

Add services

The Tour of Heroes **HeroesComponent** is currently getting and displaying fake data.

After the refactoring in this tutorial, **HeroesComponent** will be lean and focused on supporting the view. It will also be easier to unit-test with a mock service.

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

Why services

Components shouldn't fetch or save data directly and they certainly shouldn't knowingly present fake data. They should focus on presenting data and delegate data access to a service.

In this tutorial, you'll create a **HeroService** that all application classes can use to get heroes. Instead of creating that service with the **new** keyword, you'll rely on Angular *dependency injection* to inject it into the **HeroesComponent** constructor.

Services are a great way to share information among classes that *don't know each other*. You'll create a **MessageService** and inject it in two places.

1. Inject in **HeroService**, which uses the service to send a message.
2. Inject in **MessagesComponent**, which displays that message, and also displays the ID when the user clicks a hero.

Create the HeroService

Using the Angular CLI, create a service called **hero**.

ng generate service hero

The command generates a skeleton **HeroService** class in **src/app/hero.service.ts** as follows:

@Injectable() services

Notice that the new service imports the Angular **Injectable** symbol and annotates the class with the **@Injectable()** decorator. This marks the class as one that participates in the *dependency injection system*. The **HeroService** class is going to provide an injectable service, and it can also have its own injected dependencies. It doesn't have any dependencies yet, but it will soon.

The **@Injectable()** decorator accepts a metadata object for the service, the same way the **@Component()** decorator did for your component classes.

Get hero data

The **HeroService** could get hero data from anywhere—a web service, local storage, or a mock data source.

Removing data access from components means you can change your mind about the implementation anytime, without touching any components. They don't know how the service works.

The implementation in *this* tutorial will continue to deliver *mock heroes*.

Import the **Hero** and **HEROES**.

Add a **getHeroes** method to return the *mock heroes*.

```
{@a provide} ## Provide the HeroService
```

You must make the **HeroService** available to the dependency injection system before Angular can *inject* it into the **HeroesComponent** by registering a *provider*. A provider is something that can create or deliver a service; in this case, it instantiates the **HeroService** class to provide the service.

To make sure that the **HeroService** can provide this service, register it with the *injector*, which is the object that is responsible for choosing and injecting the provider where the application requires it.

By default, the Angular CLI command **ng generate service** registers a provider with the *root injector* for your service by including provider metadata, that is **providedIn: 'root'** in the **@Injectable()** decorator.

```
@Injectable({  
  providedIn: 'root',  
})
```

When you provide the service at the root level, Angular creates a single, shared instance of **HeroService** and injects into any class that asks for it. Registering the provider in the **@Injectable** metadata also allows Angular to optimize an application by removing the service if it turns out not to be used after all.

To learn more about providers, see the Providers section. To learn more about injectors, see the Dependency Injection guide.

The **HeroService** is now ready to plug into the **HeroesComponent**.

This is an interim code sample that will allow you to provide and use the **HeroService**. At this point, the code will differ from the **HeroService** in the “final code review”.

Update HeroesComponent

Open the **HeroesComponent** class file.

Delete the `HEROES` import, because you won't need that anymore. Import the `HeroService` instead.

Replace the definition of the `heroes` property with a declaration.

```
{@a inject}
```

Inject the HeroService

Add a private `heroService` parameter of type `HeroService` to the constructor.

The parameter simultaneously defines a private `heroService` property and identifies it as a `HeroService` injection site.

When Angular creates a `HeroesComponent`, the Dependency Injection system sets the `heroService` parameter to the singleton instance of `HeroService`.

Add getHeroes()

Create a method to retrieve the heroes from the service.

```
{@a oninit}
```

Call it in ngOnInit()

While you could call `getHeroes()` in the constructor, that's not the best practice.

Reserve the constructor for minimal initialization such as wiring constructor parameters to properties. The constructor shouldn't *do anything*. It certainly shouldn't call a function that makes HTTP requests to a remote server as a *real* data service would.

Instead, call `getHeroes()` inside the *ngOnInit lifecycle hook* and let Angular call `ngOnInit()` at an appropriate time *after* constructing a `HeroesComponent` instance.

See it run

After the browser refreshes, the application should run as before, showing a list of heroes and a hero detail view when you click on a hero name.

Observable data

The `HeroService.getHeroes()` method has a *synchronous signature*, which implies that the `HeroService` can fetch heroes synchronously. The `HeroesComponent` consumes the `getHeroes()` result as if heroes could be fetched synchronously.

This will not work in a real application. You're getting away with it now because the service currently returns *mock heroes*. But soon the application will fetch heroes from a remote server, which is an inherently *asynchronous* operation.

The `HeroService` must wait for the server to respond, `getHeroes()` cannot return immediately with hero data, and the browser will not block while the service waits.

`HeroService.getHeroes()` must have an *asynchronous signature* of some kind.

In this tutorial, `HeroService.getHeroes()` will return an `Observable` because it will eventually use the Angular `HttpClient.get` method to fetch the heroes and `HttpClient.get()` returns an `Observable`.

Observable HeroService

`Observable` is one of the key classes in the RxJS library.

In a later tutorial on HTTP, you'll learn that Angular's `HttpClient` methods return RxJS `Observables`. In this tutorial, you'll simulate getting data from the server with the RxJS `of()` function.

Open the `HeroService` file and import the `Observable` and `of` symbols from RxJS.

Replace the `getHeroes()` method with the following:

`of(HEROES)` returns an `Observable<Hero[]>` that emits *a single value*, the array of mock heroes.

In the HTTP tutorial, you'll call `HttpClient.get<Hero[]>()` which also returns an `Observable<Hero[]>` that emits *a single value*, an array of heroes from the body of the HTTP response.

Subscribe in HeroesComponent

The `HeroService.getHeroes` method used to return a `Hero[]`. Now it returns an `Observable<Hero[]>`.

You'll have to adjust to that difference in `HeroesComponent`.

Find the `getHeroes` method and replace it with the following code (shown side-by-side with the previous version for comparison)

`Observable.subscribe()` is the critical difference.

The previous version assigns an array of heroes to the component's `heroes` property. The assignment occurs *synchronously*, as if the server could return heroes instantly or the browser could freeze the UI while it waited for the server's response.

That *won't work* when the `HeroService` is actually making requests of a remote server.

The new version waits for the `Observable` to emit the array of heroes—which could happen now or several minutes from now. The `subscribe()` method

passes the emitted array to the callback, which sets the component's `heroes` property.

This asynchronous approach *will work* when the `HeroService` requests heroes from the server.

Show messages

This section guides you through the following:

- adding a `MessagesComponent` that displays application messages at the bottom of the screen
- creating an injectable, app-wide `MessageService` for sending messages to be displayed
- injecting `MessageService` into the `HeroService`
- displaying a message when `HeroService` fetches heroes successfully

Create MessagesComponent

Use the CLI to create the `MessagesComponent`.

```
ng generate component messages
```

The CLI creates the component files in the `src/app/messages` folder and declares the `MessagesComponent` in `AppModule`.

Modify the `AppComponent` template to display the generated `MessagesComponent`.

You should see the default paragraph from `MessagesComponent` at the bottom of the page.

Create the MessageService

Use the CLI to create the `MessageService` in `src/app`.

```
ng generate service message
```

Open `MessageService` and replace its contents with the following.

The service exposes its cache of `messages` and two methods: one to `add()` a message to the cache and another to `clear()` the cache.

```
{@a inject-message-service} ### Inject it into the HeroService
```

In `HeroService`, import the `MessageService`.

Modify the constructor with a parameter that declares a private `messageService` property. Angular will inject the singleton `MessageService` into that property when it creates the `HeroService`.

This is a typical “*service-in-service*” scenario: you inject the `MessageService` into the `HeroService` which is injected into the `HeroesComponent`.

Send a message from HeroService

Modify the `getHeroes()` method to send a message when the heroes are fetched.

Display the message from HeroService

The `MessagesComponent` should display all messages, including the message sent by the `HeroService` when it fetches heroes.

Open `MessagesComponent` and import the `MessageService`.

Modify the constructor with a parameter that declares a **public** `messageService` property. Angular will inject the singleton `MessageService` into that property when it creates the `MessagesComponent`.

The `messageService` property **must be public** because you're going to bind to it in the template.

Angular only binds to *public* component properties.

Bind to the MessageService

Replace the CLI-generated `MessagesComponent` template with the following.

This template binds directly to the component's `messageService`.

- The `*ngIf` only displays the messages area if there are messages to show.
- An `*ngFor` presents the list of messages in repeated `<div>` elements.
- An Angular event binding binds the button's click event to `MessageService.clear()`.

The messages will look better when you add the private CSS styles to `messages.component.css` as listed in one of the “final code review” tabs below.

Add additional messages to hero service

The following example shows how to send and display a message each time the user clicks on a hero, showing a history of the user's selections. This will be helpful when you get to the next section on Routing.

Refresh the browser to see the list of heroes, and scroll to the bottom to see the messages from the `HeroService`. Each time you click a hero, a new message appears to record the selection. Use the **Clear messages** button to clear the message history.

{@a final-code-review}

Final code review

Here are the code files discussed on this page.

Summary

- You refactored data access to the **HeroService** class.
- You registered the **HeroService** as the *provider* of its service at the root level so that it can be injected anywhere in the application.
- You used Angular Dependency Injection to inject it into a component.
- You gave the **HeroService** *get data* method an asynchronous signature.
- You discovered **Observable** and the RxJS *Observable* library.
- You used RxJS `of()` to return an observable of mock heroes (**Observable<Hero[]>**).
- The component's `ngOnInit` lifecycle hook calls the **HeroService** method, not the constructor.
- You created a **MessageService** for loosely-coupled communication between classes.
- The **HeroService** injected into a component is created with another injected service, **MessageService**.