Photo by Artem Bali on Unsplash

Top 10 ways to use Interceptors in Angular

Find out which interceptor superpowers you can start using today

Michael Karén Follow
Mar 5 · 9 min read

Just like Batman develops his gadgets we can use interceptors to gain superpowers.

There are many ways to use an interceptor, and I'm sure most of us have only scratched the surface. In this article, I will present my ten

favorite ways to use interceptors in Angular.

I have kept the examples as short as possible. And I'm hoping they will inspire you to think of new ways to use interceptors. This article is not about teaching interceptors as there are already so many good articles out there. But let's start with a little bit of basic knowledge before we start the count down.

Interceptors 101

HttpInterceptor was introduced with Angular 4.3. It provides a way to intercept HTTP requests and responses to transform or handle them before passing them along.

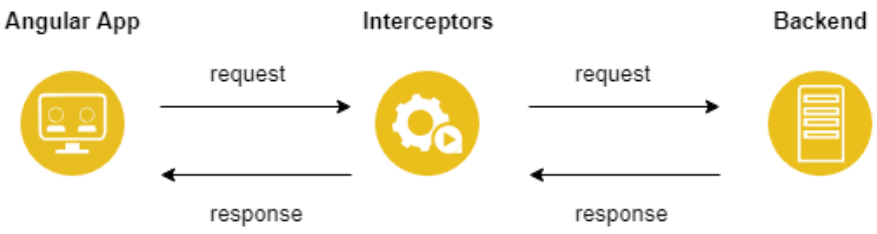
> *Although interceptors are capable of mutating requests and responses, the* `HttpRequest` *and* `HttpResponse` *instance properties are* `read-only`*, rendering them largely immutable.—Angular Docs*

This is because we might want to retry a request if it does not succeed at first. And immutability ensures that the interceptor chain can re-process the same request multiple times.

You can use multiple interceptors but keep this in mind:

> *Angular applies interceptors in the order that you provide them. If you provide interceptors A, then B, then C, requests will flow in A->B->C and responses will flow out C->B->A.*
> *You cannot change the order or remove interceptors later. If you need to enable and disable an interceptor dynamically, you'll have to build that capability into the interceptor itself.—Angular Docs*

In the example app, we have all the interceptors provided, but we only use one at a time. This is done by checking the path. If it is not the request we are looking for, we can pass it on to the next interceptor with `next.handle(req).`



Another nice thing about interceptors is that they can **process the request and response together.** This gives us some nice possibilities as we will see.

For more in-depth knowledge check out this excellent article by Max Koretskyi aka Wizard:

- Insider's guide into interceptors and HttpClient mechanics in Angular

I'm using the website JSONPlaceholder in the examples for the HTTP calls. If you want to follow along in the code you can find it here:

Example code on GitHub. 📜
Run the code on StackBlitz. 🤸

And now let's start the count down!

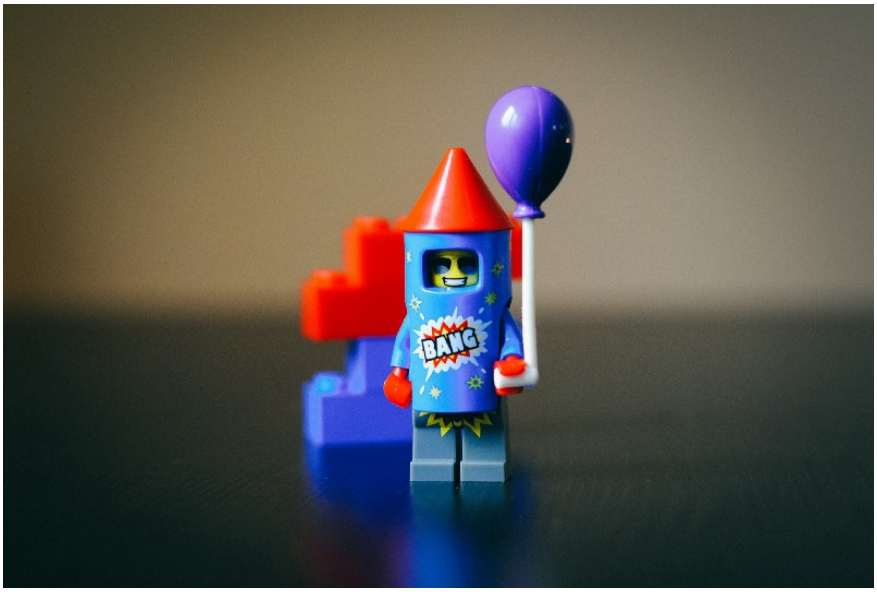


Photo by Hello I'm Nik on Unsplash

• • •

10. URL

Manipulating the URL. Sounds a bit risky when I say it out loud but let's see how easily we can do it in an interceptor.

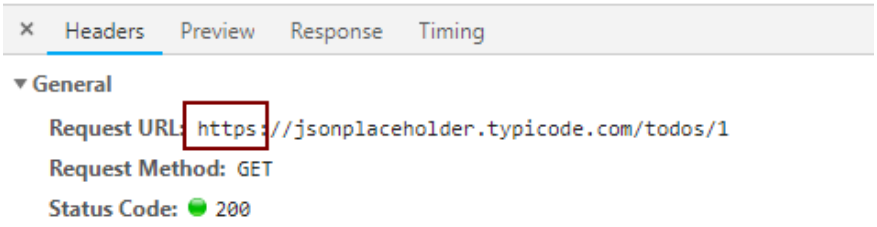
We could, for example, want to change HTTP to HTTPS.

It's as easy as cloning the request and replacing http:// with https:// at the same time. Then we send the cloned, HTTPS request to the next handler.

```
// clone request and replace 'http://' with 'https://'
const httpsReq = req.clone({
  url: req.url.replace("http://", "https://")
});

```

In the example, we set the URL with HTTP, but when we check the request, we can see that it changed to HTTPS.

```
const url = "http://jsonplaceholder.typicode.com/todos/1";
this.response = this.http.get(url);
```



Automagic https, why is this not higher up? Well, normally you would set up these things on your web server. Or if you want to switch between HTTP and HTTPS in development you could use the CLI:

```
ng serve -ssl
```

Similarly, you could change a bit more of the URL and call it an API prefix interceptor:

```
req.clone({
  url: environment.serverUrl + request.url
});
```

Or you could again do it with the CLI:

```
ng serve — serve-path=<path> — base-href <path>/
```

Thanks to David Herges for the CLI tips! 🙏

• • •

9. Loader

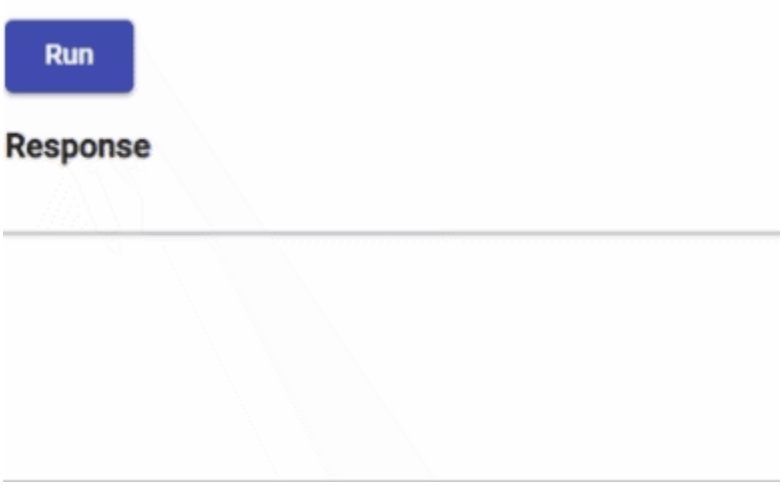
Everyone wants to see the spinning wheel of fortune when we are waiting for a response. What if I said we could set it up centrally in an interceptor so that we show a loader whenever there are active requests.

For this, we can use a **loader service** that has a show and a hide function. Before we handle the request, we call the **show** method and through finalize we can **hide** the loader when done.

```
1  const loaderService = this.injector.get(LoaderService);
2
3  loaderService.show();
4
5  return next.handle(req).pipe(
6    delay(5000),
```

This example is simplified, and in a real solution, we should take into account that there could be multiple HTTP calls intercepted. This could be solved by having a counter for requests (+1) and responses (-1).

Also, I added a delay so that we have time to see the loader.



Having a global loader sounds like a great idea so why is this one not higher up on the list? Maybe it could be for certain applications, but you might want to have more specificity for your loaders especially if you are loading more than one thing at the same time.

I'll leave you with something to ponder. What would happen if you use a switchMap that cancels the request?

• • •

8. Converting

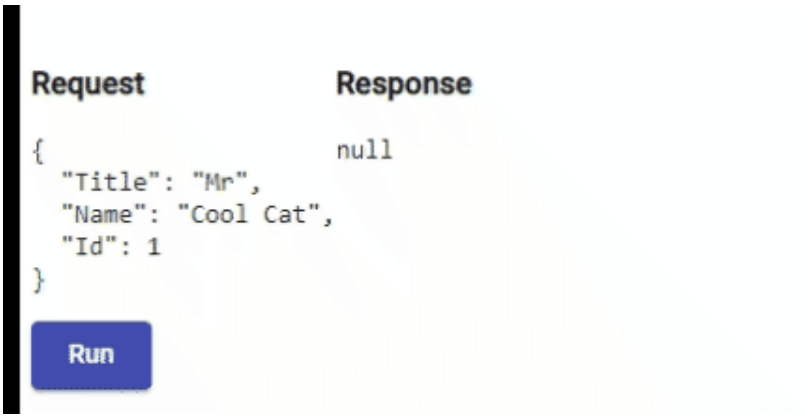
When the API returns a format we do not agree with, we can use an interceptor to format it the way we like it.

This could be converting from XML to JSON or like in this example property names from PascalCase to camelCase. If the backend doesn't care about JSON/JS conventions we can use an interceptor to rename all the property names to camelCase.

Check if there is an npm package that can do the heavy lifting for you. I'm using mapKeys and camelCase from lodash in the example.

```
return next.handle(req).pipe(
  map((event: HttpEvent<any>) => {
    if (event instanceof HttpResponse) {
      let camelCaseObject = mapKeys(event.body, (v, k)
      const modEvent = event.clone({ body: camelCaseOb

      return modEvent;
```

This one is also something that the backend should be doing so I would typically not do this. But add it to your arsenal so that you have it ready when you need it.



. . .

7. Headers

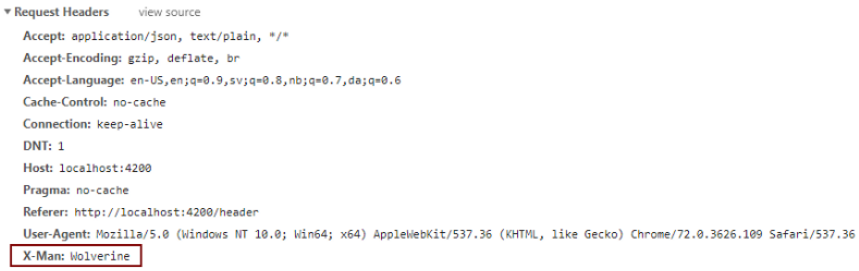
We can do a lot by manipulating headers. Some things are:

- Authentication/authorization
- Caching behavior; for example, If-Modified-Since
- XSRF protection

We can easily add headers to the request in the interceptor.

```
const modified = req.clone({
  setHeaders: { "X-Man": "Wolverine" }
});

return next.handle(modified);
```

And we can see that it gets added to the request headers in dev tools.



Angular uses interceptors for protection against Cross-Site Request Forgery (XSRF). It does this by reading the **XSRF-TOKEN** from a cookie and setting it as the **X-XSRF-TOKEN** HTTP header. Since only code that runs on your domain could read the cookie, the backend can be sure that the HTTP request came from your client application and not an attacker.

So as you can see it is straightforward to manipulate headers in the interceptor. We will see at least one more example of header manipulation further ahead.

• • •

6. Notifications

Here we have many different cases where we could show messages. In my example, I show “Object created” every time we get a 201 created status back from the server.

```
1  return next.handle(req).pipe(
2    tap((event: HttpEvent<any>) => {
3      if (event instanceof HttpResponse && event.status =
4        this.toastr.success("Object created.");
5      }
6    })
```

We could check the type of object to show “Type created” instead. Or we could have a more specific message by wrapping our data in an object together with a message:

```
{
  data: T,
  message: string
}
```

We could also show notifications from the interceptor when errors occur.

• • •

5. Errors

There are two use cases for errors that we can implement in the interceptor.

First, we can **retry the HTTP call.** For example, network interruptions are frequent in mobile scenarios, and trying again may produce a successful result. Things to consider here are how many times to retry before giving up. And should we wait before retrying or do it immediately?

For this, we use the retry operator from RxJS to resubscribe to the observable. Re-subscribing to the result of an HttpClient method call has the effect of reissuing the HTTP request.

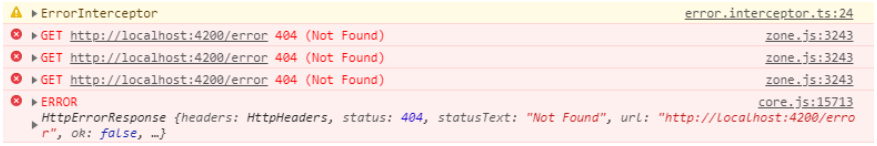
More advanced examples of this sort of behavior:

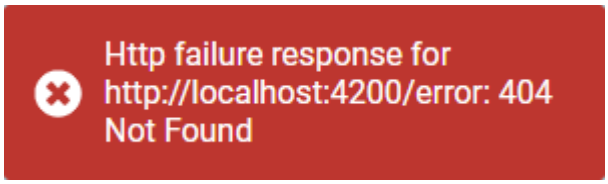
- Retry an observable sequence on error based on custom criteria
- Power of RxJS when using exponential backoff

Secondly, we can check the **status of the exception.** And depending on the status, we can decide what we should do.

```
return next.handle(req).pipe(
  retry(2),
  catchError((error: HttpErrorResponse) => {
    if (error.status !== 401) {
      // 401 handled in auth.interceptor
      this.toastr.error(error.message);
    }
```

In this example, we retry twice before checking the error status. And if the status is not 401, we show the error as a popup (toastr). And all errors are then re-thrown for further handling.





For more knowledge on error handling you can read my earlier article on this:

- Expecting the Unexpected—Best practices for Error handling in Angular

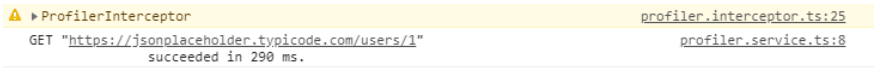
. . .

4. Profiling

Because interceptors can process the request and response *together*, they can do things like time and log an entire HTTP operation. So we can capture the time of the request and the response and log the outcome with the elapsed time.

```
const started = Date.now();
let ok: string;

return next.handle(req).pipe(
  tap(
    (event: HttpEvent<any>) => ok = event instanceof H
    (error: HttpErrorResponse) => ok = "failed"
  ),
  // Log when response observable either completes or
  finalize(() => {
    const elapsed = Date.now() - started;
```

There are a lot of possibilities here, and we could log the profiles to the database to get some statistics for example. In the example, we log to the console.



• • •

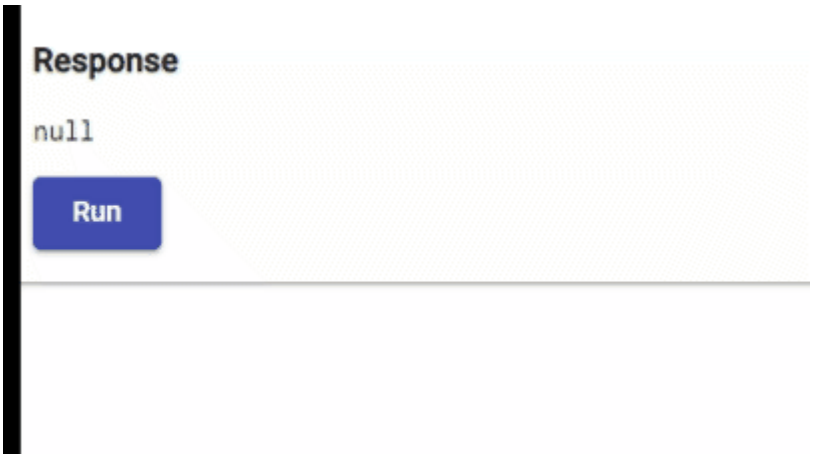
3. Fake backend

A mock or fake backend can be used in development when you do not have a backend yet. You can also use it for code hosted in StackBlitz.

We mock the response depending on the request. And then return an observable of HttpResponse.

```
const body = {
  firstName: "Mock",
  lastName: "Faker"
};
```

```
return of(new HttpResponse(
  { status: 200, body: body }
));
```



• • •

2. Caching

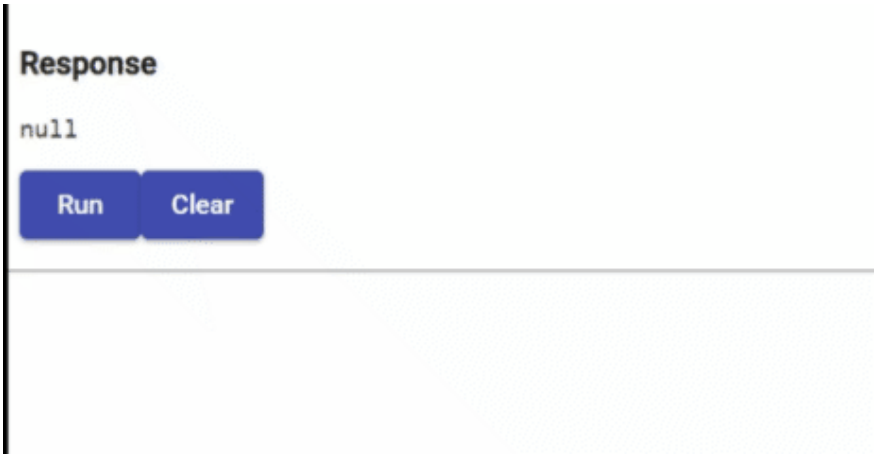
Since interceptors can handle requests by themselves, without forwarding to `next.handle()`, we can use it for caching requests.

What we do is use the URL a the key in our cache that is just a key-value map. And if we find a response in the map, we can return an observable of that response, by-passing the `next` handler.

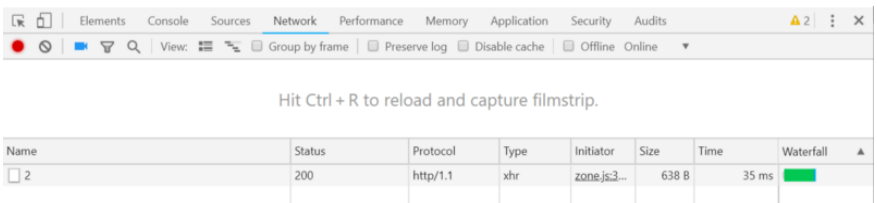
This increases performance since you don't have to go all the way to the backend when you already have the response cached.

```
import { Injectable } from '@angular/core';
import { HttpEvent, HttpRequest, HttpHandler, HttpInte
import { Observable, of } from 'rxjs';
import { tap, shareReplay } from 'rxjs/operators';

@Injectable()
export class CacheInterceptor implements HttpIntercept
  private cache = new Map<string, any>();

  intercept(request: HttpRequest<any>, next: HttpHandl
    if (request.method !== 'GET') {
      return next.handle(request);
    }

    const cachedResponse = this.cache.get(request.url)
    if (cachedResponse) {
      return of(cachedResponse);
    }
```

If we run the request, clear the response and then run again we will be using the cache.



If we check the network tab in DevTools, we can see that we only make the request once.



You will introduce some more complexity since you need to invalidate your cache if the data is updated. But let's not worry about that now. Caching is fantastic when it works!

For more knowledge on caching read this excellent article by Dominic E.:

- Advanced caching with RxJS

• • •

1. Authentication

And first on the list is authentication! It is just so fundamental for many applications that we have a proper authentication system in place. This is one of the most common use cases for interceptors and for a good reason. It fits right in!

There are several things connected to authentication we can do:

1. Add bearer token
2. Refresh Token
3. Redirect to the login page

We should also have some filtering for when we send the bearer token. If we don't have a token yet, then we are probably logging in and should not add the token. And if we are doing calls to other domains, then we would also not want to add the token. For example, if we send errors into Slack.

This is also a bit more complex than the other interceptors. Here is an example of how it can look with some explaining comments:

```
import { Injectable } from "@angular/core";
import {
  HttpEvent, HttpInterceptor, HttpHandler,
  HttpRequest, HttpErrorResponse
} from "@angular/common/http";
import { throwError, Observable, BehaviorSubject, of }
import { catchError, filter, take, switchMap } from "r

@Injectable()
export class AuthInterceptor implements HttpIntercepto
  private AUTH_HEADER = "Authorization";
  private token = "secrettoken";
  private refreshTokenInProgress = false;
  private refreshTokenSubject: BehaviorSubject<any> =

  intercept(req: HttpRequest<any>, next: HttpHandler):

    if (!req.headers.has('Content-Type')) {
      req = req.clone({
        headers: req.headers.set('Content-Type', 'appl
      });
    }

    req = this.addAuthenticationToken(req);

    return next.handle(req).pipe(
      catchError((error: HttpErrorResponse) => {
        if (error && error.status === 401) {
          // 401 errors are most likely going to be be
          if (this.refreshTokenInProgress) {
            // If refreshTokenInProgress is true, we w
            // which means the new token is ready and
            return this.refreshTokenSubject.pipe(
              filter(result => result !== null),
              take(1),
              switchMap(() => next.handle(this.addAuth
            );
          } else {
            this.refreshTokenInProgress = true;

            // Set the refreshTokenSubject to null so
            this.refreshTokenSubject.next(null);

            return this.refreshAccessToken().pipe(
```

```
45                switchMap((success: boolean) => {
46                  this.refreshTokenSubject.next(success)
47                  return next.handle(this.addAuthenticat
48                }),
49                // When the call to refreshToken complet
50                // for the next time the token needs to
```

Winner winner chicken dinner! 🚀

• • •

Conclusion

Interceptors were a great addition in Angular 4.3 and had many excellent use cases as we have seen here. Now just let your creativity flow, and I'm sure you can come up with something spectacular!

Remember you can be like Batman by using interceptors!

Photo by Zhen Hu on Unsplash

Example code on GitHub. 📜
Run the code on StackBlitz. 🤸

Thanks to all from Angular In Depth that contributed with ideas and helped with editing. Hoping I don't forget anyone I would like to thank Max Koretskyi aka Wizard, Tim Deschryver, Alex Okrushko, Alexander Poshtaruk, Lars Gyrup Brink Nielsen, Nacho Vazquez Calleja, thekiba & Alexey Zuev!

Resources

- Insider's guide into interceptors and HttpClient mechanics in Angular by Max Koretskyi, aka Wizard
- Angular Docs
- Bits and pieces from many other great articles out there.