

Get data from a server

In this tutorial, you'll add the following data persistence features with help from Angular's `HttpClient` .

- The `HeroService` gets hero data with HTTP requests.
- Users can add, edit, and delete heroes and save these changes over HTTP.
- Users can search for heroes by name.

For the sample application that this page describes, see the .

Enable HTTP services

`HttpClient` is Angular's mechanism for communicating with a remote server over HTTP.

Make `HttpClient` available everywhere in the application in two steps. First, add it to the root `AppModule` by importing it:

Next, still in the `AppModule` , add `HttpClientModule` to the `imports` array:

Simulate a data server

This tutorial sample mimics communication with a remote data server by using the [In-memory Web API](#) module.

After installing the module, the application will make requests to and receive responses from the `HttpClient` without knowing that the *In-memory Web API* is intercepting those requests, applying them to an in-memory data store, and returning simulated responses.

By using the In-memory Web API, you won't have to set up a server to learn about `HttpClient` .

Important: the In-memory Web API module has nothing to do with HTTP in Angular.

If you're reading this tutorial to learn about `HttpClient` , you can [skip over](#) this step. If you're coding along with this tutorial, stay here and add the In-memory Web API now.

Install the In-memory Web API package from npm with the following command:

```
npm install angular-in-memory-web-api --save
```

In the `AppModule` , import the `HttpClientInMemoryWebApiModule` and the `InMemoryDataService` class, which you will create in a moment.

After the `HttpClientModule` , add the `HttpClientInMemoryWebApiModule` to the `AppModule` `imports` array and configure it with the `InMemoryDataService` .

The `forRoot()` configuration method takes an `InMemoryDataService` class that primes the in-memory database.

Generate the class `src/app/in-memory-data.service.ts` with the following command:

```
ng generate service InMemoryData
```

Replace the default contents of `in-memory-data.service.ts` with the following:

The `in-memory-data.service.ts` file will take over the function of `mock-heroes.ts` . However, don't delete `mock-heroes.ts` yet, as you still need it for a few more steps of this tutorial.

When the server is ready, you'll detach the In-memory Web API, and the application's requests will go through to the server.

```
{@a import-heroes}
```

Heroes and HTTP

In the `HeroService`, import `HttpClient` and `HttpHeaders`:

Still in the `HeroService`, inject `HttpClient` into the constructor in a private property called `http`.

Notice that you keep injecting the `MessageService` but since you'll call it so frequently, wrap it in a private `log()` method:

Define the `heroesUrl` of the form `:base/:collectionName` with the address of the heroes resource on the server. Here `base` is the resource to which requests are made, and `collectionName` is the heroes data object in the `in-memory-data-service.ts`.

Get heroes with `HttpClient`

The current `HeroService.getHeroes()` uses the RxJS `of()` function to return an array of mock heroes as an `Observable<Hero[]>`.

Convert that method to use `HttpClient` as follows:

Refresh the browser. The hero data should successfully load from the mock server.

You've swapped `of()` for `http.get()` and the application keeps working without any other changes because both functions return an `Observable<Hero[]>`.

`HttpClient` methods return one value

All `HttpClient` methods return an RxJS `Observable` of something.

HTTP is a request/response protocol. You make a request, it returns a single response.

In general, an observable *can* return multiple values over time. An observable from `HttpClient` always emits a single value and then completes, never to emit again.

This particular `HttpClient.get()` call returns an `Observable<Hero[]>`; that is, "*an observable of hero arrays*". In practice, it will only return a single hero array.

`HttpClient.get()` returns response data

`HttpClient.get()` returns the body of the response as an untyped JSON object by default. Applying the optional type specifier, `<Hero[]>`, adds TypeScript capabilities, which reduce errors during compile time.

The server's data API determines the shape of the JSON data. The *Tour of Heroes* data API returns the hero data as an array.

Other APIs may bury the data that you want within an object. You might have to dig that data out by processing the `Observable` result with the RxJS `map()` operator.

Although not discussed here, there's an example of `map()` in the `getHeroNo404()` method included in the sample source code.

Error handling

Things go wrong, especially when you're getting data from a remote server. The `HeroService.getHeroes()` method should catch errors and do something appropriate.

To catch errors, you "**pipe**" the **observable** result from `http.get()` through an RxJS `catchError()` operator.

Import the `catchError` symbol from `rxjs/operators`, along with some other operators you'll need later.

Now extend the observable result with the `pipe()` method and give it a `catchError()` operator.

The `catchError()` operator intercepts an **Observable that failed**. The operator then passes the error to the error handling function.

The following `handleError()` method reports the error and then returns an innocuous result so that the application keeps working.

handleError

The following `handleError()` will be shared by many `HeroService` methods so it's generalized to meet their different needs.

Instead of handling the error directly, it returns an error handler function to `catchError` that it has configured with both the name of the operation that failed and a safe return value.

After reporting the error to the console, the handler constructs a user friendly message and returns a safe value to the application so the application can keep working.

Because each service method returns a different kind of `Observable` result, `handleError()` takes a type parameter so it can return the safe value as the type that the application expects.

Tap into the Observable

The `HeroService` methods will **tap** into the flow of observable values and send a message, using the `log()` method, to the message area at the bottom of the page.

They'll do that with the RxJS `tap()` operator, which looks at the observable values, does something with those values, and passes them along. The `tap()` call back doesn't touch the values themselves.

Here is the final version of `getHeroes()` with the `tap()` that logs the operation.

Get hero by id

Most web APIs support a *get by id* request in the form `:baseUrl/:id`.

Here, the *base URL* is the `heroesURL` defined in the [Heroes and HTTP](#) section (`api/heroes`) and *id* is the number of the hero that you want to retrieve. For example, `api/heroes/11`.

Update the `HeroService` `getHero()` method with the following to make that request:

There are three significant differences from `getHeroes()`:

- `getHero()` constructs a request URL with the desired hero's id.
- The server should respond with a single hero rather than an array of heroes.
- `getHero()` returns an `Observable<Hero>` ("*an observable of Hero objects*") rather than an observable of hero arrays.

Update heroes

Edit a hero's name in the hero detail view. As you type, the hero name updates the heading at the top of the page. But when you click the "go back button", the changes are lost.

If you want changes to persist, you must write them back to the server.

At the end of the hero detail template, add a save button with a `click` event binding that invokes a new component method named `save()`.

In the `HeroDetail` component class, add the following `save()` method, which persists hero name changes using the hero service `updateHero()` method and then navigates back to the previous view.

Add `HeroService.updateHero()`

The overall structure of the `updateHero()` method is similar to that of `getHeroes()`, but it uses `http.put()` to persist the changed hero on the server. Add the following to the `HeroService`.

The `HttpClient.put()` method takes three parameters:

- the URL
- the data to update (the modified hero in this case)
- options

The URL is unchanged. The heroes web API knows which hero to update by looking at the hero's `id`.

The heroes web API expects a special header in HTTP save requests. That header is in the `httpOptions` constant defined in the `HeroService`. Add the following to the `HeroService` class.

Refresh the browser, change a hero name and save your change. The `save()` method in `HeroDetailComponent` navigates to the previous view. The hero now appears in the list with the changed name.

Add a new hero

To add a hero, this application only needs the hero's name. You can use an `<input>` element paired with an add button.

Insert the following into the `HeroesComponent` template, after the heading:

In response to a click event, call the component's click handler, `add()`, and then clear the input field so that it's ready for another name. Add the following to the `HeroesComponent` class:

When the given name is non-blank, the handler creates a `Hero`-like object from the name (it's only missing the `id`) and passes it to the service's `addHero()` method.

When `addHero()` saves successfully, the `subscribe()` callback receives the new hero and pushes it into the `heroes` list for display.

Add the following `addHero()` method to the `HeroService` class.

`addHero()` differs from `updateHero()` in two ways:

- It calls `HttpClient.post()` instead of `put()`.

- It expects the server to generate an id for the new hero, which it returns in the `Observable<Hero>` to the caller.

Refresh the browser and add some heroes.

Delete a hero

Each hero in the heroes list should have a delete button.

Add the following button element to the `HeroesComponent` template, after the hero name in the repeated `` element.

The HTML for the list of heroes should look like this:

To position the delete button at the far right of the hero entry, add some CSS to the `heroes.component.css`. You'll find that CSS in the [final review code](#) below.

Add the `delete()` handler to the component class.

Although the component delegates hero deletion to the `HeroService`, it remains responsible for updating its own list of heroes. The component's `delete()` method immediately removes the *hero-to-delete* from that list, anticipating that the `HeroService` will succeed on the server.

There's really nothing for the component to do with the `Observable` returned by `heroService.delete()` **but it must subscribe anyway**.

If you neglect to `subscribe()`, the service will not send the delete request to the server. As a rule, an `Observable` *does nothing* until something subscribes.

Confirm this for yourself by temporarily removing the `subscribe()`, clicking "Dashboard", then clicking "Heroes". You'll see the full list of heroes again.

Next, add a `deleteHero()` method to `HeroService` like this.

Note the following key points:

- `deleteHero()` calls `HttpClient.delete()`.
- The URL is the heroes resource URL plus the `id` of the hero to delete.
- You don't send data as you did with `put()` and `post()`.
- You still send the `httpOptions`.

Refresh the browser and try the new delete functionality.

Search by name

In this last exercise, you learn to chain `Observable` operators together so you can minimize the number of similar HTTP requests and consume network bandwidth economically.

You will add a heroes search feature to the Dashboard. As the user types a name into a search box, you'll make repeated HTTP requests for heroes filtered by that name. Your goal is to issue only as many requests as necessary.

```
HeroService.searchHeroes()
```

Start by adding a `searchHeroes()` method to the `HeroService`.

The method returns immediately with an empty array if there is no search term. The rest of it closely resembles `getHeroes()`, the only significant difference being the URL, which includes a query string with the search term.

Add search to the Dashboard

Open the `DashboardComponent` template and add the hero search element, `<app-hero-search>`, to the bottom of the markup.

This template looks a lot like the `*ngFor` repeater in the `HeroesComponent` template.

For this to work, the next step is to add a component with a selector that matches `<app-hero-search>`.

Create `HeroSearchComponent`

Create a `HeroSearchComponent` with the CLI.

ng generate component hero-search

The CLI generates the three `HeroSearchComponent` files and adds the component to the `AppModule` declarations.

Replace the generated `HeroSearchComponent` template with an `<input>` and a list of matching search results, as follows.

Add private CSS styles to `hero-search.component.css` as listed in the [final code review](#) below.

As the user types in the search box, an input event binding calls the component's `search()` method with the new search box value.

```
{@a asyncpipe}
```

AsyncPipe

The `*ngFor` repeats hero objects. Notice that the `*ngFor` iterates over a list called `heroes$`, not `heroes`. The `$` is a convention that indicates `heroes$` is an `Observable`, not an array.

Since `*ngFor` can't do anything with an `Observable`, use the pipe character (`|`) followed by `async`. This identifies Angular's `AsyncPipe` and subscribes to an `Observable` automatically so you won't have to do so in the component class.

Edit the `HeroSearchComponent` class

Replace the generated `HeroSearchComponent` class and metadata as follows.

Notice the declaration of `heroes$` as an `Observable`:

You'll set it in [ngOnInit\(\)](#). Before you do, focus on the definition of `searchTerms`.

The `searchTerms` RxJS subject

The `searchTerms` property is an RxJS `Subject`.

A `Subject` is both a source of observable values and an `Observable` itself. You can subscribe to a `Subject` as you would any `Observable`.

You can also push values into that `Observable` by calling its `next(value)` method as the `search()` method does.

The event binding to the textbox's `input` event calls the `search()` method.

Every time the user types in the textbox, the binding calls `search()` with the textbox value, a "search term". The `searchTerms` becomes an `Observable` emitting a steady stream of search terms.

```
{@a search-pipe}
```

Chaining RxJS operators

Passing a new search term directly to the `searchHeroes()` after every user keystroke would create an excessive amount of HTTP requests, taxing server resources and burning through data plans.

Instead, the `ngOnInit()` method pipes the `searchTerms` observable through a sequence of RxJS operators that reduce the number of calls to the `searchHeroes()`, ultimately returning an observable of timely hero search results (each a `Hero[]`).

Here's a closer look at the code.

Each operator works as follows:

- `debounceTime(300)` waits until the flow of new string events pauses for 300 milliseconds before passing along the latest string. You'll never make requests more frequently than 300ms.
- `distinctUntilChanged()` ensures that a request is sent only if the filter text changed.
- `switchMap()` calls the search service for each search term that makes it through `debounce()` and `distinctUntilChanged()`. It cancels and discards previous search observables, returning only the latest search service observable.

With the [switchMap operator](#), every qualifying key event can trigger an `HttpClient.get()` method call. Even with a 300ms pause between requests, you could have multiple HTTP requests in flight and they may not return in the order sent.


`switchMap()` preserves the original request order while returning only the observable from the most recent HTTP method call. Results from prior calls are canceled and discarded.

Note that canceling a previous `searchHeroes()` Observable doesn't actually abort a pending HTTP request. Unwanted results are discarded before they reach your application code.

Remember that the component *class* does not subscribe to the `heroes$ observable`. That's the job of the [AsyncPipe](#) in the template.

Try it

Run the application again. In the *Dashboard*, enter some text in the search box. If you enter characters that match any existing hero names, you'll see something like this.

 Hero Search field with the letters 'm' and 'a' along with four search results that match the query displayed in a list beneath the search input

Final code review

Here are the code files discussed on this page (all in the `src/app/` folder).

```
{@a heroservice} {@a inmemorydataservice} {@a appmodule}
```

```
HeroService , InMemoryDataService , AppModule
```

```
{@a heroescomponent}
```

```
HeroesComponent
```

```
{@a herodetailcomponent}
```

```
HeroDetailComponent
```

```
{@a dashboardcomponent}
```

```
DashboardComponent
```

```
{@a herosearchcomponent}
```

```
HeroSearchComponent
```

Summary

You're at the end of your journey, and you've accomplished a lot.

- You added the necessary dependencies to use HTTP in the app.
- You refactored `HeroService` to load heroes from a web API.
- You extended `HeroService` to support `post()` , `put()` , and `delete()` methods.
- You updated the components to allow adding, editing, and deleting of heroes.
- You configured an in-memory web API.
- You learned how to use observables.

This concludes the "Tour of Heroes" tutorial. You're ready to learn more about Angular development in the fundamentals section, starting with the [Architecture](#) guide.