponential* a few times already, but what does it mean? In mathematics, it's a function of the following form:

$$f(x) = b^x$$

In our case, as new values are emitted (x in the function above) the longer the delay between them will be. In code it translates into the following method:

```
function calculateDelay(iteration, initialInterval) {
 return Math.pow(2, iteration) * initialInterval;
}
```

With iterations starting from 0 and provided initial interval of 1000 milliseconds the emitted values would be 1000, 2000, 4000, 8000...

With that out of our way, let's get into our first use case.

• • •

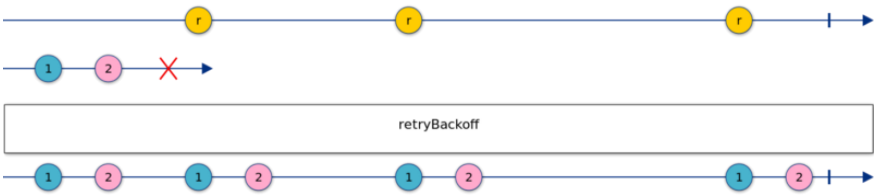
1. retryBackoff

One of the most frequent use cases for exponential backoff is to retry on error. A good example would be Google Cloud Storage (GCS) which requires this strategy to be used when retrying failed requests.

Before working on `backoff-rxjs` I found a few examples of exponential backoff retries in some gists or this stackoverflow answer, but none were flexible enough for my needs; thus I created `retryBackoff`.

`retryBackoff` takes either a number as initial delay or a `RetryBackoffConfig` for more configurations. RxJS uses marble

diagrams to visualize how operator works, so here is one for our operator.



Notice how `retryBackoff` here behaves similarly to `retry` operator and can be as simple as:

```
message$ = of('Call me!').pipe(
    switchMap(() => this.service.callBackend()),
    retryBackoff(1000),
);
```

RetryBackoffConfig

When more customization is required `retryBackoff` operator takes `RetryBackoffConfig` that has the following shape:

```

export interface RetryBackoffConfig {
  initialInterval: number;
  maxRetries?: number;
  maxInterval?: number;
  shouldRetry?: (error: any) => boolean;
```

If, for example, we want to limit the number of retries up to twelve, our call would look like this:

```
message$ = of('Call me!').pipe(
    switchMap(() => this.service.callBackend()),
    retryBackoff({
        initialInterval: 100,
        maxRetries: 12,
    })
```

Let's look into the properties of `RetryBackoffConfig`

- `initialInterval` —initial delay that is also used to calculate all the rest of the delays; this is the only require property
- `maxRetries` —the maximum number of retries
- `maxInterval` —the maximum delay between retries
- `shouldRetry` —function that lets you analyze the error and decide whether to continue retrying (return `true`) or stop retrying (return `false`)
- `backoffDelay` —function for calculating custom delays

The last two functions (`shouldRetry` and `backoffDelay`) I think deserve a bit more info.

shouldRetry function

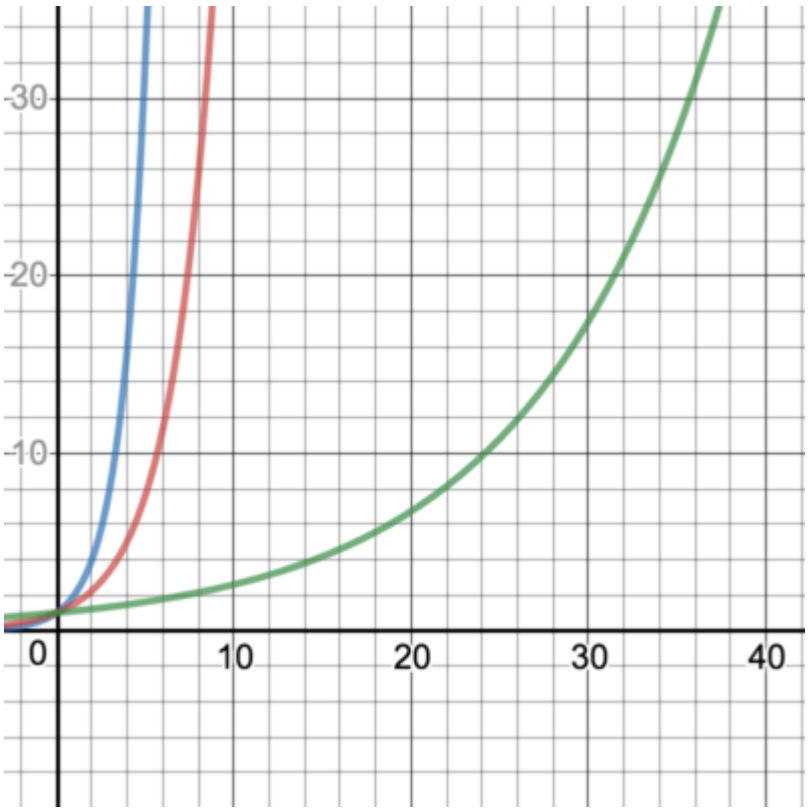
Sometimes when we get onto particular error we would like to stop retrying, for example if the return status is 404, there is little change that it will ever succeed.

```
1  // Determine if the error matches our expected type
2  // http://www.typescriptlang.org/docs/handbook/advance
3  function isHttpError(error: {}): error is HttpError {
4    // This is a type guard for interface
5    // if HttpError was a class we would use instanceof
6    return (error as HttpError).status !== undefined;
7  }
8
9  message$ = of('Call me!').pipe(
10     tap(console.log),
11     switchMap(() => this.service.callBackend()),
12     retryBackoff({
13       initialInterval: INIT_INTERVAL_MS,
14       maxInterval: MAX_INTERVAL_MS,
15       shouldRetry: (error) => {
16         // error could be anything, including HttpErro
17         // we want to handle from sevice.callBackend()
18         if (isHttpError(error)) {
```

backoffDelay function

By default delays will be doubled between each interval, however sometimes smoother backoff is needed. Through `backoffDelay` property we can provide a custom delay calculation function, e.g.:

`backoffDelay: (iteration, initialInterval) => Math.pow(1.5, iteration) * initialInterval`,
or even slower increase of the delays
`backoffDelay: (iteration, initialInterval) => Math.pow(1.1, iteration) * initialInterval`



blue: y = 2^x, red: y = 1.5^x, green: y = 1.1^x

Demo

The example of the full app can be found at StackBlitz.

exponential-backoff-retries-example—StackBlitz

Starter project for Angular apps that exports to the Angular CLI

stackblitz.com

• • •

2. intervalBackoff

Have you ever wondered what your app is doing while you are sleeping? Among many tabs that are kept open is it still working hard querying your servers, using precious resources?

The **second use case** of exponential backoff is to reduce the frequency of the requests by exponentially increasing each delay between requests. This is handy technique may be applied when the app detects that there is no user activity—for example, no mouse movement.

Let's take a look at the following piece of code.

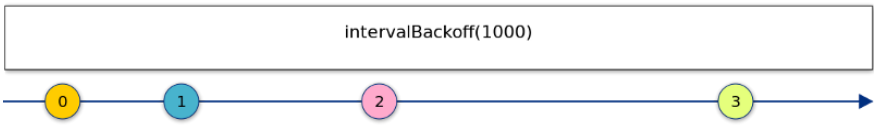
```
import {fromEvent} from 'rxjs';
import {sampleTime, startWith, switchMap} from 'rxjs/c
import {intervalBackoff} from 'backoff-rxjs';
import {service} from './service';

const newData$ = fromEvent(document, 'mousemove').pipe

    // There could be many mousemoves, we'd want to sa
    // with certain frequency
    sampleTime(1000),

    // Start immediately
    startWith(null),
```

Now let's break it down what's happening here:

- *mousemove* event is tracked on the `document` and use it as an indicator of user activity
- It is triggered very frequently when the mouse is moved, so we use `sampleTime` as a filter of those events
- `sampleTime` emits the first value only once the time specified expires. If we need to make the first call immediately (and in most cases we need it) then `startWith` helps us to do that
- now we are at the `intervalBackoff`, which is a pipeable operator that works similar to `interval`, however, instead of using the same delay between emissions it doubles the delay after each one
- Once `intervalBackoff` emits the value we do the service call

Note, that every time the *mousemove* event is detected it resets the `intervalBackoff`.

Here is the marble diagram for `intervalBackoff`:



Similarly to the `retryBackoff`, `intervalBackoff` is also configurable and can take more than just initial delay.

```
export interface IntervalBackoffConfig {
  initialInterval: number;
  maxInterval?: number;
  backoffDelay?: (iteration: number, initialInterval: r
```

Demo

Example of the app using `intervalBackoff`:

exponential-backoff-interval-example—StackBlitz

Starter project for Angular apps that exports to the Angular CLI

stackblitz.com

• • •

Summary

Exponential backoff is a very useful strategy and has at least two common use cases: **interval backoff** and **retry backoff**. `backoff-rxjs` package provides pipeable operators for both cases, and they are just a combination of existing RxJS operators.

Sources: https://github.com/alex-okrushko/backoff-rxjs

• • •

Special thanks to **Ben Lesh**, **Max NgWizard K** and **Nicholas Jamieson** for reviewing the article/operators and providing valuable feedback.

I'm also very curious about readers' feedback (maybe there is another use case for exponential backoff that I didn't cover?), or if you have questions or comments please ask them 👇

If you want to chat more, you can find me on twitter @AlexOkrushko. My DMs are open.

. . .

Thanks for reading! If you liked this article, hit that clap button below 👏. It means a lot to me and it helps other people see the story. For more advanced articles you can follow me on Twitter or on Medium.