**ng new angular-tour-of-heroes** - Create a new project named angular-tour-of-heroes with this CLI command

cd angular-tour-of-heroes

**ng serve --open** - The --open flag opens a browser to http://localhost:4200/.

**ng serve --host 0.0.0.0 --port 3000** - uses host and port param

The page you see is the application shell. The shell is controlled by an Angular component named AppComponent.

**Components are the fundamental building blocks of Angular applications. They display data on the screen, listen for user input, and take action based on that input.**

You'll find the implementation of the shell AppComponent distributed over three files:

app.component.ts— the component class code, written in TypeScript.

app.component.html— the component template, written in HTML.

app.component.css— the component's private CSS styles.

<h1>{{title}}</h1> - **The double curly braces are Angular's interpolation binding syntax.** This interpolation binding presents the component's title property value inside the HTML header tag.

Most apps strive for a consistent look across the application. The CLI generated an empty styles.css for this purpose. Put your application-wide styles there.

**ng generate component heroes** - generate a new component named heroes

**You always import the Component symbol from the Angular core library and annotate the component class with @Component.**

**@Component is a decorator function that specifies the Angular metadata for the component.**

The CLI generated three metadata properties:

**selector**— the component's CSS element selector

**templateUrl**— the location of the component's template file.

**styleUrls**— the location of the component's private CSS styles.

The CSS element selector, 'app-heroes', matches the name of the HTML element that identifies this component within a parent component's template.

**The ngOnInit is a lifecycle hook Angular calls ngOnInit shortly after creating a component. It's a good place to put initialization logic.**

**Always export the component class so you can import it elsewhere** ... like in the AppModule.

hero = 'Windstorm'; - Add hero property to the HeroesComponent for a hero named "Windstorm"

**To display the HeroesComponent, you must add it to the template of the shell AppComponent.**

**Remember that app-heroes is the element selector for the HeroesComponent.** So add an <app-heroes> element to the AppComponent template file, just below the title. e.g., <app-heroes></app-heroes>

export class Hero {

id: number;

name: string;

}

hero: Hero = {

id: 1,

name: 'Windstorm'

};

**The page no longer displays properly because you changed the hero from a string to an object.**

**Update the binding in the template to announce the hero's name and show both id and name** in a details layout like this:

<h2>{{ hero.name }} Details</h2>

<div><span>id: </span>{{hero.id}}</div>

<div><span>name: </span>{{hero.name}}</div>

<h2>{{ hero.name | uppercase }} Details</h2> - **The word uppercase in the interpolation binding, right after the pipe operator ( | ), activates the built-in UppercasePipe.**

**Pipes are a good way to format strings, currency amounts, dates and other display data.** Angular ships with several built-in pipes and you can create your own.

**Users should be able to edit the hero name in an <input> textbox.**

**The textbox should both display the hero's name property and update that property as the user types. That means data flow from the component class out to the screen and from the screen back to the class.**

**To automate that data flow, setup a two-way data binding between the <input> form element and the hero.name property.**

<div>

<label>name:

<input [(ngModel)]="hero.name" placeholder="name">

</label>

</div>

**[(ngModel)] is Angular's two-way data binding syntax.**

**Here it binds the hero.name property to the HTML textbox so that data can flow in both directions: from the hero.name property to the textbox, and from the textbox back to the hero.name.**

**Although ngModel is a valid Angular directive, it isn't available by default.**

**It belongs to the optional FormsModule and you must opt-in to using it.**

**Angular needs to know how the pieces of your application fit together and what other files and libraries the app requires. This information is called metadata.**

**Some of the metadata is in the @Component decorators that you added to your component classes. Other critical metadata is in @NgModule decorators.**

**The most important @NgModule decorator annotates the top-level AppModule class.**

**The Angular CLI generated an AppModule class in src/app/app.module.ts when it created the project. This is where you opt-in to the FormsModule.**

import { FormsModule } from '@angular/forms';

imports: [

BrowserModule,

FormsModule

],

**Every component must be declared in exactly one NgModule.**

You didn't declare the HeroesComponent. So why did the application work?

It worked because the Angular CLI declared HeroesComponent in the AppModule when it generated that component.

Open src/app/app.module.ts and find HeroesComponent imported near the top.

import { HeroesComponent } from './heroes/heroes.component';

The HeroesComponent is declared in the @NgModule.declarations array.

declarations: [

AppComponent,

HeroesComponent

],

Note that AppModule declares both application components, AppComponent and HeroesComponent.

**Create a file called mock-heroes.ts in the src/app/ folder. Define a HEROES constant as an array of ten heroes and export it.**

import { Hero } from './hero';

export const 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' }

];

**Open the HeroesComponent class file and import the mock HEROES.**

heroes = HEROES; - **Add a heroes property to the class that exposes these heroes for binding.**

Open the HeroesComponent template file and make the following changes:

Add an <h2> at the top,

Below it add an HTML unordered list (<ul>)

Insert an <li> within the <ul> that displays properties of a hero.

Sprinkle some CSS classes for styling (you'll add the CSS styles shortly).

<h2>My Heroes</h2>

<ul class="heroes">

<li \*ngFor="let hero of heroes">

<span class="badge">{{hero.id}}</span> {{hero.name}}

</li>

</ul>

**The \*ngFor is Angular's repeater directive. It repeats the host element for each element in a list.**

In this example

**<li> is the host element**

**heroes is the list from the HeroesComponent class.**

**hero holds the current hero object for each iteration through the list.**

**You define private styles either inline in the @Component.styles array or as stylesheet file(s) identified in the @Component.styleUrls array.**

When the CLI generated the HeroesComponent, it created an empty heroes.component.css stylesheet for the HeroesComponent and pointed to it in @Component.styleUrls.

@Component({

selector: 'app-heroes',

templateUrl: './heroes.component.html',

styleUrls: ['./heroes.component.css']

})

**Styles and stylesheets identified in @Component metadata are scoped to that specific component. The heroes.component.css styles apply only to the HeroesComponent and don't affect the outer HTML or the HTML in any other component.**

<li \*ngFor="let hero of heroes" (click)="onSelect(hero)"> - **Add a click event binding to the <li>**

**The parentheses around click tell Angular to listen for the <li> element's click event. When the user clicks in the <li>, Angular executes the onSelect(hero) expression.**

**onSelect() is a HeroesComponent method** that you're about to write. **Angular calls it with the hero object displayed in the clicked <li>, the same hero defined previously in the \*ngFor expression.**

Rename the component's hero property to selectedHero but don't assign it. There is no selected hero when the application starts.

Add the following **onSelect() method, which assigns the clicked hero from the template to the component's selectedHero.**

selectedHero: Hero;

onSelect(hero: Hero): void {

this.selectedHero = hero;

}

**Binding expressions in the template that refer to properties of selectedHero — expressions like {{selectedHero.name}} — must fail because there is no selected hero.**

The component should only display the selected hero details if the selectedHero exists.

**Wrap the hero detail HTML in a <div>. Add Angular's \*ngIf directive to the <div> and set it to selectedHero.**

**When selectedHero is undefined, the ngIf removes the hero detail from the DOM. There are no selectedHero bindings to worry about.**

**When the user picks a hero, selectedHero has a value and ngIf puts the hero detail into the DOM.**

It's difficult to identify the selected hero in the list when all <li> elements look alike.

If the user clicks "Magneta", that hero should render with a distinctive but subtle background color.

**That selected hero coloring is the work of the .selected CSS class in the styles you added earlier. You just have to apply the .selected class to the <li> when the user clicks it.**

**The Angular class binding makes it easy to add and remove a CSS class conditionally. Just add [class.some-css-class]="some-condition" to the element you want to style.**

**Add the following [class.selected] binding to the <li> in the HeroesComponent template**

<li \*ngFor="let hero of heroes"

[class.selected]="hero === selectedHero"

(click)="onSelect(hero)">

<span class="badge">{{hero.id}}</span> {{hero.name}}

</li>

**When the current row hero is the same as the selectedHero, Angular adds the selected CSS class. When the two heroes are different, Angular removes the class.**

At the moment, the HeroesComponent displays both the list of heroes and the selected hero's details.

**Keeping all features in one component as the application grows will not be maintainable. You'll want to split up large components into smaller sub-components, each focused on a specific task or workflow.**

In this page, you'll take the first step in that direction by moving the hero details into a separate, reusable HeroDetailComponent.

The HeroesComponent will only present the list of heroes. The HeroDetailComponent will present details of a selected hero.

**The HeroDetailComponent template binds to the component's hero property which is of type Hero.**

The hero property must be an Input property, annotated with the @Input() decorator, because the external HeroesComponent will bind to it like this-

import { Component, OnInit, Input } from '@angular/core'; - Amend the @angular/core import statement to include the Input symbol.

@Input() hero: Hero; - Add a hero property, preceded by the @Input() decorator.

That's the only change you should make to the HeroDetailComponent class. There are no more properties. There's no presentation logic. This component simply receives a hero object through its hero property and displays it.

The HeroesComponent is still a master/detail view.

It used to display the hero details on its own, before you cut that portion of the template. Now it will delegate to the HeroDetailComponent.

The two components will have a parent/child relationship. The parent HeroesComponent will control the child HeroDetailComponent by sending it a new hero to display whenever the user selects a hero from the list.

You won't change the HeroesComponent class but you will change its template.

<app-hero-detail [hero]="selectedHero"></app-hero-detail> - [hero]="selectedHero" is an Angular property binding

It's a one way data binding from the selectedHero property of the HeroesComponent to the hero property of the target element, which maps to the hero property of the HeroDetailComponent.

Now when the user clicks a hero in the list, the selectedHero changes. When the selectedHero changes, the property binding updates hero and the HeroDetailComponent displays the new hero.

As before, whenever a user clicks on a hero name, the hero detail appears below the hero list. Now the HeroDetailComponent is presenting those details instead of the HeroesComponent.

Refactoring the original HeroesComponent into two components yields benefits, both now and in the future:

You simplified the HeroesComponent by reducing its responsibilities.

You can evolve the HeroDetailComponent into a rich hero editor without touching the parent HeroesComponent.

You can evolve the HeroesComponent without touching the hero detail view.

You can re-use the HeroDetailComponent in the template of some future component.

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 new, 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:

in HeroService which uses the service to send a message.

in MessagesComponent which displays that message.

ng generate service hero

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.

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

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.

You must make the HeroService available to the dependency injection system before Angular can inject it into the HeroesComponent, as you will do below. You do this 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.

Now, you need to make sure that the HeroService is registered as the provider of this service. You are registering it with an injector, which is the object that is responsible for choosing and injecting the provider where it is required.

By default, the Angular CLI command ng generate service registers a provider with the root injector for your service by including provider metadata 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 app by removing the service if it turns out not to be used after all.

If you need to, you can register providers at different levels: in the HeroesComponent, in the AppComponent, in the AppModule. For instance, you could have told the CLI to provide the service at the module level automatically by appending --module=app.

ng generate service hero --module=app

Open the HeroesComponent class file.

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

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

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

heroes: Hero[];

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

constructor(private heroService: HeroService) { }

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.

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

Reserve the constructor for simple 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.

ngOnInit() {

this.getHeroes();

}

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.heroes = this.heroService.getHeroes();

This will not work in a real app. You're getting away with it now because the service currently returns mock heroes. But soon the app 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.

It can take a callback. It could return a Promise. It could return an Observable.

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

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.

import { Observable, of } from 'rxjs';

getHeroes(): Observable<Hero[]> {

return of(HEROES);

}

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

getHeroes(): void {

this.heroService.getHeroes()

.subscribe(heroes => this.heroes = heroes);

}

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. Then subscribe 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.

In this section you will

add a MessagesComponent that displays app messages at the bottom of the screen.

create an injectable, app-wide MessageService for sending messages to be displayed

inject MessageService into the HeroService

display a message when HeroService fetches heroes successfully.

<h1>{{title}}</h1>

<app-heroes></app-heroes>

<app-messages></app-messages>

Open MessageService and replace its contents with the following.

import { Injectable } from '@angular/core';

@Injectable({

providedIn: 'root',

})

export class MessageService {

messages: string[] = [];

add(message: string) {

this.messages.push(message);

}

clear() {

this.messages = [];

}

}

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

Re-open the HeroService and import the MessageService.

import { MessageService } from './message.service';

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.

constructor(private messageService: MessageService) { }

This is a typical "service-in-service" scenario: you inject the MessageService into the HeroService which is injected into the HeroesComponent.

getHeroes(): Observable<Hero[]> {

// TODO: send the message \_after\_ fetching the heroes

this.messageService.add('HeroService: fetched heroes');

return of(HEROES);

}

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

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

Angular only binds to public component properties.

<div \*ngIf="messageService.messages.length">

<h2>Messages</h2>

<button class="clear"

(click)="messageService.clear()">clear</button>

<div \*ngFor='let message of messageService.messages'> {{message}} </div>

</div>

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().

There are new requirements for the Tour of Heroes app:

Add a Dashboard view.

Add the ability to navigate between the Heroes and Dashboard views.

When users click a hero name in either view, navigate to a detail view of the selected hero.

When users click a deep link in an email, open the detail view for a particular hero.

An Angular best practice is to load and configure the router in a separate, top-level module that is dedicated to routing and imported by the root AppModule.

By convention, the module class name is AppRoutingModule and it belongs in the app-routing.module.ts in the src/app folder.

ng generate module app-routing --flat --module=app

--flat puts the file in src/app instead of its own folder.

--module=app tells the CLI to register it in the imports array of the AppModule.

You generally don't declare components in a routing module so you can delete the @NgModule.declarations array and delete CommonModule references too.

You'll configure the router with Routes in the RouterModule so import those two symbols from the @angular/router library.

Add an @NgModule.exports array with RouterModule in it. Exporting RouterModule makes router directives available for use in the AppModule components that will need them.

import { RouterModule, Routes } from '@angular/router';

@NgModule({

exports: [ RouterModule ]

})

Routes tell the router which view to display when a user clicks a link or pastes a URL into the browser address bar.

A typical Angular Route has two properties:

path: a string that matches the URL in the browser address bar.

component: the component that the router should create when navigating to this route.

You intend to navigate to the HeroesComponent when the URL is something like localhost:4200/heroes.

Import the HeroesComponent so you can reference it in a Route. Then define an array of routes with a single route to that component.

You first must initialize the router and start it listening for browser location changes.

Add RouterModule to the @NgModule.imports array and configure it with the routes in one step by calling RouterModule.forRoot() within the imports array, like this:

imports: [ RouterModule.forRoot(routes) ],

The method is called forRoot() because you configure the router at the application's root level. The forRoot() method supplies the service providers and directives needed for routing, and performs the initial navigation based on the current browser URL.

The <router-outlet> tells the router where to display routed views.

The RouterOutlet is one of the router directives that became available to the AppComponent because AppModule imports AppRoutingModule which exported RouterModule.

Users shouldn't have to paste a route URL into the address bar. They should be able to click a link to navigate.

Add a <nav> element and, within that, an anchor element that, when clicked, triggers navigation to the HeroesComponent.

<h1>{{title}}</h1>

<nav>

<a routerLink="/heroes">Heroes</a>

</nav>

<router-outlet></router-outlet>

<app-messages></app-messages>

A routerLink attribute is set to "/heroes", the string that the router matches to the route to HeroesComponent. The routerLink is the selector for the RouterLink directive that turns user clicks into router navigations. It's another of the public directives in the RouterModule.

Routing makes more sense when there are multiple views. So far there's only the heroes view.

The template presents a grid of hero name links.

The \*ngFor repeater creates as many links as are in the component's heroes array.

The links are styled as colored blocks by the dashboard.component.css.

The class is similar to the HeroesComponent class.

It defines a heroes array property.

The constructor expects Angular to inject the HeroService into a private heroService property.

The ngOnInit() lifecycle hook calls getHeroes.

getHeroes(): void {

this.heroService.getHeroes()

.subscribe(heroes => this.heroes = heroes.slice(1, 5));

}

This getHeroes reduces the number of heroes displayed to four (2nd, 3rd, 4th, and 5th).

When the app starts, the browsers address bar points to the web site's root. That doesn't match any existing route so the router doesn't navigate anywhere. The space below the <router-outlet> is blank.

To make the app navigate to the dashboard automatically, add the following route to the AppRoutingModule.Routes array.

{ path: '', redirectTo: '/dashboard', pathMatch: 'full' },

This route redirects a URL that fully matches the empty path to the route whose path is '/dashboard'.

The user should be able to navigate back and forth between the DashboardComponent and the HeroesComponent by clicking links in the navigation area near the top of the page.

Add a dashboard navigation link to the AppComponent shell template, just above the Heroes link.

<a routerLink="/dashboard">Dashboard</a>

The HeroDetailsComponent displays details of a selected hero. At the moment the HeroDetailsComponent is only visible at the bottom of the HeroesComponent

The user should be able to get to these details in three ways.

By clicking a hero in the dashboard.

By clicking a hero in the heroes list.

By pasting a "deep link" URL into the browser address bar that identifies the hero to display.

In this section, you'll enable navigation to the HeroDetailsComponent and liberate it from the HeroesComponent.

A URL like ~/detail/11 would be a good URL for navigating to the Hero Detail view of the hero whose id is 11.

Then add a parameterized route to the AppRoutingModule.routes array that matches the path pattern to the hero detail view.

{ path: 'detail/:id', component: HeroDetailComponent },

The colon (:) in the path indicates that :id is a placeholder for a specific hero id.

Now that the router has a route to HeroDetailComponent, fix the dashboard hero links to navigate via the parameterized dashboard route.

<a \*ngFor="let hero of heroes" class="col-1-4"

routerLink="/detail/{{hero.id}}">

You're using Angular interpolation binding within the \*ngFor repeater to insert the current iteration's hero.id into each routerLink.

The hero items in the HeroesComponent are <li> elements whose click events are bound to the component's onSelect() method.

Strip the <li> back to just its \*ngFor, wrap the badge and name in an anchor element (<a>), and add a routerLink attribute to the anchor that is the same as in the dashboard template

Previously, the parent HeroesComponent set the HeroDetailComponent.hero property and the HeroDetailComponent displayed the hero.

HeroesComponent doesn't do that anymore. Now the router creates the HeroDetailComponent in response to a URL such as ~/detail/11.

The HeroDetailComponent needs a new way to obtain the hero-to-display.

Get the route that created it,

Extract the id from the route

Acquire the hero with that id from the server via the HeroService

Inject the ActivatedRoute, HeroService, and Location services into the constructor, saving their values in private fields:

constructor(

private route: ActivatedRoute,

private heroService: HeroService,

private location: Location

) {}

The ActivatedRoute holds information about the route to this instance of the HeroDetailComponent. This component is interested in the route's bag of parameters extracted from the URL. The "id" parameter is the id of the hero to display.

The HeroService gets hero data from the remote server and this component will use it to get the hero-to-display.

The location is an Angular service for interacting with the browser. You'll use it later to navigate back to the view that navigated here.

getHero(): void {

const id = +this.route.snapshot.paramMap.get('id');

this.heroService.getHero(id)

.subscribe(hero => this.hero = hero);

}

The route.snapshot is a static image of the route information shortly after the component was created.

The paramMap is a dictionary of route parameter values extracted from the URL. The "id" key returns the id of the hero to fetch.

Route parameters are always strings. The JavaScript (+) operator converts the string to a number, which is what a hero id should be.

getHero(id: number): Observable<Hero> {

// TODO: send the message \_after\_ fetching the hero

this.messageService.add(`HeroService: fetched hero id=${id}`);

return of(HEROES.find(hero => hero.id === id));

}

Note the backticks ( ` ) that define a JavaScript template literal for embedding the id.

Like getHeroes(), getHero() has an asynchronous signature. It returns a mock hero as an Observable, using the RxJS of() function.

If you paste localhost:4200/detail/11 in the browser address bar, the router navigates to the detail view for the hero with id: 11, "Mr. Nice".

By clicking the browser's back button, you can go back to the hero list or dashboard view, depending upon which sent you to the detail view.

It would be nice to have a button on the HeroDetail view that can do that.

Add a go back button to the bottom of the component template and bind it to the component's goBack() method.

<button (click)="goBack()">go back</button>

Add a goBack() method to the component class that navigates backward one step in the browser's history stack using the Location service that you injected previously.

goBack(): void {

this.location.back();

}

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.

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

To make HttpClient available everywhere in the app,

open the root AppModule,

import the HttpClientModule symbol from @angular/common/http,

add it to the @NgModule.imports array.

## Simulate a data server

This tutorial sample mimics communication with a remote data server by using the [In-memory Web API](https://github.com/angular/in-memory-web-api) module.

After installing the module, the app will make requests to and receive responses from the [HttpClient](https://angular.io/api/common/http/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.

Install the In-memory Web API package from npm

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

Import the HttpClientInMemoryWebApiModule and the InMemoryDataService class, which you will create in a moment.

import { HttpClientInMemoryWebApiModule } from 'angular-in-memory-web-api';

import { InMemoryDataService } from './in-memory-data.service';

Add the HttpClientInMemoryWebApiModule to the @[NgModule.imports](https://angular.io/api/core/NgModule" \l "imports) array— after importing the [HttpClient](https://angular.io/api/common/http/HttpClient), —while configuring it with the InMemoryDataService.

[HttpClientModule](https://angular.io/api/common/http/HttpClientModule),

// The HttpClientInMemoryWebApiModule module intercepts HTTP requests

// and returns simulated server responses.

// Remove it when [a](https://angular.io/api/router/RouterLinkWithHref) real server is ready to receive requests.

HttpClientInMemoryWebApiModule.forRoot(

InMemoryDataService, { dataEncapsulation: false }

)

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

import { InMemoryDbService } from 'angular-in-memory-web-api';

export class InMemoryDataService implements InMemoryDbService {

createDb() {

const heroes = [

{ 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' }

];

return {heroes};

}

}

This file replaces mock-heroes.ts, which is now safe to delete.

import { [HttpClient](https://angular.io/api/common/http/HttpClient), [HttpHeaders](https://angular.io/api/common/http/HttpHeaders) } from '@angular/common/http';

getHeroes (): Observable<Hero[]> {

return this.http.get<Hero[]>(this.heroesUrl)

}

All [HttpClient](https://angular.io/api/common/http/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](https://angular.io/api/common/http/HttpClient) always emits a single value and then completes, never to emit again.

HttpClient.get returns the body of the response as an untyped JSON object by default. Applying the optional type specifier, <Hero[]> , gives you a typed result object.

### 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.

import { catchError, map, tap } from 'rxjs/operators';

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

getHeroes (): Observable<Hero[]> {

return this.http.get<Hero[]>(this.heroesUrl)

.pipe(

catchError(this.handleError('getHeroes', []))

);

}

The catchError() operator intercepts an Observable that failed. It passes the error an error handler that can do what it wants with the error.

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

The following errorHandler() 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.

The HeroService methods will tap into the flow of observable values and send a message (via log()) 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.

/\*\* GET heroes from the server \*/

getHeroes (): Observable<Hero[]> {

return this.http.get<Hero[]>(this.heroesUrl)

.pipe(

tap(heroes => this.log(`fetched heroes`)),

catchError(this.handleError('getHeroes', []))

);

}

/\*\* GET hero by id. Will 404 if id not found \*/

getHero(id: number): Observable<Hero> {

const url = `${this.heroesUrl}/${id}`;

return this.http.get<Hero>(url).pipe(

tap(\_ => this.log(`fetched hero id=${id}`)),

catchError(this.handleError<Hero>(`getHero id=${id}`))

);

}

There are three significant differences from getHeroes().

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

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().

<button (click)="save()">save</button>

Add the following save() method, which persists hero name changes using the hero service updateHero() method and then navigates back to the previous view.

save(): void {

this.heroService.updateHero(this.hero)

.subscribe(() => this.goBack());

}

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.

/\*\* PUT: update the hero on the server \*/

updateHero (hero: Hero): Observable<any> {

return this.http.put(this.heroesUrl, hero, httpOptions).pipe(

tap(\_ => this.log(`updated hero id=${hero.id}`)),

catchError(this.handleError<any>('updateHero'))

);

}

The [HttpClient.put()](https://angular.io/) 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.

const httpOptions = {

headers: new [HttpHeaders](https://angular.io/api/common/http/HttpHeaders)({ 'Content-Type': 'application/json' })

};

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

<div>

<label>Hero name:

<input #heroName />

</label>

<!-- (click) passes input value to add() and then clears the input -->

<button (click)="add(heroName.value); heroName.value=''">

add

</button>

</div>

In response to a click event, call the component's click handler and then clear the input field so that it's ready for another name.

add(name: string): void {

name = name.trim();

if (!name) { return; }

this.heroService.addHero({ name } as Hero)

.subscribe(hero => {

this.heroes.push(hero);

});

}

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 services addHero() method.

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

/\*\* POST: add [a](https://angular.io/api/router/RouterLinkWithHref) new hero to the server \*/

addHero (hero: Hero): Observable<Hero> {

return this.http.post<Hero>(this.heroesUrl, hero, httpOptions).pipe(

tap((hero: Hero) => this.log(`added hero w/ id=${hero.id}`)),

catchError(this.handleError<Hero>('addHero'))

);

}

HeroService.addHero() differs from updateHero in two ways.

* it calls [HttpClient.post()](https://angular.io/) instead of put().
* it expects the server to generates an id for the new hero, which it returns in the Observable<Hero> to the caller.

Add the following button element to the HeroesComponent template, after the hero name in the repeated <li> element.

<button class="delete" title="delete hero"

(click)="delete(hero)">x</button>

<ul class="heroes">

<li \*[ngFor](https://angular.io/api/common/NgForOf)="let hero of heroes">

<[a](https://angular.io/api/router/RouterLinkWithHref) [routerLink](https://angular.io/api/router/RouterLink)="/detail/{{hero.id}}">

<span class="badge">{{hero.id}}</span> {{hero.name}}

</[a](https://angular.io/api/router/RouterLinkWithHref)>

<button class="delete" title="delete hero"

(click)="delete(hero)">x</button>

</li>

</ul>

delete(hero: Hero): void {

this.heroes = this.heroes.filter(h => h !== hero);

this.heroService.deleteHero(hero).subscribe();

}

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(). 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!

/\*\* DELETE: delete the hero from the server \*/

deleteHero (hero: Hero | number): Observable<Hero> {

const id = typeof hero === 'number' ? hero : hero.id;

const url = `${this.heroesUrl}/${id}`;

return this.http.delete<Hero>(url, httpOptions).pipe(

tap(\_ => this.log(`deleted hero id=${id}`)),

catchError(this.handleError<Hero>('deleteHero'))

);

}

Note that

* it 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.

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.

/\* GET heroes whose name contains search term \*/

searchHeroes(term: string): Observable<Hero[]> {

if (!term.trim()) {

// if not search term, return empty hero array.

return of([]);

}

return this.http.get<Hero[]>(`${this.heroesUrl}/?name=${term}`).pipe(

tap(\_ => this.log(`found heroes matching "${term}"`)),

catchError(this.handleError<Hero[]>('searchHeroes', []))

);

}

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 is the URL, which includes a query string with the search term.

ng generate component hero-search

<div id="search-component">

<h4>Hero Search</h4>

<input #searchBox id="search-box" (keyup)="search(searchBox.value)" />

<ul class="search-result">

<li \*ngFor="let hero of heroes$ | async" >

<a routerLink="/detail/{{hero.id}}">

{{hero.name}}

</a>

</li>

</ul>

</div>

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

As expected, the \*[ngFor](https://angular.io/api/common/NgForOf) repeats hero objects.

Look closely and you'll see that the \*[ngFor](https://angular.io/api/common/NgForOf) iterates over a list called heroes$, not heroes.

<li \*[ngFor](https://angular.io/api/common/NgForOf)="let hero of heroes$ | [async](https://angular.io/api/core/testing/async)" >

The $ is a convention that indicates heroes$ is an Observable, not an array.

The \*[ngFor](https://angular.io/api/common/NgForOf) can't do anything with an Observable. But there's also a pipe character (|) followed by [async](https://angular.io/api/core/testing/async), which identifies Angular's [AsyncPipe](https://angular.io/api/common/AsyncPipe).

The [AsyncPipe](https://angular.io/api/common/AsyncPipe) subscribes to an Observable automatically so you won't have to do so in the component class.

import { Component, OnInit } from '@angular/core';

import { Observable, Subject } from 'rxjs';

import {

debounceTime, distinctUntilChanged, switchMap

} from 'rxjs/operators';

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

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

@Component({

selector: 'app-hero-search',

templateUrl: './hero-search.component.html',

styleUrls: [ './hero-search.component.css' ]

})

export class HeroSearchComponent implements OnInit {

heroes$: Observable<Hero[]>;

private searchTerms = new Subject<string>();

constructor(private heroService: HeroService) {}

// Push a search term into the observable stream.

search(term: string): void {

this.searchTerms.next(term);

}

ngOnInit(): void {

this.heroes$ = this.searchTerms.pipe(

// wait 300ms after each keystroke before considering the term

debounceTime(300),

// ignore new term if same as previous term

distinctUntilChanged(),

// switch to new search observable each time the term changes

switchMap((term: string) => this.heroService.searchHeroes(term)),

);

}

}

Notice the declaration of heroes$ as an Observable

heroes$: Observable<Hero[]>;

The searchTerms property is declared as an RxJS Subject.

private searchTerms = new Subject<string>();

// Push [a](https://angular.io/api/router/RouterLinkWithHref) search term into the observable stream.

search(term: string): void {

this.searchTerms.next(term);

}

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 search() method is called via an event binding to the textbox's keystroke event.

<input #searchBox id="search-box" (keyup)="search(searchBox.value)" />

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.

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 the cellular network data plan.

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[]).

* 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](http://www.learnrxjs.io/operators/transformation/switchmap.html), every qualifying key event can trigger an [HttpClient.get()](https://angular.io/) 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 simply 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](https://angular.io/tutorial/toh-pt6" \l "asyncpipe) in the template.