

# Communicating with backend services using HTTP



## Contents >

Setup for server communication

Requesting data from a server

Requesting a typed response

...

Most front-end applications need to communicate with a server over the HTTP protocol, in order to download or upload data and access other back-end services. Angular provides a simplified client HTTP API for Angular applications, the `HttpClient` service class in `@angular/common/http`.

The HTTP client service offers the following major features.

- The ability to request [typed response objects](#).
- Streamlined [error handling](#).
- [Testability](#) features.
- Request and response [interception](#).

## Prerequisites

Before working with the `HttpClientModule`, you should have a basic understanding of the following:

- TypeScript programming
- Usage of the HTTP protocol
- Angular app-design fundamentals, as described in [Angular Concepts](#)
- Observable techniques and operators. See the [Observables](#) guide.

## Setup for server communication

Before you can use `HttpClient`, you need to import the Angular `HttpClientModule`. Most apps do so in the root `AppModule`.

app/app.module.ts (excerpt)

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  imports: [
```



```

    BrowserModule,
    // import HttpClientModule after BrowserModule.
    HttpClientModule,
  ],
  declarations: [
    AppComponent,
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule {}

```

You can then inject the `HttpClient` service as a dependency of an application class, as shown in the following `ConfigService` example.

#### app/config/config.service.ts (excerpt)

```

import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Injectable()
export class ConfigService {
  constructor(private http: HttpClient) { }
}

```

The `HttpClient` service makes use of [observables](#) for all transactions. You must import the RxJS observable and operator symbols that appear in the example snippets. These `ConfigService` imports are typical.

#### app/config/config.service.ts (RxJS imports)

```

import { Observable, throwError } from 'rxjs';
import { catchError, retry } from 'rxjs/operators';

```

You can run the [live example](#) / [download example](#) that accompanies this guide.

The sample app does not require a data server. It relies on the [Angular \*in-memory-web-api\*](#), which replaces the `HttpClient` module's `HttpBackend`. The replacement service simulates the behavior of a REST-like backend.

Look at the `AppModule` *imports* to see how it is configured.

## Requesting data from a server

Use the `HttpClient.get()` method to fetch data from a server. The asynchronous method sends an HTTP request, and returns an `Observable` that emits the requested data when the response is received. The return type varies based on the `observe` and `responseType` values that you pass to the call.

The `get()` method takes two arguments; the endpoint URL from which to fetch, and an *options* object that you can use to configure the request.

```
options: {
  headers?: HttpHeaders | {[header: string]: string | string[]},
  observe?: 'body' | 'events' | 'response',
  params?: HttpParams | {[param: string]: string | string[]},
  reportProgress?: boolean,
  responseType?: 'arraybuffer' | 'blob' | 'json' | 'text',
  withCredentials?: boolean,
}
```

Important options include the *observe* and *responseType* properties.

- The *observe* option specifies how much of the response to return.
- The *responseType* option specifies the format in which to return data.

You can use the `options` object to configure various other aspects of an outgoing request. In [Adding headers](#), for example, the service set the default headers using the `headers` option property.

Use the `params` property to configure a request with [HTTP URL parameters](#), and the `reportProgress` option to [listen for progress events](#) when transferring large amounts of data.

Applications often request JSON data from a server. In the `ConfigService` example, the app needs a configuration file on the server, `config.json`, that specifies resource URLs.

assets/config.json

```
{
  "heroesUrl": "api/heroes",
  "textfile": "assets/textfile.txt"
}
```

To fetch this kind of data, the `get()` call needs the following options: `{observe: 'body', responseType: 'json'}`. These are the default values for those options, so the following examples do not pass the options object. Later sections show some of the additional option possibilities.

The example conforms to the best practices for creating scalable solutions by defining a re-usable [injectable service](#) to perform the data-handling functionality. In addition to fetching data, the service can post-process the data, add error handling, and add retry logic.

The `ConfigService` fetches this file using the `HttpClient.get()` method.

app/config/config.service.ts (getConfig v.1)

```
configUrl = 'assets/config.json';
```

```
getConfig() {  
    return this.http.get(this.configUrl);  
}
```

The `ConfigComponent` injects the `ConfigService` and calls the `getConfig` service method.

Because the service method returns an `Observable` of configuration data, the component *subscribes* to the method's return value. The subscription callback performs minimal post-processing. It copies the data fields into the component's `config` object, which is data-bound in the component template for display.

#### app/config/config.component.ts (showConfig v.1)

```
showConfig() {  
    this.configService.getConfig()  
        .subscribe((data: Config) => this.config = {  
            heroesUrl: data.heroesUrl,  
            textfile: data.textfile  
        });  
}
```

## Requesting a typed response

You can structure your `HttpClient` request to declare the type of the response object, to make consuming the output easier and more obvious. Specifying the response type acts as a type assertion at compile time.

Specifying the response type is a declaration to TypeScript that it should treat your response as being of the given type. This is a build-time check and doesn't guarantee that the server will actually respond with an object of this type. It is up to the server to ensure that the type specified by the server API is returned.

To specify the response object type, first define an interface with the required properties. Use an interface rather than a class, because the response is a plain object that cannot be automatically converted to an instance of a class.

```
export interface Config {  
    heroesUrl: string;  
    textfile: string;  
}
```

Next, specify that interface as the `HttpClient.get()` call's type parameter in the service.

#### app/config/config.service.ts (getConfig v.2)

```
getConfig() {  
    // now returns an Observable of Config  
    return this.http.get<Config>(this.configUrl);  
}
```

When you pass an interface as a type parameter to the `HttpClient.get()` method, you can use the [RxJS map operator](#) to transform the response data as needed by the UI. You can then pass the transformed data to the [async pipe](#).

The callback in the updated component method receives a typed data object, which is easier and safer to consume:

#### app/config/config.component.ts (showConfig v.2)

```
config: Config;  
  
showConfig() {  
    this.configService.getConfig()  
        // clone the data object, using its known Config shape  
        .subscribe((data: Config) => this.config = { ...data });  
}
```

To access properties that are defined in an interface, you must explicitly convert the plain object you get from the JSON to the required response type. For example, the following `subscribe` callback receives data as an `Object`, and then type-casts it in order to access the properties.

```
.subscribe(data => this.config = {  
    heroesUrl: (data as any).heroesUrl,  
    textfile: (data as any).textfile,  
});
```

#### \*OBSERVE\* AND \*RESPONSE\* TYPES

The types of the observe and response options are *string unions*, rather than plain strings.

```
options: {  
    ...  
    observe?: 'body' | 'events' | 'response',  
    ...  
    responseType?: 'arraybuffer' | 'blob' | 'json' | 'text',  
}
```

```
...
}
```

This can cause confusion. For example:

```
// this works
client.get('/foo', {responseType: 'text'})

// but this does NOT work
const options = {
  responseType: 'text',
};
client.get('/foo', options)
```



In the second case, TypeScript infers the type of `options` to be `{responseType: string}`. The type is too wide to pass to `HttpClient.get` which is expecting the type of `responseType` to be one of the *specific* strings. `HttpClient` is typed explicitly this way so that the compiler can report the correct return type based on the options you provided.

Use `as const` to let TypeScript know that you really do mean to use a constant string type:

```
const options = {
  responseType: 'text' as const,
};
client.get('/foo', options);
```



## Reading the full response

In the previous example, the call to `HttpClient.get()` did not specify any options. By default, it returned the JSON data contained in the response body.

You might need more information about the transaction than is contained in the response body. Sometimes servers return special headers or status codes to indicate certain conditions that are important to the application workflow.

Tell `HttpClient` that you want the full response with the `observe` option of the `get()` method:

```
getConfigResponse(): Observable<HttpResponse<Config>> {
  return this.http.get<Config>(
```



```
    this.configUrl, { observe: 'response' });  
  }  
}
```

Now `HttpClient.get()` returns an `Observable` of type `HttpResponse` rather than just the JSON data contained in the body.

The component's `showConfigResponse()` method displays the response headers as well as the configuration:

#### app/config/config.component.ts (showConfigResponse)

```
showConfigResponse() {  
  this.configService.getConfigResponse()  
    // resp is of type `HttpResponse<Config>`  
    .subscribe(resp => {  
      // display its headers  
      const keys = resp.headers.keys();  
      this.headers = keys.map(key =>  
        `${key}: ${resp.headers.get(key)}`);  
  
      // access the body directly, which is typed as `Config`.  
      this.config = { ... resp.body };  
    });  
}
```

As you can see, the response object has a `body` property of the correct type.

## Making a JSONP request

Apps can use the `HttpClient` to make [JSONP](#) requests across domains when a server doesn't support [CORS protocol](#).

Angular JSONP requests return an `Observable`. Follow the pattern for subscribing to observables and use the `RxJS` `map` operator to transform the response before using the [async pipe](#) to manage the results.

In Angular, use JSONP by including `HttpClientJsonpModule` in the `NgModule` imports. In the following example, the `searchHeroes()` method uses a JSONP request to query for heroes whose names contain the search term.

```
/* GET heroes whose name contains search term */  
searchHeroes(term: string): Observable {  
  term = term.trim();  
  
  const heroesURL = `${this.heroesURL}?${term}`;  
  return this.http.jsonp(heroesUrl, 'callback').pipe(  
    catchError(this.handleError('searchHeroes', [])) // then handle the error  
  );  
}
```

This request passes the `heroesURL` as the first parameter and the callback function name as the second parameter. The response is wrapped in the callback function, which takes the observables returned by the `JSONP` method and pipes them through to the error handler.

## Requesting non-JSON data

Not all APIs return JSON data. In this next example, a `DownloaderService` method reads a text file from the server and logs the file contents, before returning those contents to the caller as an `Observable<string>`.

app/downloader/downloader.service.ts (getTextFile)

```
getTextFile(filename: string) {  
  // The Observable returned by get() is of type Observable<string>  
  // because a text response was specified.  
  // There's no need to pass a <string> type parameter to get().  
  return this.http.get(filename, {responseType: 'text'})  
    .pipe(  
      tap( // Log the result or error  
        data => this.log(filename, data),  
        error => this.logError(filename, error)  
      )  
    );  
}
```

`HttpClient.get()` returns a string rather than the default JSON because of the `responseType` option.

The RxJS `tap` operator (as in "wiretap") lets the code inspect both success and error values passing through the observable without disturbing them.

A `download()` method in the `DownloaderComponent` initiates the request by subscribing to the service method.

app/downloader/downloader.component.ts (download)

```
download() {  
  this.downloaderService.getTextFile('assets/textfile.txt')  
    .subscribe(results => this.contents = results);  
}
```

## Handling request errors

If the request fails on the server, `HttpClient` returns an *error* object instead of a successful response.

The same service that performs your server transactions should also perform error inspection, interpretation, and resolution.

When an error occurs, you can obtain details of what failed in order to inform your user. In some cases, you might also automatically [retry the request](#).

## Getting error details



An app should give the user useful feedback when data access fails. A raw error object is not particularly useful as feedback. In addition to detecting that an error has occurred, you need to get error details and use those details to compose a user-friendly response.

Two types of errors can occur.

- The server backend might reject the request, returning an HTTP response with a status code such as 404 or 500. These are error *responses*.
- Something could go wrong on the client-side such as a network error that prevents the request from completing successfully or an exception thrown in an RxJS operator. These errors produce JavaScript `ErrorEvent` objects.

`HttpClient` captures both kinds of errors in its `HttpErrorResponse`. You can inspect that response to identify the error's cause.

The following example defines an error handler in the previously defined `ConfigService`.

#### app/config/config.service.ts (handleError)

```
private handleError(error: HttpErrorResponse) {  
  if (error.error instanceof ErrorEvent) {  
    // A client-side or network error occurred. Handle it accordingly.  
    console.error('An error occurred:', error.error.message);  
  } else {  
    // The backend returned an unsuccessful response code.  
    // The response body may contain clues as to what went wrong.  
    console.error(  
      `Backend returned code ${error.status}, ` +  
      `body was: ${error.error}`);  
  }  
  // Return an observable with a user-facing error message.  
  return throwError(  
    'Something bad happened; please try again later.');  
}
```

The handler returns an RxJS `ErrorObservable` with a user-friendly error message. The following code updates the `getConfig()` method, using a `pipe` to send all observables returned by the `HttpClient.get()` call to the error handler.

#### app/config/config.service.ts (getConfig v.3 with error handler)

```
getConfig() {  
  return this.http.get<Config>(this.configUrl)  
    .pipe(  
      catchError(this.handleError)  
    );  
}
```

## Retrying a failed request

Sometimes the error is transient and goes away automatically if you try again. For example, network interruptions are common in mobile scenarios, and trying again can produce a successful result.

The [RxJS library](#) offers several *retry* operators. For example, the `retry()` operator automatically re-subscribes to a failed `Observable` a specified number of times. *Re-subscribing* to the result of an `HttpClient` method call has the effect of reissuing the HTTP request.

The following example shows how you can pipe a failed request to the `retry()` operator before passing it to the error handler.

app/config/config.service.ts (getConfig with retry)

```
getConfig() {  
  return this.http.get<Config>(this.configUrl)  
    .pipe(  
      retry(3), // retry a failed request up to 3 times  
      catchError(this.handleError) // then handle the error  
    );  
}
```

## Sending data to a server

In addition to fetching data from a server, `HttpClient` supports other HTTP methods such as PUT, POST, and DELETE, which you can use to modify the remote data.

The sample app for this guide includes a simplified version of the "Tour of Heroes" example that fetches heroes and enables users to add, delete, and update them. The following sections show examples of the data-update methods from the sample's `HeroesService`.

## Making a POST request

Apps often send data to a server with a POST request when submitting a form. In the following example, the `HeroesService` makes an HTTP POST request when adding a hero to the database.

app/heroes/heroes.service.ts (addHero)

```
/** POST: add a new hero to the database */  
addHero(hero: Hero): Observable<Hero> {  
  return this.http.post<Hero>(this.heroesUrl, hero, httpOptions)  
    .pipe(  
      catchError(this.handleError('addHero', hero))  
    );  
}
```

The `HttpClient.post()` method is similar to `get()` in that it has a type parameter, which you can use to specify that you expect the server to return data of a given type. The method takes a resource URL and two additional

parameters:

- *body* - The data to POST in the body of the request.
- *options* - An object containing method options which, in this case, [specify required headers](#).

The example catches errors as [described above](#).

The HeroesComponent initiates the actual POST operation by subscribing to the Observable returned by this service method.

#### app/heroes/heroes.component.ts (addHero)

```
this.heroesService
  .addHero(newHero)
  .subscribe(hero => this.heroes.push(hero));
```

When the server responds successfully with the newly added hero, the component adds that hero to the displayed heroes list.

## Making a DELETE request

This application deletes a hero with the HttpClient.delete method by passing the hero's id in the request URL.

#### app/heroes/heroes.service.ts (deleteHero)

```
/** DELETE: delete the hero from the server */
deleteHero(id: number): Observable<{}> {
  const url = `${this.heroesUrl}/${id}`; // DELETE api/heroes/42
  return this.http.delete(url, httpOptions)
    .pipe(
      catchError(this.handleError('deleteHero'))
    );
}
```

The HeroesComponent initiates the actual DELETE operation by subscribing to the Observable returned by this service method.

#### app/heroes/heroes.component.ts (deleteHero)

```
this.heroesService
  .deleteHero(hero.id)
  .subscribe();
```

The component isn't expecting a result from the delete operation, so it subscribes without a callback. Even though you are not using the result, you still have to subscribe. Calling the subscribe() method *executes* the observable, which is what initiates the DELETE request.

You must call *subscribe()* or nothing happens. Just calling `HeroesService.deleteHero()` does not initiate the DELETE request.

```
// oops ... subscribe() is missing so nothing happens
this.heroesService.deleteHero(hero.id);
```



### Always *subscribe*!

An `HttpClient` method does not begin its HTTP request until you call `subscribe()` on the observable returned by that method. This is true for *all* `HttpClient` *methods*.

The `AsyncPipe` subscribes (and unsubscribes) for you automatically.

All observables returned from `HttpClient` methods are *cold* by design. Execution of the HTTP request is *deferred*, allowing you to extend the observable with additional operations such as `tap` and `catchError` before anything actually happens.

Calling `subscribe(...)` triggers execution of the observable and causes `HttpClient` to compose and send the HTTP request to the server.

You can think of these observables as *blueprints* for actual HTTP requests.

In fact, each `subscribe()` initiates a separate, independent execution of the observable. Subscribing twice results in two HTTP requests.

```
const req = http.get<Heroes>('/api/heroes');
// 0 requests made - .subscribe() not called.
req.subscribe();
// 1 request made.
req.subscribe();
// 2 requests made.
```



## Making a PUT request

An app can send PUT requests using the HTTP client service. The following `HeroesService` example, like the POST example, replaces a resource with updated data.

app/heroes/heroes.service.ts (updateHero)

```
/** PUT: update the hero on the server. Returns the updated hero upon success. */
```



```

updateHero(hero: Hero): Observable<Hero> {
  return this.http.put<Hero>(this.heroesUrl, hero, httpOptions)
    .pipe(
      catchError(this.handleError('updateHero', hero))
    );
}

```

As for any of the HTTP methods that return an observable, the caller, `HeroesComponent.update()` **must** `subscribe()` to the observable returned from the `HttpClient.put()` in order to initiate the request.

## Adding and updating headers

Many servers require extra headers for save operations. For example, a server might require an authorization token, or "Content-Type" header to explicitly declare the MIME type of the request body.

### Adding headers

The `HeroesService` defines such headers in an `httpOptions` object that are passed to every `HttpClient` save method.

app/heroes/heroes.service.ts (httpOptions)

```

import { HttpHeaders } from '@angular/common/http';

const httpOptions = {
  headers: new HttpHeaders({
    'Content-Type': 'application/json',
    Authorization: 'my-auth-token'
  })
};

```

### Updating headers

You can't directly modify the existing headers within the previous options object because instances of the `HttpHeaders` class are immutable. Use the `set()` method instead, to return a clone of the current instance with the new changes applied.

The following example shows how, when an old token has expired, you can update the authorization header before making the next request.

```

httpOptions.headers =
  httpOptions.headers.set('Authorization', 'my-new-auth-token');

```

## Configuring HTTP URL parameters

Use the `HttpParams` class with the `params` request option to add URL query strings in your `HttpRequest`.

The following example, the `searchHeroes()` method queries for heroes whose names contain the search term.

Start by importing `HttpParams` class.

```
import {HttpParams} from "@angular/common/http";
```

```
/* GET heroes whose name contains search term */
searchHeroes(term: string): Observable<Hero[]> {
  term = term.trim();

  // Add safe, URL encoded search parameter if there is a search term
  const options = term ?
    { params: new HttpParams().set('name', term) } : {};

  return this.http.get<Hero[]>(this.heroesUrl, options)
    .pipe(
      catchError(this.handleError<Hero[]>('searchHeroes', []))
    );
}
```



If there is a search term, the code constructs an options object with an HTML URL-encoded search parameter. If the term is "cat", for example, the GET request URL would be `api/heroes?name=cat`.

The `HttpParams` object is immutable. If you need to update the options, save the returned value of the `.set()` method.

You can also create HTTP parameters directly from a query string by using the `fromString` variable:

```
const params = new HttpParams({fromString: 'name=foo'});
```

## Intercepting requests and responses

With interception, you declare *interceptors* that inspect and transform HTTP requests from your application to a server. The same interceptors can also inspect and transform a server's responses on their way back to the application. Multiple interceptors form a *forward-and-backward* chain of request/response handlers.

Interceptors can perform a variety of *implicit* tasks, from authentication to logging, in a routine, standard way, for every HTTP request/response.

Without interception, developers would have to implement these tasks *explicitly* for each `HttpClient` method call.

### Write an interceptor

To implement an interceptor, declare a class that implements the `intercept()` method of the `HttpInterceptor` interface.

Here is a do-nothing *noop* interceptor that simply passes the request through without touching it:

```
import { Injectable } from '@angular/core';
import {
  HttpEvent, HttpInterceptor, HttpHandler, HttpRequest
} from '@angular/common/http';

import { Observable } from 'rxjs';

/** Pass untouched request through to the next request handler. */
@Injectable()
export class NoopInterceptor implements HttpInterceptor {

  intercept(req: HttpRequest<any>, next: HttpHandler):
    Observable<HttpEvent<any>> {
    return next.handle(req);
  }
}
```

The `intercept` method transforms a request into an `Observable` that eventually returns the HTTP response. In this sense, each interceptor is fully capable of handling the request entirely by itself.

Most interceptors inspect the request on the way in and forward the (perhaps altered) request to the `handle()` method of the next object which implements the `HttpHandler` interface.

```
export abstract class HttpHandler {
  abstract handle(req: HttpRequest<any>): Observable<HttpEvent<any>>;
}
```

Like `intercept()`, the `handle()` method transforms an HTTP request into an `Observable` of `HttpEvents` which ultimately include the server's response. The `intercept()` method could inspect that observable and alter it before returning it to the caller.

This *no-op* interceptor simply calls `next.handle()` with the original request and returns the observable without doing a thing.

## The *next* object

The `next` object represents the next interceptor in the chain of interceptors. The final `next` in the chain is the `HttpClient` backend handler that sends the request to the server and receives the server's response.

Most interceptors call `next.handle()` so that the request flows through to the next interceptor and, eventually, the backend handler. An interceptor *could* skip calling `next.handle()`, short-circuit the chain, and [return its own](#) `Observable` with an artificial server response.

This is a common middleware pattern found in frameworks such as `Express.js`.

## Provide the interceptor

The `NoopInterceptor` is a service managed by Angular's [dependency injection \(DI\)](#) system. Like other services, you must provide the interceptor class before the app can use it.

Because interceptors are (optional) dependencies of the `HttpClient` service, you must provide them in the same injector (or a parent of the injector) that provides `HttpClient`. Interceptors provided *after* DI creates the `HttpClient` are ignored.

This app provides `HttpClient` in the app's root injector, as a side-effect of importing the `HttpClientModule` in `AppModule`. You should provide interceptors in `AppModule` as well.

After importing the `HTTP_INTERCEPTORS` injection token from `@angular/common/http`, write the `NoopInterceptor` provider like this:

```
{ provide: HTTP_INTERCEPTORS, useClass: NoopInterceptor, multi: true },
```

Note the `multi: true` option. This required setting tells Angular that `HTTP_INTERCEPTORS` is a token for a *multi-provider* that injects an array of values, rather than a single value.

You *could* add this provider directly to the providers array of the `AppModule`. However, it's rather verbose and there's a good chance that you'll create more interceptors and provide them in the same way. You must also pay [close attention to the order](#) in which you provide these interceptors.

Consider creating a "barrel" file that gathers all the interceptor providers into an `httpInterceptorProviders` array, starting with this first one, the `NoopInterceptor`.

app/http-interceptors/index.ts

```
/* "Barrel" of Http Interceptors */
import { HTTP_INTERCEPTORS } from '@angular/common/http';

import { NoopInterceptor } from './noop-interceptor';

/** Http interceptor providers in outside-in order */
export const httpInterceptorProviders = [
  { provide: HTTP_INTERCEPTORS, useClass: NoopInterceptor, multi: true },
];
```

Then import and add it to the `AppModule` *providers array* like this:

app/app.module.ts (interceptor providers)

```
providers: [
  httpInterceptorProviders
],
```

As you create new interceptors, add them to the `httpInterceptorProviders` array and you won't have to revisit the `AppModule`.



There are many more interceptors in the complete sample code.

## Interceptor order

Angular applies interceptors in the order that you provide them. If you provide interceptors *A*, then *B*, then *C*, requests flow in *A*->*B*->*C* and responses 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.

## Handling interceptor events

Most `HttpClient` methods return observables of `HttpResponse<any>`. The `HttpResponse` class itself is actually an event, whose type is `HttpEventType.Response`. A single HTTP request can, however, generate multiple events of other types, including upload and download progress events. The methods `HttpInterceptor.intercept()` and `HttpHandler.handle()` return observables of `HttpEvent<any>`.

Many interceptors are only concerned with the outgoing request and return the event stream from `next.handle()` without modifying it. Some interceptors, however, need to examine and modify the response from `next.handle()`; these operations can see all of these events in the stream.

Although interceptors are capable of modifying requests and responses, the `HttpRequest` and `HttpResponse` instance properties are `readonly`, rendering them largely immutable. They are immutable for a good reason: an app might retry a request several times before it succeeds, which means that the interceptor chain can re-process the same request multiple times. If an interceptor could modify the original request object, the re-tried operation would start from the modified request rather than the original. Immutability ensures that interceptors see the same request for each try.

Your interceptor should return every event without modification unless it has a compelling reason to do otherwise.

TypeScript prevents you from setting `HttpRequest` read-only properties.

```
// Typescript disallows the following assignment because req.url is readonly
req.url = req.url.replace('http://', 'https://');
```



If you must alter a request, clone it first and modify the clone before passing it to `next.handle()`. You can clone and modify the request in a single step, as shown in the following example.

app/http-interceptors/ensure-https-interceptor.ts (excerpt)

```
// clone request and replace 'http://' with 'https://' at the same time
const secureReq = req.clone({
  url: req.url.replace('http://', 'https://')
});
```



```
});  
// send the cloned, "secure" request to the next handler.  
return next.handle(secureReq);
```

The `clone()` method's `hash` argument allows you to mutate specific properties of the request while copying the others.

## Modifying a request body

The `readonly` assignment guard can't prevent deep updates and, in particular, it can't prevent you from modifying a property of a request body object.

```
req.body.name = req.body.name.trim(); // bad idea!
```

If you must modify the request body, follow these steps.

1. Copy the body and make your change in the copy.
2. Clone the request object, using its `clone()` method.
3. Replace the clone's body with the modified copy.

### app/http-interceptors/trim-name-interceptor.ts (excerpt)

```
// copy the body and trim whitespace from the name property  
const newBody = { ...body, name: body.name.trim() };  
// clone request and set its body  
const newReq = req.clone({ body: newBody });  
// send the cloned request to the next handler.  
return next.handle(newReq);
```

## Clearing the request body in a clone

Sometimes you need to clear the request body rather than replace it. To do this, set the cloned request body to `null`.

**Tip:** If you set the cloned request body to `undefined`, Angular assumes you intend to leave the body as is.

```
newReq = req.clone({ ... }); // body not mentioned => preserve original body  
newReq = req.clone({ body: undefined }); // preserve original body  
newReq = req.clone({ body: null }); // clear the body
```

## Setting default headers

Apps often use an interceptor to set default headers on outgoing requests.

The sample app has an `AuthService` that produces an authorization token. Here is its `AuthInterceptor` that injects that service to get the token and adds an authorization header with that token to every outgoing request:

app/http-interceptors/auth-interceptor.ts

```
import { AuthService } from '../auth.service';

@Injectable()
export class AuthInterceptor implements HttpInterceptor {

  constructor(private auth: AuthService) {}

  intercept(req: HttpRequest<any>, next: HttpHandler) {
    // Get the auth token from the service.
    const authToken = this.auth.getAuthorizationToken();

    // Clone the request and replace the original headers with
    // cloned headers, updated with the authorization.
    const authReq = req.clone({
      headers: req.headers.set('Authorization', authToken)
    });

    // send cloned request with header to the next handler.
    return next.handle(authReq);
  }
}
```

The practice of cloning a request to set new headers is so common that there's a `setHeaders` shortcut for it:

```
// Clone the request and set the new header in one step.
const authReq = req.clone({ setHeaders: { Authorization: authToken } });
```

An interceptor that alters headers can be used for a number of different operations, including:

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

## Using interceptors for logging

Because interceptors can process the request and response *together*, they can perform tasks such as timing and logging an entire HTTP operation.

Consider the following `LoggingInterceptor`, which captures the time of the request, the time of the response, and logs the outcome with the elapsed time with the injected `MessageService`.



```
import { finalize, tap } from 'rxjs/operators';
import { MessageService } from '../message.service';

@Injectable()
export class LoggingInterceptor implements HttpInterceptor {
  constructor(private messenger: MessageService) {}

  intercept(req: HttpRequest<any>, next: HttpHandler) {
    const started = Date.now();
    let ok: string;

    // extend server response observable with logging
    return next.handle(req)
      .pipe(
        tap(
          // Succeeds when there is a response; ignore other events
          event => ok = event instanceof HttpResponse ? 'succeeded' : '',
          // Operation failed; error is an HttpErrorResponse
          error => ok = 'failed'
        ),
        // Log when response observable either completes or errors
        finalize(() => {
          const elapsed = Date.now() - started;
          const msg = `${req.method} "${req.urlWithParams}"
            ${ok} in ${elapsed} ms.`;
          this.messenger.add(msg);
        })
      );
  }
}
```

The RxJS `tap` operator captures whether the request succeeded or failed. The RxJS `finalize` operator is called when the response observable either errors or completes (which it must), and reports the outcome to the `MessageService`.

Neither `tap` nor `finalize` touch the values of the observable stream returned to the caller.

## Using interceptors for caching

Interceptors can handle requests by themselves, without forwarding to `next.handle()`.

For example, you might decide to cache certain requests and responses to improve performance. You can delegate caching to an interceptor without disturbing your existing data services.

The `CachingInterceptor` in the following example demonstrates this approach.



```

@Injectable()
export class CachingInterceptor implements HttpInterceptor {
  constructor(private cache: RequestCache) {}

  intercept(req: HttpRequest<any>, next: HttpHandler) {
    // continue if not cacheable.
    if (!isCacheable(req)) { return next.handle(req); }

    const cachedResponse = this.cache.get(req);
    return cachedResponse ?
      of(cachedResponse) : sendRequest(req, next, this.cache);
  }
}

```

- The `isCacheable()` function determines if the request is cacheable. In this sample, only GET requests to the npm package search api are cacheable.
- If the request is not cacheable, the interceptor simply forwards the request to the next handler in the chain.
- If a cacheable request is found in the cache, the interceptor returns an `of()` *observable* with the cached response, by-passing the next handler (and all other interceptors downstream).
- If a cacheable request is not in cache, the code calls `sendRequest()`. This function creates a [request clone](#) without headers, because the npm API forbids them. The function then forwards the clone of the request to `next.handle()` which ultimately calls the server and returns the server's response.



```

/**
 * Get server response observable by sending request to `next()`.
 * Will add the response to the cache on the way out.
 */
function sendRequest(
  req: HttpRequest<any>,
  next: HttpHandler,
  cache: RequestCache): Observable<HttpEvent<any>> {

  // No headers allowed in npm search request
  const noHeaderReq = req.clone({ headers: new HttpHeaders() });

  return next.handle(noHeaderReq).pipe(
    tap(event => {
      // There may be other events besides the response.
      if (event instanceof HttpResponse) {
        cache.put(req, event); // Update the cache.
      }
    })
  )
}

```

```
);  
}
```

Note how `sendRequest()` intercepts the response on its way back to the application. This method pipes the response through the `tap()` operator, whose callback adds the response to the cache.

The original response continues untouched back up through the chain of interceptors to the application caller.

Data services, such as `PackageSearchService`, are unaware that some of their `HttpClient` requests actually return cached responses.

## Using interceptors to request multiple values

The `HttpClient.get()` method normally returns an observable that emits a single value, either the data or an error. An interceptor can change this to an observable that emits [multiple values](#).

The following revised version of the `CachingInterceptor` optionally returns an observable that immediately emits the cached response, sends the request on to the npm web API, and emits again later with the updated search results.

```
// cache-then-refresh  
if (req.headers.get('x-refresh')) {  
  const results$ = sendRequest(req, next, this.cache);  
  return cachedResponse ?  
    results$.pipe( startWith(cachedResponse) ) :  
    results$;  
}  
// cache-or-fetch  
return cachedResponse ?  
  of(cachedResponse) : sendRequest(req, next, this.cache);
```



The *cache-then-refresh* option is triggered by the presence of a custom `x-refresh` header.

A checkbox on the `PackageSearchComponent` toggles a `withRefresh` flag, which is one of the arguments to `PackageSearchService.search()`. That `search()` method creates the custom `x-refresh` header and adds it to the request before calling `HttpClient.get()`.

The revised `CachingInterceptor` sets up a server request whether there's a cached value or not, using the same `sendRequest()` method described [above](#). The `results$` observable makes the request when subscribed.

- If there's no cached value, the interceptor returns `results$`.
- If there is a cached value, the code *pipes* the cached response onto `results$`, producing a recomposed observable that emits twice, the cached response first (and immediately), followed later by the response from the server. Subscribers see a sequence of two responses.

## Tracking and showing request progress

Sometimes applications transfer large amounts of data and those transfers can take a long time. File uploads are a typical example. You can give the users a better experience by providing feedback on the progress of such transfers.

To make a request with progress events enabled, you can create an instance of `HttpRequest` with the `reportProgress` option set `true` to enable tracking of progress events.

#### app/uploader/uploader.service.ts (upload request)

```
const req = new HttpRequest('POST', '/upload/file', file, {
  reportProgress: true
});
```



**Tip:** Every progress event triggers change detection, so only turn them on if you need to report progress in the UI.

When using `HttpClient.request()` with an HTTP method, configure the method with `observe: 'events'` to see all events, including the progress of transfers.

Next, pass this request object to the `HttpClient.request()` method, which returns an `Observable` of `HttpEvents` (the same events processed by [interceptors](#)).

#### app/uploader/uploader.service.ts (upload body)

```
// The `HttpClient.request` API produces a raw event stream
// which includes start (sent), progress, and response events.
return this.http.request(req).pipe(
  map(event => this.getEventMessage(event, file)),
  tap(message => this.showProgress(message)),
  last(), // return last (completed) message to caller
  catchError(this.handleError(file))
);
```



The `getEventMessage` method interprets each type of `HttpEvent` in the event stream.

#### app/uploader/uploader.service.ts (getEventMessage)

```
/** Return distinct message for sent, upload progress, & response events */
private getEventMessage(event: HttpEvent<any>, file: File) {
  switch (event.type) {
    case HttpEventType.Sent:
      return `Uploading file "${file.name}" of size ${file.size}`;

    case HttpEventType.UploadProgress:
      // Compute and show the % done:
      const percentDone = Math.round(100 * event.loaded / event.total);
```



```

        return `File "${file.name}" is ${percentDone}% uploaded.`;

    case HttpEventType.Response:
        return `File "${file.name}" was completely uploaded!`;

    default:
        return `File "${file.name}" surprising upload event: ${event.type}.`;
    }
}

```

The sample app for this guide doesn't have a server that accepts uploaded files. The `UploadInterceptor` in `app/http-interceptors/upload-interceptor.ts` intercepts and short-circuits upload requests by returning an observable of simulated events.

## Optimizing server interaction with debouncing

If you need to make an HTTP request in response to user input, it's not efficient to send a request for every keystroke. It's better to wait until the user stops typing and then send a request. This technique is known as debouncing.

Consider the following template, which lets a user enter a search term to find an npm package by name. When the user enters a name in a search-box, the `PackageSearchComponent` sends a search request for a package with that name to the npm web API.

app/package-search/package-search.component.html (search)

```

<input (keyup)="search($event.target.value)" id="name" placeholder="Search" />

<ul>
  <li *ngFor="let package of packages$ | async">
    <b>{{package.name}} v.{{package.version}}</b> -
    <i>{{package.description}}</i>
  </li>
</ul>

```

Here, the `keyup` event binding sends every keystroke to the component's `search()` method. The following snippet implements debouncing for this input using RxJS operators.

app/package-search/package-search.component.ts (excerpt)

```

withRefresh = false;
packages$: Observable<NpmPackageInfo[]>;
private searchText$ = new Subject<string>();

search(packageName: string) {

```



```

    this.searchText$.next(packageName);
  }

  ngOnInit() {
    this.packages$ = this.searchText$.pipe(
      debounceTime(500),
      distinctUntilChanged(),
      switchMap(packageName =>
        this.searchService.search(packageName, this.withRefresh))
    );
  }

  constructor(private searchService: PackageSearchService) { }

```

The `searchText$` is the sequence of search-box values coming from the user. It's defined as an `RxJS Subject`, which means it is a multicasting `Observable` that can also emit values for itself by calling `next(value)`, as happens in the `search()` method.

Rather than forward every `searchText` value directly to the injected `PackageSearchService`, the code in `ngOnInit()` pipes search values through three operators, so that a search value reaches the service only if it's a new value and the user has stopped typing.

- `debounceTime(500)`—Wait for the user to stop typing (1/2 second in this case).
- `distinctUntilChanged()`—Wait until the search text changes.
- `switchMap()`—Send the search request to the service.

The code sets `packages$` to this re-composed `Observable` of search results. The template subscribes to `packages$` with the [AsyncPipe](#) and displays search results as they arrive.

See [Using interceptors to request multiple values](#) for more about the `withRefresh` option.



## Using the *switchMap()* operator

The `switchMap()` operator takes a function argument that returns an `Observable`. In the example, `PackageSearchService.search` returns an `Observable`, as other data service methods do. If a previous search request is still in-flight (as when the network connection is poor), the operator cancels that request and sends a new one.

Note that `switchMap()` returns service responses in their original request order, even if the server returns them out of order.

If you think you'll reuse this debouncing logic, consider moving it to a utility function or into the `PackageSearchService` itself.

# Security: XSRF protection

[Cross-Site Request Forgery \(XSRF or CSRF\)](#)  is an attack technique by which the attacker can trick an authenticated user into unknowingly executing actions on your website. `HttpClient` supports a [common mechanism](#)  used to prevent XSRF attacks. When performing HTTP requests, an interceptor reads a token from a cookie, by default `XSRF-TOKEN`, and sets it as an HTTP header, `X-XSRF-TOKEN`. Since only code that runs on your domain could read the cookie, the backend can be certain that the HTTP request came from your client application and not an attacker.

By default, an interceptor sends this header on all mutating requests (such as POST) to relative URLs, but not on GET/HEAD requests or on requests with an absolute URL.

To take advantage of this, your server needs to set a token in a JavaScript readable session cookie called `XSRF-TOKEN` on either the page load or the first GET request. On subsequent requests the server can verify that the cookie matches the `X-XSRF-TOKEN` HTTP header, and therefore be sure that only code running on your domain could have sent the request. The token must be unique for each user and must be verifiable by the server; this prevents the client from making up its own tokens. Set the token to a digest of your site's authentication cookie with a salt for added security.

In order to prevent collisions in environments where multiple Angular apps share the same domain or subdomain, give each application a unique cookie name.

*`HttpClient` supports only the client half of the XSRF protection scheme.* Your backend service must be configured to set the cookie for your page, and to verify that the header is present on all eligible requests. Failing to do so renders Angular's default protection ineffective.

## Configuring custom cookie/header names

If your backend service uses different names for the XSRF token cookie or header, use `HttpClientXsrfModule.withOptions()` to override the defaults.

```
imports: [  
  HttpClientModule,  
  HttpClientModule.withOptions({  
    cookieName: 'My-Xsrf-Cookie',  
    headerName: 'My-Xsrf-Header',  
  } ),  
],
```



## Testing HTTP requests

As for any external dependency, you must mock the HTTP backend so your tests can simulate interaction with a remote server. The `@angular/common/http/testing` library makes it straightforward to set up such mocking.

Angular's HTTP testing library is designed for a pattern of testing in which the app executes code and makes requests first. The test then expects that certain requests have or have not been made, performs assertions against

those requests, and finally provides responses by "flushing" each expected request.

At the end, tests can verify that the app has made no unexpected requests.

You can run [these sample tests](#) / [download example](#) in a live coding environment.

The tests described in this guide are in `src/testing/http-client.spec.ts`. There are also tests of an application data service that call `HttpClient` in `src/app/heroes/heroes.service.spec.ts`.

## Setup for testing

To begin testing calls to `HttpClient`, import the `HttpClientTestingModule` and the mocking controller, `HttpTestingController`, along with the other symbols your tests require.

### app/testing/http-client.spec.ts (imports)

```
// Http testing module and mocking controller
import { HttpClientTestingModule, HttpTestingController } from
  '@angular/common/http/testing';

// Other imports
import { TestBed } from '@angular/core/testing';
import { HttpClient, HttpResponseError } from '@angular/common/http';
```

Then add the `HttpClientTestingModule` to the `TestBed` and continue with the setup of the *service-under-test*.

### app/testing/http-client.spec.ts(setup)

```
describe('HttpClient testing', () => {
  let httpClient: HttpClient;
  let httpTestingController: HttpTestingController;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [ HttpClientTestingModule ]
    });

    // Inject the http service and test controller for each test
    httpClient = TestBed.inject(HttpClient);
    httpTestingController = TestBed.inject(HttpTestingController);
  });

  /// Tests begin ///
});
```

Now requests made in the course of your tests hit the testing backend instead of the normal backend.

This setup also calls `TestBed.inject()` to inject the `HttpClient` service and the mocking controller so they can be referenced during the tests.

## Expecting and answering requests

Now you can write a test that expects a GET Request to occur and provides a mock response.

app/testing/http-client.spec.ts(httpClient.get)

```
it('can test HttpClient.get', () => {
  const testData: Data = {name: 'Test Data'};

  // Make an HTTP GET request
  httpClient.get<Data>(testUrl)
    .subscribe(data =>
      // When observable resolves, result should match test data
      expect(data).toEqual(testData)
    );

  // The following `expectOne()` will match the request's URL.
  // If no requests or multiple requests matched that URL
  // `expectOne()` would throw.
  const req = httpTestingController.expectOne('/data');

  // Assert that the request is a GET.
  expect(req.request.method).toEqual('GET');

  // Respond with mock data, causing Observable to resolve.
  // Subscribe callback asserts that correct data was returned.
  req.flush(testData);

  // Finally, assert that there are no outstanding requests.
  httpTestingController.verify();
});
```

The last step, verifying that no requests remain outstanding, is common enough for you to move it into an `afterEach()` step:

```
afterEach(() => {
  // After every test, assert that there are no more pending requests.
  httpTestingController.verify();
});
```

## Custom request expectations

If matching by URL isn't sufficient, it's possible to implement your own matching function. For example, you could look for an outgoing request that has an authorization header:

```
// Expect one request with an authorization header
const req = httpTestingController.expectOne(
  request => request.headers.has('Authorization')
);
```

As with the previous `expectOne()`, the test fails if 0 or 2+ requests satisfy this predicate.

## Handling more than one request

If you need to respond to duplicate requests in your test, use the `match()` API instead of `expectOne()`. It takes the same arguments but returns an array of matching requests. Once returned, these requests are removed from future matching and you are responsible for flushing and verifying them.

```
// get all pending requests that match the given URL
const requests = httpTestingController.match(testUrl);
expect(requests.length).toEqual(3);

// Respond to each request with different results
requests[0].flush([]);
requests[1].flush([testData[0]]);
requests[2].flush(testData);
```

## Testing for errors

You should test the app's defenses against HTTP requests that fail.

Call `request.flush()` with an error message, as seen in the following example.

```
it('can test for 404 error', () => {
  const emsg = 'deliberate 404 error';

  httpClient.get<Data[]>(testUrl).subscribe(
    data => fail('should have failed with the 404 error'),
    (error: HttpResponseError) => {
      expect(error.status).toEqual(404, 'status');
      expect(error.error).toEqual(emsg, 'message');
    }
  );

  const req = httpTestingController.expectOne(testUrl);

  // Respond with mock error
```

```
req.flush(msg, { status: 404, statusText: 'Not Found' });  
});
```

Alternatively, you can call `request.error()` with an `ErrorEvent`.

```
it('can test for network error', () => {  
  const msg = 'simulated network error';  
  
  httpClient.get<Data[]>(testUrl).subscribe(  
    data => fail('should have failed with the network error'),  
    (error: HttpResponse) => {  
      expect(error.error.message).toEqual(msg, 'message');  
    }  
  );  
  
  const req = httpTestingController.expectOne(testUrl);  
  
  // Create mock ErrorEvent, raised when something goes wrong at the network level.  
  // Connection timeout, DNS error, offline, etc  
  const mockError = new ErrorEvent('Network error', {  
    message: msg,  
  });  
  
  // Respond with mock error  
  req.error(mockError);  
});
```

## RESOURCES

[About](#)  
[Resource Listing](#)  
[Press Kit](#)  
[Blog](#)  
[Usage Analytics](#)

## HELP

[Stack Overflow](#)  
[Gitter](#)  
[Report Issues](#)  
[Code of Conduct](#)

## COMMUNITY

[Events](#)  
[Meetups](#)  
[Twitter](#)  
[GitHub](#)  
[Contribute](#)

## LANGUAGES

[简体中文版](#)  
[正體中文版](#)  
[日本語版](#)  
[한국어](#)

Super-powered by Google ©2010-2020. Code licensed under an MIT-style License. Documentation licensed under CC BY 4.0.

Version 10.0.10-local+sha.84d1ba792b.