HttpClient

Most front-end applications communicate with backend services over the HTTP protocol. Modern browsers support two different APIs for making HTTP requests: the `XMLHttpRequest` interface and the `fetch()` API.

With `HttpClient`, `@angular/common/http` provides a simplified API for HTTP functionality for use with Angular applications, building on top of the `XMLHttpRequest` interface exposed by browsers. Additional benefits of `HttpClient` include testability support, strong typing of request and response objects, request and response interceptor support, and better error handling via apis based on Observables.

Setup: installing the module

Before you can use the `HttpClient`, you need to install the `HttpClientModule` which provides it. This can be done in your application module, and is only necessary once.

```
// app.module.ts:

import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';

// Import HttpClientModule from @angular/common/http
import {HttpClientModule} from '@angular/common/http';

@NgModule({
  imports: [
    BrowserModule,
    // Include it under 'imports' in your application module
    // after BrowserModule.
    HttpClientModule,
  ],
})
export class MyAppModule {}
```

Once you import `HttpClientModule` into your app module, you can inject `HttpClient` into your components and services.

Making a request for JSON data

The most common type of request applications make to a backend is to request JSON data. For example, suppose you have an API endpoint that lists items, `/api/items`, which returns a JSON object of the form:

```
{
  "results": [
    "Item 1",
    "Item 2",
  ]
}
```

The `get()` method on `HttpClient` makes accessing this data straightforward.

```
@Component(...)
export class MyComponent implements OnInit {

  results: string[];

  // Inject HttpClient into your component or service.
  constructor(private http: HttpClient) {}

  ngOnInit(): void {
    // Make the HTTP request:
    this.http.get('/api/items').subscribe(data => {
      // Read the result field from the JSON response.
      this.results = data['results'];
    });
  }
}
```

Typechecking the response

In the above example, the `data['results']` field access stands out because you use bracket notation to access the results field. If you tried to write `data.results`, TypeScript would correctly complain that the `Object` coming back from HTTP does not have a `results` property. That's because while `HttpClient` parsed the JSON response into an `Object`, it doesn't know what shape that object is.

You can, however, tell `HttpClient` what type the response will be, which is recommended. To do so, first you define an interface with the correct shape:

```
interface ItemsResponse {
  results: string[];
}
```

Then, when you make the `HttpClient.get` call, pass a type parameter:

```
http.get<ItemsResponse>('/api/items').subscribe(data => {
  // data is now an instance of type ItemsResponse, so you can do this:
  this.results = data.results;
});
```

Reading the full response

The response body doesn't return all the data you may need. Sometimes servers return special headers or status codes to indicate certain conditions, and inspecting those can be necessary. To do this, you can tell `HttpClient` you want the full response instead of just the body with the `observe` option:

```
http
  .get<MyJsonData>('/data.json', {observe: 'response'})
  .subscribe(resp => {
    // Here, resp is of type HttpResponse<MyJsonData>.
    // You can inspect its headers:
    console.log(resp.headers.get('X-Custom-Header'));
    // And access the body directly, which is typed as MyJsonData as requested.
    console.log(resp.body.someField);
  });
```

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

Error handling

What happens if the request fails on the server, or if a poor network connection prevents it from even reaching the server? `HttpClient` will return an *error* instead of a successful response.

To handle it, add an error handler to your `.subscribe()` call:

```
http
  .get<ItemsResponse>('/api/items')
  .subscribe(
    // Successful responses call the first callback.
    data => {...},
    // Errors will call this callback instead:
    err => {
      console.log('Something went wrong!');
    }
  );
```

Getting error details

Detecting that an error occurred is one thing, but it's more useful to know what error actually occurred. The `err` parameter to the callback above is of type `HttpErrorResponse`, and contains useful information on what went wrong.

There are two types of errors that can occur. If the backend returns an unsuccessful response code (404, 500, etc.), it gets returned as an error. Also, if something goes wrong client-side, such as an exception gets thrown in an RxJS operator, or if a network error prevents the request from completing successfully, an actual `Error` will be thrown.

In both cases, you can look at the `HttpErrorResponse` to figure out what happened.

```
http
  .get<ItemsResponse>('/api/items')
  .subscribe(
    data => {...},
    (err: HttpErrorResponse) => {
      if (err.error instanceof Error) {
        // A client-side or network error occurred. Handle it accordingly.
        console.log('An error occurred:', err.error.message);
      } else {
        // The backend returned an unsuccessful response code.
        // The response body may contain clues as to what went wrong,
        console.log(`Backend returned code ${err.status}, body was: ${err.error}`);
      }
    }
  );
```

`.retry()`

One way to deal with errors is to simply retry the request. This strategy can be useful when the errors are transient and unlikely to repeat.

RxJS has a useful operator called `.retry()`, which automatically resubscribes to an Observable, thus reissuing the request, upon encountering an error.

First, import it:

```
import 'rxjs/add/operator/retry';
```

Then, you can use it with HTTP Observables like this:

```
http
  .get<ItemsResponse>('/api/items')
  // Retry this request up to 3 times.
  .retry(3)
  // Any errors after the 3rd retry will fall through to the app.
  .subscribe(...);
```

Requesting non-JSON data

Not all APIs return JSON data. Suppose you want to read a text file on the server. You have to tell `HttpClient` that you expect a textual response:

```
http
  .get('/textfile.txt', {responseType: 'text'})
  // 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().
  .subscribe(data => console.log(data));
```

Sending data to the server

In addition to fetching data from the server, `HttpClient` supports mutating requests, that is, sending data to the server in various forms.

Making a POST request

One common operation is to POST data to a server; for example when submitting a form. The code for sending a POST request is very similar to the code for GET:

```
const body = {name: 'Brad'};

http
  .post('/api/developers/add', body)
  // See below - subscribe() is still necessary when using post().
  .subscribe(...);
```

Note the `subscribe()` method. All Observables returned from `HttpClient` are _cold_, which is to say that they are _blueprints_ for making requests. Nothing will happen until you call `subscribe()`, and every such call will make a separate request. For example, this code sends a POST request with the same data twice: ```javascript const req = http.post('/api/items/add', body); // 0 requests made - .subscribe() not called.

req.subscribe(); // 1 request made. req.subscribe(); // 2 requests made. ```

Configuring other parts of the request

Besides the URL and a possible request body, there are other aspects of an outgoing request which you may wish to configure. All of these are available via an options object, which you pass to the request.

Headers

One common task is adding an `Authorization` header to outgoing requests. Here's how you do that:

```
http
  .post('/api/items/add', body, {
    headers: new HttpHeaders().set('Authorization', 'my-auth-token'),
  })
  .subscribe();
```

The `HttpHeaders` class is immutable, so every `set()` returns a new instance and applies the changes.

URL Parameters

Adding URL parameters works in the same way. To send a request with the `id` parameter set to `3`, you would do:

```
http
  .post('/api/items/add', body, {
    params: new HttpParams().set('id', '3'),
  })
  .subscribe();
```

In this way, you send the POST request to the URL `/api/items/add?id=3`.

Advanced usage

The above sections detail how to use the basic HTTP functionality in `@angular/common/http`, but sometimes you need to do more than just make requests and get data back.

Intercepting all requests or responses

A major feature of `@angular/common/http` is *interception*, the ability to declare interceptors which sit in between your application and the backend. When your application makes a request, interceptors transform it

before sending it to the server, and the interceptors can transform the response on its way back before your application sees it. This is useful for everything from authentication to logging.

Writing an interceptor

To implement an interceptor, you declare a class that implements `HttpInterceptor`, which has a single `intercept()` method. Here is a simple interceptor which does nothing but forward the request through without altering it:

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

@Injectable()
export class NoopInterceptor implements HttpInterceptor {
  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    return next.handle(req);
  }
}
```

`intercept` is a method which transforms a request into an Observable that eventually returns the response. In this sense, each interceptor is entirely responsible for handling the request by itself.

Most of the time, though, interceptors will make some minor change to the request and forward it to the rest of the chain. That's where the `next` parameter comes in. `next` is an `HttpHandler`, an interface that, similar to `intercept`, transforms a request into an Observable for the response. In an interceptor, `next` always represents the next interceptor in the chain, if any, or the final backend if there are no more interceptors. So most interceptors will end by calling `next` on the request they transformed.

Our do-nothing handler simply calls `next.handle` on the original request, forwarding it without mutating it at all.

This pattern is similar to those in middleware frameworks such as Express.js.

Providing your interceptor

Simply declaring the `NoopInterceptor` above doesn't cause your app to use it. You need to wire it up in your app module by providing it as an interceptor, as follows:

```
import {NgModule} from '@angular/core';
import {HTTP_INTERCEPTORS} from '@angular/common/http';

@NgModule({
  providers: [{
    provide: HTTP_INTERCEPTORS,
    useClass: NoopInterceptor,
    multi: true,
  }],
})
export class AppModule {}
```

Note the `multi: true` option. This is required and tells Angular that `HTTP_INTERCEPTORS` is an array of values, rather than a single value.

Events

You may have also noticed that the Observable returned by `intercept` and `HttpHandler.handle` is not an `Observable<HttpResponse<any>>` but an `Observable<HttpEvent<any>>`. That's because interceptors work at a lower level than the `HttpClient` interface. A single request can generate multiple events, including upload and download progress events. The `HttpResponse` class is actually an event itself, with a `type` of `HttpEventType.HttpResponseEvent`.

An interceptor must pass through all events that it does not understand or intend to modify. It must not filter out events it didn't expect to process. Many interceptors are only concerned with the outgoing request, though, and will simply return the event stream from `next` without modifying it.

Ordering

When you provide multiple interceptors in an application, Angular applies them in the order that you provided them.

Immutability

Interceptors exist to examine and mutate outgoing requests and incoming responses. However, it may be surprising to learn that the `HttpRequest` and `HttpResponse` classes are largely immutable.

This is for a reason: because the app may retry requests, the interceptor chain may process an individual request multiple times. If requests were mutable, a retried request would be different than the original request. Immutability ensures the interceptors see the same request for each try.

There is one case where type safety cannot protect you when writing interceptors—the request body. It is

invalid to mutate a request body within an interceptor, but this is not checked by the type system.

If you have a need to mutate the request body, you need to copy the request body, mutate the copy, and then use `clone()` to copy the request and set the new body.

Since requests are immutable, they cannot be modified directly. To mutate them, use `clone()`:

```
intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
  // This is a duplicate. It is exactly the same as the original.
  const dupReq = req.clone();

  // Change the URL and replace 'http://' with 'https://'
  const secureReq = req.clone({url: req.url.replace('http://', 'https://')});
}
```

As you can see, the hash accepted by `clone()` allows you to mutate specific properties of the request while copying the others.

Setting new headers

A common use of interceptors is to set default headers on outgoing responses. For example, assuming you have an injectable `AuthService` which can provide an authentication token, here is how you would write an interceptor which adds it to all outgoing requests:

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

@Injectable()
export class AuthInterceptor implements HttpInterceptor {
  constructor(private auth: AuthService) {}

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    // Get the auth header from the service.
    const authHeader = this.auth.getAuthorizationHeader();
    // Clone the request to add the new header.
    const authReq = req.clone({headers: req.headers.set('Authorization', authHeader)}
);
    // Pass on the cloned request instead of the original request.
    return next.handle(authReq);
  }
}
```

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

```
const authReq = req.clone({setHeaders: {Authorization: authHeader}});
```

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

Logging

Because interceptors can process the request and response *together*, they can do things like log or time requests. Consider this interceptor which uses `console.log` to show how long each request takes:

```
import 'rxjs/add/operator/do';

export class TimingInterceptor implements HttpInterceptor {
  constructor(private auth: AuthService) {}

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    const started = Date.now();
    return next
      .handle(req)
      .do(event => {
        if (event instanceof HttpResponse) {
          const elapsed = Date.now() - started;
          console.log(`Request for ${req.urlWithParams} took ${elapsed} ms.`);
        }
      });
  }
}
```

Notice the RxJS `do()` operator—it adds a side effect to an Observable without affecting the values on the stream. Here, it detects the `HttpResponse` event and logs the time the request took.

Caching

You can also use interceptors to implement caching. For this example, assume that you've written an HTTP cache with a simple interface:

```
abstract class HttpCache {
  /**
   * Returns a cached response, if any, or null if not present.
   */
  abstract get(req: HttpRequest<any>): HttpResponse<any>|null;

  /**
   * Adds or updates the response in the cache.
   */
  abstract put(req: HttpRequest<any>, resp: HttpResponse<any>): void;
}
```

An interceptor can apply this cache to outgoing requests.

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

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    // Before doing anything, it's important to only cache GET requests.
    // Skip this interceptor if the request method isn't GET.
    if (req.method !== 'GET') {
      return next.handle(req);
    }

    // First, check the cache to see if this request exists.
    const cachedResponse = this.cache.get(req);
    if (cachedResponse) {
      // A cached response exists. Serve it instead of forwarding
      // the request to the next handler.
      return Observable.of(cachedResponse);
    }

    // No cached response exists. Go to the network, and cache
    // the response when it arrives.
    return next.handle(req).do(event => {
      // Remember, there may be other events besides just the response.
      if (event instanceof HttpResponse) {
        // Update the cache.
        this.cache.put(req, event);
      }
    });
  }
}
```

Obviously this example glosses over request matching, cache invalidation, etc., but it's easy to see that interceptors have a lot of power beyond just transforming requests. If desired, they can be used to completely take over the request flow.

To really demonstrate their flexibility, you can change the above example to return *two* response events if the request exists in cache—the cached response first, and an updated network response later.

```
intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
  // Still skip non-GET requests.
  if (req.method !== 'GET') {
    return next.handle(req);
  }

  // This will be an Observable of the cached value if there is one,
  // or an empty Observable otherwise. It starts out empty.
  let maybeCachedResponse: Observable<HttpEvent<any>> = Observable.empty();

  // Check the cache.
  const cachedResponse = this.cache.get(req);
  if (cachedResponse) {
    maybeCachedResponse = Observable.of(cachedResponse);
  }

  // Create an Observable (but don't subscribe) that represents making
  // the network request and caching the value.
  const networkResponse = next.handle(req).do(event => {
    // Just like before, check for the HttpResponse event and cache it.
    if (event instanceof HttpResponse) {
      this.cache.put(req, event);
    }
  });

  // Now, combine the two and send the cached response first (if there is
  // one), and the network response second.
  return Observable.concat(maybeCachedResponse, networkResponse);
}
```

Now anyone doing `http.get(url)` will receive *two* responses if that URL has been cached before.

Listening to progress events

Sometimes applications need to transfer large amounts of data, and those transfers can take time. It's a good user experience practice to provide feedback on the progress of such transfers; for example, uploading files—

and `@angular/common/http` supports this.

To make a request with progress events enabled, first create an instance of `HttpRequest` with the special `reportProgress` option set:

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

This option enables tracking of progress events. Remember, every progress event triggers change detection, so only turn them on if you intend to actually update the UI on each event.

Next, make the request through the `request()` method of `HttpClient`. The result will be an Observable of events, just like with interceptors:

```
http.request(req).subscribe(event => {
  // Via this API, you get access to the raw event stream.
  // Look for upload progress events.
  if (event.type === HttpEventType.UploadProgress) {
    // This is an upload progress event. Compute and show the % done:
    const percentDone = Math.round(100 * event.loaded / event.total);
    console.log(`File is ${percentDone}% uploaded.`);
  } else if (event instanceof HttpResponse) {
    console.log('File is completely uploaded!');
  }
});
```

Security: XSRF Protection

Cross-Site Request Forgery (XSRF) 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 cookie on all mutating requests (POST, etc.) 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.

Note that `HttpClient`'s support is 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. If not, Angular's default protection will be ineffective.

Configuring custom cookie/header names

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

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

Testing HTTP requests

Like any external dependency, the HTTP backend needs to be mocked as part of good testing practice. `@angular/common/http` provides a testing library `@angular/common/http/testing` that makes setting up such mocking straightforward.

Mocking philosophy

Angular's HTTP testing library is designed for a pattern of testing where the app executes code and makes requests first. After that, tests expect that certain requests have or have not been made, perform assertions against those requests, and finally provide responses by "flushing" each expected request, which may trigger more new requests, etc. At the end, tests can optionally verify that the app has made no unexpected requests.

Setup

To begin testing requests made through `HttpClient`, import `HttpClientTestingModule` and add it to your `TestBed` setup, like so:

```
import {HttpClientTestingModule} from '@angular/common/http/testing';

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

That's it. Now requests made in the course of your tests will hit the testing backend instead of the normal backend.

Expecting and answering requests

With the mock installed via the module, you can write a test that expects a GET Request to occur and provides a mock response. The following example does this by injecting both the `HttpClient` into the test and a class called `HttpTestingController`

```
it('expects a GET request', inject([HttpClient, HttpTestingController], (http: HttpCl
ient, httpMock: HttpTestingController) => {
  // Make an HTTP GET request, and expect that it return an object
  // of the form {name: 'Test Data'}.
  http
    .get('/data')
    .subscribe(data => expect(data['name']).toEqual('Test Data'));

  // At this point, the request is pending, and no response has been
  // sent. The next step is to expect that the request happened.
  const req = httpMock.expectOne('/data');

  // If no request with that URL was made, or if multiple requests match,
  // expectOne() would throw. However this test makes only one request to
  // this URL, so it will match and return a mock request. The mock request
  // can be used to deliver a response or make assertions against the
  // request. In this case, the test asserts that the request is a GET.
  expect(req.request.method).toEqual('GET');

  // Next, fulfill the request by transmitting a response.
  req.flush({name: 'Test Data'});

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

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

```
afterEach(inject([HttpTestingController], (httpMock: HttpTestingController) => {
  httpMock.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:

```
const req = httpMock.expectOne((req) => req.headers.has('Authorization'));
```

Just as with the `expectOne()` by URL in the test above, if 0 or 2+ requests match this expectation, it will throw.

Handling more than one request

If you need to respond to duplicate requests in your test, use the `match()` API instead of `expectOne()`, which takes the same arguments but returns an array of matching requests. Once returned, these requests are removed from future matching and are your responsibility to verify and flush.

```
// Expect that 5 pings have been made and flush them.
const reqs = httpMock.match('/ping');
expect(reqs.length).toBe(5);
reqs.forEach(req => req.flush());
```