SERVICES

We create a reusable service to manage our hero data calls

Services

The Tour of Heroes is evolving and we anticipate adding more components in the near future.

Multiple components will need access to hero data and we don't want to copy and paste the same code over and over. Instead, we'll create a single reusable data service and learn to inject it in the components that need it.

Refactoring data access to a separate service keeps the component lean and focused on supporting the view. It also makes it easier to unit test the component with a mock service.

Because data services are invariably asynchronous, we'll finish the chapter with a promise-based version of the data service.

Run the live example for part 4

Where We Left Off

Before we continue with our Tour of Heroes, let’s verify we have the following structure. If not, we’ll need to go back and follow the previous chapters.

```
angular2-tour-of-heroes
  app
    app.component.ts
    hero.ts
    hero-detail.component.ts
    main.ts
  node_modules ...
  typings ...
  index.html
  package.json
  tsconfig.json
  typings.json
```

Keep the app transpiling and running

Open a terminal/console window. Start the TypeScript compiler, watch for changes, and start our server by entering the command:

```
npm start
```

The application runs and updates automatically as we continue to build the Tour of Heroes.

Creating a Hero Service

Our stakeholders have shared their larger vision for our app. They tell us they want to show the heroes in various ways on different pages. We already can select a hero from a list. Soon we'll add a dashboard with the top performing heroes and create a separate view for editing hero details. All three views need hero data.

At the moment the `AppComponent` defines mock heroes for display. We have at least two objections. First, defining heroes is not the component's job. Second, we can't easily share that list of heroes with other components and views.

We can refactor this hero data acquisition business to a single service that provides heroes and share that service with all components that need heroes.

Create the HeroService

Create a file in the `app` folder called `hero.service.ts`.

> We've adopted a convention in which we spell the name of a service in lowercase followed by `.service`. If the service name were multi-word, we'd spell the base filename with lower dash case (AKA kebab-case). The `SpecialSuperHeroService` would be defined in the `special-super-hero.service.ts` file.

We name the class `HeroService` and export it for others to import.

hero.service.ts (exported class)

```
import {Injectable} from 'angular2/core';

@Injectable()
export class HeroService {
}
```

Injectable Services

Notice that we imported the Angular `Injectable` function and applied that function as an `@Injectable()` decorator.

Don't forget the parentheses! Neglecting them leads to an error that's difficult to diagnose.

TypeScript sees the `@Injectable()` decorator and emits metadata about our service, metadata that Angular may need to inject other dependencies into this service.

The `HeroService` doesn't have any dependencies *at the moment*. Add the decorator anyway. It is a "best practice" to apply the `@Injectable()` decorator *from the start* both for consistency and for future-proofing.

Getting Heroes

Add a `getHeroes` method stub.

hero.service.ts (getHeroes stub)

```
@Injectable()
export class HeroService {
  getHeroes() {
  }
}
```

We're holding back on the implementation for a moment to make an important point.

The consumer of our service doesn't know how the service gets the data. Our `HeroService` could get `Hero` data from anywhere. It could get the data from a web service or local storage or from a mock data source.

That's the beauty of removing data access from the component. We can change our minds about the implementation as often as we like, for whatever reason, without touching any of the components that need heroes.

Mock Heroes

We already have mock `Hero` data sitting in the `AppComponent`. It doesn't belong there. It doesn't belong *here* either. We'll move the mock data to its own file.

Cut the `HEROES` array from `app.component.ts` and paste it to a new file in the `app` folder named `mock-heroes.ts`. We copy the `import {Hero} ...` statement as well because the heroes array uses the `Hero` interface.

mock-heroes.ts (Heroes array)

```
import {Hero} from './hero';

export var HEROES: Hero[] = [
    {"id": 11, "name": "Mr. Nice"},
    {"id": 12, "name": "Narco"},
    {"id": 13, "name": "Bombasto"},
    {"id": 14, "name": "Celeritas"},
    {"id": 15, "name": "Magneta"},
    {"id": 16, "name": "RubberMan"},
    {"id": 17, "name": "Dynama"},
    {"id": 18, "name": "Dr IQ"},
    {"id": 19, "name": "Magma"},
    {"id": 20, "name": "Tornado"}
];
```

We export the `HEROES` constant so we can import it elsewhere — such as our `HeroService`.

Meanwhile, back in `app.component.ts` where we cut away the `HEROES` array, we leave behind an uninitialized `heroes` property:

app.component.ts (heroes property)

```
heroes: Hero[];
```

Return Mocked Heroes

Back in the `HeroService` we import the mock `HEROES` and return it from the `getHeroes` method. Our `HeroService` looks like this:

hero.service.ts

```
import {HEROES} from './mock-heroes';
import {Injectable} from 'angular2/core';

@Injectable()
export class HeroService {
  getHeroes() {
    return HEROES;
  }
}
```

Use the Hero Service

We're ready to use the `HeroService` in other components starting with our `AppComponent`.

We begin, as usual, by importing the thing we want to use, the `HeroService`.

app.component.ts (import HeroService)

```
import {HeroService} from './hero.service';
```

Importing the service allows us to *reference* it in our code. How should the `AppComponent` acquire a runtime concrete `HeroService` instance?

Do we *new* the *HeroService*? No way!

We could create a new instance of the `HeroService` with "new" like this:

```
heroService = new HeroService(); // don't do this
```

That's a bad idea for several reasons including

- Our component has to know how to create a `HeroService`. If we ever change the `HeroService` constructor, we'll have to find every place we create the service and fix it. Running around patching code is error prone and adds to the test burden.

- We create a new service each time we use "new". What if the service should cache heroes and share that cache with others? We couldn't do that.

- We're locking the `AppComponent` into a specific implementation of the `HeroService`. It will be hard to switch implementations for different scenarios. Can we operate offline? Will we need different mocked versions under test? Not easy.

What if ... what if ... Hey, we've got work to do!

We get it. Really we do. But it is so ridiculously easy to avoid these problems that there is no excuse for doing it wrong.

Inject the *HeroService*

Two lines replace the one line of *new*:

1. we add a constructor.
2. we add to the component's `providers` metadata

Here's the constructor:

app.component.ts (constructor)

```
constructor(private _heroService: HeroService) { }
```

The constructor itself does nothing. The parameter simultaneously defines a private `_heroService` property and identifies it as a `HeroService` injection site.

> We prefix private variables with an underscore (_) to warn readers of our code that this variable is not part of the component's public API.

Now Angular will know to supply an instance of the `HeroService` when it creates a new `AppComponent`.

Angular has to get that instance from somewhere. That's the role of the Angular *Dependency Injector*. The **Injector** has a **container** of previously created services. Either it finds and returns a pre-existing `HeroService` from its container or it creates a new instance, adds it to the container, and returns it to Angular.

> Learn more about Dependency Injection in the Dependency Injection chapter.

The *injector* does not know yet how to create a `HeroService`. If we ran our code now, Angular would fail with an error:

```
EXCEPTION: No provider for HeroService! (AppComponent ->
    HeroService)
```

We have to teach the *injector* how to make a `HeroService` by registering a `HeroService` **provider**. Do that by adding the following `providers` array property to the bottom of the component metadata in the `@Component` call.

app.component.ts (providing HeroService)

```
providers: [HeroService]
```

The `providers` array tells Angular to create a fresh instance of the `HeroService` when it creates a new `AppComponent`. The `AppComponent` can use that service to get heroes and so can every child component of its component tree.

> ### Services and the component tree
>
> Recall that the `AppComponent` creates an instance of `HeroDetail` by virtue of the `<my-hero-detail>` tag at the bottom of its template. That `HeroDetail` is a child of the `AppComponent`.
>
> If the `HeroDetailComponent` needed its parent component's `HeroService`, it would ask Angular to inject the service into its constructor which would look just like the one for `AppComponent`:
>
> hero-detail.component.ts (constructor)
>
> ```
> constructor(private _heroService: HeroService) { }
> ```
>
> The `HeroDetailComponent` must *not* repeat its parent's `providers` array! Guess why.
>
> The `AppComponent` is the top level component of our application. There should be only one instance of that component and only one instance of the `HeroService` in our entire app.

getHeroes in the *AppComponent*

We've got the service in a `_heroService` private variable. Let's use it.

We pause to think. We can call the service and get the data in one line.

```
this.heroes = this._heroService.getHeroes();
```

We don't really need a dedicated method to wrap one line. We write it anyway:

app.component.ts (getHeroes)

```
  getHeroes() {
    this.heroes = this._heroService.getHeroes();
  }
```

The *ngOnInit* Lifecycle Hook

`AppComponent` should fetch and display heroes without a fuss. Where do we call the `getHeroes` method? In a constructor? We do *not*!

Years of experience and bitter tears have taught us to keep complex logic out of the constructor, especially anything that might call a server as a data access method is sure to do.

The constructor is for simple initializations like wiring constructor parameters to properties. It's not for heavy lifting. We should be able to create a component in a test and not worry that it might do real work — like calling a server! — before we tell it to do so.

If not the constructor, something has to call `getHeroes`.

Angular will call it if we implement the Angular **ngOnInit** *Lifecycle Hook*. Angular offers a number of interfaces for tapping into critical moments in the component lifecycle: at creation, after each change, and at its eventual destruction.

Each interface has a single method. When the component implements that method, Angular calls it at the appropriate time.

> Learn more about lifecycle hooks in the Lifecycle Hooks chapter.

Here's the essential outline for the `OnInit` interface:

app.component.ts (OnInit protocol)

```
import {OnInit} from 'angular2/core';

export class AppComponent implements OnInit {
  ngOnInit() {
  }
}
```

We write an `ngOnInit` method with our initialization logic inside and leave it to Angular to call it at the right time. In our case, we initialize by calling `getHeroes`.

app.component.ts (OnInit protocol)

```
  ngOnInit() {
    this.getHeroes();
  }
```

Our application should be running as expected, showing a list of heroes and a hero detail view when we click on a hero name.

We're getting closer. But something isn't quite right.

Async Services and Promises

Our `HeroService` returns a list of mock heroes immediately. Its `getHeroes` signature is synchronous

```
this.heroes = this._heroService.getHeroes();
```

Ask for heroes and they are there in the returned result.

Someday we're going to get heroes from a remote server. We don’t call http yet, but we aspire to in later chapters.

When we do, we'll have to wait for the server to respond and we won't be able to block the UI while we wait, even if we want to (which we shouldn't) because the browser won't block.

We'll have to use some kind of asynchronous technique and that will change the signature of our `getHeroes` method.

We'll use *promises*.

The Hero Service makes a promise

A **promise** is ... well it's a promise to call us back later when the results are ready. We ask an asynchronous service to do some work and give it a callback function. It does that work (somewhere) and eventually it calls our function with the results of the work or an error.

> We are simplifying. Learn about ES2015 Promises here and elsewhere on the web.

Update the `HeroService` with this promise-returning `getHeroes` method:

hero.service.ts (getHeroes)

```
getHeroes() {
  return Promise.resolve(HEROES);
}
```

We're still mocking the data. We're simulating the behavior of an ultra-fast, zero-latency server, by returning an **immediately resolved promise** with our mock heroes as the result.

Act on the Promise

Returning to the `AppComponent` and its `getHeroes` method, we see that it still looks like this:

app.component.ts (getHeroes - old)

```
getHeroes() {
  this.heroes = this._heroService.getHeroes();
}
```

As a result of our change to `HeroService`, we're now setting `this.heroes` to a promise rather than an array of heroes.

We have to change our implementation to *act on the promise when it resolves*. When the promise resolves successfully, *then* we will have heroes to display.

We pass our callback function as an argument to the promise's **then** method:

app.component.ts (getHeroes - revised)

```
getHeroes() {
  this._heroService.getHeroes().then(heroes => this.heroes =
heroes);
}
```

> The [ES2015 arrow function](#) in the callback is more succinct than the equivalent function expression and gracefully handles *this*.

Our callback sets the component's `heroes` property to the array of heroes returned by the service. That's all there is to it!

Our app should still be running, still showing a list of heroes, and still responding to a name selection with a detail view.

> Checkout the "[Take it slow](#)" appendix to see what the app might be like with a poor connection.

Review the App Structure

Let's verify that we have the following structure after all of our good refactoring in this chapter:

```
angular2-tour-of-heroes
  app
    app.component.ts
    hero.ts
    hero-detail.component.ts
    hero.service.ts
    main.ts
    mock-heroes.ts
  node_modules ...
  typings ...
  index.html
  package.json
  tsconfig.json
  typings.json
```

Here are the code files we discussed in this chapter.

```
import {Hero} from './hero';
import {HEROES} from './mock-heroes';
import {Injectable} from 'angular2/core';

@Injectable()
export class HeroService {
  getHeroes() {
    return Promise.resolve(HEROES);
  }
  // See the "Take it slow" appendix
  getHeroesSlowly() {
    return new Promise<Hero[]>(resolve =>
      setTimeout(()=>resolve(HEROES), 2000) // 2 seconds
    );
  }
}
```

The Road We've Travelled

Let's take stock of what we've built.

- We created a service class that can be shared by many components
- We used the `ngOnInit` Lifecycle Hook to get our heroes when our `AppComponent` activates
- We defined our `HeroService` as a provider for our `AppComponent`
- We created mock hero data and imported them into our service
- We designed our service to return a promise and our component to get our data from the promise

Run the live example for part 4

The Road Ahead

Our Tour of Heroes has become more reusable using shared components and services. We want to create a dashboard, add menu links that route between the views, and format data in a template. As our app evolves, we'll learn how to design it to make it easier to grow and maintain.

We learn about Angular Component Router and navigation among the views in the next tutorial chapter.

Appendix: Take it slow

We can simulate a slow connection.

Import the `Hero` symbol and add the following `getHeroesSlowly` method to the `HeroService`

hero.service.ts (getHeroesSlowy)

```
getHeroesSlowly() {
  return new Promise<Hero[]>(resolve =>
    setTimeout(()=>resolve(HEROES), 2000) // 2 seconds
  );
  }
```

Like `getHeroes`, it also returns a promise. But this promise waits 2 seconds before resolving the promise with mock heroes.

Back in the `AppComponent`, swap `_heroService.getHeroesSlowly` for `_heroService.getHeroes` and see how the app behaves.

Appendix: Shadowing the parent's service

We stated [earlier]() that if we injected the parent `AppComponent` `HeroService` into the `HeroDetailComponent`, *we must not add a providers array* to the `HeroDetailComponent` metadata.

Why? Because that tells Angular to create a new instance of the `HeroService` at the `HeroDetailComponent` level. The `HeroDetailComponent` doesn't want its *own* service instance; it wants its *parent's* service instance. Adding the `providers` array creates a new service instance that shadows the parent instance.

Think carefully about where and when to register a provider. Understand the scope of that registration. Be careful not to create a new service instance at the wrong level.

> **Next Step**
>
> [Routing]()