The Application Shell

You begin by creating an initial application using the Angular CLI. Throughout this tutorial, you’ll modify and extend that starter application to create the Tour of Heroes app.

In this part of the tutorial, you'll do the following:

1. Set up your environment.
2. Create a new workspace and initial app project.
3. Serve the application.
4. Make changes to the application.

Set up your environment

To set up your development environment, follow the instructions in [Local Environment Setup](https://angular.io/guide/setup-local).

Create a new workspace and an initial application

You develop apps in the context of an Angular [workspace](https://angular.io/guide/glossary#workspace). A workspace contains the files for one or more [projects](https://angular.io/guide/glossary#project). A project is the set of files that comprise an app, a library, or end-to-end (e2e) tests. For this tutorial, you will create a new workspace.

To create a new workspace and an initial app project:

1. Ensure that you are not already in an Angular workspace folder. For example, if you have previously created the Getting Started workspace, change to the parent of that folder.
2. Run the CLI command ng new and provide the name angular-tour-of-heroes, as shown here:

ng new angular-tour-of-heroes

1. The ng new command prompts you for information about features to include in the initial app project. Accept the defaults by pressing the Enter or Return key.

The Angular CLI installs the necessary Angular npm packages and other dependencies. This can take a few minutes.

It also creates the following workspace and starter project files:

* A new workspace, with a root folder named angular-tour-of-heroes.
* An initial skeleton app project, also called angular-tour-of-heroes (in the src subfolder).
* An end-to-end test project (in the e2e subfolder).
* Related configuration files.

The initial app project contains a simple Welcome app, 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 app, starts the development server, watches the source files, and rebuilds the app as you make changes to those files.

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

You should see the app 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 and navigate to the src/app folder to make some changes to the starter app.

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

1. app.component.ts— the component class code, written in TypeScript.
2. app.component.html— the component template, written in HTML.
3. app.component.css— the component's private CSS styles.

Change the application title

Open the component class file (app.component.ts) and change the value of the title property to 'Tour of Heroes'.

app.component.ts (class title property)

content\_copytitle = 'Tour of Heroes';

Open the component template file (app.component.html) and delete the default template generated by the Angular CLI. Replace it with the following line of HTML.

app.component.html (template)

content\_copy<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. The CLI generated 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)

content\_copy

1. /\* Application-wide [Styles](https://angular.io/) \*/
2. h1 {
3. color: #369;
4. font-family: Arial, Helvetica, sans-serif;
5. font-size: 250%;
6. }
7. h2, h3 {
8. color: #444;
9. font-family: Arial, Helvetica, sans-serif;
10. font-weight: lighter;
11. }
12. body {
13. margin: 2em;
14. }
15. body, input[type="text"], button {
16. color: #333;
17. font-family: Cambria, Georgia;
18. }
19. /\* everywhere else \*/
20. \* {
21. font-family: Arial, Helvetica, sans-serif;
22. }

Final code review

The source code for this tutorial and the complete *Tour of Heroes* global styles are available in the [live example](https://angular.io/generated/live-examples/toh-pt0/stackblitz.html) / [download example](https://angular.io/generated/zips/toh-pt0/toh-pt0.zip).

Here are the code files discussed on this page.

src/app/app.component.ts

src/app/app.component.html

src/styles.css (excerpt)

content\_copyimport { [Component](https://angular.io/api/core/Component) } from '@angular/core';

@[Component](https://angular.io/api/core/Component)({

selector: 'app-root',

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

[styleUrls](https://angular.io/api/core/Component#styleUrls): ['./app.component.css']

})

export class AppComponent {

title = 'Tour of Heroes';

}

Summary

* You created the initial application structure using the Angular CLI.
* You learned that Angular components display data.
* You used the double curly braces of interpolation to display the app title.

# The Hero Editor

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

## Create the heroes component

Using the Angular CLI, generate a new component named heroes.

content\_copyng generate component heroes

The CLI creates a new folder, src/app/heroes/, and generates the four files of the HeroesComponent.

The HeroesComponent class file is as follows:

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

content\_copyimport { [Component](https://angular.io/api/core/Component), [OnInit](https://angular.io/api/core/OnInit) } from '@angular/core';

@[Component](https://angular.io/api/core/Component)({

selector: 'app-heroes',

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

[styleUrls](https://angular.io/api/core/Component#styleUrls): ['./heroes.component.css']

})

export class HeroesComponent implements [OnInit](https://angular.io/api/core/OnInit) {

constructor() { }

ngOnInit() {

}

}

You always import the [Component](https://angular.io/api/core/Component) symbol from the Angular core library and annotate the component class with @[Component](https://angular.io/api/core/Component).

@[Component](https://angular.io/api/core/Component) is a decorator function that specifies the Angular metadata for the component.

The CLI generated three metadata properties:

1. selector— the component's CSS element selector
2. templateUrl— the location of the component's template file.
3. [styleUrls](https://angular.io/api/core/Component#styleUrls)— the location of the component's private CSS styles.

The [CSS element selector](https://developer.mozilla.org/en-US/docs/Web/CSS/Type_selectors), 'app-heroes', matches the name of the HTML element that identifies this component within a parent component's template.

The ngOnInit is a [lifecycle hook](https://angular.io/guide/lifecycle-hooks#oninit). Angular calls ngOnInit shortly after creating a component. It's a good place to put initialization logic.

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

### Add a hero property

Add a hero property to the HeroesComponent for a hero named "Windstorm."

heroes.component.ts (hero property)

content\_copyhero = 'Windstorm';

### Show the hero

Open the heroes.component.html template file. Delete the default text generated by the Angular CLI and replace it with a data binding to the new hero property.

heroes.component.html

content\_copy{{hero}}

## 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](https://angular.io/tutorial/toh-pt1#selector) for the HeroesComponent. So add an <app-heroes> element to the AppComponent template file, just below the title.

src/app/app.component.html

content\_copy<h1>{{title}}</h1>

<app-heroes></app-heroes>

Assuming that the CLI ng serve command is still running, the browser should refresh and display both the application title and the hero name.

## Create a Hero class

A real hero is more than a name.

Create a Hero class in its own file in the src/app folder. Give it id and name properties.

src/app/hero.ts

content\_copyexport class Hero {

id: number;

name: string;

}

Return to the HeroesComponent class and import the Hero class.

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

content\_copyimport { [Component](https://angular.io/api/core/Component), [OnInit](https://angular.io/api/core/OnInit) } from '@angular/core';

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

@[Component](https://angular.io/api/core/Component)({

selector: 'app-heroes',

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

[styleUrls](https://angular.io/api/core/Component#styleUrls): ['./heroes.component.css']

})

export class HeroesComponent implements [OnInit](https://angular.io/api/core/OnInit) {

hero: Hero = {

id: 1,

name: 'Windstorm'

};

constructor() { }

ngOnInit() {

}

}

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 layout like this:

heroes.component.html (HeroesComponent's template)

content\_copy<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

Modify the hero.name binding like this.

content\_copy<h2>{{hero.name | [uppercase](https://angular.io/api/common/UpperCasePipe)}} Details</h2>

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

The word [uppercase](https://angular.io/api/common/UpperCasePipe) in the interpolation binding, right after the pipe operator ( | ), activates the built-in UppercasePipe.

[Pipes](https://angular.io/guide/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 name in an <input> textbox.

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

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

### 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)

content\_copy<div>

<label>name:

<input [([ngModel](https://angular.io/api/forms/NgModel))]="hero.name" placeholder="name"/>

</label>

</div>

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

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

### The missing FormsModule

Notice that the app stopped working when you added [([ngModel](https://angular.io/api/forms/NgModel))].

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

content\_copy[Template](https://angular.io/) parse errors:

Can't bind to '[ngModel](https://angular.io/api/forms/NgModel)' since it isn't [a](https://angular.io/api/router/RouterLinkWithHref) known property of 'input'.

Although [ngModel](https://angular.io/api/forms/NgModel) is a valid Angular directive, it isn't available by default.

It belongs to the optional [FormsModule](https://angular.io/api/forms/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 app requires. This information is called metadata

Some of the metadata is in the @[Component](https://angular.io/api/core/Component) decorators that you added to your component classes. Other critical metadata is in [@NgModule](https://angular.io/guide/ngmodules) decorators.

The most important @[NgModule](https://angular.io/api/core/NgModule) decorator annotates the top-level **AppModule** class.

The Angular CLI generated an AppModule class in src/app/app.module.ts when it created the project. This is where you opt-in to the [FormsModule](https://angular.io/api/forms/FormsModule).

### Import FormsModule

Open AppModule (app.module.ts) and import the [FormsModule](https://angular.io/api/forms/FormsModule) symbol from the @angular/forms library.

app.module.ts (FormsModule symbol import)

content\_copyimport { [FormsModule](https://angular.io/api/forms/FormsModule) } from '@angular/forms'; // <-- [NgModel](https://angular.io/api/forms/NgModel) lives here

Then add [FormsModule](https://angular.io/api/forms/FormsModule) to the @[NgModule](https://angular.io/api/core/NgModule) metadata's imports array, which contains a list of external modules that the app needs.

app.module.ts ( @NgModule imports)

content\_copyimports: [

[BrowserModule](https://angular.io/api/platform-browser/BrowserModule),

[FormsModule](https://angular.io/api/forms/FormsModule)

],

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

### Declare HeroesComponent

Every component must be declared in exactly one [NgModule](https://angular.io/guide/ngmodules).

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

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

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

content\_copyimport { HeroesComponent } from './heroes/heroes.component';

The HeroesComponent is declared in the @[NgModule.declarations](https://angular.io/api/core/NgModule" \l "declarations) array.

content\_copydeclarations: [

AppComponent,

HeroesComponent

],

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

## Final code review

Your app should look like this [live example](https://angular.io/generated/live-examples/toh-pt1/stackblitz.html) / [download example](https://angular.io/generated/zips/toh-pt1/toh-pt1.zip). 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.ts

src/app/app.component.ts

src/app/app.component.html

src/app/hero.ts

content\_copy

1. import { [Component](https://angular.io/api/core/Component), [OnInit](https://angular.io/api/core/OnInit) } from '@angular/core';
2. import { Hero } from '../hero';
4. @[Component](https://angular.io/api/core/Component)({
5. selector: 'app-heroes',
6. templateUrl: './heroes.component.html',
7. [styleUrls](https://angular.io/api/core/Component#styleUrls): ['./heroes.component.css']
8. })
9. export class HeroesComponent implements [OnInit](https://angular.io/api/core/OnInit) {
10. hero: Hero = {
11. id: 1,
12. name: 'Windstorm'
13. };
15. constructor() { }
17. ngOnInit() {
18. }
20. }

## Summary

* You used the CLI 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](https://angular.io/api/forms/NgModel) directive.
* You learned about the AppModule.
* You imported the [FormsModule](https://angular.io/api/forms/FormsModule) in the AppModule so that Angular would recognize and apply the [ngModel](https://angular.io/api/forms/NgModel)directive.
* You learned the importance of declaring components in the AppModule and appreciated that the CLI declared it for you.

# Display a Heroes List

In this page, you'll expand the Tour of Heroes app to display a list of heroes, and allow users to select a hero and display the hero's details.

## Create mock heroes

You'll need some heroes to display.

Eventually you'll get them from a remote data server. For now, you'll create some mock heroes and pretend they came from the server.

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

src/app/mock-heroes.ts

content\_copyimport { Hero } from './hero';

export const HEROES: Hero[] = [

{ id: 11, name: 'Dr Nice' },

{ id: 12, name: 'Narco' },

{ id: 13, name: 'Bombasto' },

{ id: 14, name: 'Celeritas' },

{ id: 15, name: 'Magneta' },

{ id: 16, name: 'RubberMan' },

{ id: 17, name: 'Dynama' },

{ id: 18, name: 'Dr IQ' },

{ id: 19, name: 'Magma' },

{ id: 20, name: 'Tornado' }

];

## Displaying heroes

You're about to display the list of heroes at the top of the HeroesComponent.

Open the HeroesComponent class file and import the mock HEROES.

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

content\_copyimport { HEROES } from '../mock-heroes';

In the same file (HeroesComponent class), define a component property called heroes to expose HEROES array for binding.

content\_copyexport class HeroesComponent implements [OnInit](https://angular.io/api/core/OnInit) {

heroes = HEROES;

### List heroes with \*ngFor

Open the HeroesComponent template file and make the following changes:

* Add an <h2> at the top,
* Below it add an HTML unordered list (<ul>)
* Insert an <li> within the <ul> that displays properties of a hero.
* Sprinkle some CSS classes for styling (you'll add the CSS styles shortly).

Make it look like this:

heroes.component.html (heroes template)

content\_copy<h2>My Heroes</h2>

<ul class="heroes">

<li>

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

</li>

</ul>

Now change the <li> to this:

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

The [\*ngFor](https://angular.io/guide/template-syntax#ngFor) is Angular's repeater directive. It repeats the host element for each element in a list.

In this example

* <li> is the host element
* heroes is the list from the HeroesComponent class.
* hero holds the current hero object for each iteration through the list.

Don't forget the asterisk (\*) in front of [ngFor](https://angular.io/api/common/NgForOf). It's a critical part of the syntax.

After the browser refreshes, the list of heroes appears.

### 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](https://angular.io/tutorial/toh-pt0#app-wide-styles), you set the basic styles for the entire application in styles.css. That stylesheet didn't include styles for this list of heroes.

You could add more styles to styles.css and keep growing that stylesheet as you add components.

You may prefer instead to define private styles for a specific component and keep everything a component needs— 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](https://angular.io/api/core/Component" \l "styles) array or as stylesheet file(s) identified in the @[Component.styleUrls](https://angular.io/api/core/Component" \l "styleUrls) array.

When the CLI generated the HeroesComponent, it created an empty heroes.component.css stylesheet for the HeroesComponent and pointed to it in @[Component.styleUrls](https://angular.io/api/core/Component" \l "styleUrls) like this.

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

content\_copy@[Component](https://angular.io/api/core/Component)({

selector: 'app-heroes',

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

[styleUrls](https://angular.io/api/core/Component#styleUrls): ['./heroes.component.css']

})

Open the heroes.component.css file and paste in the private CSS styles for the HeroesComponent. You'll find them in the [final code review](https://angular.io/tutorial/toh-pt2#final-code-review) at the bottom of this guide.

Styles and stylesheets identified in @[Component](https://angular.io/api/core/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.

## Master/Detail

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

In this section, you'll listen for the hero item click event and update the hero detail.

### Add a click event binding

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

heroes.component.html (template excerpt)

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

This is an example of Angular's [event binding](https://angular.io/guide/template-syntax#event-binding) syntax.

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

onSelect() is a HeroesComponent method that you're about to write. Angular calls it with the hero object displayed in the clicked <li>, the same hero defined previously in the \*[ngFor](https://angular.io/api/common/NgForOf) expression.

### Add the click event handler

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

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

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

content\_copyselectedHero: Hero;

onSelect(hero: Hero): void {

this.selectedHero = hero;

}

### Update the details template

The template still refers to the component's old hero property which no longer exists. Rename hero to selectedHero.

heroes.component.html (selected hero details)

content\_copy<h2>{{selectedHero.name | [uppercase](https://angular.io/api/common/UpperCasePipe)}} Details</h2>

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

<div>

<label>name:

<input [([ngModel](https://angular.io/api/forms/NgModel))]="selectedHero.name" placeholder="name"/>

</label>

</div>

After the browser refreshes, the application is broken.

Open the browser developer tools and look in the console for an error message like this:

content\_copyHeroesComponent.html:3 ERROR TypeError: Cannot [read](https://angular.io/api/core/Query#read) property 'name' of undefined

#### **What happened?**

When the app starts, the selectedHero is undefined by design.

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

#### **The fix - hide empty details with \*ngIf**

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

Wrap the hero detail HTML in a <div>. Add Angular's \*[ngIf](https://angular.io/api/common/NgIf) directive to the <div> and set it to selectedHero.

Don't forget the asterisk (\*) in front of [ngIf](https://angular.io/api/common/NgIf). It's a critical part of the syntax.

src/app/heroes/heroes.component.html (\*ngIf)

content\_copy<div \*[ngIf](https://angular.io/api/common/NgIf)="selectedHero">

<h2>{{selectedHero.name | [uppercase](https://angular.io/api/common/UpperCasePipe)}} Details</h2>

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

<div>

<label>name:

<input [([ngModel](https://angular.io/api/forms/NgModel))]="selectedHero.name" placeholder="name"/>

</label>

</div>

</div>

After the browser refreshes, the list of names reappears. The details area is blank. Click a hero in the list of heroes and its details appear. The app seems to be working again. The heroes appear in a list and details about the clicked hero appear at the bottom of the page.

#### **Why it works**

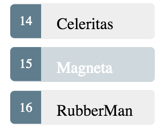
When selectedHero is undefined, the [ngIf](https://angular.io/api/common/NgIf) removes the hero detail from the DOM. There are no selectedHerobindings to worry about.

When the user picks a hero, selectedHero has a value and [ngIf](https://angular.io/api/common/NgIf) puts the hero detail into the DOM.

### Style the selected hero

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

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



That selected hero coloring is the work of the .selected CSS class in the [styles you added earlier](https://angular.io/tutorial/toh-pt2#styles). You just have to apply the .selected class to the <li> when the user clicks it.

The Angular [class binding](https://angular.io/guide/template-syntax#class-binding) makes it easy to add and remove a CSS class conditionally. Just add [class.some-css-class]="some-condition" to the element you want to style.

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

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

content\_copy[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)

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

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

(click)="onSelect(hero)">

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

</li>

## Final code review

Your app should look like this [live example](https://angular.io/generated/live-examples/toh-pt2/stackblitz.html) / [download example](https://angular.io/generated/zips/toh-pt2/toh-pt2.zip).

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.html

src/app/heroes/heroes.component.css

content\_copy

1. import { Hero } from './hero';
3. export const HEROES: Hero[] = [
4. { id: 11, name: 'Dr Nice' },
5. { id: 12, name: 'Narco' },
6. { id: 13, name: 'Bombasto' },
7. { id: 14, name: 'Celeritas' },
8. { id: 15, name: 'Magneta' },
9. { id: 16, name: 'RubberMan' },
10. { id: 17, name: 'Dynama' },
11. { id: 18, name: 'Dr IQ' },
12. { id: 19, name: 'Magma' },
13. { id: 20, name: 'Tornado' }
14. ];

## Summary

* The Tour of Heroes app displays a list of heroes in a Master/Detail view.
* The user can select a hero and see that hero's details.
* You used \*[ngFor](https://angular.io/api/common/NgForOf) to display a list.
* You used \*[ngIf](https://angular.io/api/common/NgIf) to conditionally include or exclude a block of HTML.
* You can toggle a CSS style class with a class binding.

# Master/Detail Components

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

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

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

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

## Make the HeroDetailComponent

Use the Angular CLI to generate a new component named hero-detail.

content\_copyng generate component hero-detail

The command scaffolds the following:

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

Inside that directory four files are generated:

* 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](https://angular.io/api/core/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 generated boilerplate in the HeroDetailComponent template.

The pasted HTML refers to a selectedHero. The new HeroDetailComponent can present any hero, not just a selected hero. So 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

content\_copy<div \*[ngIf](https://angular.io/api/common/NgIf)="hero">

<h2>{{hero.name | [uppercase](https://angular.io/api/common/UpperCasePipe)}} Details</h2>

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

<div>

<label>name:

<input [([ngModel](https://angular.io/api/forms/NgModel))]="hero.name" placeholder="name"/>

</label>

</div>

</div>

### Add the @[Input](https://angular.io/api/core/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)

content\_copyimport { Hero } from '../hero';

The hero property [must be an Input property](https://angular.io/guide/template-syntax#inputs-outputs), annotated with the @[Input](https://angular.io/api/core/Input)() decorator, because the externalHeroesComponent [will bind to it](https://angular.io/tutorial/toh-pt3#heroes-component-template) like this.

content\_copy<app-hero-detail [hero]="selectedHero"></app-hero-detail>

Amend the @angular/core import statement to include the [Input](https://angular.io/api/core/Input) symbol.

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

content\_copyimport { [Component](https://angular.io/api/core/Component), [OnInit](https://angular.io/api/core/OnInit), [Input](https://angular.io/api/core/Input) } from '@angular/core';

Add a hero property, preceded by the @[Input](https://angular.io/api/core/Input)() decorator.

content\_copy@[Input](https://angular.io/api/core/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 simply receives a hero object through its hero property and displays it.

## Show the HeroDetailComponent

The HeroesComponent is still a master/detail view.

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

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

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

### 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)

content\_copy<app-hero-detail [hero]="selectedHero"></app-hero-detail>

[hero]="selectedHero" is an Angular [property binding](https://angular.io/guide/template-syntax#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

content\_copy<h2>My Heroes</h2>

<ul class="heroes">

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

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

(click)="onSelect(hero)">

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

</li>

</ul>

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

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

## What changed?

As [before](https://angular.io/tutorial/toh-pt2), 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 simplified the HeroesComponent by reducing its 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 and your app should look like this [live example](https://angular.io/generated/live-examples/toh-pt3/stackblitz.html) / [download example](https://angular.io/generated/zips/toh-pt3/toh-pt3.zip).

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

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

src/app/heroes/heroes.component.html

src/app/app.module.ts

content\_copy

1. import { [Component](https://angular.io/api/core/Component), [OnInit](https://angular.io/api/core/OnInit), [Input](https://angular.io/api/core/Input) } from '@angular/core';
2. import { Hero } from '../hero';
4. @[Component](https://angular.io/api/core/Component)({
5. selector: 'app-hero-detail',
6. templateUrl: './hero-detail.component.html',
7. [styleUrls](https://angular.io/api/core/Component#styleUrls): ['./hero-detail.component.css']
8. })
9. export class HeroDetailComponent implements [OnInit](https://angular.io/api/core/OnInit) {
10. @[Input](https://angular.io/api/core/Input)() hero: Hero;
12. constructor() { }
14. ngOnInit() {
15. }
17. }

## Summary

* You created a separate, reusable HeroDetailComponent.
* You used a [property binding](https://angular.io/guide/template-syntax#property-binding) to give the parent HeroesComponent control over the child HeroDetailComponent.
* You used the [@Input decorator](https://angular.io/guide/template-syntax#inputs-outputs) to make the hero property available for binding by the external HeroesComponent.

# Services

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

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

## Why services

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

In this tutorial, you'll create a HeroService that all application classes can use to get heroes. Instead of creating that service with new, you'll rely on Angular [dependency injection](https://angular.io/guide/dependency-injection) to inject it into the HeroesComponent constructor.

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

1. in HeroService which uses the service to send a message.
2. in MessagesComponent which displays that message.

## Create the HeroService

Using the Angular CLI, create a service called hero.

content\_copyng generate service hero

The command generates skeleton HeroService class in src/app/hero.service.ts The HeroService class should look like the following example.

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

content\_copyimport { [Injectable](https://angular.io/api/core/Injectable) } from '@angular/core';

@[Injectable](https://angular.io/api/core/Injectable)({

[providedIn](https://angular.io/api/core/Injectable#providedIn): 'root',

})

export class HeroService {

constructor() { }

}

### @Injectable() services

Notice that the new service imports the Angular [Injectable](https://angular.io/api/core/Injectable) symbol and annotates the class with the @[Injectable](https://angular.io/api/core/Injectable)()decorator. This marks the class as one that participates in the dependency injection system. The HeroService class is going to provide an injectable service, and it can also have its own injected dependencies. It doesn't have any dependencies yet, but [it will soon](https://angular.io/tutorial/toh-pt4#inject-message-service).

The @[Injectable](https://angular.io/api/core/Injectable)() decorator accepts a metadata object for the service, the same way the @[Component](https://angular.io/api/core/Component)() decorator did for your component classes.

### Get hero data

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

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

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

Import the Hero and HEROES.

content\_copyimport { Hero } from './hero';

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

Add a getHeroes method to return the mock heroes.

content\_copygetHeroes(): 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, as you will do [below](https://angular.io/tutorial/toh-pt4#inject). You do this by registering a provider. A provider is something that can create or deliver a service; in this case, it instantiates the HeroService class to provide the service.

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

By default, the Angular CLI command ng generate service registers a provider with the root injector for your service by including provider metadata in the @[Injectable](https://angular.io/api/core/Injectable) decorator.

If you look at the @[Injectable](https://angular.io/api/core/Injectable)() statement right before the HeroService class definition, you can see that the [providedIn](https://angular.io/api/core/Injectable" \l "providedIn) metadata value is 'root':

content\_copy@[Injectable](https://angular.io/api/core/Injectable)({

[providedIn](https://angular.io/api/core/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](https://angular.io/api/core/Injectable) metadata also allows Angular to optimize an app by removing the service if it turns out not to be used after all.

To learn more about providers, see the [Providers section](https://angular.io/guide/providers). To learn more about injectors, see the [Dependency Injection guide](https://angular.io/guide/dependency-injection).

The HeroService is now ready to plug into the HeroesComponent.

This is an interim code sample that will allow you to provide and use the HeroService. At this point, the code will differ from the HeroService in the ["final code review"](https://angular.io/tutorial/toh-pt4#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)

content\_copyimport { HeroService } from '../hero.service';

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

content\_copyheroes: Hero[];

### Inject the HeroService

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

content\_copyconstructor(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](https://angular.io/guide/dependency-injection) system sets the heroService parameter to the singleton instance of HeroService.

### Add getHeroes()

Create a function to retrieve the heroes from the service.

content\_copygetHeroes(): 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 simple initialization such as wiring constructor parameters to properties. The constructor shouldn't do anything. It certainly shouldn't call a function that makes HTTP requests to a remote server as a real data service would.

Instead, call getHeroes() inside the [ngOnInit lifecycle hook](https://angular.io/guide/lifecycle-hooks) and let Angular call ngOnInit at an appropriate time afterconstructing a HeroesComponent instance.

content\_copyngOnInit() {

this.getHeroes();

}

### See it run

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

## Observable data

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

content\_copythis.heroes = this.heroService.getHeroes();

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

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

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

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

In this tutorial, HeroService.getHeroes() will return an Observable in part because it will eventually use the Angular HttpClient.get method to fetch the heroes and [HttpClient.get() returns an Observable](https://angular.io/guide/http).

### Observable HeroService

Observable is one of the key classes in the [RxJS library](http://reactivex.io/rxjs/).

In a [later tutorial on HTTP](https://angular.io/tutorial/toh-pt6), you'll learn that Angular's [HttpClient](https://angular.io/api/common/http/HttpClient) methods return RxJS Observables. In this tutorial, you'll simulate getting data from the server with the RxJS of() function.

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

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

content\_copyimport { Observable, of } from 'rxjs';

Replace the getHeroes method with this one.

content\_copygetHeroes(): Observable<Hero[]> {

return of(HEROES);

}

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

In the [HTTP tutorial](https://angular.io/tutorial/toh-pt6), you'll call HttpClient.get<Hero[]>() which also returns an Observable<Hero[]>that emits a single value, an array of heroes from the body of the HTTP response.

### Subscribe in HeroesComponent

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

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

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

heroes.component.ts (Observable)

heroes.component.ts (Original)

content\_copygetHeroes(): 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. Then subscribe passes the emitted array to the callback, which sets the component's heroes property.

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

## Show messages

In this section you will

* add a MessagesComponent that displays app messages at the bottom of the screen.
* create an injectable, app-wide MessageService for sending messages to be displayed
* inject MessageService into the HeroService
* display a message when HeroService fetches heroes successfully.

### Create MessagesComponent

Use the CLI to create the MessagesComponent.

content\_copyng generate component [messages](https://angular.io/api/service-worker/SwPush#messages)

The CLI creates the component files in the src/app/[messages](https://angular.io/api/service-worker/SwPush#messages) folder and declares the MessagesComponent in AppModule.

Modify the AppComponent template to display the generated MessagesComponent

/src/app/app.component.html

content\_copy<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 the CLI to create the MessageService in src/app.

content\_copyng generate service [message](https://angular.io/api/common/http/HttpErrorResponse#message)

Open MessageService and replace its contents with the following.

/src/app/message.service.ts

content\_copy

1. import { [Injectable](https://angular.io/api/core/Injectable) } from '@angular/core';
3. @[Injectable](https://angular.io/api/core/Injectable)({
4. [providedIn](https://angular.io/api/core/Injectable#providedIn): 'root',
5. })
6. export class MessageService {
7. [messages](https://angular.io/api/service-worker/SwPush#messages): string[] = [];
9. add([message](https://angular.io/api/common/http/HttpErrorResponse#message): string) {
10. this.messages.push([message](https://angular.io/api/common/http/HttpErrorResponse#message));
11. }
13. clear() {
14. this.messages = [];
15. }
16. }

The service exposes its cache of [messages](https://angular.io/api/service-worker/SwPush#messages) and two methods: one to add() a message to the cache and another to [clear()](https://angular.io/api/forms/FormArray#clear) the cache.

### Inject it into the HeroService

Re-open the HeroService and import the MessageService.

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

content\_copyimport { MessageService } from './message.service';

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

content\_copyconstructor(private messageService: MessageService) { }

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

### Send a message from HeroService

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

content\_copygetHeroes(): Observable<Hero[]> {

// TODO: send the [message](https://angular.io/api/common/http/HttpErrorResponse#message) \_after\_ fetching the heroes

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

return of(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)

content\_copyimport { MessageService } from '../message.service';

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

content\_copyconstructor(public messageService: MessageService) {}

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

Angular only binds to public component properties.

### Bind to the MessageService

Replace the CLI-generated MessagesComponent template with the following.

src/app/messages/messages.component.html

content\_copy<div \*[ngIf](https://angular.io/api/common/NgIf)="messageService.messages.length">

<h2>Messages</h2>

<button class="clear"

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

<div \*[ngFor](https://angular.io/api/common/NgForOf)='let [message](https://angular.io/api/common/http/HttpErrorResponse#message) of messageService.messages'> {{[message](https://angular.io/api/common/http/HttpErrorResponse#message)}} </div>

</div>

This template binds directly to the component's messageService.

* The \*[ngIf](https://angular.io/api/common/NgIf) only displays the messages area if there are messages to show.
* An \*[ngFor](https://angular.io/api/common/NgForOf) presents the list of messages in repeated <div> elements.
* An Angular [event binding](https://angular.io/guide/template-syntax#event-binding) binds the button's click event to MessageService.clear().

The messages will look better when you add the private CSS styles to messages.component.css as listed in one of the ["final code review"](https://angular.io/tutorial/toh-pt4#final-code-review) tabs below.

The browser refreshes and the page displays the list of heroes. Scroll to the bottom to see the message from the HeroService in the message area. Click the "clear" button and the message area disappears.

## Final code review

Here are the code files discussed on this page and your app should look like this [live example](https://angular.io/generated/live-examples/toh-pt4/stackblitz.html) / [download example](https://angular.io/generated/zips/toh-pt4/toh-pt4.zip).

src/app/hero.service.ts

src/app/message.service.ts

src/app/heroes/heroes.component.ts

src/app/messages/messages.component.ts

src/app/messages/messages.component.html

src/app/messages/messages.component.css

src/app/app.module.ts

src/app/app.component.html

content\_copy

1. import { [Injectable](https://angular.io/api/core/Injectable) } from '@angular/core';
3. import { Observable, of } from 'rxjs';
5. import { Hero } from './hero';
6. import { HEROES } from './mock-heroes';
7. import { MessageService } from './message.service';
9. @[Injectable](https://angular.io/api/core/Injectable)({
10. [providedIn](https://angular.io/api/core/Injectable#providedIn): 'root',
11. })
12. export class HeroService {
14. constructor(private messageService: MessageService) { }
16. getHeroes(): Observable<Hero[]> {
17. // TODO: send the [message](https://angular.io/api/common/http/HttpErrorResponse#message) \_after\_ fetching the heroes
18. this.messageService.add('HeroService: fetched heroes');
19. return of(HEROES);
20. }
21. }

## 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 app.
* You used [Angular Dependency Injection](https://angular.io/guide/dependency-injection) to inject it into a component.
* You gave the HeroService get data method an asynchronous signature.
* You discovered Observable and the RxJS Observable library.
* You used RxJS of() to return an observable of mock heroes (Observable<Hero[]>).
* The component's ngOnInit lifecycle hook calls the HeroService method, not the constructor.
* You created a MessageService for loosely-coupled communication between classes.
* The HeroService injected into a component is created with another injected service, MessageService.

# Routing

There are new requirements for the Tour of Heroes app:

* Add a Dashboard view.
* Add the ability to navigate between the Heroes and Dashboard views.
* When users click a hero name in either view, navigate to a detail view of the selected hero.
* When users click a deep link in an email, open the detail view for a particular hero.

When you’re done, users will be able to navigate the app like this:



## Add the AppRoutingModule

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

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

Use the CLI to generate it.

content\_copyng generate module app-routing --flat --module=app

--flat puts the file in src/app instead of its own folder.  
--module=app tells the CLI to register it in the imports array of the AppModule.

The generated file looks like this:

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

content\_copyimport { [NgModule](https://angular.io/api/core/NgModule) } from '@angular/core';

import { [CommonModule](https://angular.io/api/common/CommonModule) } from '@angular/common';

@[NgModule](https://angular.io/api/core/NgModule)({

imports: [

[CommonModule](https://angular.io/api/common/CommonModule)

],

declarations: []

})

export class AppRoutingModule { }

You generally don't declare components in a routing module so you can delete the @[NgModule.declarations](https://angular.io/api/core/NgModule" \l "declarations) array and delete [CommonModule](https://angular.io/api/common/CommonModule) references too.

You'll configure the router with [Routes](https://angular.io/api/router/Routes) in the [RouterModule](https://angular.io/api/router/RouterModule) so import those two symbols from the @angular/routerlibrary.

Add an @[NgModule.exports](https://angular.io/api/core/NgModule" \l "exports) array with [RouterModule](https://angular.io/api/router/RouterModule) in it. Exporting [RouterModule](https://angular.io/api/router/RouterModule) makes router directives available for use in the AppModule components that will need them.

AppRoutingModule looks like this now:

src/app/app-routing.module.ts (v1)

content\_copyimport { [NgModule](https://angular.io/api/core/NgModule) } from '@angular/core';

import { [RouterModule](https://angular.io/api/router/RouterModule), [Routes](https://angular.io/api/router/Routes) } from '@angular/router';

@[NgModule](https://angular.io/api/core/NgModule)({

exports: [ [RouterModule](https://angular.io/api/router/RouterModule) ]

})

export class AppRoutingModule {}

### Add routes

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

A typical Angular [Route](https://angular.io/api/router/Route) has two properties:

1. path: a string that matches the URL in the browser address bar.
2. component: the component that the router should create when navigating to this route.

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

Import the HeroesComponent so you can reference it in a [Route](https://angular.io/api/router/Route). Then define an array of routes with a single route to that component.

content\_copyimport { HeroesComponent } from './heroes/heroes.component';

const routes: [Routes](https://angular.io/api/router/Routes) = [

{ path: 'heroes', component: HeroesComponent }

];

Once you've finished setting up, the router will match that URL to path: 'heroes' and display the HeroesComponent.

### RouterModule.forRoot()

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

Add [RouterModule](https://angular.io/api/router/RouterModule) to the @[NgModule.imports](https://angular.io/api/core/NgModule" \l "imports) array and configure it with the routes in one step by calling[RouterModule.forRoot()](https://angular.io/api/router/RouterModule#forRoot) within the imports array, like this:

content\_copyimports: [ RouterModule.forRoot(routes) ],

The method is called [forRoot()](https://angular.io/api/router/RouterModule" \l "forRoot) because you configure the router at the application's root level. The [forRoot()](https://angular.io/api/router/RouterModule" \l "forRoot) method supplies the service providers and directives needed for routing, and performs the initial navigation based on the current browser URL.

## Add RouterOutlet

Open the AppComponent template and replace the <app-heroes> element with a <[router-outlet](https://angular.io/api/router/RouterOutlet)> element.

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

content\_copy<h1>{{title}}</h1>

<[router-outlet](https://angular.io/api/router/RouterOutlet)></[router-outlet](https://angular.io/api/router/RouterOutlet)>

<app-messages></app-messages>

You removed <app-heroes> because you will only display the HeroesComponent when the user navigates to it.

The <[router-outlet](https://angular.io/api/router/RouterOutlet)> tells the router where to display routed views.

The [RouterOutlet](https://angular.io/api/router/RouterOutlet) is one of the router directives that became available to the AppComponent because AppModule imports AppRoutingModule which exported [RouterModule](https://angular.io/api/router/RouterModule).

#### **Try it**

You should still be running with this CLI command.

content\_copyng serve

The browser should refresh and display the app 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 master/detail view.

## Add a navigation link ([routerLink](https://angular.io/api/router/RouterLink))

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

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

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

content\_copy<h1>{{title}}</h1>

<nav>

<[a](https://angular.io/api/router/RouterLinkWithHref) [routerLink](https://angular.io/api/router/RouterLink)="/heroes">Heroes</[a](https://angular.io/api/router/RouterLinkWithHref)>

</nav>

<[router-outlet](https://angular.io/api/router/RouterOutlet)></[router-outlet](https://angular.io/api/router/RouterOutlet)>

<app-messages></app-messages>

A [routerLink attribute](https://angular.io/tutorial/toh-pt5" \l "routerlink) is set to "/heroes", the string that the router matches to the route to HeroesComponent. The [routerLink](https://angular.io/api/router/RouterLink) is the selector for the [RouterLink directive](https://angular.io/tutorial/toh-pt5" \l "routerlink) that turns user clicks into router navigations. It's another of the public directives in the [RouterModule](https://angular.io/api/router/RouterModule).

The browser refreshes and displays the app 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](https://angular.io/tutorial/toh-pt5#appcomponent) below.

## Add a dashboard view

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

Add a DashboardComponent using the CLI:

content\_copyng generate component dashboard

The CLI generates the files for the DashboardComponent and declares it in AppModule.

Replace the default file content in these three files as follows and then return for a little discussion:

src/app/dashboard/dashboard.component.html

src/app/dashboard/dashboard.component.ts

src/app/dashboard/dashboard.component.css

content\_copy<h3>Top Heroes</h3>

<div class="grid grid-pad">

<[a](https://angular.io/api/router/RouterLinkWithHref) \*[ngFor](https://angular.io/api/common/NgForOf)="let hero of heroes" class="col-1-4">

<div class="module hero">

<h4>{{hero.name}}</h4>

</div>

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

</div>

The template presents a grid of hero name links.

* The \*[ngFor](https://angular.io/api/common/NgForOf) 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 but [they will shortly](https://angular.io/tutorial/toh-pt5#hero-details).

The class is similar to the HeroesComponent class.

* It defines a heroes array property.
* The constructor expects Angular to inject the HeroService into a private heroService property.
* The ngOnInit() lifecycle hook calls getHeroes.

This getHeroes returns the sliced list of heroes at positions 1 and 5, returning only four of the Top Heroes (2nd, 3rd, 4th, and 5th).

content\_copygetHeroes(): 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 AppRoutingModule.

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

content\_copyimport { DashboardComponent } from './dashboard/dashboard.component';

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

content\_copy{ path: 'dashboard', component: DashboardComponent },

### Add a default route

When the app starts, the browsers address bar points to the web site's root. That doesn't match any existing route so the router doesn't navigate anywhere. The space below the <[router-outlet](https://angular.io/api/router/RouterOutlet)> is blank.

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

content\_copy{ path: '', [redirectTo](https://angular.io/api/router/Route#redirectTo): '/dashboard', [pathMatch](https://angular.io/api/router/Route#pathMatch): '[full](https://angular.io/api/core/Version#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 back and forth between the DashboardComponent and the HeroesComponent by clicking links in the navigation area near the top of the page.

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

src/app/app.component.html

content\_copy<h1>{{title}}</h1>

<nav>

<[a](https://angular.io/api/router/RouterLinkWithHref) [routerLink](https://angular.io/api/router/RouterLink)="/dashboard">Dashboard</[a](https://angular.io/api/router/RouterLinkWithHref)>

<[a](https://angular.io/api/router/RouterLinkWithHref) [routerLink](https://angular.io/api/router/RouterLink)="/heroes">Heroes</[a](https://angular.io/api/router/RouterLinkWithHref)>

</nav>

<[router-outlet](https://angular.io/api/router/RouterOutlet)></[router-outlet](https://angular.io/api/router/RouterOutlet)>

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

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

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.

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

### Delete hero details from HeroesComponent

When the user clicks a hero item in the HeroesComponent, the app 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 HeroesComponent template (heroes/heroes.component.html) and delete the <app-hero-detail>element from the bottom.

Clicking a hero item now does nothing. You'll [fix that shortly](https://angular.io/tutorial/toh-pt5#heroes-component-links) 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 AppRoutingModule and import HeroDetailComponent.

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

content\_copyimport { HeroDetailComponent } from './hero-detail/hero-detail.component';

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

content\_copy{ path: 'detail/:id', component: HeroDetailComponent },

The colon (:) 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)

content\_copyconst routes: [Routes](https://angular.io/api/router/Routes) = [

{ path: '', [redirectTo](https://angular.io/api/router/Route#redirectTo): '/dashboard', [pathMatch](https://angular.io/api/router/Route#pathMatch): '[full](https://angular.io/api/core/Version#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 via the parameterized dashboard route.

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

content\_copy<[a](https://angular.io/api/router/RouterLinkWithHref) \*[ngFor](https://angular.io/api/common/NgForOf)="let hero of heroes" class="col-1-4"

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

<div class="module hero">

<h4>{{hero.name}}</h4>

</div>

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

You're using Angular [interpolation binding](https://angular.io/guide/template-syntax#interpolation) within the \*[ngFor](https://angular.io/api/common/NgForOf) repeater to insert the current iteration's hero.id into each[routerLink](https://angular.io/tutorial/toh-pt5#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)

content\_copy<ul class="heroes">

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

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

(click)="onSelect(hero)">

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

</li>

</ul>

Strip the <li> back to just its \*[ngFor](https://angular.io/api/common/NgForOf), wrap the badge and name in an anchor element (<[a](https://angular.io/api/router/RouterLinkWithHref)>), and add a [routerLink](https://angular.io/api/router/RouterLink)attribute to the anchor that is the same as in the dashboard template

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

content\_copy<ul class="heroes">

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

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

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

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

</li>

</ul>

You'll have to fix the private stylesheet (heroes.component.css) to make the list look as it did before. Revised styles are in the [final code review](https://angular.io/tutorial/toh-pt5#heroescomponent) 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 up and you'll be grateful to yourself later. Here's the class after pruning away the dead code.

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

content\_copyexport class HeroesComponent implements [OnInit](https://angular.io/api/core/OnInit) {

heroes: Hero[];

constructor(private heroService: HeroService) { }

ngOnInit() {

this.getHeroes();

}

getHeroes(): void {

this.heroService.getHeroes()

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

}

}

## Routable HeroDetailComponent

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

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

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

* Get the route that created it,
* Extract the id from the route
* Acquire the hero with that id from the server via the HeroService

Add the following imports:

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

content\_copyimport { [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) } from '@angular/router';

import { [Location](https://angular.io/api/common/Location) } from '@angular/common';

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

Inject the [ActivatedRoute](https://angular.io/api/router/ActivatedRoute), HeroService, and [Location](https://angular.io/api/common/Location) services into the constructor, saving their values in private fields:

content\_copyconstructor(

private route: [ActivatedRoute](https://angular.io/api/router/ActivatedRoute),

private heroService: HeroService,

private location: [Location](https://angular.io/api/common/Location)

) {}

The [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) holds information about the route to this instance of the HeroDetailComponent. This component is interested in the route's bag of parameters extracted from the URL. The "id" parameter is the id of the hero to display.

The [HeroService](https://angular.io/tutorial/toh-pt4) gets hero data from the remote server and this component will use it to get the hero-to-display.

The [location](https://angular.io/api/common/Location) is an Angular service for interacting with the browser. You'll use it [later](https://angular.io/tutorial/toh-pt5#goback) to navigate back to the view that navigated here.

### Extract the id route parameter

In the ngOnInit() [lifecycle hook](https://angular.io/guide/lifecycle-hooks#oninit) call getHero() and define it as follows.

content\_copyngOnInit(): void {

this.getHero();

}

getHero(): void {

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

this.heroService.getHero(id)

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

}

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

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

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

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

### Add HeroService.getHero()

Open HeroService and add this getHero() method

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

content\_copygetHero(id: number): Observable<Hero> {

// TODO: send the [message](https://angular.io/api/common/http/HttpErrorResponse#message) \_after\_ fetching the hero

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

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

}

Note the backticks ( ` ) that define a JavaScript [template literal](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals) for embedding the id.

Like [getHeroes()](https://angular.io/tutorial/toh-pt4" \l "observable-heroservice), getHero() has an asynchronous signature. It returns a mock hero as an Observable, using the RxJS of() function.

You'll be able to re-implement getHero() as a real Http request without having to change the HeroDetailComponentthat calls it.

#### **Try it**

The browser refreshes and the app 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/11 in the browser address bar, the router navigates to the detail view for the hero with id: 11, "Dr Nice".

### Find the way back

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

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

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

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

content\_copy<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](https://angular.io/api/common/Location) service that you [injected previously](https://angular.io/tutorial/toh-pt5#hero-detail-ctor).

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

content\_copygoBack(): void {

this.location.back();

}

Refresh the browser and start clicking. Users can navigate around the app, from the dashboard to hero details and back, from heroes list to the mini detail to the hero details and back to the heroes again.

You've met all of the navigational requirements that propelled this page.

## Final code review

Here are the code files discussed on this page and your app should look like this [live example](https://angular.io/generated/live-examples/toh-pt5/stackblitz.html) / [download example](https://angular.io/generated/zips/toh-pt5/toh-pt5.zip).

#### **AppRoutingModule, AppModule, and HeroService**

src/app/app-routing.module.ts

src/app/app.module.ts

src/app/hero.service.ts

content\_copy

1. import { [NgModule](https://angular.io/api/core/NgModule) } from '@angular/core';
2. import { [RouterModule](https://angular.io/api/router/RouterModule), [Routes](https://angular.io/api/router/Routes) } from '@angular/router';
4. import { DashboardComponent } from './dashboard/dashboard.component';
5. import { HeroesComponent } from './heroes/heroes.component';
6. import { HeroDetailComponent } from './hero-detail/hero-detail.component';
8. const routes: [Routes](https://angular.io/api/router/Routes) = [
9. { path: '', [redirectTo](https://angular.io/api/router/Route#redirectTo): '/dashboard', [pathMatch](https://angular.io/api/router/Route#pathMatch): '[full](https://angular.io/api/core/Version#full)' },
10. { path: 'dashboard', component: DashboardComponent },
11. { path: 'detail/:id', component: HeroDetailComponent },
12. { path: 'heroes', component: HeroesComponent }
13. ];
15. @[NgModule](https://angular.io/api/core/NgModule)({
16. imports: [ RouterModule.forRoot(routes) ],
17. exports: [ [RouterModule](https://angular.io/api/router/RouterModule) ]
18. })
19. export class AppRoutingModule {}

#### **AppComponent**

src/app/app.component.html

src/app/app.component.css

content\_copy<h1>{{title}}</h1>

<nav>

<[a](https://angular.io/api/router/RouterLinkWithHref) [routerLink](https://angular.io/api/router/RouterLink)="/dashboard">Dashboard</[a](https://angular.io/api/router/RouterLinkWithHref)>

<[a](https://angular.io/api/router/RouterLinkWithHref) [routerLink](https://angular.io/api/router/RouterLink)="/heroes">Heroes</[a](https://angular.io/api/router/RouterLinkWithHref)>

</nav>

<[router-outlet](https://angular.io/api/router/RouterOutlet)></[router-outlet](https://angular.io/api/router/RouterOutlet)>

<app-messages></app-messages>

#### **DashboardComponent**

src/app/dashboard/dashboard.component.html

src/app/dashboard/dashboard.component.ts

src/app/dashboard/dashboard.component.css

content\_copy<h3>Top Heroes</h3>

<div class="grid grid-pad">

<[a](https://angular.io/api/router/RouterLinkWithHref) \*[ngFor](https://angular.io/api/common/NgForOf)="let hero of heroes" class="col-1-4"

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

<div class="module hero">

<h4>{{hero.name}}</h4>

</div>

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

</div>

#### **HeroesComponent**

src/app/heroes/heroes.component.html

src/app/heroes/heroes.component.ts

src/app/heroes/heroes.component.css

content\_copy<h2>My Heroes</h2>

<ul class="heroes">

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

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

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

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

</li>

</ul>

#### **HeroDetailComponent**

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

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

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

content\_copy<div \*[ngIf](https://angular.io/api/common/NgIf)="hero">

<h2>{{hero.name | [uppercase](https://angular.io/api/common/UpperCasePipe)}} Details</h2>

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

<div>

<label>name:

<input [([ngModel](https://angular.io/api/forms/NgModel))]="hero.name" placeholder="name"/>

</label>

</div>

<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](https://angular.io/api/router/RouterLinkWithHref)> links and a <[router-outlet](https://angular.io/api/router/RouterOutlet)>.
* You configured the router in an AppRoutingModule
* You defined simple routes, a redirect route, and a parameterized route.
* You used the [routerLink](https://angular.io/api/router/RouterLink) directive in anchor elements.
* You refactored a tightly-coupled master/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 among multiple components.