Communicating with backend services using HTTP

Most front-end applications need to communicate with a server over the HTTP protocol, to download or upload data and access other back-end services. Angular provides a 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`.

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

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.

You can run the 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 is used to configure the request.

```
options: {
    headers?: HttpHeaders | {[header: string]: string | string[]},
    observe?: 'body' | 'events' | 'response',
    params?: HttpParams|{[param: string]: string | number | boolean |
ReadonlyArray<string | number | boolean>},
```

```
  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.

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.

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.

{@a config-service}

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.

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.

{@a typed-response}

Requesting a typed response

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 actually responds 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.

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

When you pass an interface as a type parameter to the `HttpClient.get()` method, 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:

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, });
{@a string-union-types}

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:

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:

<code-example path="http/src/app/config/config.component.ts" region="showConfigResponse" header="app/config/config.component.ts (showConfigResponse)"

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>`.

`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.

{@a error-handling}

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.

{@a error-details}

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 have `status` set to `0` and the `error` property contains a `ProgressEvent` object, whose `type` might provide further information.

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

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

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.

{@a retry}

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 to pipe a failed request to the `retry()` operator before passing it to the error handler.

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 an abridged 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.

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.

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.

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

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.

{@a always-subscribe} **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*, letting you 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.

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.

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.

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 expires, you can update the authorization header before making the next request.

{@a url-params}

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";

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'});

{@a intercepting-requests-and-responses}

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 passes the request through without touching it:

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 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:

Note the `multi: true` option. This required setting tells Angular that `HTTP_INTERCEPTORS` is a token for a *multiprovider* 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`.

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

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. For example, consider a situation in which you want to handle the authentication of your HTTP requests and log them before sending them to a server. To accomplish this task, you could provide an `AuthInterceptor` service and then a `LoggingInterceptor` service. Outgoing requests would flow from the `AuthInterceptor` to the `LoggingInterceptor`. Responses from these requests would flow in the other direction, from `LoggingInterceptor` back to `AuthInterceptor`. The following is a visual representation of the process:

Interceptor in order of HttpClient, AuthInterceptor, AuthInterceptor, HttpBackend, Server, and back in opposite order to show the two-way flow

The last interceptor in the process is always the `HttpBackend` that handles communication with the server.

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.

{@a interceptor-events}

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.

{@a immutability}

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.

The `clone()` method's hash argument lets you 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.

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
```

Http interceptor use-cases

Following are a number of common uses for interceptors.

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:

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

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 request and response pairs

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`.

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.

{@a custom-json-parser}

Custom JSON parsing

Interceptors can be used to replace the built-in JSON parsing with a custom implementation.

The `CustomJsonInterceptor` in the following example demonstrates how to achieve this. If the intercepted request expects a `'json'` response, the `responseType` is changed to `'text'` to disable the built-in JSON parsing. Then the response is parsed via the injected `JsonParser`.

You can then implement your own custom `JsonParser`. Here is a custom JsonParser that has a special date reviver.

You provide the `CustomParser` along with the `CustomJsonInterceptor`.

{@a caching}

Caching requests

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.

- The `isCacheable()` function determines if the request is cacheable. In this sample, only GET requests to the package search API are cacheable.
- If the request is not cacheable, the interceptor 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 forwards the request to `next.handle()` which ultimately calls the server and returns the server's response.

{@a send-request}

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.

{@a cache-refresh}

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 package search API, and emits again later with the updated search results.

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.

{@a report-progress}

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, create an instance of `HttpRequest` with the `reportProgress` option set true to enable tracking of progress events.

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]()).

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

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 a 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 package search API.

Here, the `keyup` event binding sends every keystroke to the component's `search()` method.

The type of `$event.target` is only `EventTarget` in the template. In the `getValue()` method, the target is cast to an `HTMLInputElement` to let type-safe have access to its `value` property.

The following snippet implements debouncing for this input using RxJS operators.

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 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`. Because 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.

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.

{@a testing-requests}

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 made no unexpected requests.

You can run these sample tests 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.

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

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.

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

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:

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.

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.

Alternatively, call `request.error()` with a `ProgressEvent`.

Passing metadata to interceptors

Many interceptors require or benefit from configuration. Consider an interceptor that retries failed requests. By default, the interceptor might retry a request three times, but you might want to override this retry count for particularly error-prone or sensitive requests.

`HttpClient` requests contain a *context* that can carry metadata about the request. This context is available for interceptors to read or modify, though it is not transmitted to the backend server when the request is sent. This lets applications or other interceptors tag requests with configuration parameters, such as how many times to retry a request.

Creating a context token

Angular stores and retrieves a value in the context using an `HttpContextToken`. You can create a context token using the `new` operator, as in the following example:

The lambda function `() => 3` passed during the creation of the `HttpContextToken` serves two purposes:

1. It lets TypeScript infer the type of this token: `HttpContextToken<number>`. The request context is type-safe—reading a token from a request's context returns a value of the appropriate type.

2. It sets the default value for the token. This is the value that the request context returns if no other value was set for this token. Using a default value avoids the need to check if a particular value is set.

Setting context values when making a request

When making a request, you can provide an `HttpContext` instance, in which you have already set the context values.

Reading context values in an interceptor

Within an interceptor, you can read the value of a token in a given request's context with `HttpContext.get()`. If you have not explicitly set a value for the token, Angular returns the default value specified in the token.

Contexts are mutable

Unlike most other aspects of `HttpRequest` instances, the request context is mutable and persists across other immutable transformations of the request. This lets interceptors coordinate operations through the context. For instance, the `RetryInterceptor` example could use a second context token to track how many errors occur during the execution of a given request: