Tour of Heroes application and tutorial

GETTING STARTED

In this tutorial, you build your own Angular application from the start. This is a good way to experience a typical development process as you learn Angular application-design concepts, tools, and terminology.

If you're new to Angular, try the **Try it now** quick-start application first. **Try it now** is based on a ready-made partially completed project. You can edit the application in StackBlitz and see the results in real time.

Try it now covers the same major topics —components, template syntax, routing, services, and accessing data using HTTP— in a condensed format, following best practices.

This *Tour of Heroes* tutorial provides an introduction to the fundamentals of Angular and shows you how to:

- Set up your local Angular development environment.
- Use the Angular CLI to develop an application.

The *Tour of Heroes* application that you build helps a staffing agency manage its stable of heroes. The application has many of the features that you'd expect to find in any data-driven application.

The finished application:

- Gets a list of heroes
- Displays the heroes in a list
- Edits a selected hero's details
- Navigates between different views of heroic data

This tutorial helps you gain confidence that Angular can do whatever you need it to do by showing you how to:

- Use Angular directives to show and hide elements and display lists of hero data.
- Create Angular components to display hero details and show an array of heroes.
- Use one-way data binding for read-only data.
- Add editable fields to update a model with two-way data binding.
- Bind component methods to user events, like keystrokes and clicks.
- Enable users to select a hero from a list and edit that hero in the details view.
- Format data with pipes.
- Create a shared service to assemble the heroes.
- Use routing to navigate among different views and their components.

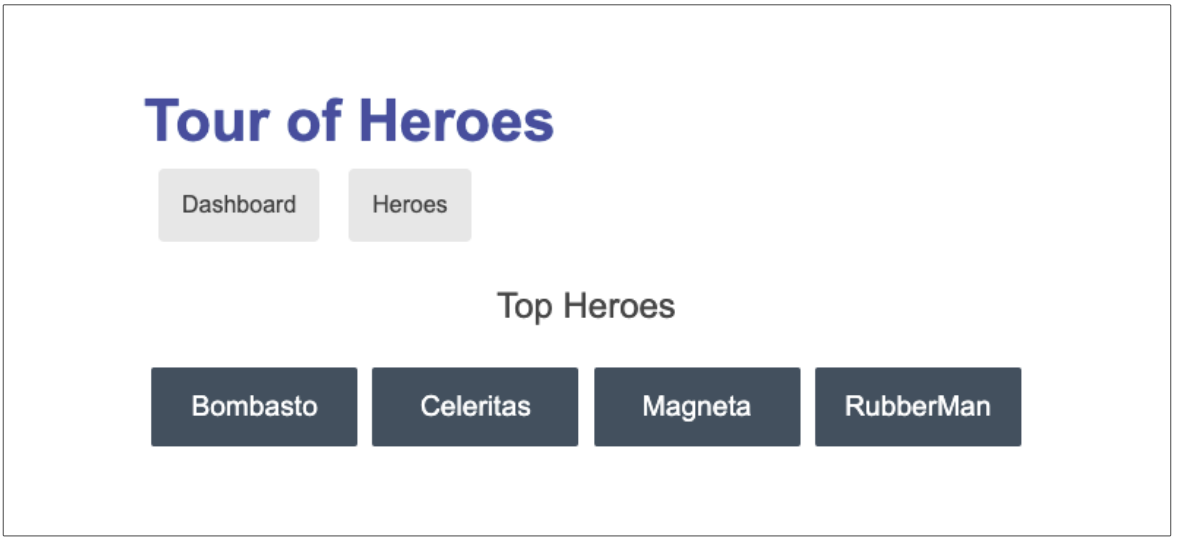
SOLUTION

After you complete all tutorial steps, the final application looks like this example.

live example / download example.

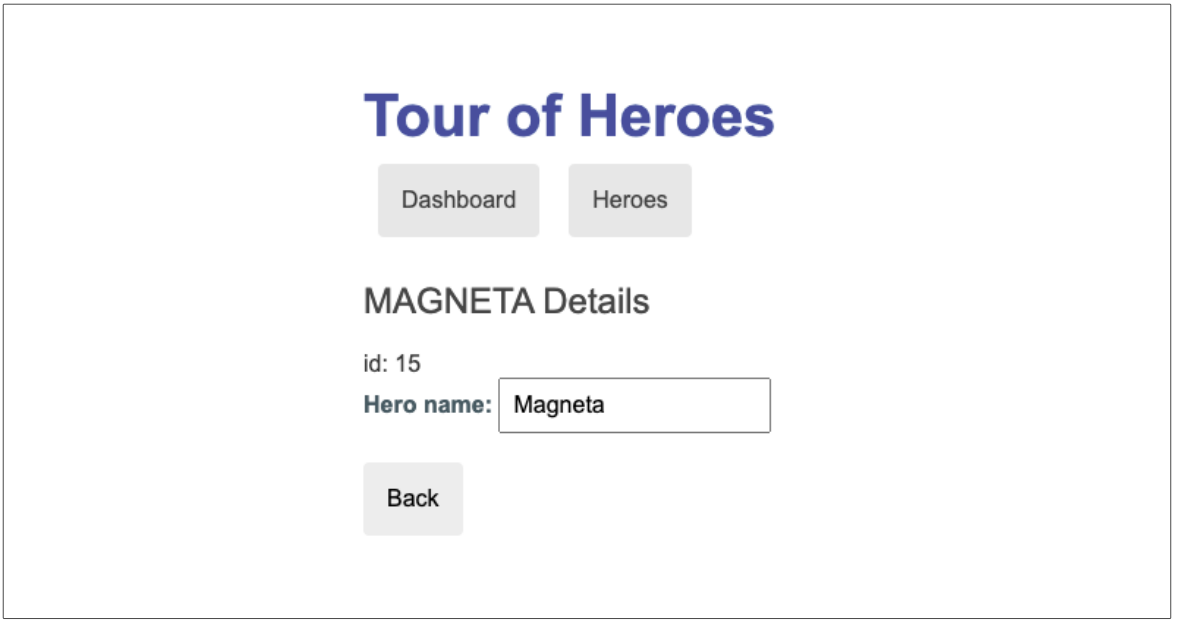
Design your new application

Here's an image of where this tutorial leads, showing the Dashboard view and the most heroic heroes:

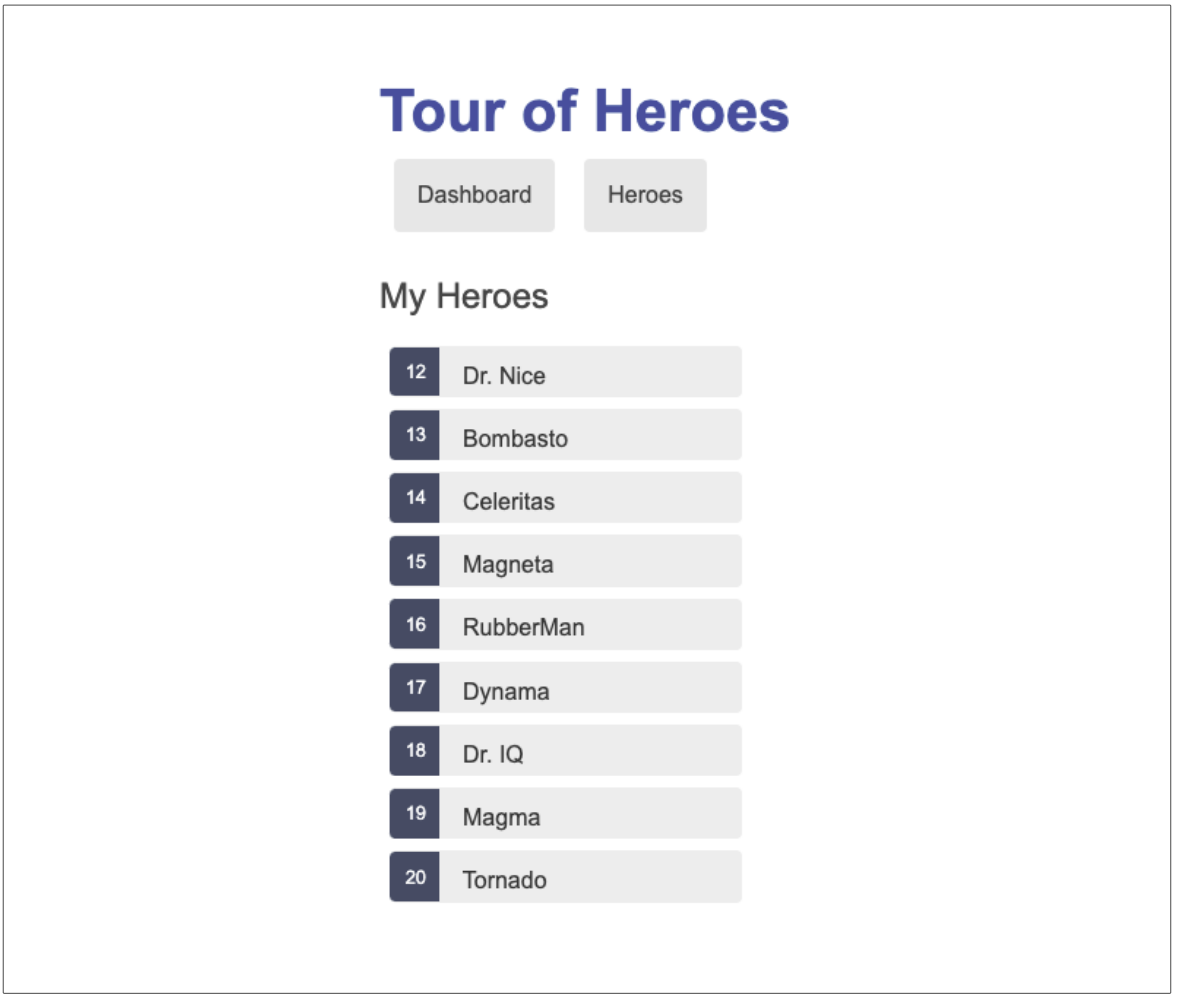


You can click the **Dashboard** and **Heroes** links in the dashboard to navigate between the views.

If you click the dashboard hero "Magneta," the router opens a "Hero Details" view where you can change the hero's name.



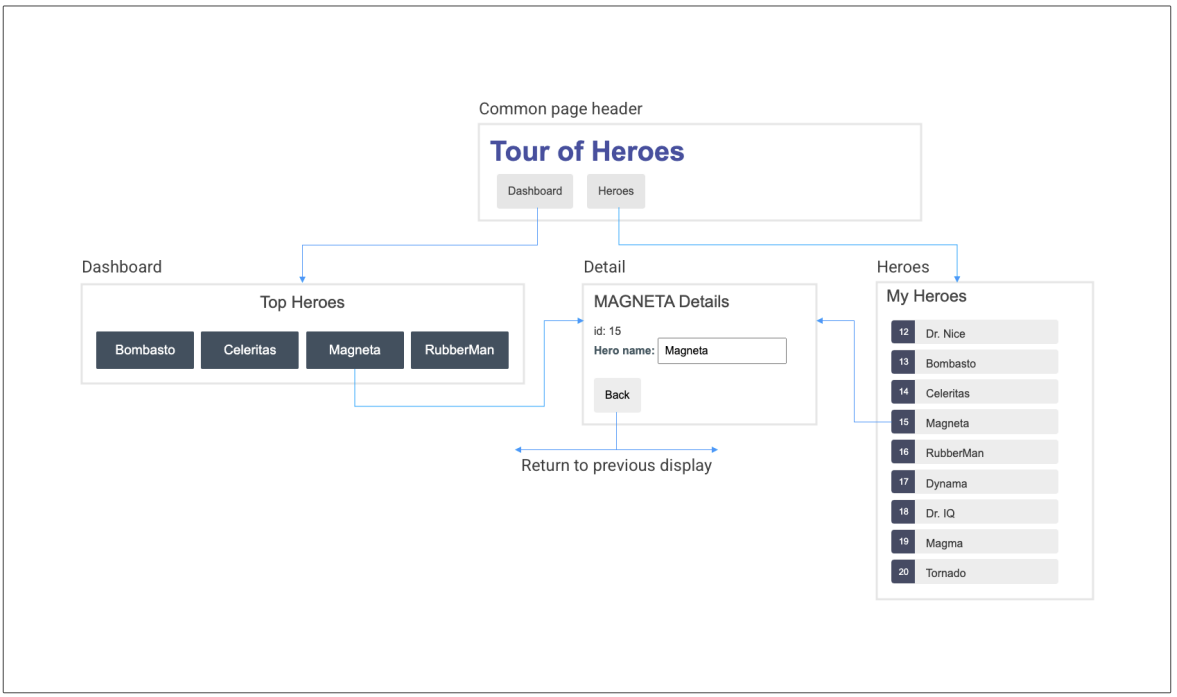
Clicking the "Back" button returns you to the Dashboard. Links at the top take you to either of the main views. If you click "Heroes," the application displays the "Heroes" list view.



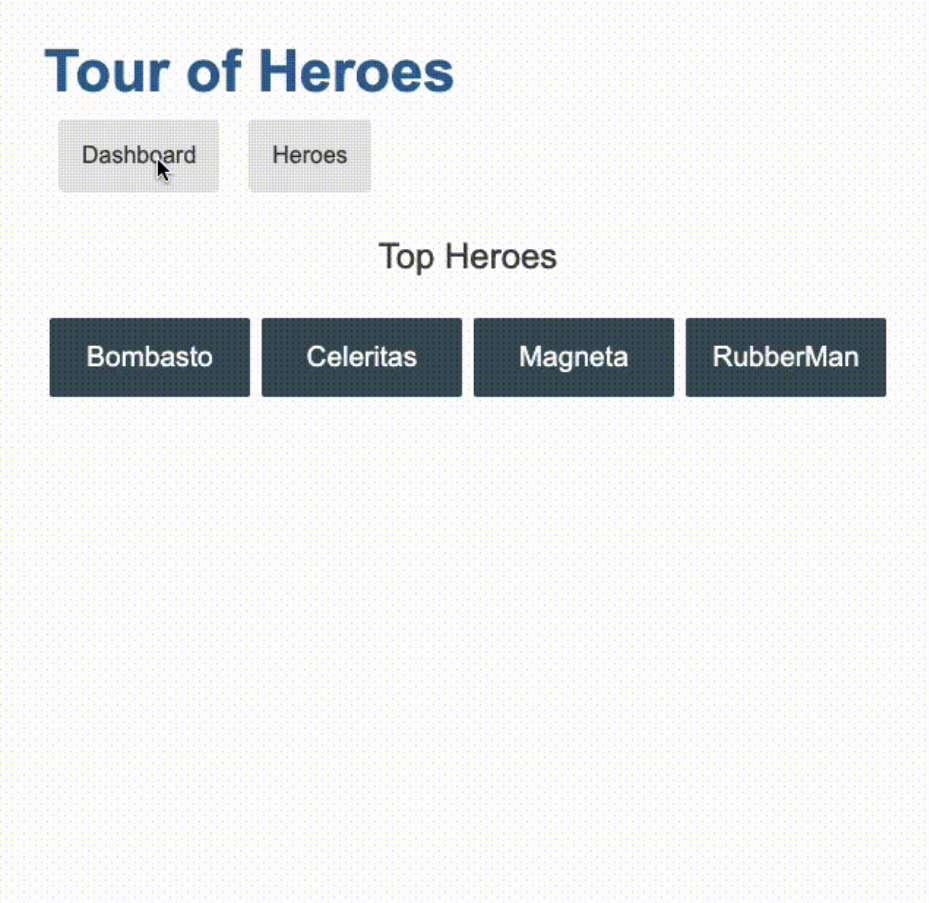
When you click a different hero name, the read-only mini detail beneath the list reflects the new choice.

You can click the "View Details" button to drill into the editable details of the selected hero.

The following diagram illustrates the navigation options.



Here's the application in action:



Last reviewed on Mon May 16 2022

Create a new project

Use the `ng new` command to start creating your **Tour of Heroes** application.

This tutorial:

1. Sets up your environment.
2. Creates a new workspace and initial application project.
3. Serves the application.
4. Makes changes to the new application.

> To view the application's code, see the live example / download example.

Set up your environment

To set up your development environment, follow the instructions in Local Environment Setup.

Create a new workspace and an initial application

You develop applications in the context of an Angular workspace. A *workspace* contains the files for one or more projects. A *project* is the set of files that make up an application or a library.

To create a new workspace and an initial project:

1. Ensure that you aren't already in an Angular workspace directory. For example, if you're in the Getting Started workspace from an earlier exercise, navigate to its parent.
2. Run `ng new` followed by the application name as shown here:

```
ng new angular-tour-of-heroes
```

3. `ng new` prompts you for information about features to include in the initial project. Accept the defaults by pressing the Enter or Return key.

`ng new` installs the necessary `npm` packages and other dependencies that Angular requires. This can take a few minutes.

`ng new` also creates the following workspace and starter project files:

- A new workspace, with a root directory named `angular-tour-of-heroes`
- An initial skeleton application project in the `src/app` subdirectory
- Related configuration files

The initial application project contains a simple application that's ready to run.

Serve the application

Go to the workspace directory and launch the application.

```
cd angular-tour-of-heroes
ng serve --open
```

> The `ng serve` command:
>
> - Builds the application
> - Starts the development server
> - Watches the source files
> - Rebuilds the application as you make changes
>
> The `--open` flag opens a browser to `http://localhost:4200`.

You should see the application running in your browser.

Angular components

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.

Make changes to the application

Open the project in your favorite editor or IDE. Navigate to the `src/app` directory to edit the starter application.
In the IDE, locate these files, which make up the `AppComponent` that you just created:

FILES	DETAILS
`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.

When you ran `ng new`, Angular created test specifications for your new application. Unfortunately, making these changes breaks your newly created specifications.

That won't be a problem because Angular testing is outside the scope of this tutorial and won't be used.

To learn more about testing with Angular, see Testing.

Change the application title

Open the `app.component.ts` and change the `title` property value to 'Tour of Heroes'.

app.component.ts (class title property)

```
title = 'Tour of Heroes';
```

Open `app.component.html` and delete the default template that `ng new` created. Replace it with the following line of HTML.

app.component.html (template)

```
<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.

The browser refreshes and displays the new application title.

Add application styles

Most apps strive for a consistent look across the application. `ng new` created an empty `styles.css` for this purpose. Put your application-wide styles there.

Open `src/styles.css` and add the code below to the file.

src/styles.css (excerpt)

```
/* Application-wide Styles */
h1 {
  color: #369;
  font-family: Arial, Helvetica, sans-serif;
  font-size: 250%;
}
h2, h3 {
  color: #444;
  font-family: Arial, Helvetica, sans-serif;
  font-weight: lighter;
}
body {
  margin: 2em;
}
body, input[type="text"], button {
  color: #333;
  font-family: Cambria, Georgia, serif;
}
button {
  background-color: #eee;
  border: none;
  border-radius: 4px;
  cursor: pointer;
  color: black;
  font-size: 1.2rem;
  padding: 1rem;
  margin-right: 1rem;
  margin-bottom: 1rem;
  margin-top: 1rem;
}
button:hover {
  background-color: black;
  color: white;
}
button:disabled {
  background-color: #eee;
  color: #aaa;
  cursor: auto;
}

/* everywhere else */
* {
  font-family: Arial, Helvetica, sans-serif;
}
```

Final code review

Here are the code files discussed on this page.

src/app/app.component.ts *src/app/app.component.html* *src/styles.css (excerpt)*

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Tour of Heroes';
}
```

Summary

- You created the initial application structure using `ng new`.
- You learned that Angular components display data
- You used the double curly braces of interpolation to display the application title

Last reviewed on Mon Feb 28 2022

The hero editor

The application now has a basic title. Next, create a new component to display hero information and place that component in the application shell.

> For the sample application that this page describes, see the live example / download example.

Create the heroes component

Use `ng generate` to create a new component named `heroes`.

```
ng generate component heroes
```

`ng generate` creates a new directory , `src/app/heroes/`, and generates the three files of the `HeroesComponent` along with a test file.

The `HeroesComponent` class file is as follows:

app/heroes/heroes.component.ts (initial version)

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-heroes',
  templateUrl: './heroes.component.html',
  styleUrls: ['./heroes.component.css']
})
export class HeroesComponent {
}
```

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.

`ng generate` created three metadata properties:

PROPERTIES	DETAILS
`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.

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

Add a `hero` property

Add a `hero` property to the `HeroesComponent` for a hero named, `Windstorm`.

heroes.component.ts (hero property)

```
hero = 'Windstorm';
```

Show the hero

Open the `heroes.component.html` template file. Delete the default text that `ng generate` created and replace it with a data binding to the new `hero` property.

heroes.component.html

```
<h2>{{hero}}</h2>
```

Show the `HeroesComponent` view

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`. Add an `<app-heroes>` element to the `AppComponent` template file, just below the title.

src/app/app.component.html

```
<h1>{{title}}</h1>
<app-heroes></app-heroes>
```

If `ng serve` is still running, the browser should refresh and display both the application title and the hero's name.

Create a `Hero` interface

A real hero is more than a name.

Create a `Hero` interface in its own file in the `src/app` directory . Give it `id` and `name` properties.

src/app/hero.ts

```
export interface Hero {
  id: number;
  name: string;
}
```

Return to the `HeroesComponent` class and import the `Hero` interface.

Refactor the component's `hero` property to be of type `Hero`. Initialize it with an `id` of `1` and the name `Windstorm`.

The revised `HeroesComponent` class file should look like this:

src/app/heroes/heroes.component.ts

```
import { Component } from '@angular/core';
import { Hero } from '../hero';

@Component({
  selector: 'app-heroes',
  templateUrl: './heroes.component.html',
  styleUrls: ['./heroes.component.css']
})
export class HeroesComponent {
  hero: Hero = {
    id: 1,
    name: 'Windstorm'
  };
}
```

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

Show the hero object

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

heroes.component.html (HeroesComponent template)

```
<h2>{{hero.name}} Details</h2>
<div><span>id: </span>{{hero.id}}</div>
<div><span>name: </span>{{hero.name}}</div>
```

The browser refreshes and displays the hero's information.

Format with the `UppercasePipe`

Edit the `hero.name` binding like this:

src/app/heroes/heroes.component.html

```
<h2>{{hero.name | uppercase}} Details</h2>
```

The browser refreshes and now the hero's name is displayed in capital letters.

The word `uppercase` in the interpolation binding after the pipe `|` character, 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.

Edit the hero

Users should be able to edit the hero's name in an `<input>` text box.

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

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

Two-way binding

Refactor the details area in the `HeroesComponent` template so it looks like this:

src/app/heroes/heroes.component.html (HeroesComponent's template)

```
<div>
  <label for="name">Hero name: </label>
  <input id="name" [(ngModel)]="hero.name" placeholder="name">
</div>
```

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

Here it binds the `hero.name` property to the HTML text box so that data can flow *in both directions*. Data can flow from the `hero.name` property to the text box and from the text box back to the `hero.name`.

The missing `FormsModule`

Notice that the application stopped working when you added `[(ngModel)]`.

To see the error, open the browser development tools and look in the console for a message like

```
Template parse errors:
Can't bind to 'ngModel' since it isn't a known property of 'input'.
```

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.

`AppModule`

Angular needs to know how the pieces of your application fit together and what other files and libraries the application 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.

`ng new` created 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`

Open `app.module.ts` and import the `FormsModule` symbol from the `@angular/forms` library.

app.module.ts (FormsModule symbol import)

```
import { FormsModule } from '@angular/forms'; // <-- NgModel lives here
```

Add `FormsModule` to the `imports` array in `@NgModule`. The `imports` array contains the list of external modules that the application needs.

app.module.ts (@NgModule imports)

```
imports: [
  BrowserModule,
  FormsModule
],
```

When the browser refreshes, the application should work again. You can edit the hero's name and see the changes reflected immediately in the `<h2>` above the text box.

Declare `HeroesComponent`

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

You didn't declare the `HeroesComponent`. Why did the application work?

It worked because the `ng generate` declared `HeroesComponent` in `AppModule` when it created that component.

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

src/app/app.module.ts

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

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

src/app/app.module.ts

```
declarations: [
  AppComponent,
  HeroesComponent
],
```

`AppModule` declares both application components, `AppComponent` and `HeroesComponent`.

Final code review

Here are the code files discussed on this page.

src/app/heroes/heroes.component.ts *src/app/heroes/heroes.component.html* *src/app/app.module*

```
import { Component } from '@angular/core';
import { Hero } from '../hero';

@Component({
  selector: 'app-heroes',
  templateUrl: './heroes.component.html',
  styleUrls: ['./heroes.component.css']
})
export class HeroesComponent {
  hero: Hero = {
    id: 1,
    name: 'Windstorm'
  };
}
```

Summary

- You used `ng generate` to create a second `HeroesComponent`.
- You displayed the `HeroesComponent` by adding it to the `AppComponent` shell.
- You applied the `UppercasePipe` to format the name.
- You used two-way data binding with the `ngModel` directive.
- You learned about the `AppModule`.
- You imported the `FormsModule` in the `AppModule` so that Angular would recognize and apply the `ngModel` directive.
- You learned the importance of declaring components in the `AppModule`.

Last reviewed on Mon Feb 28 2022

Display a selection list

This tutorial shows you how to:

- Expand the Tour of Heroes application to display a list of heroes.
- Allow users to select a hero and display the hero's details.

> For the sample application that this page describes, see the live example / download example.

Create mock heroes

The first step is to create some heroes to display.

Create a file called `mock-heroes.ts` in the `src/app/` directory. Define a `HEROES` constant as an array of ten heroes and export it. The file should look like this.

src/app/mock-heroes.ts

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

export const HEROES: Hero[] = [
  { id: 12, name: 'Dr. Nice' },
  { 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' }
];
```

Displaying heroes

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

src/app/heroes/heroes.component.ts (import HEROES)

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

In `HeroesComponent` class, define a component property called `heroes` to expose the `HEROES` array for binding.

src/app/heroes/heroes.component.ts

```
export class HeroesComponent {

  heroes = HEROES;
}
```

List heroes with *ngFor

Open the `HeroesComponent` template file and make the following changes:

1. Add an `<h2>` at the top.
2. Below the `<h2>`, add a `<ul>` element.
3. In the `<ul>` element, insert an `<li>`.
4. Place a `<button>` inside the `<li>` that displays properties of a `hero` inside `<span>` elements.
5. Add CSS classes to style the component.

to look like this:

heroes.component.html (heroes template)

```
<h2>My Heroes</h2>
<ul class="heroes">
  <li>
    <button type="button">
      <span class="badge">{{hero.id}}</span>
      <span class="name">{{hero.name}}</span>
    </button>
  </li>
</ul>
```

That displays an error since the `hero` property doesn't exist. To have access to each individual hero and list them all, add an `*ngFor` to the `<li>` to iterate through the list of heroes:

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

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

The syntax in this example is as follows:

SYNTAX	DETAILS
`<li>`	The host element.
`heroes`	Holds the mock heroes list from the `HeroesComponent` class, the mock heroes list.
`hero`	Holds the current hero object for each iteration through the list.

> Don't forget to put the asterisk `*` in front of `ngFor`. It's a critical part of the syntax.

After the browser refreshes, the list of heroes appears.

INTERACTIVE ELEMENTS

Inside the `<li>` element, add a `<button>` element to wrap the hero's details, and then make the hero clickable. To improve accessibility, use HTML elements that are inherently interactive instead of adding an event listener to a non-interactive element. In this case, the interactive `<button>` element is used instead of adding an event to the `<li>` element.

For more details on accessibility, see Accessibility in Angular.

Style the heroes

The heroes list should be attractive and should respond visually when users hover over and select a hero from the list.

In the first tutorial, you set the basic styles for the entire application in `styles.css`. That style sheet didn't include styles for this list of heroes.

You could add more styles to `styles.css` and keep growing that style sheet as you add components.

You may prefer instead to define private styles for a specific component. This keeps everything a component needs, such as the code, the HTML, and the CSS, together in one place.

This approach makes it easier to re-use the component somewhere else and deliver the component's intended appearance even if the global styles are different.

You define private styles either inline in the `@Component.styles` array or as style sheet files identified in the `@Component.styleUrls` array.

When the `ng generate` created the `HeroesComponent`, it created an empty `heroes.component.css` style sheet for the `HeroesComponent` and pointed to it in `@Component.styleUrls` like this.

src/app/heroes/heroes.component.ts (@Component)

```
@Component({
  selector: 'app-heroes',
  templateUrl: './heroes.component.html',
  styleUrls: ['./heroes.component.css']
})
```

Open the `heroes.component.css` file and paste in the private CSS styles for the `HeroesComponent` from the final code review.

> Styles and style sheets 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.

Viewing details

When the user clicks a hero in the list, the component should display the selected hero's details at the bottom of the page.

The code in this section listens for the hero item click event and display/update the hero details.

Add a click event binding

Add a click event binding to the `<button>` in the `<li>` like this:

heroes.component.html (template excerpt)

```
<li *ngFor="let hero of heroes">
  <button type="button" (click)="onSelect(hero)">
  <!-- ... -->
```

This is an example of Angular's event binding syntax.

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

In the next section, define an `onSelect()` method in `HeroesComponent` to display the hero that was defined in the `*ngFor` expression.

Add the click event handler

Rename the component's `hero` property to `selectedHero` but don't assign any value to it since 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`.

src/app/heroes/heroes.component.ts (onSelect)

```
selectedHero?: Hero;
onSelect(hero: Hero): void {
  this.selectedHero = hero;
}
```

Add a details section

Currently, you have a list in the component template. To show details about a hero when you click their name in the list, add a section in the template that displays their details. Add the following to `heroes.component.html` beneath the list section:

heroes.component.html (selected hero details)

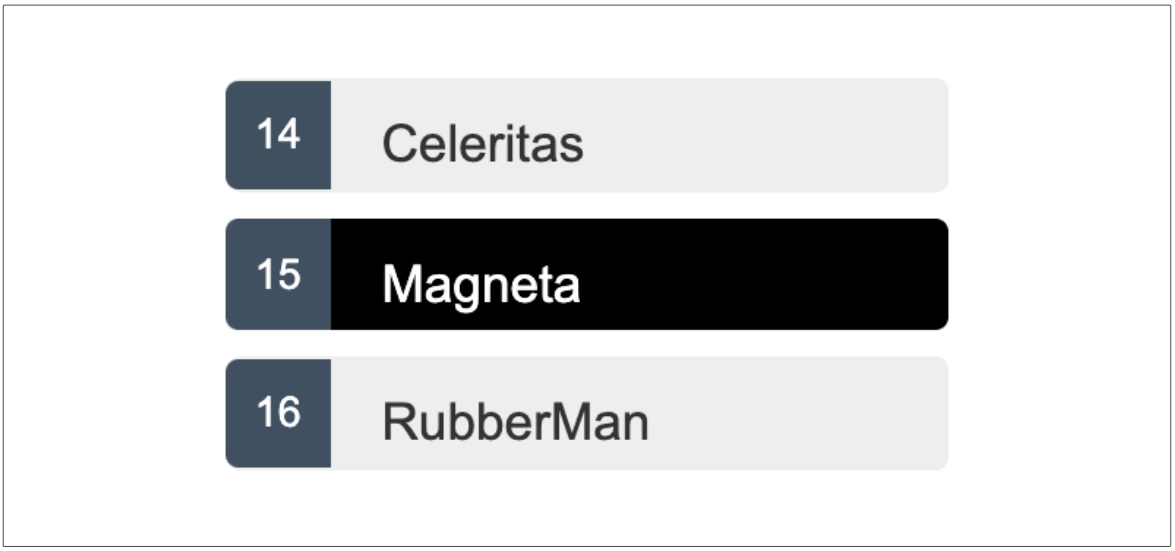
```
<div *ngIf="selectedHero">
  <h2>{{selectedHero.name | uppercase}} Details</h2>
  <div>id: {{selectedHero.id}}</div>
  <div>
    <label for="hero-name">Hero name: </label>
    <input id="hero-name" [(ngModel)]="selectedHero.name" placeholder="name">
  </div>
</div>
```

The hero details should only be displayed when a hero is selected. When a component is created initially, there is no selected hero. Add the `*ngIf` directive to the `<div>` that wraps the hero details. This directive tells Angular to render the section only when the `selectedHero` is defined after it has been selected by clicking on a hero.

> Don't forget the asterisk `*` character in front of `ngIf`. It's a critical part of the syntax.

Style the selected hero

To help identify the selected hero, you can use the `.selected` CSS class in the styles you added earlier. To apply the `.selected` class to the `<li>` when the user clicks it, use class binding.



Angular's class binding can add and remove a CSS class conditionally. Add `[class.some-css-class]="some-condition"` to the element you want to style.

Add the following `[class.selected]` binding to the `<button>` in the `HeroesComponent` template:

heroes.component.html (toggle the 'selected' CSS class)

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

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.

The finished `<li>` looks like this:

heroes.component.html (list item hero)

```
<li *ngFor="let hero of heroes">
  <button [class.selected]="hero === selectedHero" type="button"
(click)="onSelect(hero)">
    <span class="badge">{{hero.id}}</span>
    <span class="name">{{hero.name}}</span>
  </button>
</li>
```

Final code review

Here are the code files discussed on this page, including the `HeroesComponent` styles.

src/app/mock-heroes.ts *src/app/heroes/heroes.component.ts* *src/app/heroes/heroes.component.h*

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

export const HEROES: Hero[] = [
  { id: 12, name: 'Dr. Nice' },
  { 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' }
];
```

Summary

- The Tour of Heroes application displays a list of heroes with a detail view.
- The user can select a hero and see that hero's details.
- You used `*ngFor` to display a list.
- You used `*ngIf` to conditionally include or exclude a block of HTML.
- You can toggle a CSS style class with a `class` binding.

Last reviewed on Mon May 23 2022

Create a feature component

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 won't be maintainable. This tutorial splits up large components into smaller subcomponents, each focused on a specific task or workflow.

The first step is to move the hero details into a separate, reusable `HeroDetailComponent` and end up with:

- A `HeroesComponent` that presents the list of heroes.
- A `HeroDetailComponent` that presents the details of a selected hero.

> For the sample application that this page describes, see the live example / download example.

Make the `HeroDetailComponent`

Use this `ng generate` command to create a new component named `hero-detail`.

```
ng generate component hero-detail
```

The command scaffolds the following:

- Creates a directory `src/app/hero-detail`.

Inside that directory, four files are created:

- A CSS file for the component styles.
- An HTML file for the component template.
- A TypeScript file with a component class named `HeroDetailComponent`.
- A test file for the `HeroDetailComponent` class.

The command also adds the `HeroDetailComponent` as a declaration in the `@NgModule` decorator of the `src/app/app.module.ts` file.

Write the template

Cut the HTML for the hero detail from the bottom of the `HeroesComponent` template and paste it over the boilerplate content in the `HeroDetailComponent` template.

The pasted HTML refers to a `selectedHero`. The new `HeroDetailComponent` can present *any* hero, not just a selected hero. Replace `selectedHero` with `hero` everywhere in the template.

When you're done, the `HeroDetailComponent` template should look like this:

src/app/hero-detail/hero-detail.component.html

```
<div *ngIf="hero">

  <h2>{{hero.name | uppercase}} Details</h2>
  <div><span>id: </span>{{hero.id}}</div>
  <div>
    <label for="hero-name">Hero name: </label>
    <input id="hero-name" [(ngModel)]="hero.name" placeholder="name">
  </div>

</div>
```

Add the `@Input()` hero property

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

Open the `HeroDetailComponent` class file and import the `Hero` symbol.

src/app/hero-detail/hero-detail.component.ts (import Hero)

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

The `hero` property must be an `Input` property, annotated with the `@Input()` decorator, because the *external* `HeroesComponent` binds to it like this.

```
<app-hero-detail [hero]="selectedHero"></app-hero-detail>
```

Amend the `@angular/core` import statement to include the `Input` symbol.

src/app/hero-detail/hero-detail.component.ts (import Input)

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

Add a `hero` property, preceded by the `@Input()` decorator.

src/app/hero-detail/hero-detail.component.ts

```
@Input() hero?: Hero;
```

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

Show the `HeroDetailComponent`

The `HeroesComponent` used to display the hero details on its own, before you removed that part of the template. This section guides you through delegating logic to the `HeroDetailComponent`.

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

You don't need to change the `HeroesComponent` *class*, instead change its *template*.

Update the `HeroesComponent` template

The `HeroDetailComponent` selector is `'app-hero-detail'`. Add an `<app-hero-detail>` element near the bottom of the `HeroesComponent` template, where the hero detail view used to be.

Bind the `HeroesComponent.selectedHero` to the element's `hero` property like this.

heroes.component.html (HeroDetail binding)

```
<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.

The revised `HeroesComponent` template should look like this:

heroes.component.html

```
<h2>My Heroes</h2>

<ul class="heroes">
  <li *ngFor="let hero of heroes">
    <button [class.selected]="hero === selectedHero" type="button"
(click)="onSelect(hero)">
      <span class="badge">{{hero.id}}</span>
      <span class="name">{{hero.name}}</span>
    </button>
  </li>
</ul>

<app-hero-detail [hero]="selectedHero"></app-hero-detail>
```

The browser refreshes and the application starts working again as it did before.

What changed?

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:

1. You reduced the `HeroesComponent` responsibilities.
2. You can evolve the `HeroDetailComponent` into a rich hero editor without touching the parent `HeroesComponent`.
3. You can evolve the `HeroesComponent` without touching the hero detail view.
4. You can re-use the `HeroDetailComponent` in the template of some future component.

Final code review

Here are the code files discussed on this page.

src/app/hero-detail/hero-detail.component.ts *src/app/hero-detail/hero-detail.component.html* *src/*

```
import { Component, Input } from '@angular/core';
import { Hero } from '../hero';

@Component({
  selector: 'app-hero-detail',
  templateUrl: './hero-detail.component.html',
  styleUrls: ['./hero-detail.component.css']
})
export class HeroDetailComponent {
  @Input() hero?: Hero;
}
```

Summary

- You created a separate, reusable `HeroDetailComponent`.
- You used a property binding to give the parent `HeroesComponent` control over the child `HeroDetailComponent`.
- You used the `@Input` decorator to make the `hero` property available for binding by the external `HeroesComponent`.

Add services

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

Refactoring the `HeroesComponent` focuses on supporting the view and making it easier to unit-test with a mock service.

> For the sample application that this page describes, see the live example / download example.

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.

This tutorial creates a `HeroService` that all application classes can use to get heroes. Instead of creating that service with the `new` keyword, use the *dependency injection* that Angular supports to inject it into the `HeroesComponent` constructor.

Services are a great way to share information among classes that *don't know each other*. Create a `MessageService` next and inject it in these two places.

- Inject in `HeroService`, which uses the service to send a message
- Inject in `MessagesComponent`, which displays that message, and also displays the ID when the user clicks a hero

---

Create the `HeroService`

Run `ng generate` to create a service called `hero`.

```
ng generate service hero
```

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

src/app/hero.service.ts (new service)

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

@Injectable({
  providedIn: 'root',
})
export class HeroService {

  constructor() { }

}
```

`@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.

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 such as 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 continues to deliver *mock heroes*.

Import the `Hero` and `HEROES`.

src/app/hero.service.ts

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

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

src/app/hero.service.ts

```
getHeroes(): Hero[] {
  return HEROES;
}
```

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*. The *injector* is the object that chooses and injects the provider where the application requires it.

By default, `ng generate service` registers a provider with the *root injector* for your service by including provider metadata, that's `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 isn't used.

> 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 allows you to provide and use the `HeroService`. At this point, the code differs 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.

src/app/heroes/heroes.component.ts (import HeroService)

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

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

src/app/heroes/heroes.component.ts

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

Inject the `HeroService`

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

src/app/heroes/heroes.component.ts

```
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`.

Add `getHeroes()`

Create a method to retrieve the heroes from the service.

src/app/heroes/heroes.component.ts

```
getHeroes(): void {
  this.heroes = this.heroService.getHeroes();
}
```

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.

src/app/heroes/heroes.component.ts

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

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 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.

src/app/heroes/heroes.component.ts

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

This approach won't work in a real application that uses asynchronous calls. It works now because your service synchronously returns *mock heroes*.

If `getHeroes()` can't return immediately with hero data, it shouldn't be synchronous, because that would block the browser as it waits to return data.

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

In this tutorial, `HeroService.getHeroes()` returns an `Observable` so that it can use the Angular `HttpClient.get` method to fetch the heroes and have `HttpClient.get()` return an `Observable`.

Observable `HeroService`

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

In the tutorial on HTTP, you can see how Angular's `HttpClient` methods return RxJS `Observable` objects. This tutorial simulates getting data from the server with the RxJS `of()` function.

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

src/app/hero.service.ts (Observable imports)

```
import { Observable, of } from 'rxjs';
```

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

src/app/hero.service.ts

```
getHeroes(): Observable<Hero[]> {
  const heroes = of(HEROES);
  return heroes;
}
```

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

> The HTTP tutorial shows you how to 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 need to adjust your application to work with that change to `HeroesComponent`.

Find the `getHeroes` method and replace it with the following code. the new code is shown side-by-side with the current version for comparison.

heroes.component.ts (Observable) *heroes.component.ts (Original)*

```
getHeroes(): void {
  this.heroService.getHeroes()
      .subscribe(heroes => this.heroes = heroes);
}
```

`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 *works* 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, application-wide `MessageService` for sending messages to be displayed
- Injecting `MessageService` into the `HeroService`
- Displaying a message when `HeroService` fetches heroes successfully

Create `MessagesComponent`

Use `ng generate` to create the `MessagesComponent`.

```
ng generate component messages
```

`ng generate` creates the component files in the `src/app/messages` directory and declares the `MessagesComponent` in `AppModule`.

Edit the `AppComponent` template to display the `MessagesComponent`.

src/app/app.component.html

```
<h1>{{title}}</h1>
<app-heroes></app-heroes>
<app-messages></app-messages>
```

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

Create the `MessageService`

Use `ng generate` to create the `MessageService` in `src/app`.

```
ng generate service message
```

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

src/app/message.service.ts

```
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.
- Another to `clear()` the cache.

Inject it into the `HeroService`

In `HeroService`, import the `MessageService`.

src/app/hero.service.ts (import MessageService)

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

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

src/app/hero.service.ts

```
constructor(private messageService: MessageService) { }
```

> This is an example of a typical *service-in-service* scenario in which you inject the `MessageService` into the `HeroService` which is injected into the `HeroesComponent`.

Send a message from `HeroService`

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

src/app/hero.service.ts

```
getHeroes(): Observable<Hero[]> {
  const heroes = of(HEROES);
  this.messageService.add('HeroService: fetched heroes');
  return heroes;
}
```

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`.

src/app/messages/messages.component.ts (import MessageService)

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

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

src/app/messages/messages.component.ts

```
constructor(public messageService: MessageService) {}
```

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 `MessagesComponent` template created by `ng generate` with the following.

src/app/messages/messages.component.html

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

  <h2>Messages</h2>
  <button type="button" class="clear"
          (click)="messageService.clear()">Clear messages</button>
  <div *ngFor='let message of messageService.messages'> {{message}} </div>

</div>
```

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

	DETAILS
`*ngIf`	Only displays the messages area if there are messages to show.
`*ngFor`	Presents the list of messages in repeated `<div>` elements.
Angular event binding	Binds the button's click event to `MessageService.clear()`.

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

Add MessageService to HeroesComponent

The following example shows how to display a history of each time the user clicks on a hero. This helps when you get to the next section on Routing.

src/app/heroes/heroes.component.ts

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

import { Hero } from '../hero';
import { HeroService } from '../hero.service';
import { MessageService } from '../message.service';

@Component({
  selector: 'app-heroes',
  templateUrl: './heroes.component.html',
  styleUrls: ['./heroes.component.css']
})
export class HeroesComponent implements OnInit {

  selectedHero?: Hero;

  heroes: Hero[] = [];

  constructor(private heroService: HeroService, private messageService:
MessageService) { }

  ngOnInit(): void {
    this.getHeroes();
  }

  onSelect(hero: Hero): void {
    this.selectedHero = hero;
    this.messageService.add(`HeroesComponent: Selected hero id=${hero.id}`);
  }

  getHeroes(): void {
    this.heroService.getHeroes()
        .subscribe(heroes => this.heroes = heroes);
  }
}
```

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.

Final code review

Here are the code files discussed on this page.

src/app/hero.service.ts *src/app/message.service.ts* *src/app/heroes/heroes.component.ts* *sr*

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

import { Observable, of } from 'rxjs';

import { Hero } from './hero';
import { HEROES } from './mock-heroes';
import { MessageService } from './message.service';

@Injectable({
  providedIn: 'root',
})
export class HeroService {

  constructor(private messageService: MessageService) { }

  getHeroes(): Observable<Hero[]> {
    const heroes = of(HEROES);
    this.messageService.add('HeroService: fetched heroes');
    return heroes;
  }
}
```

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 `Observable<Hero[]>`, an observable of mock heroes.
- 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`.

Last reviewed on Mon Feb 28 2022

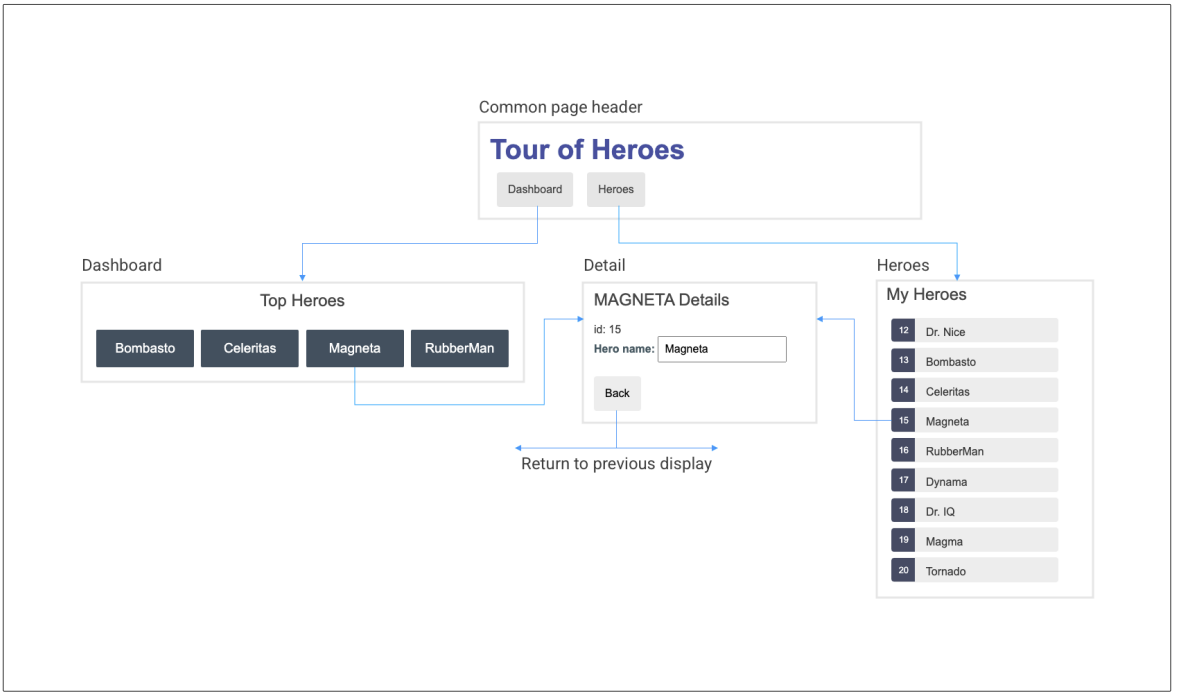
Add navigation with routing

The Tour of Heroes application has new requirements:

- 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

> For the sample application that this page describes, see the live example / download example.

When you're done, users can navigate the application like this:



Add the `AppRoutingModule`

In Angular, the best practice is to load and configure the router in a separate, top-level module. The router 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` directory.

Run `ng generate` to create the application routing module.

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

PARAMETER	DETAILS
`--flat`	Puts the file in `src/app` instead of its own directory.
`--module=app`	Tells `ng generate` to register it in the `imports` array of the `AppModule`.

The file that `ng generate` creates looks like this:

src/app/app-routing.module.ts (generated)

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

@NgModule({
  imports: [
    CommonModule
  ],
  declarations: []
})
export class AppRoutingModule { }
```

Replace it with the following:

src/app/app-routing.module.ts (updated)

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { HeroesComponent } from './heroes/heroes.component';

const routes: Routes = [
  { path: 'heroes', component: HeroesComponent }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

First, the `app-routing.module.ts` file imports `RouterModule` and `Routes` so the application can have routing capability. The next import, `HeroesComponent`, gives the Router somewhere to go once you configure the routes.

Notice that the `CommonModule` references and `declarations` array are unnecessary, so are no longer part of `AppRoutingModule`. The following sections explain the rest of the `AppRoutingModule` in more detail.

Routes

The next part of the file is where you configure your routes. *Routes* tell the Router which view to display when a

user clicks a link or pastes a URL into the browser address bar.

Since `app-routing.module.ts` already imports `HeroesComponent`, you can use it in the `routes` array:

src/app/app-routing.module.ts

```
const routes: Routes = [
  { path: 'heroes', component: HeroesComponent }
];
```

A typical Angular `Route` has two properties:

PROPERTIES	DETAILS
`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.

This tells the router to match that URL to `path: 'heroes'` and display the `HeroesComponent` when the URL is something like `localhost:4200/heroes`.

`RouterModule.forRoot()`

The `@NgModule` metadata initializes the router and starts it listening for browser location changes.

The following line adds the `RouterModule` to the `AppRoutingModule` `imports` array and configures it with the `routes` in one step by calling `RouterModule.forRoot()`:

src/app/app-routing.module.ts

```
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.

Next, `AppRoutingModule` exports `RouterModule` to be available throughout the application.

src/app/app-routing.module.ts (exports array)

```
exports: [ RouterModule ]
```

Add `RouterOutlet`

Open the `AppComponent` template and replace the `<app-heroes>` element with a `<router-outlet>` element.

src/app/app.component.html (router-outlet)

```
<h1>{{title}}</h1>
<router-outlet></router-outlet>
<app-messages></app-messages>
```

The `AppComponent` template no longer needs `<app-heroes>` because the application only displays the `HeroesComponent` when the user navigates to it.

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`. The `ng generate` command you ran at the start of this tutorial added this import because of the `--module=app` flag. If you didn't use the `ng generate` command to create `app-routing.module.ts`, import `AppRoutingModule` into `app.module.ts` and add it to the `imports` array of the `NgModule`.

Try it

If you're not still serving your application, run `ng serve` to see your application in the browser.

The browser should refresh and display the application title but not the list of heroes.

Look at the browser's address bar. The URL ends in `/`. The route path to `HeroesComponent` is `/heroes`.

Append `/heroes` to the URL in the browser address bar. You should see the familiar heroes overview/detail view.

Remove `/heroes` from the URL in the browser address bar. The browser should refresh and display the application title but not the list of heroes.

Add a navigation link using `routerLink`

Ideally, users should be able to click a link to navigate rather than pasting a route URL into the address bar.

Add a `<nav>` element and, within that, an anchor element that, when clicked, triggers navigation to the `HeroesComponent`. The revised `AppComponent` template looks like this:

src/app/app.component.html (heroes RouterLink)

```
<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`.

The browser refreshes and displays the application title and heroes link, but not the heroes list.

Click the link. The address bar updates to `/heroes` and the list of heroes appears.

> Make this and future navigation links look better by adding private CSS styles to `app.component.css` as listed in the final code review below.

Add a dashboard view

Routing makes more sense when your application has more than one view, yet the *Tour of Heroes* application has only the heroes view.

To add a `DashboardComponent`, run `ng generate` as shown here:

```
ng generate component dashboard
```

`ng generate` creates the files for the `DashboardComponent` and declares it in `AppModule`.

Replace the default content in these files as shown here:

src/app/dashboard/dashboard.component.html *src/app/dashboard/dashboard.component.ts* *src/a*

```
<h2>Top Heroes</h2>
<div class="heroes-menu">
  <a *ngFor="let hero of heroes">
    {{hero.name}}
  </a>
</div>
```

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 links don't go anywhere yet.

The *class* is like 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()`

This `getHeroes()` returns the sliced list of heroes at positions 1 and 5, returning only Heroes two, three, four, and five.

src/app/dashboard/dashboard.component.ts

```
getHeroes(): void {
  this.heroService.getHeroes()
    .subscribe(heroes => this.heroes = heroes.slice(1, 5));
}
```

Add the dashboard route

To navigate to the dashboard, the router needs an appropriate route.

Import the `DashboardComponent` in the `app-routing.module.ts` file.

src/app/app-routing.module.ts (import DashboardComponent)

```
import { DashboardComponent } from './dashboard/dashboard.component';
```

Add a route to the `routes` array that matches a path to the `DashboardComponent`.

src/app/app-routing.module.ts

```
{ path: 'dashboard', component: DashboardComponent },
```

Add a default route

When the application starts, the browser's 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 application navigate to the dashboard automatically, add the following route to the `routes` array.

src/app/app-routing.module.ts

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

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

After the browser refreshes, the router loads the `DashboardComponent` and the browser address bar shows the `/dashboard` URL.

Add dashboard link to the shell

The user should be able to navigate 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.

src/app/app.component.html

```
<h1>{{title}}</h1>
<nav>
  <a routerLink="/dashboard">Dashboard</a>
  <a routerLink="/heroes">Heroes</a>
</nav>
<router-outlet></router-outlet>
<app-messages></app-messages>
```

After the browser refreshes you can navigate freely between the two views by clicking the links.

Navigating to hero details

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

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

1. By clicking a hero in the dashboard.
2. By clicking a hero in the heroes list.
3. By pasting a "deep link" URL into the browser address bar that identifies the hero to display.

This section enables navigation to the `HeroDetailComponent` and liberates it from the `HeroesComponent`.

Delete *hero details* from `HeroesComponent`

When the user clicks a hero in `HeroesComponent`, the application should navigate to the `HeroDetailComponent`, replacing the heroes list view with the hero detail view. The heroes list view should no longer show hero details as it does now.

Open the `heroes/heroes.component.html` and delete the `<app-hero-detail>` element from the bottom.

Clicking a hero item now does nothing. You can fix that after you enable routing to the `HeroDetailComponent`.

Add a *hero detail* route

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

Open `app-routing.module.ts` and import `HeroDetailComponent`.

src/app/app-routing.module.ts (import HeroDetailComponent)

```
import { HeroDetailComponent } from './hero-detail/hero-detail.component';
```

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

src/app/app-routing.module.ts

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

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

At this point, all application routes are in place.

src/app/app-routing.module.ts (all routes)

```
const routes: Routes = [
  { path: '', redirectTo: '/dashboard', pathMatch: 'full' },
  { path: 'dashboard', component: DashboardComponent },
  { path: 'detail/:id', component: HeroDetailComponent },
  { path: 'heroes', component: HeroesComponent }
];
```

`DashboardComponent` hero links

The `DashboardComponent` hero links do nothing at the moment.

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

src/app/dashboard/dashboard.component.html (hero links)

```
<a *ngFor="let hero of heroes"
  routerLink="/detail/{{hero.id}}">
  {{hero.name}}
</a>
```

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

`HeroesComponent` hero links

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

src/app/heroes/heroes.component.html (list with onSelect)

```
<ul class="heroes">
  <li *ngFor="let hero of heroes">
    <button type="button" (click)="onSelect(hero)" [class.selected]="hero ===
selectedHero">
      <span class="badge">{{hero.id}}</span>
      <span class="name">{{hero.name}}</span>
    </button>
  </li>
</ul>
```

Remove the inner HTML of `<li>`. Wrap the badge and name in an anchor `<a>` element. Add a `routerLink` attribute to the anchor that's the same as in the dashboard template.

src/app/heroes/heroes.component.html (list with links)

```
<ul class="heroes">
  <li *ngFor="let hero of heroes">
    <a routerLink="/detail/{{hero.id}}">
      <span class="badge">{{hero.id}}</span> {{hero.name}}
    </a>
  </li>
</ul>
```

Be sure to fix the private style sheet in `heroes.component.css` to make the list look as it did before. Revised styles are in the final code review at the bottom of this guide.

Remove dead code - optional

While the `HeroesComponent` class still works, the `onSelect()` method and `selectedHero` property are no longer used.

It's nice to tidy things up for your future self. Here's the class after pruning away the dead code.

src/app/heroes/heroes.component.ts (cleaned up)

```
export class HeroesComponent implements OnInit {
  heroes: Hero[] = [];

  constructor(private heroService: HeroService) { }

  ngOnInit(): void {
    this.getHeroes();
  }

  getHeroes(): void {
    this.heroService.getHeroes()
    .subscribe(heroes => this.heroes = heroes);
  }
}
```

Routable `HeroDetailComponent`

The parent `HeroesComponent` used to 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/12`.

The `HeroDetailComponent` needs a new way to get the hero to display. This section explains the following:

- Get the route that created it
- Extract the `id` from the route
- Get the hero with that `id` from the server using the `HeroService`

Add the following imports:

src/app/hero-detail/hero-detail.component.ts

```
import { ActivatedRoute } from '@angular/router';
import { Location } from '@angular/common';

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

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

src/app/hero-detail/hero-detail.component.ts

```
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 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 uses it to get the hero-to-display.

The `location` is an Angular service for interacting with the browser. This service lets you navigate back to the previous view.

Extract the `id` route parameter

In the `ngOnInit()` lifecycle hook call `getHero()` and define it as follows.

src/app/hero-detail/hero-detail.component.ts

```
ngOnInit(): void {
  this.getHero();
}

getHero(): void {
  const id = Number(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 `Number` function converts the string to a number, which is what a hero `id` should be.

The browser refreshes and the application crashes with a compiler error. `HeroService` doesn't have a `getHero()` method. Add it now.

Add `HeroService.getHero()`

Open `HeroService` and add the following `getHero()` method with the `id` after the `getHeroes()` method:

src/app/hero.service.ts (getHero)

```
getHero(id: number): Observable<Hero> {
  // For now, assume that a hero with the specified `id` always exists.
  // Error handling will be added in the next step of the tutorial.
  const hero = HEROES.find(h => h.id === id)!;
  this.messageService.add(`HeroService: fetched hero id=${id}`);
  return of(hero);
}
```

> **IMPORTANT**:
> The backtick (`` ` ``) characters 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.

You can rewrite `getHero()` as a real `Http` request without having to change the `HeroDetailComponent` that calls it.

Try it

The browser refreshes and the application is working again. You can click a hero in the dashboard or in the heroes list and navigate to that hero's detail view.

If you paste `localhost:4200/detail/12` in the browser address bar, the router navigates to the detail view for the hero with `id: 12`, **Dr Nice**.

Find the way back

By clicking the browser's back button, you can go back to the previous page. This could be 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.

src/app/hero-detail/hero-detail.component.html (back button)

```
<button type="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 used to inject.

src/app/hero-detail/hero-detail.component.ts (goBack)

```
goBack(): void {
  this.location.back();
}
```

Refresh the browser and start clicking. Users can now navigate around the application using the new buttons.

The details look better when you add the private CSS styles to `hero-detail.component.css` as listed in one of the "final code review" tabs below.

Final code review

Here are the code files discussed on this page.

`AppRoutingModule`, `AppModule`, and `HeroService`

src/app/app.module.ts *src/app/app-routing.module.ts* *src/app/hero.service.ts*

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';

import { AppComponent } from './app.component';
import { DashboardComponent } from './dashboard/dashboard.component';
import { HeroDetailComponent } from './hero-detail/hero-detail.component';
import { HeroesComponent } from './heroes/heroes.component';
import { MessagesComponent } from './messages/messages.component';

import { AppRoutingModule } from './app-routing.module';

@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    AppRoutingModule
  ],
  declarations: [
    AppComponent,
    DashboardComponent,
    HeroesComponent,
    HeroDetailComponent,
    MessagesComponent
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

`AppComponent`

src/app/app.component.html *src/app/app.component.ts* *src/app/app.component.css*

```
<h1>{{title}}</h1>
<nav>
  <a routerLink="/dashboard">Dashboard</a>
  <a routerLink="/heroes">Heroes</a>
</nav>
<router-outlet></router-outlet>
<app-messages></app-messages>
```

DashboardComponent

src/app/dashboard/dashboard.component.html *src/app/dashboard/dashboard.component.ts* *src/a*

```
<h2>Top Heroes</h2>
<div class="heroes-menu">
  <a *ngFor="let hero of heroes"
    routerLink="/detail/{{hero.id}}">
    {{hero.name}}
  </a>
</div>
```

HeroesComponent

src/app/heroes/heroes.component.html *src/app/heroes/heroes.component.ts* *src/app/heroes/hero*

```
<h2>My Heroes</h2>
<ul class="heroes">
  <li *ngFor="let hero of heroes">
    <a routerLink="/detail/{{hero.id}}">
      <span class="badge">{{hero.id}}</span> {{hero.name}}
    </a>
  </li>
</ul>
```

HeroDetailComponent

src/app/hero-detail/hero-detail.component.html *src/app/hero-detail/hero-detail.component.ts* *src/*

```
<div *ngIf="hero">
  <h2>{{hero.name | uppercase}} Details</h2>
  <div><span>id: </span>{{hero.id}}</div>
  <div>
    <label for="hero-name">Hero name: </label>
    <input id="hero-name" [(ngModel)]="hero.name" placeholder="Hero name"/>
  </div>
  <button type="button" (click)="goBack()">go back</button>
</div>
```

Summary

- You added the Angular router to navigate among different components
- You turned the `AppComponent` into a navigation shell with `<a>` links and a `<router-outlet>`
- You configured the router in an `AppRoutingModule`
- You defined routes, a redirect route, and a parameterized route
- You used the `routerLink` directive in anchor elements
- You refactored a tightly coupled main/detail view into a routed detail view
- You used router link parameters to navigate to the detail view of a user-selected hero
- You shared the `HeroService` with other components

Last reviewed on Mon Feb 28 2022

Get data from a server

This tutorial adds 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 live example / download example.

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:

src/app/app.module.ts (HttpClientModule import)

```
import { HttpClientModule } from '@angular/common/http';
```

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

src/app/app.module.ts (imports array excerpt)

```
@NgModule({
  imports: [
    HttpClientModule,
  ],
})
```

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 makes requests to and receive responses from the `HttpClient`. The application doesn't know 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
```

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:

src/app/in-memory-data.service.ts

```
import { Injectable } from '@angular/core';
import { InMemoryDbService } from 'angular-in-memory-web-api';
import { Hero } from './hero';

@Injectable({
  providedIn: 'root',
})
export class InMemoryDataService implements InMemoryDbService {
  createDb() {
    const heroes = [
      { id: 12, name: 'Dr. Nice' },
      { 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};
  }

  // Overrides the genId method to ensure that a hero always has an id.
  // If the heroes array is empty,
  // the method below returns the initial number (11).
  // if the heroes array is not empty, the method below returns the highest
  // hero id + 1.
  genId(heroes: Hero[]): number {
    return heroes.length > 0 ? Math.max(...heroes.map(hero => hero.id)) + 1 : 11;
  }
}
```

In the `AppModule`, import the `HttpClientInMemoryWebApiModule` and the `InMemoryDataService` class, which you create next.

src/app/app.module.ts (In-memory Web API imports)

```
import { HttpClientInMemoryWebApiModule } from 'angular-in-memory-web-api';
import { InMemoryDataService } from './in-memory-data.service';
```

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

src/app/app.module.ts (imports array excerpt)

```
HttpClientModule,

// The HttpClientInMemoryWebApiModule module intercepts HTTP requests
// and returns simulated server responses.
// Remove it when a 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.

The `in-memory-data.service.ts` file takes over the function of `mock-heroes.ts`. Don't delete `mock-heroes.ts` yet. You still need it for a few more steps of this tutorial.

After the server is ready, detach the In-memory Web API so the application's requests can go through to the server.

Heroes and HTTP

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

src/app/hero.service.ts (import HTTP symbols)

```
import { HttpClient, HttpHeaders } from '@angular/common/http';
```

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

src/app/hero.service.ts

```
constructor(
  private http: HttpClient,
  private messageService: MessageService) { }
```

Notice that you keep injecting the `MessageService` but since your application calls it so frequently, wrap it in a private `log()` method:

src/app/hero.service.ts

```
/** Log a HeroService message with the MessageService */
private log(message: string) {
  this.messageService.add(`HeroService: ${message}`);
}
```

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`.

src/app/hero.service.ts

```
private heroesUrl = 'api/heroes';  // URL to web api
```

Get heroes with `HttpClient`

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

src/app/hero.service.ts (getHeroes with RxJs 'of()')

```
getHeroes(): Observable<Hero[]> {
  const heroes = of(HEROES);
  return heroes;
}
```

Convert that method to use `HttpClient` as follows:

src/app/hero.service.ts

```
/** GET heroes from the server */
getHeroes(): Observable<Hero[]> {
  return this.http.get<Hero[]>(this.heroesUrl)
}
```

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 more than one value over time. An observable from `HttpClient` always emits a single value and then completes, never to emit again.

This particular call to `HttpClient.get()` returns an `Observable<Hero[]>`, which is *an observable of hero arrays*. In practice, it only returns 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 to use later.

src/app/hero.service.ts

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

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

src/app/hero.service.ts

```
getHeroes(): Observable<Hero[]> {
  return this.http.get<Hero[]>(this.heroesUrl)
    .pipe(
      catchError(this.handleError<Hero[]>('getHeroes', []))
    );
}
```

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()` can 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`. This function is configured with both the name of the operation that failed and a safe return value.

src/app/hero.service.ts

```
/**
 * Handle Http operation that failed.
 * Let the app continue.
 *
 * @param operation - name of the operation that failed
 * @param result - optional value to return as the observable result
 */
private handleError<T>(operation = 'operation', result?: T) {
  return (error: any): Observable<T> => {

    // TODO: send the error to remote logging infrastructure
    console.error(error); // log to console instead

    // TODO: better job of transforming error for user consumption
    this.log(`${operation} failed: ${error.message}`);

    // Let the app keep running by returning an empty result.
    return of(result as T);
  };
}
```

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

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

Tap into the Observable

The `getHeros()` method taps into the flow of observable values and sends a message, using the `log()` method, to the message area at the bottom of the page.

The RxJS `tap()` operator enables this ability by looking at the observable values, doing something with those values, and passing them along. The `tap()` callback doesn't access the values themselves.

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

src/app/hero.service.ts

```
/** GET heroes from the server */
getHeroes(): Observable<Hero[]> {
  return this.http.get<Hero[]>(this.heroesUrl)
    .pipe(
      tap(_ => this.log('fetched heroes')),
      catchError(this.handleError<Hero[]>('getHeroes', []))
    );
}
```

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 in `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:

src/app/hero.service.ts

```
/** 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}`))
  );
}
```

`getHero()` has 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>`, which is 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, yet when you click **Go back**, your 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()`.

src/app/hero-detail/hero-detail.component.html (save)

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

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.

src/app/hero-detail/hero-detail.component.ts (save)

```
save(): void {
  if (this.hero) {
    this.heroService.updateHero(this.hero)
      .subscribe(() => this.goBack());
  }
}
```

Add `HeroService.updateHero()`

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

src/app/hero.service.ts (update)

```
/** PUT: update the hero on the server */
updateHero(hero: Hero): Observable<any> {
  return this.http.put(this.heroesUrl, hero, this.httpOptions).pipe(
    tap(_ => this.log(`updated hero id=${hero.id}`)),
    catchError(this.handleError<any>('updateHero'))
  );
}
```

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

- The URL
- The data to update, which is 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.

src/app/hero.service.ts

```
httpOptions = {
  headers: new HttpHeaders({ 'Content-Type': 'application/json' })
};
```

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:

src/app/heroes/heroes.component.html (add)

```
<div>
  <label for="new-hero">Hero name: </label>
  <input id="new-hero" #heroName />

  <!-- (click) passes input value to add() and then clears the input -->
  <button type="button" class="add-button" (click)="add(heroName.value);
heroName.value=''">
    Add hero
  </button>
</div>
```

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:

src/app/heroes/heroes.component.ts (add)

```
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 isn't blank, the handler creates an object based on the hero's name. The handler passes the object name to the service's `addHero()` method.

When `addHero()` creates a new object, the `subscribe()` callback receives the new hero and pushes it into to the `heroes` list for display.

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

src/app/hero.service.ts (addHero)

```
/** POST: add a new hero to the server */
addHero(hero: Hero): Observable<Hero> {
  return this.http.post<Hero>(this.heroesUrl, hero, this.httpOptions).pipe(
    tap((newHero: Hero) => this.log(`added hero w/ id=${newHero.id}`)),
    catchError(this.handleError<Hero>('addHero'))
  );
}
```

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

- It calls `HttpClient.post()` instead of `put()`
- It expects the server to create 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 `<li>` element.

src/app/heroes/heroes.component.html

```
<button type="button" class="delete" title="delete hero"
  (click)="delete(hero)">x</button>
```

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

src/app/heroes/heroes.component.html (list of heroes)

```
<ul class="heroes">
  <li *ngFor="let hero of heroes">
    <a routerLink="/detail/{{hero.id}}">
      <span class="badge">{{hero.id}}</span> {{hero.name}}
    </a>
    <button type="button" class="delete" title="delete hero"
      (click)="delete(hero)">x</button>
  </li>
</ul>
```

To position the delete button at the far right of the hero entry, add some CSS from the final review code to the `heroes.component.css`.

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

src/app/heroes/heroes.component.ts (delete)

```
delete(hero: Hero): void {
  this.heroes = this.heroes.filter(h => h !== hero);
  this.heroService.deleteHero(hero.id).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` succeeds on the server.

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

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

src/app/hero.service.ts (delete)

```
/** DELETE: delete the hero from the server */
deleteHero(id: number): Observable<Hero> {
  const url = `${this.heroesUrl}/${id}`;

  return this.http.delete<Hero>(url, this.httpOptions).pipe(
    tap(_ => this.log(`deleted hero id=${id}`)),
    catchError(this.handleError<Hero>('deleteHero'))
  );
}
```

Notice 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 capability.

If you neglect to `subscribe()`, the service can't 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**. This shows the full list of heroes again.

Search by name

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

Add a heroes search feature to the Dashboard

As the user types a name into a search box, your application makes 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`.

src/app/hero.service.ts

```
/* 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(x => x.length ?
       this.log(`found heroes matching "${term}"`) :
       this.log(`no heroes matching "${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 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.

src/app/dashboard/dashboard.component.html

```
<h2>Top Heroes</h2>
<div class="heroes-menu">
  <a *ngFor="let hero of heroes"
      routerLink="/detail/{{hero.id}}">
      {{hero.name}}
  </a>
</div>

<app-hero-search></app-hero-search>
```

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`

Run `ng generate` to create a `HeroSearchComponent`.

```
ng generate component hero-search
```

`ng generate` creates the three `HeroSearchComponent` files and adds the component to the `AppModule` declarations.

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

src/app/hero-search/hero-search.component.html

```
<div id="search-component">
  <label for="search-box">Hero Search</label>
  <input #searchBox id="search-box" (input)="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>
```

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.

`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.

src/app/hero-search/hero-search.component.html

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

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 `HeroSearchComponent` class and metadata as follows.

src/app/hero-search/hero-search.component.ts

```
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`:

src/app/hero-search/hero-search.component.ts

```
heroes$!: Observable<Hero[]>;
```

Set this in `ngOnInit()`. Before you do, focus on the definition of `searchTerms`.

The `searchTerms` RxJS subject

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

src/app/hero-search/hero-search.component.ts

```
private searchTerms = new Subject<string>();

// Push a 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 event binding to the text box's `input` event calls the `search()` method.

src/app/hero-search/hero-search.component.html

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

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

Chaining RxJS operators

Passing a new search term directly to the `searchHeroes()` after every user keystroke creates excessive HTTP requests, which taxes server resources and burns 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, this returns an observable of timely hero search results where each one is a `Hero[]`.

Here's a closer look at the code.

src/app/hero-search/hero-search.component.ts

```ts
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)),
);
```

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. Requests aren't likely to happen more frequently than 300 ms.
- `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 300 ms pause between requests, you could have many HTTP requests in flight, and they may not return in the order sent.

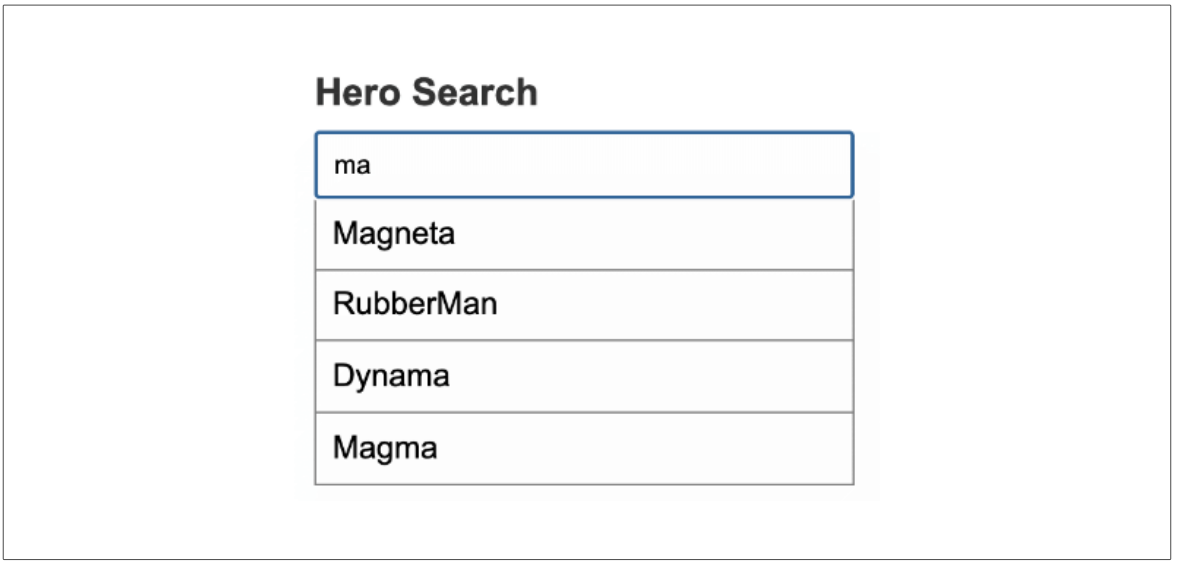
`switchMap()` preserves the original request order while returning only the observable from the most recent HTTP method call. Results from prior calls are canceled and discarded.

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

Remember that the component *class* doesn't 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. Enter characters that match any existing hero names, and look for something like this.



Final code review

Here are the code files discussed on this page. They're found in the `src/app/` directory.

`HeroService`, `InMemoryDataService`, `AppModule`

hero.service.ts *in-memory-data.service.ts* *app.module.ts*

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';

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

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


@Injectable({ providedIn: 'root' })
export class HeroService {

  private heroesUrl = 'api/heroes';  // URL to web api

  httpOptions = {
    headers: new HttpHeaders({ 'Content-Type': 'application/json' })
  };

  constructor(
    private http: HttpClient,
    private messageService: MessageService) { }

  /** GET heroes from the server */
  getHeroes(): Observable<Hero[]> {
    return this.http.get<Hero[]>(this.heroesUrl)
      .pipe(
        tap(_ => this.log('fetched heroes')),
        catchError(this.handleError<Hero[]>('getHeroes', []))
      );
  }

  /** GET hero by id. Return `undefined` when id not found */
  getHeroNo404<Data>(id: number): Observable<Hero> {
    const url = `${this.heroesUrl}/?id=${id}`;
    return this.http.get<Hero[]>(url)
      .pipe(
        map(heroes => heroes[0]), // returns a {0|1} element array
        tap(h => {
          const outcome = h ? 'fetched' : 'did not find';
          this.log(`${outcome} hero id=${id}`);
        }),
        catchError(this.handleError<Hero>(`getHero id=${id}`))
      );
  }

  /** 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}`))
    );
  }

  /* 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(x => x.length ?
       this.log(`found heroes matching "${term}"`) :
       this.log(`no heroes matching "${term}"`)),
    catchError(this.handleError<Hero[]>('searchHeroes', []))
  );
}

//////// Save methods //////////

/** POST: add a new hero to the server */
addHero(hero: Hero): Observable<Hero> {
  return this.http.post<Hero>(this.heroesUrl, hero, this.httpOptions).pipe(
    tap((newHero: Hero) => this.log(`added hero w/ id=${newHero.id}`)),
    catchError(this.handleError<Hero>('addHero'))
  );
}

/** DELETE: delete the hero from the server */
deleteHero(id: number): Observable<Hero> {
  const url = `${this.heroesUrl}/${id}`;

  return this.http.delete<Hero>(url, this.httpOptions).pipe(
    tap(_ => this.log(`deleted hero id=${id}`)),
    catchError(this.handleError<Hero>('deleteHero'))
  );
}

/** PUT: update the hero on the server */
updateHero(hero: Hero): Observable<any> {
  return this.http.put(this.heroesUrl, hero, this.httpOptions).pipe(
    tap(_ => this.log(`updated hero id=${hero.id}`)),
    catchError(this.handleError<any>('updateHero'))
  );
}

/**
 * Handle Http operation that failed.
 * Let the app continue.
 *
 * @param operation - name of the operation that failed
 * @param result - optional value to return as the observable result
 */
private handleError<T>(operation = 'operation', result?: T) {
  return (error: any): Observable<T> => {

    // TODO: send the error to remote logging infrastructure
    console.error(error); // log to console instead

    // TODO: better job of transforming error for user consumption
    this.log(`${operation} failed: ${error.message}`);

    // Let the app keep running by returning an empty result.
    return of(result as T);
  };
}

/** Log a HeroService message with the MessageService */
```

```
  private log(message: string) {
    this.messageService.add(`HeroService: ${message}`);
  }
}
```

HeroesComponent

heroes/heroes.component.html *heroes/heroes.component.ts* *heroes/heroes.component.css*

```
<h2>My Heroes</h2>

<div>
  <label for="new-hero">Hero name: </label>
  <input id="new-hero" #heroName />

  <!-- (click) passes input value to add() and then clears the input -->
  <button type="button" class="add-button" (click)="add(heroName.value);
heroName.value=''">
    Add hero
  </button>
</div>

<ul class="heroes">
  <li *ngFor="let hero of heroes">
    <a routerLink="/detail/{{hero.id}}">
      <span class="badge">{{hero.id}}</span> {{hero.name}}
    </a>
    <button type="button" class="delete" title="delete hero"
      (click)="delete(hero)">x</button>
  </li>
</ul>
```

HeroDetailComponent

hero-detail/hero-detail.component.html *hero-detail/hero-detail.component.ts* *hero-detail/hero-deta*

```
<div *ngIf="hero">
  <h2>{{hero.name | uppercase}} Details</h2>
  <div><span>id: </span>{{hero.id}}</div>
  <div>
    <label for="hero-name">Hero name: </label>
    <input id="hero-name" [(ngModel)]="hero.name" placeholder="Hero name"/>
  </div>
  <button type="button" (click)="goBack()">go back</button>
  <button type="button" (click)="save()">save</button>
</div>
```

DashboardComponent

dashboard/dashboard.component.html *dashboard/dashboard.component.ts* *dashboard/dashboard*

```
<h2>Top Heroes</h2>
<div class="heroes-menu">
  <a *ngFor="let hero of heroes"
      routerLink="/detail/{{hero.id}}">
      {{hero.name}}
  </a>
</div>

<app-hero-search></app-hero-search>
```

`HeroSearchComponent`

hero-search/hero-search.component.html *hero-search/hero-search.component.ts* *hero-search/herc*

```
<div id="search-component">
  <label for="search-box">Hero Search</label>
  <input #searchBox id="search-box" (input)="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>
```

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 application
- 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.

Last reviewed on Mon Feb 28 2022