Tour of Heroes App and Tutorial

<https://angular.io/tutorial>

GETTING STARTED TUTORIAL

If you're new to Angular, see the [Getting Started tutorial.](https://angular.io/start) The Getting Started tutorial covers the same major topics as this Tour of Heroes—components, template syntax, routing, services, and accessing data via HTTP—in a condensed format, following the most current best practices.

This Tour of Heroes tutorial is the conceptual basis for many examples in this documentation set. Reading this introduction page provides sufficient context for working with those examples. You do not need to do this tutorial to understand those other examples. The Tour of Heroes tutorial is maintained here for context and continuity.

This *Tour of Heroes* tutorial provides an introduction to the fundamentals of Angular. It shows you how to set up your local development environment and develop an app using the [Angular CLI tool](https://angular.io/cli).

In this *Tour of Heroes* tutorial, you will build an app that helps a staffing agency manage its stable of heroes.

This app has many of the features you'd expect to find in a data-driven application. It acquires and displays a list of heroes, edits a selected hero's detail, and navigates among different views of heroic data.

By the end of this tutorial you will be able to do the following:

* Use built-in 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 master 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.

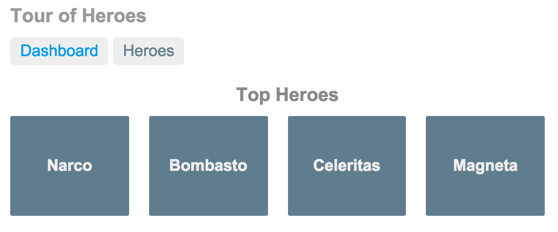
You'll learn enough Angular to get started and gain confidence that Angular can do whatever you need it to do.

SOLUTION

After completing all tutorial steps, the final app will look like this: [live example](https://angular.io/generated/live-examples/toh-pt6/stackblitz.html) / [download example](https://angular.io/generated/zips/toh-pt6/toh-pt6.zip).

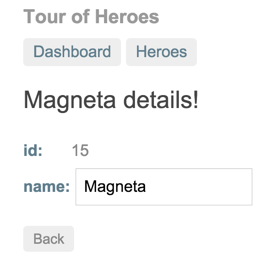
What you'll build

Here's a visual idea of where this tutorial leads, beginning with the "Dashboard" view and the most heroic heroes:



You can click the two links above the dashboard ("Dashboard" and "Heroes") to navigate between this Dashboard view and a Heroes view.

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 app displays the "Heroes" master list view.



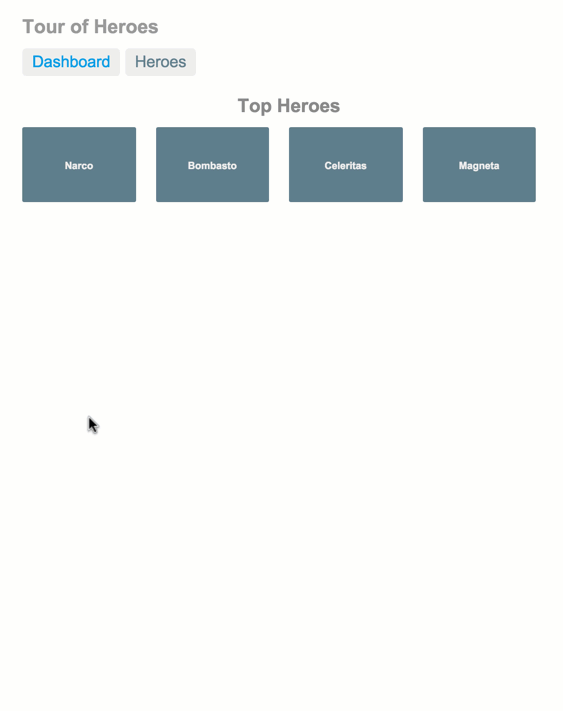
When you click a different hero name, the read-only mini detail beneath the list reflects the new choice.

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

The following diagram captures all of the navigation options.



Here's the app in action:



# 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)

**title = '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)

**<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)

**/\* Application-wide** [**Styles**](https://angular.io/) **\*/**

**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;**

**}**

**/\* everywhere else \*/**

**\* {**

**font-family: Arial, Helvetica, sans-serif;**

**}**

## 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)**

**import {** [**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.

**ng generate component heroes**

The CLI creates a new folder, 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**](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)

**hero = '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

**{{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

**<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

**export 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

**import {** [**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)

**<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.src/app/heroes/heroes.component.html

**<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 flows from the component class out to the screenand from the screen back to the class.

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

### **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>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.name property 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

[**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)

**import {** [**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)

**imports: [**

[**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.src/app/app.module.ts

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

The HeroesComponent is declared in the @[NgModule.declarations](https://angular.io/api/core/NgModule#declarations) array.src/app/app.module.ts

**declarations: [**

**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

**import {** [**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() {**

**}**

**}**

## 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

import { 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

Open the HeroesComponent class file and import the mock HEROES.

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

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

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

src/app/heroes/heroes.component.ts

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

heroes = HEROES;

}

### **List heroes with \***[**ngFor**](https://angular.io/api/common/NgForOf)

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)

<h2>My Heroes</h2>

<ul class="heroes">

<li>

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

</li>

</ul>

That shows one hero. To list them all, add an \*[ngFor](https://angular.io/api/common/NgForOf) to the <li> to iterate through the list of heroes:

**<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.

The syntax in this example is as follows:

* <li> is 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 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#styles) array or as stylesheet file(s) identified in the @[Component.styleUrls](https://angular.io/api/core/Component#styleUrls) array.

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

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

**@**[**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)

**<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.

In the next section, define an onSelect() method in HeroesComponent to display the hero that was defined 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)

**selectedHero: Hero;**

**onSelect(hero: Hero): void {**

**this.selectedHero = hero;**

**}**

### **Add a details section**

Currently, you have a list in the component template. To click on a hero on the list and reveal details about that hero, you need a section for the details to render in the template. Add the following to heroes.component.html beneath the list section:

heroes.component.html (selected hero details)

**<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:

HeroesComponent.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)

**<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**

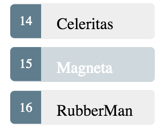
When selectedHero is undefined, the [ngIf](https://angular.io/api/common/NgIf) removes the hero detail from the DOM. There are no selectedHero bindings to consider.

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)

**[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**](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

**import { 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' }**

**];**

## 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.

ng 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

**<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)

**import { 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 external HeroesComponent [will bind to it](https://angular.io/tutorial/toh-pt3#heroes-component-template) like this.

**<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)

**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';**

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

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

**@**[**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)

**<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

**<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

**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';**

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

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

**selector: 'app-hero-detail',**

**templateUrl: './hero-detail.component.html',**

[**styleUrls**](https://angular.io/api/core/Component#styleUrls)**: ['./hero-detail.component.css']**

**})**

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

**@**[**Input**](https://angular.io/api/core/Input)**() hero: Hero;**

**constructor() { }**

**ngOnInit() {**

**}**

**}**

## 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.

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**](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**](https://angular.io/api/core/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.

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, which is the object that is responsible for choosing and injecting the provider where the app requires it.

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

**@**[**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)

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

Replace the definition of the heroes property with a simple 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](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.

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 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 after constructing a HeroesComponent instance.

src/app/heroes/heroes.component.ts

**ngOnInit() {**

**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 HeroesComponentconsumes the getHeroes() result as if heroes could be fetched synchronously.

src/app/heroes/heroes.component.ts

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

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

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

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

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

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

Replace the getHeroes() method with the following:

src/app/hero.service.ts

**getHeroes(): 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)

**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 will work when the HeroService requests heroes from the server.

## Show messages

This section guides you through the following:

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

### **Create MessagesComponent**

Use the CLI to create the MessagesComponent.

**ng generate component** [**messages**](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

**<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.

ng 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

**import {** [**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 MessageService {**

[**messages**](https://angular.io/api/service-worker/SwPush#messages)**: string[] = [];**

**add(**[**message**](https://angular.io/api/common/http/HttpErrorResponse#message)**: string) {**

**this.messages.push(**[**message**](https://angular.io/api/common/http/HttpErrorResponse#message)**);**

**}**

**clear() {**

**this.messages = [];**

**}**

**}**

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**

In HeroService, import the MessageService.

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

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

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

src/app/hero.service.ts

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

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

### **Send a message from HeroService**

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

src/app/hero.service.ts

**getHeroes(): 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)

**import { 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.

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 CLI-generated MessagesComponent template with the following.

src/app/messages/messages.component.html

**<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

**import {** [**Injectable**](https://angular.io/api/core/Injectable) **} from '@angular/core';**

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

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

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

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

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

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

**})**

**export class HeroService {**

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

**getHeroes(): 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);**

**}**

**}**

## 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.

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

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

The generated file looks like this:

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

**import {** [**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 { }**

Replace it with the following:

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

**import {** [**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';**

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

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

**{ path: 'heroes', component: HeroesComponent }**

**];**

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

**imports: [RouterModule.forRoot(routes)],**

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

**})**

**export class AppRoutingModule { }**

First, AppRoutingModule imports [RouterModule](https://angular.io/api/router/RouterModule) and [Routes](https://angular.io/api/router/Routes) so the app can have routing functionality. The next import, HeroesComponent, will give the Router somewhere to go once you configure the routes.

Notice that the [CommonModule](https://angular.io/api/common/CommonModule) references and declarations array are unecessary, 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 AppRoutingModule already imports HeroesComponent, you can use it in the routes array:

src/app/app-routing.module.ts

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

**{ path: 'heroes', component: HeroesComponent }**

**];**

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

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

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()**](https://angular.io/api/router/RouterModule#forRoot)

The @[NgModule](https://angular.io/api/core/NgModule) metadata initializes the router and starts it listening for browser location changes.

The following line adds the [RouterModule](https://angular.io/api/router/RouterModule) to the AppRoutingModule imports array and configures it with the routes in one step by calling [RouterModule.forRoot()](https://angular.io/api/router/RouterModule#forRoot):

src/app/app-routing.module.ts

**imports: [ RouterModule.forRoot(routes) ],**

The method is called [forRoot()](https://angular.io/api/router/RouterModule#forRoot) because you configure the router at the application's root level. The [forRoot()](https://angular.io/api/router/RouterModule#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](https://angular.io/api/router/RouterModule) so it will be available throughout the app.

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

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

## Add [RouterOutlet](https://angular.io/api/router/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)

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

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

**<app-messages></app-messages>**

The AppComponent template no longer needs <app-heroes> because the app 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.

**ng 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))

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**](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#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 [RouterLinkdirective](https://angular.io/api/router/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:

ng 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:

src/app/dashboard/dashboard.component.html

src/app/dashboard/dashboard.component.ts

src/app/dashboard/dashboard.component.css

**<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).

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 AppRoutingModule.

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

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

Add a route to the AppRoutingModule.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 app 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](https://angular.io/api/router/RouterOutlet)> is blank.

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

src/app/app-routing.module.ts

**{ 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

**<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)

**import { 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.

src/app/app-routing.module.ts

**{ 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)

**const 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)

**<**[**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)

**<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)

**<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)

**export 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. This section explains the following:

* 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

**import {** [**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:

toh-pt5/src/app/hero-detail/hero-detail.component.ts

**constructor(**

**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 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.

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

**ngOnInit(): 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 the following getHero() method with the id after the getHeroes() method:

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

**getHero(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#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 HeroDetailComponent that 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)

**<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)

**goBack(): 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.

## 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

import { [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';

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

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

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

const 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 }

];

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

imports: [ RouterModule.forRoot(routes) ],

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

})

export class AppRoutingModule {}

#### **AppComponent**

src/app/app.component.html

src/app/app.component.css

**<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

**<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

**<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

**<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.

# HTTP

In this tutorial, you'll add the following data persistence features with help from Angular's [HttpClient](https://angular.io/api/common/http/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.

When you're done with this page, the app should look like this [live example](https://angular.io/generated/live-examples/toh-pt6/stackblitz.html) / [download example](https://angular.io/generated/zips/toh-pt6/toh-pt6.zip).

## Enable HTTP services

[HttpClient](https://angular.io/api/common/http/HttpClient) is Angular's mechanism for communicating with a remote server over HTTP.

Make [HttpClient](https://angular.io/api/common/http/HttpClient) available everywhere in the app in two steps. First, add it to the root AppModule by importing it:

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

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

Next, still in the AppModule, add [HttpClient](https://angular.io/api/common/http/HttpClient) to the imports array:

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

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

**imports: [**

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

**],**

**})**

## Simulate a data server

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

After installing the module, the app will make requests to and receive responses from the [HttpClient](https://angular.io/api/common/http/HttpClient) without knowing that the In-memory Web API is intercepting those requests, applying them to an in-memory data store, and returning simulated responses.

By using the In-memory Web API, you won't have to set up a server to learn about [HttpClient](https://angular.io/api/common/http/HttpClient).

**Important:** the In-memory Web API module has nothing to do with HTTP in Angular.

If you're just reading this tutorial to learn about [HttpClient](https://angular.io/api/common/http/HttpClient), you can [skip over](https://angular.io/tutorial/toh-pt6#import-heroes) 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**

In the AppModule, import the HttpClientInMemoryWebApiModule and the InMemoryDataService class, which you will create in a moment.

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](https://angular.io/api/common/http/HttpClientModule), add the HttpClientInMemoryWebApiModule to the AppModule imports array and configure it with the InMemoryDataService.

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

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

**// The HttpClientInMemoryWebApiModule module intercepts HTTP requests**

**// and returns simulated server responses.**

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

**HttpClientInMemoryWebApiModule.forRoot(**

**InMemoryDataService, { dataEncapsulation: false }**

**)**

The [forRoot()](https://angular.io/api/router/RouterModule#forRoot) configuration method takes an InMemoryDataService class that primes the in-memory database.

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

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

**import {** [**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 InMemoryDataService implements InMemoryDbService {**

**createDb() {**

**const heroes = [**

**{ 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' }**

**];**

**return {heroes};**

**}**

**// Overrides the genId method to ensure that** [**a**](https://angular.io/api/router/RouterLinkWithHref) **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;**

**}**

**}**

The in-memory-data.service.ts file replaces mock-heroes.ts, which is now safe to delete.

When the server is ready, you'll detach the In-memory Web API, and the app's requests will go through to the server.

## Heroes and HTTP

In the HeroService, import [HttpClient](https://angular.io/api/common/http/HttpClient) and [HttpHeaders](https://angular.io/api/common/http/HttpHeaders):

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

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

Still in the HeroService, inject [HttpClient](https://angular.io/api/common/http/HttpClient) into the constructor in a private property called [http](https://angular.io/api/common/http).

src/app/hero.service.ts

**constructor(**

**private** [**http**](https://angular.io/api/common/http)**:** [**HttpClient**](https://angular.io/api/common/http/HttpClient)**,**

**private messageService: MessageService) { }**

Notice that you keep injecting the MessageService but since you'll call it so frequently, wrap it in a private [log](https://angular.io/api/animations/browser/testing/MockAnimationDriver#log)() method:

src/app/hero.service.ts

**/\*\* Log** [**a**](https://angular.io/api/router/RouterLinkWithHref) **HeroService** [**message**](https://angular.io/api/common/http/HttpErrorResponse#message) **with the MessageService \*/**

**private** [**log**](https://angular.io/api/animations/browser/testing/MockAnimationDriver#log)**(**[**message**](https://angular.io/api/common/http/HttpErrorResponse#message)**: string) {**

**this.messageService.add(`HeroService: ${**[**message**](https://angular.io/api/common/http/HttpErrorResponse#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**](https://angular.io/api/core/SecurityContext#URL) **to web api**

### **Get heroes with**[**HttpClient**](https://angular.io/api/common/http/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[]> {**

**return of(HEROES);**

**}**

Convert that method to use [HttpClient](https://angular.io/api/common/http/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 app keeps working without any other changes because both functions return an Observable<Hero[]>.

### [**HttpClient**](https://angular.io/api/common/http/HttpClient)**methods return one value**

All [HttpClient](https://angular.io/api/common/http/HttpClient) methods return an RxJS Observable of something.

HTTP is a request/response protocol. You make a request, it returns a single response.

In general, an observable can return multiple values over time. An observable from [HttpClient](https://angular.io/api/common/http/HttpClient) always emits a single value and then completes, never to emit again.

This particular [HttpClient.get()](https://angular.io/api/common/http/HttpClient#get) call returns an Observable<Hero[]>; that is, "an observable of hero arrays". In practice, it will only return a single hero array.

### [**HttpClient.get()**](https://angular.io/api/common/http/HttpClient#get)**returns response data**

[HttpClient.get()](https://angular.io/api/common/http/HttpClient#get) returns the body of the response as an untyped JSON object by default. Applying the optional type specifier, <Hero[]> , gives you a typed result object.

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()](https://angular.io/api/core/QueryList#map)operator.

Although not discussed here, there's an example of [map()](https://angular.io/api/core/QueryList#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 you'll need 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**. It passes the error an error handler that can do what it wants with the error.

The following [handleError()](https://angular.io/api/core/ErrorHandler#handleError) method reports the error and then returns an innocuous result so that the application keeps working.

#### **handleError**

The following [handleError()](https://angular.io/api/core/ErrorHandler#handleError) will be shared by many HeroService methods so it's generalized to meet their different needs.

Instead of handling the error directly, it returns an error handler function to catchError that it has configured with both the name of the operation that failed and a safe return value.

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**](https://angular.io/api/animations/AnimationQueryOptions#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**](https://angular.io/api/animations/browser/testing/MockAnimationDriver#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 user friendly message and returns a safe value to the app so the app can keep working.

Because each service method returns a different kind of Observable result, [handleError()](https://angular.io/api/core/ErrorHandler#handleError) takes a type parameter so it can return the safe value as the type that the app expects.

### **Tap into the Observable**

The HeroService methods will **tap** into the flow of observable values and send a message, via the [log](https://angular.io/api/animations/browser/testing/MockAnimationDriver#log)() method, to the message area at the bottom of the page.

They'll do that with the RxJS tap() operator, which looks at the observable values, does something with those values, and passes them along. The tap() call back doesn't touch the values themselves.

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(**[**\_**](https://angular.io/) **=> 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](https://angular.io/tutorial/toh-pt6#heroes-and-http) section (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(**[**\_**](https://angular.io/) **=> this.log(`fetched hero id=${id}`)),**

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

**);**

**}**

There are 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> ("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. But when you click the "go back button", the 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 (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 {**

**this.heroService.updateHero(this.hero)**

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

**}**

#### **Add HeroService.updateHero()**

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

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

**/\*\* PUT:** [**update**](https://angular.io/api/forms/NgModel#update) **the hero on the server \*/**

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

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

**tap(**[**\_**](https://angular.io/) **=> this.log(`updated hero id=${hero.id}`)),**

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

**);**

**}**

The [HttpClient.put()](https://angular.io/api/common/http/HttpClient#put) method takes three parameters:

* the URL
* the data to update (the modified hero in this case)
* options

The URL is unchanged. The heroes web API knows which hero to update by looking at the hero's id.

The heroes web API expects a special header in HTTP save requests. That header is in the httpOptions constant defined in the HeroService. Add the following to the HeroService class.

src/app/hero.service.ts

**httpOptions = {**

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

**};**

Refresh the browser, change a hero name and save your change. The save() method in HeroDetailComponentnavigates to the previous view. The hero now appears in the list with the changed name.

## Add a new hero

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

Insert the following into the HeroesComponent template, just after the heading:

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

**<div>**

**<label>Hero name:**

**<input #heroName />**

**</label>**

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

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

**add**

**</button>**

**</div>**

In response to a click event, call the component's click handler, add(), and then clear the input field so that it's ready for another name. Add the following to theHeroesComponent 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 is non-blank, the handler creates a Hero-like object from the name (it's only missing the id) and passes it to the services addHero() method.

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

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

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

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

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

**return this.http.post<Hero>(this.heroesUrl, hero, 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()](https://angular.io/api/common/http/HttpClient#post) instead of [put()](https://angular.io/api/common/http/HttpClient#put).
* It expects the server to generate 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 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**](https://angular.io/api/common/NgForOf)**="let hero of heroes">**

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

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

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

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

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

**</li>**

**</ul>**

To position the delete button at the far right of the hero entry, add some CSS to the heroes.component.css. You'll find that CSS in the [final review code](https://angular.io/tutorial/toh-pt6#heroescomponent) below.

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).subscribe();**

}

Although the component delegates hero deletion to the HeroService, it remains responsible for updating its own list of heroes. The component's delete() method immediately removes the hero-to-delete from that list, anticipating that the HeroService will succeed on the server.

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

If you neglect to subscribe(), the service will not send the delete request to the server. As a rule, an Observable does nothing until something subscribes.

Confirm this for yourself by temporarily removing the subscribe(), clicking "Dashboard", then clicking "Heroes". You'll see the full list of heroes again.

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

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

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

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

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

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

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

**tap(**[**\_**](https://angular.io/) **=> this.log(`deleted hero id=${id}`)),**

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

**);**

**}**

Note the following key points:

* deleteHero() calls [HttpClient.delete()](https://angular.io/api/common/http/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()](https://angular.io/api/common/http/HttpClient#put) and [post()](https://angular.io/api/common/http/HttpClient#post).
* You still send the httpOptions.

Refresh the browser and try the new delete functionality.

## Search by name

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

You will add a heroes search feature to the Dashboard. As the user types a name into a search box, you'll make repeated HTTP requests for heroes filtered by that name. Your goal is to issue only as many requests as necessary.

#### **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(**[**\_**](https://angular.io/) **=> this.log(`found heroes matching "${term}"`)),**

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

**);**

**}**

The method returns immediately with an empty array if there is no search term. The rest of it closely resembles getHeroes(), the only significant difference 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

**<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>**

**<app-hero-search></app-hero-search>**

This template looks a lot like the \*[ngFor](https://angular.io/api/common/NgForOf) 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**

Create a HeroSearchComponent with the CLI.

ng generate component hero-search

The CLI generates the three HeroSearchComponent files and adds the component to the AppModule declarations.

Replace the generated 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">**

**<h4><label for="search-box">Hero Search</label></h4>**

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

**<ul class="search-result">**

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

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

**{{hero.name}}**

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

**</li>**

**</ul>**

**</div>**

Add private CSS styles to hero-search.component.css as listed in the [final code review](https://angular.io/tutorial/toh-pt6#herosearchcomponent) below.

As the user types in the search box, an input event binding calls the component's [search()](https://angular.io/api/common/upgrade/$locationShim#search) method with the new search box value.

### [**AsyncPipe**](https://angular.io/api/common/AsyncPipe)

The \*[ngFor](https://angular.io/api/common/NgForOf) repeats hero objects. Notice that the \*[ngFor](https://angular.io/api/common/NgForOf) 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**](https://angular.io/api/common/NgForOf)**="let hero of heroes$ |** [**async**](https://angular.io/api/core/testing/async)**" >**

Since \*[ngFor](https://angular.io/api/common/NgForOf) can't do anything with an Observable, use the pipe character (|) followed by [async](https://angular.io/api/core/testing/async). This identifies Angular's [AsyncPipe](https://angular.io/api/common/AsyncPipe) and subscribes to an Observableautomatically so you won't have to do so in the component class.

### **Edit the HeroSearchComponent class**

Replace the generated HeroSearchComponent class and metadata as follows.

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

**import {** [**Component**](https://angular.io/api/core/Component)**,** [**OnInit**](https://angular.io/api/core/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**](https://angular.io/api/core/Component)**({**

**selector: 'app-hero-search',**

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

[**styleUrls**](https://angular.io/api/core/Component#styleUrls)**: [ './hero-search.component.css' ]**

**})**

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

**heroes$: Observable<Hero[]>;**

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

**constructor(private heroService: HeroService) {}**

**// Push** [**a**](https://angular.io/api/router/RouterLinkWithHref) **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**](https://angular.io/api/service-worker/UpdateActivatedEvent#previous) **term**

**distinctUntilChanged(),**

**// switch to new search observable each** [**time**](https://angular.io/) **the term** [**changes**](https://angular.io/api/core/QueryList#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[]>;

You'll set it in [ngOnInit()](https://angular.io/tutorial/toh-pt6#search-pipe). 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**](https://angular.io/api/router/RouterLinkWithHref) **search term into the observable stream.**

**search(term: string): void {**

**this.searchTerms.next(term);**

**}**

A Subject is both a source of observable values and an Observable itself. You can subscribe to a Subject as you would any Observable.

You can also push values into that Observable by calling its next(value) method as the [search()](https://angular.io/api/common/upgrade/$locationShim#search) method does.

The event binding to the textbox's input event calls the [search()](https://angular.io/api/common/upgrade/$locationShim#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 textbox, the binding calls [search()](https://angular.io/api/common/upgrade/$locationShim#search) with the textbox value, 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 would create an excessive amount of HTTP requests, taxing server resources and burning 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 returning an observable of timely hero search results (each a Hero[]).

Here's a closer look at the code.

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

**this.heroes$ = this.searchTerms.pipe(**

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

**debounceTime(300),**

**// ignore new term if same as** [**previous**](https://angular.io/api/service-worker/UpdateActivatedEvent#previous) **term**

**distinctUntilChanged(),**

**// switch to new search observable each** [**time**](https://angular.io/) **the term** [**changes**](https://angular.io/api/core/QueryList#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. You'll never make requests more frequently than 300ms.
* distinctUntilChanged() ensures that a request is sent only if the filter text changed.
* switchMap() calls the search service for each search term that makes it through debounce() and distinctUntilChanged(). It cancels and discards previous search observables, returning only the latest search service observable.

With the [switchMap operator](http://www.learnrxjs.io/operators/transformation/switchmap.html), every qualifying key event can trigger an [HttpClient.get()](https://angular.io/api/common/http/HttpClient#get) method call. Even with a 300ms pause between requests, you could have multiple HTTP requests in flight and they may not return in the order sent.

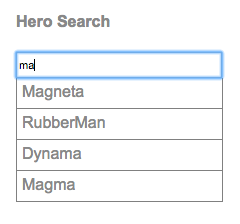
switchMap() preserves the original request order while returning only the observable from the most recent HTTP method call. Results from prior calls are canceled and discarded.

Note that canceling a previous searchHeroes() Observable doesn't actually abort a pending HTTP request. Unwanted results are simply discarded before they reach your application code.

Remember that the component class does not subscribe to the heroes$ observable. That's the job of the [AsyncPipe](https://angular.io/tutorial/toh-pt6#asyncpipe) in the template.

#### **Try it**

Run the app again. In the Dashboard, enter some text in the search box. If you enter characters that match any existing hero names, you'll see something like this.



## Final code review

Your app should look like this [live example](https://angular.io/generated/live-examples/toh-pt6/stackblitz.html) / [download example](https://angular.io/generated/zips/toh-pt6/toh-pt6.zip).

Here are the code files discussed on this page (all in the src/app/ folder).

#### **HeroService, InMemoryDataService, AppModule**

hero.service.ts

in-memory-data.service.ts

app.module.ts

**import {** [**Injectable**](https://angular.io/api/core/Injectable) **} from '@angular/core';**

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

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

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

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

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

**@**[**Injectable**](https://angular.io/api/core/Injectable)**({** [**providedIn**](https://angular.io/api/core/Injectable#providedIn)**: 'root' })**

**export class HeroService {**

**private heroesUrl = 'api/heroes'; //** [**URL**](https://angular.io/api/core/SecurityContext#URL) **to web api**

**httpOptions = {**

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

**};**

**constructor(**

**private** [**http**](https://angular.io/api/common/http)**:** [**HttpClient**](https://angular.io/api/common/http/HttpClient)**,**

**private messageService: MessageService) { }**

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

**getHeroes (): Observable<Hero[]> {**

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

**.pipe(**

**tap(**[**\_**](https://angular.io/) **=> this.log('fetched heroes')),**

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

**);**

**}**

**/\*\* GET hero by id. Return `undefined` when id not found \*/**

**getHeroNo404<**[**Data**](https://angular.io/api/router/Data)**>(id: number): Observable<Hero> {**

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

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

**.pipe(**

**map(heroes => heroes[0]), // returns** [**a**](https://angular.io/api/router/RouterLinkWithHref) **{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(**[**\_**](https://angular.io/) **=> 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(**[**\_**](https://angular.io/) **=> this.log(`found heroes matching "${term}"`)),**

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

**);**

**}**

**//////// Save methods //////////**

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

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

**return this.http.post<Hero>(this.heroesUrl, hero, 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 (hero: Hero | number): Observable<Hero> {**

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

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

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

**tap(**[**\_**](https://angular.io/) **=> this.log(`deleted hero id=${id}`)),**

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

**);**

**}**

**/\*\* PUT:** [**update**](https://angular.io/api/forms/NgModel#update) **the hero on the server \*/**

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

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

**tap(**[**\_**](https://angular.io/) **=> 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**](https://angular.io/api/animations/AnimationQueryOptions#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**](https://angular.io/api/animations/browser/testing/MockAnimationDriver#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**](https://angular.io/api/router/RouterLinkWithHref) **HeroService** [**message**](https://angular.io/api/common/http/HttpErrorResponse#message) **with the MessageService \*/**

**private** [**log**](https://angular.io/api/animations/browser/testing/MockAnimationDriver#log)**(**[**message**](https://angular.io/api/common/http/HttpErrorResponse#message)**: string) {**

**this.messageService.add(`HeroService: ${**[**message**](https://angular.io/api/common/http/HttpErrorResponse#message)**}`);**

**}**

**}**

#### **HeroesComponent**

heroes/heroes.component.html

heroes/heroes.component.ts

heroes/heroes.component.css

**<h2>My Heroes</h2>**

**<div>**

**<label>Hero name:**

**<input #heroName />**

**</label>**

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

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

**add**

**</button>**

**</div>**

**<ul class="heroes">**

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

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

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

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

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

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

**</li>**

**</ul>**

#### **HeroDetailComponent**

hero-detail/hero-detail.component.html

hero-detail/hero-detail.component.ts

**<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>**

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

**</div>**

#### **DashboardComponent**

src/app/dashboard/dashboard.component.html

**<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>**

**<app-hero-search></app-hero-search>**

#### **HeroSearchComponent**

hero-search/hero-search.component.html

hero-search/hero-search.component.ts

hero-search/hero-search.component.css

**<div id="search-component">**

**<h4><label for="search-box">Hero Search</label></h4>**

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

**<ul class="search-result">**

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

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

**{{hero.name}}**

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

**</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 app.
* You refactored HeroService to load heroes from a web API.
* You extended HeroService to support [post()](https://angular.io/api/common/http/HttpClient#post), [put()](https://angular.io/api/common/http/HttpClient#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.

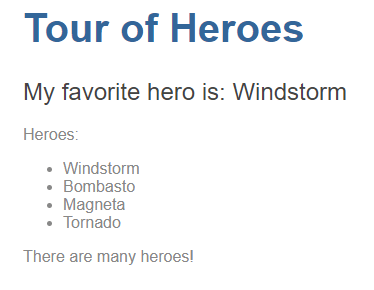
This concludes the "Tour of Heroes" tutorial. You're ready to learn more about Angular development in the fundamentals section, starting with the [Architecture](https://angular.io/guide/architecture) guide.

# Displaying Data

You can display data by binding controls in an HTML template to properties of an Angular component.

In this page, you'll create a component with a list of heroes. You'll display the list of hero names and conditionally show a message below the list.

The final UI looks like this:



The [live example](https://angular.io/generated/live-examples/displaying-data/stackblitz.html) / [download example](https://angular.io/generated/zips/displaying-data/displaying-data.zip) demonstrates all of the syntax and code snippets described in this page.

## Showing component properties with interpolation

The easiest way to display a component property is to bind the property name through interpolation. With interpolation, you put the property name in the view template, enclosed in double curly braces: {{myHero}}.

Use the CLI command [ng new displaying-data](https://angular.io/cli/new) to create a workspace and app named displaying-data.

Delete the app.component.html file. It is not needed for this example.

Then modify the app.component.ts file by changing the template and the body of the component.

When you're done, it should look like this:

src/app/app.component.ts

**import {** [**Component**](https://angular.io/api/core/Component) **} from '@angular/core';**

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

**selector: 'app-root',**

**template: `**

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

**<h2>My favorite hero is: {{myHero}}</h2>**

**`**

**})**

**export class AppComponent {**

**title = 'Tour of Heroes';**

**myHero = 'Windstorm';**

**}**

You added two properties to the formerly empty component: title and myHero.

The template displays the two component properties using double curly brace interpolation:

src/app/app.component.ts (template)

**template: `**

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

**<h2>My favorite hero is: {{myHero}}</h2>**

`

The template is a multi-line string within ECMAScript 2015 backticks (`). The backtick (`)—which is not the same character as a single quote (')—allows you to compose a string over several lines, which makes the HTML more readable.

Angular automatically pulls the value of the title and myHero properties from the component and inserts those values into the browser. Angular updates the display when these properties change.

More precisely, the redisplay occurs after some kind of asynchronous event related to the view, such as a keystroke, a timer completion, or a response to an HTTP request.

Notice that you don't call **new** to create an instance of the AppComponent class. Angular is creating an instance for you. How?

The CSS selector in the @[Component](https://angular.io/api/core/Component) decorator specifies an element named <app-root>. That element is a placeholder in the body of your index.html file:

src/index.html (body)

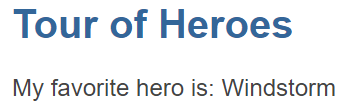
**<body>**

**<app-root></app-root>**

**</body>**

When you bootstrap with the AppComponent class (in main.ts), Angular looks for a <app-root> in the index.html, finds it, instantiates an instance of AppComponent, and renders it inside the <app-root> tag.

Now run the app. It should display the title and hero name:



The next few sections review some of the coding choices in the app.

## Template inline or template file?

You can store your component's template in one of two places. You can define it inline using the template property, or you can define the template in a separate HTML file and link to it in the component metadata using the @[Component](https://angular.io/api/core/Component) decorator's templateUrl property.

The choice between inline and separate HTML is a matter of taste, circumstances, and organization policy. Here the app uses inline HTML because the template is small and the demo is simpler without the additional HTML file.

In either style, the template data bindings have the same access to the component's properties.

By default, the Angular CLI command [ng generate component](https://angular.io/cli/generate) generates components with a template file. You can override that with:

ng generate component hero -it

## Constructor or variable initialization?

Although this example uses variable assignment to initialize the components, you could instead declare and initialize the properties using a constructor:

**export class AppComponent {**

**title: string;**

**myHero: string;**

**constructor() {**

**this.title = 'Tour of Heroes';**

**this.myHero = 'Windstorm';**

**}**

**}**

This app uses more terse "variable assignment" style simply for brevity.

## Showing an array property with **\*ngFor**

To display a list of heroes, begin by adding an array of hero names to the component and redefine myHero to be the first name in the array.

src/app/app.component.ts (class)

**export class AppComponent {**

**title = 'Tour of Heroes';**

**heroes = ['Windstorm', 'Bombasto', 'Magneta', 'Tornado'];**

**myHero = this.heroes[0];**

**}**

Now use the Angular [ngFor](https://angular.io/api/common/NgForOf) directive in the template to display each item in the heroes list.

src/app/app.component.ts (template)

**template: `**

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

**<h2>My favorite hero is: {{myHero}}</h2>**

**<p>Heroes:</p>**

**<ul>**

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

**{{ hero }}**

**</li>**

**</ul>**

**`**

This UI uses the HTML unordered list with <ul> and <li> tags. The \*[ngFor](https://angular.io/api/common/NgForOf) in the <li> element is the Angular "repeater" directive. It marks that <li> element (and its children) as the "repeater template":

src/app/app.component.ts (li)

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

**{{ hero }}**

**</li>**

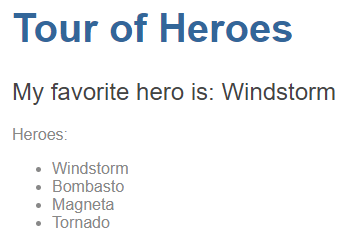
Don't forget the leading asterisk (\*) in \*[ngFor](https://angular.io/api/common/NgForOf). It is an essential part of the syntax. For more information, see the [Template Syntax](https://angular.io/guide/template-syntax#ngFor) page.

Notice the hero in the [ngFor](https://angular.io/api/common/NgForOf) double-quoted instruction; it is an example of a template input variable. Read more about template input variables in the [microsyntax](https://angular.io/guide/template-syntax#microsyntax) section of the [Template Syntax](https://angular.io/guide/template-syntax) page.

Angular duplicates the <li> for each item in the list, setting the hero variable to the item (the hero) in the current iteration. Angular uses that variable as the context for the interpolation in the double curly braces.

In this case, [ngFor](https://angular.io/api/common/NgForOf) is displaying an array, but [ngFor](https://angular.io/api/common/NgForOf) can repeat items for any [iterable](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols) object.

Now the heroes appear in an unordered list.



## Creating a class for the data

The app's code defines the data directly inside the component, which isn't best practice. In a simple demo, however, it's fine.

At the moment, the binding is to an array of strings. In real applications, most bindings are to more specialized objects.

To convert this binding to use specialized objects, turn the array of hero names into an array of Hero objects. For that you'll need a Hero class:

ng generate class hero

With the following code:

src/app/hero.ts

**export class Hero {**

**constructor(**

**public id: number,**

**public name: string) { }**

**}**

You've defined a class with a constructor and two properties: id and name.

It might not look like the class has properties, but it does. The declaration of the constructor parameters takes advantage of a TypeScript shortcut.

Consider the first parameter:

src/app/hero.ts (id)

**public id: number,**

That brief syntax does a lot:

* Declares a constructor parameter and its type.
* Declares a public property of the same name.
* Initializes that property with the corresponding argument when creating an instance of the class.

### **Using the Hero class**

After importing the Hero class, the AppComponent.heroes property can return a typed array of Hero objects:

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

**heroes = [**

**new Hero(1, 'Windstorm'),**

**new Hero(13, 'Bombasto'),**

**new Hero(15, 'Magneta'),**

**new Hero(20, 'Tornado')**

**];**

**myHero = this.heroes[0];**

Next, update the template. At the moment it displays the hero's id and name. Fix that to display only the hero's name property.

src/app/app.component.ts (template)

**template: `**

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

**<h2>My favorite hero is: {{myHero.name}}</h2>**

**<p>Heroes:</p>**

**<ul>**

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

**{{ hero.name }}**

**</li>**

**</ul>**

**`**

The display looks the same, but the code is clearer.

## Conditional display with NgIf

Sometimes an app needs to display a view or a portion of a view only under specific circumstances.

Let's change the example to display a message if there are more than three heroes.

The Angular [ngIf](https://angular.io/api/common/NgIf) directive inserts or removes an element based on a truthy/falsy condition. To see it in action, add the following paragraph at the bottom of the template:

src/app/app.component.ts (message)

<p \*[ngIf](https://angular.io/api/common/NgIf)="heroes.length > 3">There are many heroes!</p>

Don't forget the leading asterisk (\*) in \*[ngIf](https://angular.io/api/common/NgIf). It is an essential part of the syntax. Read more about [ngIf](https://angular.io/api/common/NgIf) and \* in the [ngIf section](https://angular.io/guide/template-syntax#ngIf) of the [Template Syntax](https://angular.io/guide/template-syntax) page.

The template expression inside the double quotes, \*[ngIf](https://angular.io/api/common/NgIf)="heroes.length > 3", looks and behaves much like TypeScript. When the component's list of heroes has more than three items, Angular adds the paragraph to the DOM and the message appears. If there are three or fewer items, Angular omits the paragraph, so no message appears. For more information, see the [template expressions](https://angular.io/guide/template-syntax#template-expressions) section of the [Template Syntax](https://angular.io/guide/template-syntax) page.

Angular isn't showing and hiding the message. It is adding and removing the paragraph element from the DOM. That improves performance, especially in larger projects when conditionally including or excluding big chunks of HTML with many data bindings.

Try it out. Because the array has four items, the message should appear. Go back into app.component.ts and delete or comment out one of the elements from the heroes array. The browser should refresh automatically and the message should disappear.

## Summary

Now you know how to use:

* **Interpolation** with double curly braces to display a component property.
* **ngFor** to display an array of items.
* A TypeScript class to shape the **model data** for your component and display properties of that model.
* **ngIf** to conditionally display a chunk of HTML based on a boolean expression.

Here's the final code:

src/app/app.component.ts

src/app/hero.ts

src/app/app.module.ts

main.ts

**import {** [**Component**](https://angular.io/api/core/Component) **} from '@angular/core';**

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

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

**selector: 'app-root',**

**template: `**

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

**<h2>My favorite hero is: {{myHero.name}}</h2>**

**<p>Heroes:</p>**

**<ul>**

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

**{{ hero.name }}**

**</li>**

**</ul>**

**<p \***[**ngIf**](https://angular.io/api/common/NgIf)**="heroes.length > 3">There are many heroes!</p>**

**`**

**})**

**export class AppComponent {**

**title = 'Tour of Heroes';**

**heroes = [**

**new Hero(1, 'Windstorm'),**

**new Hero(13, 'Bombasto'),**

**new Hero(15, 'Magneta'),**

**new Hero(20, 'Tornado')**

**];**

**myHero = this.heroes[0];**

**}**

# Template Syntax

The Angular application manages what the user sees and can do, achieving this through the interaction of a component class instance (the component) and its user-facing template.

You may be familiar with the component/template duality from your experience with model-view-controller (MVC) or model-view-viewmodel (MVVM). In Angular, the component plays the part of the controller/viewmodel, and the template represents the view.

This page is a comprehensive technical reference to the Angular template language. It explains basic principles of the template language and describes most of the syntax that you'll encounter elsewhere in the documentation.

Many code snippets illustrate the points and concepts, all of them available in the [Template Syntax Live Code](https://angular.io/generated/live-examples/template-syntax/stackblitz.html) / [download example](https://angular.io/generated/zips/template-syntax/template-syntax.zip).

## HTML in templates

HTML is the language of the Angular template. Almost all HTML syntax is valid template syntax. The <script> element is a notable exception; it is forbidden, eliminating the risk of script injection attacks. In practice, <script> is ignored and a warning appears in the browser console. See the [Security](https://angular.io/guide/security) page for details.

Some legal HTML doesn't make much sense in a template. The <html>, <body>, and <base> elements have no useful role. Pretty much everything else is fair game.

You can extend the HTML vocabulary of your templates with components and directives that appear as new elements and attributes. In the following sections, you'll learn how to get and set DOM (Document Object Model) values dynamically through data binding.

Begin with the first form of data binding—interpolation—to see how much richer template HTML can be.

## Interpolation and Template Expressions

Interpolation allows you to incorporate calculated strings into the text between HTML element tags and within attribute assignments. Template expressions are what you use to calculate those strings.

The interpolation [live example](https://angular.io/generated/live-examples/template-syntax/stackblitz.html) / [download example](https://angular.io/generated/zips/template-syntax/template-syntax.zip) demonstrates all of the syntax and code snippets described in this section.

### **Interpolation {{...}}**

Interpolation refers to embedding expressions into marked up text. By default, interpolation uses as its delimiter the double curly braces, {{ and }}.

In the following snippet, {{ currentCustomer }} is an example of interpolation.

src/app/app.component.html

**<h3>Current customer: {{ currentCustomer }}</h3>**

The text between the braces is often the name of a component property. Angular replaces that name with the string value of the corresponding component property.

src/app/app.component.html

**<p>{{title}}</p>**

**<div><img src="{{itemImageUrl}}"></div>**

In the example above, Angular evaluates the title and itemImageUrl properties and fills in the blanks, first displaying some title text and then an image.

More generally, the text between the braces is a **template expression** that Angular first **evaluates** and then **converts to a string**. The following interpolation illustrates the point by adding two numbers:

src/app/app.component.html

**<!-- "The sum of 1 + 1 is 2" -->**

**<p>The sum of 1 + 1 is {{1 + 1}}.</p>**

The expression can invoke methods of the host component such as getVal() in the following example:

src/app/app.component.html

**<!-- "The sum of 1 + 1 is not 4" -->**

**<p>The sum of 1 + 1 is not {{1 + 1 + getVal()}}.</p>**

Angular evaluates all expressions in double curly braces, converts the expression results to strings, and links them with neighboring literal strings. Finally, it assigns this composite interpolated result to an **element or directive property**.

You appear to be inserting the result between element tags and assigning it to attributes.

However, interpolation is a special syntax that Angular converts into a property binding.

If you'd like to use something other than {{ and }}, you can configure the interpolation delimiter via the [interpolation](https://angular.io/api/core/Component#interpolation) option in the [Component](https://angular.io/api/core/Component) metadata.

### **Template expressions**

A template **expression** produces a value and appears within the double curly braces, {{ }}. Angular executes the expression and assigns it to a property of a binding target; the target could be an HTML element, a component, or a directive.

The interpolation braces in {{1 + 1}} surround the template expression 1 + 1. In the property binding, a template expression appears in quotes to the right of the =symbol as in [property]="expression".

In terms of syntax, template expressions are similar to JavaScript. Many JavaScript expressions are legal template expressions, with a few exceptions.

You can't use JavaScript expressions that have or promote side effects, including:

* Assignments (=, +=, -=, ...)
* Operators such as new, typeof, instanceof, etc.
* Chaining expressions with ; or ,
* The increment and decrement operators ++ and --
* Some of the ES2015+ operators

Other notable differences from JavaScript syntax include:

* No support for the bitwise operators such as | and &
* New template expression operators, such as |, ?. and !

### **Expression context**

The expression context is typically the component instance. In the following snippets, the recommended within double curly braces and the itemImageUrl2 in quotes refer to properties of the AppComponent.

src/app/app.component.html

**<h4>{{recommended}}</h4>**

**<img [src]="itemImageUrl2">**

An expression may also refer to properties of the template's context such as a template input variable,

let customer, or a template reference variable, #customerInput.

src/app/app.component.html (template input variable)

**<ul>**

**<li \***[**ngFor**](https://angular.io/api/common/NgForOf)**="let customer of customers">{{customer.name}}</li>**

**</ul>**

src/app/app.component.html (template reference variable)

**<label>**[**Type**](https://angular.io/api/core/Type) **something:**

**<input #customerInput>{{customerInput.value}}**

**</label>**

The context for terms in an expression is a blend of the template variables, the directive's context object (if it has one), and the component's members. If you reference a name that belongs to more than one of these namespaces, the template variable name takes precedence, followed by a name in the directive's context, and, lastly, the component's member names.

The previous example presents such a name collision. The component has a customer property and the \*[ngFor](https://angular.io/api/common/NgForOf) defines a customer template variable.

The customer in {{customer.name}} refers to the template input variable, not the component's property.

Template expressions cannot refer to anything in the global namespace, except undefined. They can't refer to window or [document](https://angular.io/api/platform-server/PlatformConfig#document). Additionally, they can't call console.log() or Math.max() and they are restricted to referencing members of the expression context.

### **Expression guidelines**

When using template expressions follow these guidelines:

* [No visible side effects](https://angular.io/guide/template-syntax#no-visible-side-effects)
* [Quick execution](https://angular.io/guide/template-syntax#quick-execution)
* [Simplicity](https://angular.io/guide/template-syntax#simplicity)

### **No visible side effects**

A template expression should not change any application state other than the value of the target property.

This rule is essential to Angular's "unidirectional data flow" policy. You should never worry that reading a component value might change some other displayed value. The view should be stable throughout a single rendering pass.

An [idempotent](https://en.wikipedia.org/wiki/Idempotence) expression is ideal because it is free of side effects and improves Angular's change detection performance.

In Angular terms, an idempotent expression always returns exactly the same thing until one of its dependent values changes.

Dependent values should not change during a single turn of the event loop. If an idempotent expression returns a string or a number, it returns the same string or number when called twice in a row. If the expression returns an object, including an array, it returns the same object reference when called twice in a row.

There is one exception to this behavior that applies to \*[ngFor](https://angular.io/api/common/NgForOf). \*[ngFor](https://angular.io/api/common/NgForOf) has trackBy functionality that can deal with referential inequality of objects that when iterating over them.

For more information, see the [\*ngFor with trackBy](https://angular.io/guide/template-syntax#ngfor-with-trackby) section of this guide.

### **Quick execution**

Angular executes template expressions after every change detection cycle. Change detection cycles are triggered by many asynchronous activities such as promise resolutions, HTTP results, timer events, key presses and mouse moves.

Expressions should finish quickly or the user experience may drag, especially on slower devices. Consider caching values when their computation is expensive.

### **Simplicity**

Although it's possible to write complex template expressions, it's a better practice to avoid them.

A property name or method call should be the norm, but an occasional Boolean negation, !, is OK. Otherwise, confine application and business logic to the component, where it is easier to develop and test.

## Template statements

A template **statement** responds to an **event** raised by a binding target such as an element, component, or directive. You'll see template statements in the [event binding](https://angular.io/guide/template-syntax#event-binding)section, appearing in quotes to the right of the = symbol as in (event)="statement".

src/app/app.component.html

<button (click)="deleteHero()">Delete hero</button>

A template statement has a side effect. That's the whole point of an event. It's how you update application state from user action.

Responding to events is the other side of Angular's "unidirectional data flow". You're free to change anything, anywhere, during this turn of the event loop.

Like template expressions, template statements use a language that looks like JavaScript. The template statement parser differs from the template expression parser and specifically supports both basic assignment (=) and chaining expressions (with ; or ,).

However, certain JavaScript syntax is not allowed:

* new
* increment and decrement operators, ++ and --
* operator assignment, such as += and -=
* the bitwise operators | and &
* the [template expression operators](https://angular.io/guide/template-syntax#expression-operators)

### **Statement context**

As with expressions, statements can refer only to what's in the statement context such as an event handling method of the component instance.

The statement context is typically the component instance. The deleteHero in (click)="deleteHero()" is a method of the data-bound component.

src/app/app.component.html

<button (click)="deleteHero()">Delete hero</button>

The statement context may also refer to properties of the template's own context. In the following examples, the template $event object, a [template input variable](https://angular.io/guide/template-syntax#template-input-variable) (let hero), and a [template reference variable](https://angular.io/guide/template-syntax#ref-vars) (#heroForm) are passed to an event handling method of the component.

src/app/app.component.html

**<button (click)="onSave($event)">Save</button>**

**<button \***[**ngFor**](https://angular.io/api/common/NgForOf)**="let hero of heroes" (click)="deleteHero(hero)">{{hero.name}}</button>**

**<form #heroForm (ngSubmit)="onSubmit(heroForm)"> ... </form>**

Template context names take precedence over component context names. In deleteHero(hero) above, the hero is the template input variable, not the component's heroproperty.

Template statements cannot refer to anything in the global namespace. They can't refer to window or [document](https://angular.io/api/platform-server/PlatformConfig#document). They can't call console.log or Math.max.

### **Statement guidelines**

As with expressions, avoid writing complex template statements. A method call or simple property assignment should be the norm.

Now that you have a feel for template expressions and statements, you're ready to learn about the varieties of data binding syntax beyond interpolation.

## Binding syntax: an overview

Data-binding is a mechanism for coordinating what users see, specifically with application data values. While you could push values to and pull values from HTML, the application is easier to write, read, and maintain if you turn these tasks over to a binding framework. You simply declare bindings between binding sources, target HTML elements, and let the framework do the rest.

For a demonstration of the syntax and code snippets in this section, see the [binding syntax example](https://angular.io/generated/live-examples/binding-syntax/stackblitz.html) / [download example](https://angular.io/generated/zips/binding-syntax/binding-syntax.zip).

Angular provides many kinds of data-binding. Binding types can be grouped into three categories distinguished by the direction of data flow:

* From the source-to-view
* From view-to-source
* Two-way sequence: view-to-source-to-view

|  |  |  |
| --- | --- | --- |
| **Type** | **Syntax** | **Category** |
| Interpolation Property Attribute Class Style | {{expression}}  [[target](https://angular.io/api/router/RouterLinkWithHref#target)]="expression"  bind-target="expression" | One-way from data source to view target |
| Event | ([target](https://angular.io/api/router/RouterLinkWithHref#target))="statement"  on-target="statement" | One-way from view target to data source |
| Two-way | [([target](https://angular.io/api/router/RouterLinkWithHref#target))]="expression"  bindon-target="expression" | Two-way |

Binding types other than interpolation have a **target name** to the left of the equal sign, either surrounded by punctuation, [] or (), or preceded by a prefix: bind-, on-, bindon-.

The target of a binding is the property or event inside the binding punctuation: [], () or [()].

Every public member of a **source** directive is automatically available for binding. You don't have to do anything special to access a directive member in a template expression or statement.

## Data-binding and HTML

In the normal course of HTML development, you create a visual structure with HTML elements, and you modify those elements by setting element attributes with string constants.

**<div class="special">Plain old** [**HTML**](https://angular.io/api/core/SecurityContext#HTML)**</div>**

**<img src="images/item.png">**

**<button disabled>Save</button>**

With data-binding, you can control things like the state of a button:

src/app/app.component.html

**<!-- Bind button disabled** [**state**](https://angular.io/api/animations/state) **to `isUnchanged` property -->**

**<button [disabled]="isUnchanged">Save</button>**

Notice that the binding is to the disabled property of the button's DOM element, **not** the attribute. This applies to data-binding in general. Data-binding works with properties of DOM elements, components, and directives, not HTML attributes.

## HTML attribute vs. DOM property

The distinction between an HTML attribute and a DOM property is key to understanding how Angular binding works. **Attributes are defined by HTML. Properties are accessed from DOM, or the Document Object Model, nodes.**

* A few HTML attributes have 1:1 mapping to properties; for example, id.
* Some HTML attributes don't have corresponding properties; for example, aria-\*.
* Some DOM properties don't have corresponding attributes; for example, textContent.

This general rule can help you build a mental model of attributes and DOM properties: **attributes initialize DOM properties and then they are done. Property values can change; attribute values can't.**

There is, of course, an exception to this rule because attributes can be changed by [setAttribute()](https://angular.io/), which will re-initialize corresponding DOM properties again.

Comparing the [<td> attributes](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/td) attributes to the [<td> properties](https://developer.mozilla.org/en-US/docs/Web/API/HTMLTableCellElement) provides a helpful example for differentiation. In particular, you can navigate from the attributes page to the properties via "DOM interface" link, and navigate the inheritance hierarchy up to HTMLTableCellElement.

**The HTML attribute and the DOM property are not the same thing, even when they have the same name.**

For more information, see the [MDN Interfaces documentation](https://developer.mozilla.org/en-US/docs/Web/API#Interfaces) which has API docs for all the standard DOM elements and their properties.

### **Example 1: an <input>**

When the browser renders <input type="text" value="Sarah">, it creates a corresponding DOM node with a value property initialized to "Sarah".

**<input type="text" value="Sarah">**

When the user enters "Sally" into the <input>, the DOM element value property becomes "Sally". However, if you look at the HTML attribute value using input.getAttribute('value'), you can see that the attribute remains unchanged—it returns "Sarah".

The HTML attribute value specifies the initial value; the DOM value property is the current value.

To see attributes versus DOM properties in a functioning app, see the [live example](https://angular.io/generated/live-examples/binding-syntax/stackblitz.html) / [download example](https://angular.io/generated/zips/binding-syntax/binding-syntax.zip) especially for binding syntax.

### **Example 2: a disabled button**

The disabled attribute is another example. A button's disabled property is false by default so the button is enabled.

When you add the disabled attribute, its presence alone initializes the button's disabled property to true so the button is disabled.

**<button disabled>Test Button</button>**

Adding and removing the disabled attribute disables and enables the button. However, the value of the attribute is irrelevant, which is why you cannot enable a button by writing <button disabled="false">Still Disabled</button>.

To control the state of the button, set the disabled property,

**Note:** Though you could technically set the [attr.disabled] attribute binding, the values are different in that the property binding requires to a boolean value, while its corresponding attribute binding relies on whether the value is null or not. Consider the following:

**<input [disabled]="condition ? true : false">**

**<input [attr.disabled]="condition ? 'disabled' : null">**

Generally, use property binding over attribute binding as it is more intuitive (being a boolean value), has a shorter syntax, and is more performant.

**The HTML attribute and the DOM property are different things, even when they have the same name.**

**Template binding works with**properties**and**events**, not**attributes**.**

To see the disabled button example in a functioning app, see the [live example](https://angular.io/generated/live-examples/binding-syntax/stackblitz.html) / [download example](https://angular.io/generated/zips/binding-syntax/binding-syntax.zip) especially for binding syntax. This example shows you how to toggle the disabled property from the component.

### **Angular and attributes**

In Angular, the only role of attributes is to initialize element and directive state. When you write a data-binding, you're dealing exclusively with properties and events of the target object.

## Binding targets

The **target of a data-binding** is something in the DOM. Depending on the binding type, the target can be a property (element, component, or directive), an event (element, component, or directive), or sometimes an attribute name. The following table summarizes:

|  |  |  |
| --- | --- | --- |
| **Type** | **Target** | **Examples** |
| Property | Element property Component property Directive property | src, hero, and [ngClass](https://angular.io/api/common/NgClass) in the following:  <img [src]="heroImageUrl">  <app-hero-detail [hero]="currentHero"></app-hero-detail>  <div [[ngClass](https://angular.io/api/common/NgClass)]="{'special': isSpecial}"></div> |
| Event | Element event Component event Directive event | click, deleteRequest, and myClick in the following:  <button (click)="onSave()">Save</button>  <app-hero-detail (deleteRequest)="deleteHero()"></app-hero-detail>  <div (myClick)="clicked=$event" clickable>click me</div> |
| Two-way | Event and property | <input [([ngModel](https://angular.io/api/forms/NgModel))]="name"> |
| Attribute | Attribute (the exception) | <button [attr.aria-label]="help">help</button> |
| Class | class property | <div [class.special]="isSpecial">Special</div> |
| Style | [style](https://angular.io/api/animations/style) property | <button [style.color]="isSpecial ? 'red' : 'green'"> |

## Property binding [property]

Use property binding to set properties of target elements or directive @[Input](https://angular.io/api/core/Input)() decorators. For an example demonstrating all of the points in this section, see the [property binding example](https://angular.io/generated/live-examples/property-binding/stackblitz.html) / [download example](https://angular.io/generated/zips/property-binding/property-binding.zip).

### **One-way in**

Property binding flows a value in one direction, from a component's property into a target element property.

You can't use property binding to read or pull values out of target elements. Similarly, you cannot use property binding to call a method on the target element. If the element raises events, you can listen to them with an [event binding](https://angular.io/guide/template-syntax#event-binding).

If you must read a target element property or call one of its methods, see the API reference for [ViewChild](https://angular.io/api/core/ViewChild) and [ContentChild](https://angular.io/api/core/ContentChild).

### **Examples**

The most common property binding sets an element property to a component property value. An example is binding the src property of an image element to a component's itemImageUrl property:

src/app/app.component.html

**<img [src]="itemImageUrl">**

Here's an example of binding to the colSpan property. Notice that it's not colspan, which is the attribute, spelled with a lowercase s.

src/app/app.component.html

**<!-- Notice the colSpan property is camel case -->**

**<tr><td [colSpan]="2">Span 2 columns</td></tr>**

For more details, see the [MDN HTMLTableCellElment](https://developer.mozilla.org/en-US/docs/Web/API/HTMLTableCellElement) documentation.

Another example is disabling a button when the component says that it isUnchanged:

src/app/app.component.html

**<!-- Bind button disabled** [**state**](https://angular.io/api/animations/state) **to `isUnchanged` property -->**

**<button [disabled]="isUnchanged">Disabled Button</button>**

Another is setting a property of a directive:

src/app/app.component.html

**<p [**[**ngClass**](https://angular.io/api/common/NgClass)**]="**[**classes**](https://angular.io/api/core/DebugElement#classes)**">[**[**ngClass**](https://angular.io/api/common/NgClass)**] binding to the** [**classes**](https://angular.io/api/core/DebugElement#classes) **property making this blue</p>**

Yet another is setting the model property of a custom component—a great way for parent and child components to communicate:

src/app/app.component.html

**<app-item-detail [childItem]="parentItem"></app-item-detail>**

### **Binding target**

An element property between enclosing square brackets identifies the target property. The target property in the following code is the image element's src property.

src/app/app.component.html

**<img [src]="itemImageUrl">**

There's also the bind- prefix alternative:

src/app/app.component.html

**<img bind-src="itemImageUrl">**

In most cases, the target name is the name of a property, even when it appears to be the name of an attribute. So in this case, src is the name of the <img> element property.

Element properties may be the more common targets, but Angular looks first to see if the name is a property of a known directive, as it is in the following example:

src/app/app.component.html

**<p [**[**ngClass**](https://angular.io/api/common/NgClass)**]="**[**classes**](https://angular.io/api/core/DebugElement#classes)**">[**[**ngClass**](https://angular.io/api/common/NgClass)**] binding to the** [**classes**](https://angular.io/api/core/DebugElement#classes) **property making this blue</p>**

Technically, Angular is matching the name to a directive @[Input](https://angular.io/api/core/Input)(), one of the property names listed in the directive's inputs array or a property decorated with @[Input](https://angular.io/api/core/Input)(). Such inputs map to the directive's own properties.

If the name fails to match a property of a known directive or element, Angular reports an “unknown directive” error.

Though the target name is usually the name of a property, there is an automatic attribute-to-property mapping in Angular for several common attributes. These include class/className, innerHtml/innerHTML, and tabindex/tabIndex.

### **Avoid side effects**

Evaluation of a template expression should have no visible side effects. The expression language itself, or the way you write template expressions, helps to a certain extent; you can't assign a value to anything in a property binding expression nor use the increment and decrement operators.

For example, you could have an expression that invoked a property or method that had side effects. The expression could call something like getFoo() where only you know what getFoo() does. If getFoo() changes something and you happen to be binding to that something, Angular may or may not display the changed value. Angular may detect the change and throw a warning error. As a best practice, stick to properties and to methods that return values and avoid side effects.

### **Return the proper type**

The template expression should evaluate to the type of value that the target property expects. Return a string if the target property expects a string, a number if it expects a number, an object if it expects an object, and so on.

In the following example, the childItem property of the ItemDetailComponent expects a string, which is exactly what you're sending in the property binding:

src/app/app.component.html

**<app-item-detail [childItem]="parentItem"></app-item-detail>**

You can confirm this by looking in the ItemDetailComponent where the @[Input](https://angular.io/api/core/Input) type is set to a string:src/app/item-detail/item-detail.component.ts (setting the @Input() type

**@**[**Input**](https://angular.io/api/core/Input)**() childItem: string;**

As you can see here, the parentItem in AppComponent is a string, which the ItemDetailComponent expects:src/app/app.component.ts

**parentItem = 'lamp';**

#### **Passing in an object**

The previous simple example showed passing in a string. To pass in an object, the syntax and thinking are the same.

In this scenario, ListItemComponent is nested within AppComponent and the [item](https://angular.io/api/core/IterableChangeRecord#item) property expects an object.

src/app/app.component.html

<app-list-item [items]="currentItem"></app-list-item>

The [item](https://angular.io/api/core/IterableChangeRecord#item) property is declared in the ListItemComponent with a type of Item and decorated with @[Input](https://angular.io/api/core/Input)():

src/app/list-item.component.ts

@[Input](https://angular.io/api/core/Input)() items: Item[];

In this sample app, an Item is an object that has two properties; an id and a name.

src/app/item.ts

export class Item {

id: number;

name: string;

}

While a list of items exists in another file, mock-items.ts, you can specify a different item in app.component.ts so that the new item will render:

src/app.component.ts

currentItem = [{

id: 21,

name: 'phone'

}];

You just have to make sure, in this case, that you're supplying an object because that's the type of [item](https://angular.io/api/core/IterableChangeRecord#item) and is what the nested component, ListItemComponent, expects.

In this example, AppComponent specifies a different [item](https://angular.io/api/core/IterableChangeRecord#item) object (currentItem) and passes it to the nested ListItemComponent. ListItemComponent was able to use currentItem because it matches what an Item object is according to item.ts. The item.ts file is where ListItemComponent gets its definition of an [item](https://angular.io/api/core/IterableChangeRecord#item).

### **Remember the brackets**

The brackets, [], tell Angular to evaluate the template expression. If you omit the brackets, Angular treats the string as a constant and initializes the target property with that string:

src/app.component.html

<app-item-detail childItem="parentItem"></app-item-detail>

Omitting the brackets will render the string parentItem, not the value of parentItem.

### **One-time string initialization**

You should omit the brackets when all of the following are true:

* The target property accepts a string value.
* The string is a fixed value that you can put directly into the template.
* This initial value never changes.

You routinely initialize attributes this way in standard HTML, and it works just as well for directive and component property initialization. The following example initializes the prefix property of the StringInitComponent to a fixed string, not a template expression. Angular sets it and forgets about it.

src/app/app.component.html

<app-string-init prefix="This is [a](https://angular.io/api/router/RouterLinkWithHref) one-time initialized string."></app-string-init>

The [[item](https://angular.io/api/core/IterableChangeRecord#item)] binding, on the other hand, remains a live binding to the component's currentItem property.

### **Property binding vs. interpolation**

You often have a choice between interpolation and property binding. The following binding pairs do the same thing:

src/app/app.component.html

<p><img src="{{itemImageUrl}}"> is the <i>interpolated</i> image.</p>

<p><img [src]="itemImageUrl"> is the <i>property bound</i> image.</p>

<p><span>"{{interpolationTitle}}" is the <i>interpolated</i> title.</span></p>

<p>"<span [innerHTML]="propertyTitle"></span>" is the <i>property bound</i> title.</p>

Interpolation is a convenient alternative to property binding in many cases. When rendering data values as strings, there is no technical reason to prefer one form to the other, though readability tends to favor interpolation. However, when setting an element property to a non-string data value, you must use property binding.

### **Content security**

Imagine the following malicious content.

src/app/app.component.ts

evilTitle = '[Template](https://angular.io/) <script>alert("evil never sleeps")</script> Syntax';

In the component template, the content might be used with interpolation:

src/app/app.component.ts

<p><span>"{{evilTitle}}" is the <i>interpolated</i> evil title.</span></p>

Fortunately, Angular data binding is on alert for dangerous HTML. In the above case, the HTML displays as is, and the Javascript does not execute. Angular **does not** allow HTML with script tags to leak into the browser, neither with interpolation nor property binding.

In the following example, however, Angular [sanitizes](https://angular.io/guide/security#sanitization-and-security-contexts) the values before displaying them.

src/app/app.component.html

<!--

Angular generates [a](https://angular.io/api/router/RouterLinkWithHref) warning for the following line as it sanitizes them

WARNING: sanitizing [HTML](https://angular.io/api/core/SecurityContext#HTML) stripped some content (see http://g.co/ng/security#xss).

-->

<p>"<span [innerHTML]="evilTitle"></span>" is the <i>property bound</i> evil title.</p>

Interpolation handles the <script> tags differently than property binding but both approaches render the content harmlessly. The following is the browser output of the evilTitle examples.

"[Template](https://angular.io/) <script>alert("evil never sleeps")</script> Syntax" is the interpolated evil title.

"[Template](https://angular.io/) alert("evil never sleeps")Syntax" is the property bound evil title.

## Attribute, class, and style bindings

The template syntax provides specialized one-way bindings for scenarios less well-suited to property binding.

To see attribute, class, and style bindings in a functioning app, see the [live example](https://angular.io/generated/live-examples/attribute-binding/stackblitz.html) / [download example](https://angular.io/generated/zips/attribute-binding/attribute-binding.zip) especially for this section.

### **Attribute binding**

Set the value of an attribute directly with an **attribute binding**. This is the only exception to the rule that a binding sets a target property and the only binding that creates and sets an attribute.

Usually, setting an element property with a [property binding](https://angular.io/guide/template-syntax#property-binding) is preferable to setting the attribute with a string. However, sometimes there is no element property to bind, so attribute binding is the solution.

Consider the [ARIA](https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA) and [SVG](https://developer.mozilla.org/en-US/docs/Web/SVG). They are purely attributes, don't correspond to element properties, and don't set element properties. In these cases, there are no property targets to bind to.

Attribute binding syntax resembles property binding, but instead of an element property between brackets, start with the prefix attr, followed by a dot (.), and the name of the attribute. You then set the attribute value, using an expression that resolves to a string, or remove the attribute when the expression resolves to null.

One of the primary use cases for attribute binding is to set ARIA attributes, as in this example:

src/app/app.component.html

<!-- create and set an aria attribute for assistive technology -->

<button [attr.aria-label]="actionName">{{actionName}} with Aria</button>

#### **colspan and colSpan**

Notice the difference between the colspan attribute and the colSpan property.

If you wrote something like this:

<tr><td colspan="{{1 + 1}}">Three-Four</td></tr>

You'd get this error:

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

Can't bind to 'colspan' since it isn't [a](https://angular.io/api/router/RouterLinkWithHref) known [native](https://angular.io/) property

As the message says, the <td> element does not have a colspan property. This is true because colspan is an attribute—colSpan, with a capital S, is the corresponding property. Interpolation and property binding can set only properties, not attributes.

Instead, you'd use property binding and write it like this:

src/app/app.component.html

<!-- Notice the colSpan property is camel case -->

<tr><td [colSpan]="1 + 1">Three-Four</td></tr>

### **Class binding**

Add and remove CSS class names from an element's class attribute with a **class binding**.

Here's how to set the attribute without binding in plain HTML:

<!-- standard class attribute setting -->

<div class="[item](https://angular.io/api/core/IterableChangeRecord#item) clearance special">Item clearance special</div>

Class binding syntax resembles property binding, but instead of an element property between brackets, start with the prefix class, optionally followed by a dot (.) and the name of a CSS class: [class.class-name].

You can replace that with a binding to a string of the desired class names; this is an all-or-nothing, replacement binding.

src/app/app.component.html

<h3>Overwrite all existing [classes](https://angular.io/api/core/DebugElement#classes) with [a](https://angular.io/api/router/RouterLinkWithHref) new class:</h3>

<div class="[item](https://angular.io/api/core/IterableChangeRecord#item) clearance special" [attr.class]="resetClasses">Reset all [classes](https://angular.io/api/core/DebugElement#classes) at once</div>

You can also add append a class to an element without overwriting the classes already on the element:

src/app/app.component.html

<h3>Add [a](https://angular.io/api/router/RouterLinkWithHref) class:</h3>

<div class="[item](https://angular.io/api/core/IterableChangeRecord#item) clearance special" [class.item-clearance]="itemClearance">Add another class</div>

Finally, you can bind to a specific class name. Angular adds the class when the template expression evaluates to truthy. It removes the class when the expression is falsy.

src/app/app.component.html

<h3>toggle the "special" class on/off with [a](https://angular.io/api/router/RouterLinkWithHref) property:</h3>

<div [class.special]="isSpecial">The class binding is special.</div>

<h3>binding to class.special [overrides](https://angular.io/api/platform-browser/HammerGestureConfig#overrides) the class attribute:</h3>

<div class="special" [class.special]="!isSpecial">This one is not so special.</div>

<h3>Using the bind- syntax:</h3>

<div bind-class.special="isSpecial">This class binding is special too.</div>

While this technique is suitable for toggling a single class name, consider the [NgClass](https://angular.io/guide/template-syntax#ngClass) directive when managing multiple class names at the same time.

### **Style binding**

You can set inline styles with a **style binding**.

Style binding syntax resembles property binding. Instead of an element property between brackets, start with the prefix [style](https://angular.io/api/animations/style), followed by a dot (.) and the name of a CSS style property: [style.style-property].

src/app/app.component.html

<button [style.color]="isSpecial ? 'red': 'green'">Red</button>

<button [style.background-color]="canSave ? 'cyan': 'grey'" >Save</button>

Some style binding styles have a unit extension. The following example conditionally sets the font size in “em” and “%” units .

src/app/app.component.html

<button [style.font-size.em]="isSpecial ? 3 : 1" >Big</button>

<button [style.font-size.%]="!isSpecial ? 150 : 50" >Small</button>

**This technique is suitable for setting a single style, but consider the**[NgStyle](https://angular.io/guide/template-syntax#ngStyle)**directive when setting several inline styles at the same time.**

Note that a style property name can be written in either [dash-case](https://angular.io/guide/glossary#dash-case), as shown above, or [camelCase](https://angular.io/guide/glossary#camelcase), such as fontSize.

## Event binding (event)

Event binding allows you to listen for certain events such as keystrokes, mouse movements, clicks, and touches. For an example demonstrating all of the points in this section, see the [event binding example](https://angular.io/generated/live-examples/event-binding/stackblitz.html) / [download example](https://angular.io/generated/zips/event-binding/event-binding.zip).

Angular event binding syntax consists of a **target event** name within parentheses on the left of an equal sign, and a quoted template statement on the right. The following event binding listens for the button's click events, calling the component's onSave() method whenever a click occurs:

### **Target event**

As above, the target is the button's click event.

src/app/app.component.html

<button (click)="onSave($event)">Save</button>

Alternatively, use the on- prefix, known as the canonical form:

src/app/app.component.html

<button on-click="onSave($event)">on-click Save</button>

Element events may be the more common targets, but Angular looks first to see if the name matches an event property of a known directive, as it does in the following example:

src/app/app.component.html

<h4>myClick is an event on the custom ClickDirective:</h4>

<button (myClick)="clickMessage=$event" clickable>click with myClick</button>

{{clickMessage}}

If the name fails to match an element event or an output property of a known directive, Angular reports an “unknown directive” error.

### **$event and event handling statements**

In an event binding, Angular sets up an event handler for the target event.

When the event is raised, the handler executes the template statement. The template statement typically involves a receiver, which performs an action in response to the event, such as storing a value from the HTML control into a model.

The binding conveys information about the event. This information can include data values such as an event object, string, or number named $event.

The target event determines the shape of the $event object. If the target event is a native DOM element event, then $event is a [DOM event object](https://developer.mozilla.org/en-US/docs/Web/Events), with properties such as [target](https://angular.io/api/router/RouterLinkWithHref#target) and target.value.

Consider this example:

src/app/app.component.html

<input [value]="currentItem.name"

(input)="currentItem.name=$event.target.value" >

without [NgModel](https://angular.io/api/forms/NgModel)

This code sets the <input> value property by binding to the name property. To listen for changes to the value, the code binds to the input event of the <input> element. When the user makes changes, the input event is raised, and the binding executes the statement within a context that includes the DOM event object, $event.

To update the name property, the changed text is retrieved by following the path $event.target.value.

If the event belongs to a directive—recall that components are directives—$event has whatever shape the directive produces.

### **Custom events with**[**EventEmitter**](https://angular.io/api/core/EventEmitter)

Directives typically raise custom events with an Angular [EventEmitter](https://angular.io/api/core/EventEmitter). The directive creates an [EventEmitter](https://angular.io/api/core/EventEmitter) and exposes it as a property. The directive calls EventEmitter.emit(payload) to fire an event, passing in a message payload, which can be anything. Parent directives listen for the event by binding to this property and accessing the payload through the $event object.

Consider an ItemDetailComponent that presents item information and responds to user actions. Although the ItemDetailComponent has a delete button, it doesn't know how to delete the hero. It can only raise an event reporting the user's delete request.

Here are the pertinent excerpts from that ItemDetailComponent:

src/app/item-detail/item-detail.component.html (template)

<img src="{{itemImageUrl}}" [style.display]="displayNone">

<span [style.text-decoration]="lineThrough">{{ item.name }}

</span>

<button (click)="delete()">Delete</button>

src/app/item-detail/item-detail.component.ts (deleteRequest)

// This component makes [a](https://angular.io/api/router/RouterLinkWithHref) [request](https://angular.io/api/common/http/testing/TestRequest#request) but it can't actually delete [a](https://angular.io/api/router/RouterLinkWithHref) hero.

@[Output](https://angular.io/api/core/Output)() deleteRequest = new [EventEmitter](https://angular.io/api/core/EventEmitter)<Item>();

delete() {

this.deleteRequest.emit(this.item);

this.displayNone = this.displayNone ? '' : 'none';

this.lineThrough = this.lineThrough ? '' : 'line-through';

}

The component defines a deleteRequest property that returns an [EventEmitter](https://angular.io/api/core/EventEmitter). When the user clicks delete, the component invokes the delete() method, telling the [EventEmitter](https://angular.io/api/core/EventEmitter) to emit an Item object.

Now imagine a hosting parent component that binds to the deleteRequest event of the ItemDetailComponent.

src/app/app.component.html (event-binding-to-component)

<app-item-detail (deleteRequest)="deleteItem($event)" [[item](https://angular.io/api/core/IterableChangeRecord#item)]="currentItem"></app-item-detail>

When the deleteRequest event fires, Angular calls the parent component's deleteItem() method, passing the item-to-delete (emitted by ItemDetail) in the $eventvariable.

### **Template statements have side effects**

Though [template expressions](https://angular.io/guide/template-syntax#template-expressions) shouldn't have [side effects](https://angular.io/guide/template-syntax#avoid-side-effects), template statements usually do. The deleteItem() method does have a side effect: it deletes an item.

Deleting an item updates the model, and depending on your code, triggers other changes including queries and saving to a remote server. These changes propagate through the system and ultimately display in this and other views.

## Two-way binding [(...)]

Two-way binding gives your app a way to share data between a component class and its template.

For a demonstration of the syntax and code snippets in this section, see the [two-way binding example](https://angular.io/generated/live-examples/two-way-binding/stackblitz.html) / [download example](https://angular.io/generated/zips/two-way-binding/two-way-binding.zip).

### **Basics of two-way binding**

Two-way binding does two things:

1. Sets a specific element property.
2. Listens for an element change event.

Angular offers a special two-way data binding syntax for this purpose, [()]. The [()] syntax combines the brackets of property binding, [], with the parentheses of event binding, ().

[( )] = BANANA IN A BOX

Visualize a banana in a box to remember that the parentheses go inside the brackets.

The [()] syntax is easy to demonstrate when the element has a settable property called x and a corresponding event named xChange. Here's a SizerComponent that fits this pattern. It has a size value property and a companion sizeChange event:

src/app/sizer.component.ts

import { [Component](https://angular.io/api/core/Component), [Input](https://angular.io/api/core/Input), [Output](https://angular.io/api/core/Output), [EventEmitter](https://angular.io/api/core/EventEmitter) } from '@angular/core';

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

selector: 'app-sizer',

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

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

})

export class SizerComponent {

@[Input](https://angular.io/api/core/Input)() size: number | string;

@[Output](https://angular.io/api/core/Output)() sizeChange = new [EventEmitter](https://angular.io/api/core/EventEmitter)<number>();

dec() { this.resize(-1); }

inc() { this.resize(+1); }

resize(delta: number) {

this.size = Math.min(40, Math.max(8, +this.size + delta));

this.sizeChange.emit(this.size);

}

}

The initial size is an input value from a property binding. Clicking the buttons increases or decreases the size, within min/max value constraints, and then raises, or emits, the sizeChange event with the adjusted size.

Here's an example in which the AppComponent.fontSizePx is two-way bound to the SizerComponent:

src/app/app.component.html (two-way-1)

<app-sizer [(size)]="fontSizePx"></app-sizer>

<div [style.font-size.px]="fontSizePx">Resizable Text</div>

The AppComponent.fontSizePx establishes the initial SizerComponent.size value.

src/app/app.component.ts

fontSizePx = 16;

Clicking the buttons updates the AppComponent.fontSizePx via the two-way binding. The revised AppComponent.fontSizePx value flows through to the style binding, making the displayed text bigger or smaller.

The two-way binding syntax is really just syntactic sugar for a property binding and an event binding. Angular desugars the SizerComponent binding into this:

src/app/app.component.html (two-way-2)

<app-sizer [size]="fontSizePx" (sizeChange)="fontSizePx=$event"></app-sizer>

The $event variable contains the payload of the SizerComponent.sizeChange event. Angular assigns the $event value to the AppComponent.fontSizePx when the user clicks the buttons.

## Two-way binding in forms

The two-way binding syntax is a great convenience compared to separate property and event bindings. It would be convenient to use two-way binding with HTML form elements like <input> and <select>. However, no native HTML element follows the x value and xChange event pattern.

For more on how to use two-way binding in forms, see Angular [NgModel](https://angular.io/guide/template-syntax#ngModel).

## Built-in directives

Angular offers two kinds of built-in directives: attribute directives and structural directives. This segment reviews some of the most common built-in directives, classified as either [attribute directives](https://angular.io/guide/template-syntax#attribute-directives) or [structural directives](https://angular.io/guide/template-syntax#structural-directives) and has its own [built-in directives example](https://angular.io/generated/live-examples/built-in-directives/stackblitz.html) / [download example](https://angular.io/generated/zips/built-in-directives/built-in-directives.zip).

For more detail, including how to build your own custom directives, see [Attribute Directives](https://angular.io/guide/attribute-directives) and [Structural Directives](https://angular.io/guide/structural-directives).

### **Built-in attribute directives**

Attribute directives listen to and modify the behavior of other HTML elements, attributes, properties, and components. You usually apply them to elements as if they were HTML attributes, hence the name.

Many NgModules such as the [RouterModule](https://angular.io/guide/router) and the [FormsModule](https://angular.io/guide/forms) define their own attribute directives. The most common attribute directives are as follows:

* [NgClass](https://angular.io/guide/template-syntax#ngClass)—adds and removes a set of CSS classes.
* [NgStyle](https://angular.io/guide/template-syntax#ngStyle)—adds and removes a set of HTML styles.
* [NgModel](https://angular.io/guide/template-syntax#ngModel)—adds two-way data binding to an HTML form element.

### [**NgClass**](https://angular.io/api/common/NgClass)

Add or remove several CSS classes simultaneously with [ngClass](https://angular.io/api/common/NgClass).

src/app/app.component.html

<!-- toggle the "special" class on/off with [a](https://angular.io/api/router/RouterLinkWithHref) property -->

<div [[ngClass](https://angular.io/api/common/NgClass)]="isSpecial ? 'special' : ''">This div is special</div>

To add or remove a single class, use [class binding](https://angular.io/guide/template-syntax#class-binding) rather than [NgClass](https://angular.io/api/common/NgClass).

Consider a setCurrentClasses() component method that sets a component property, currentClasses, with an object that adds or removes three classes based on thetrue/false state of three other component properties. Each key of the object is a CSS class name; its value is true if the class should be added, false if it should be removed.

src/app/app.component.ts

currentClasses: {};

setCurrentClasses() {

// CSS [classes](https://angular.io/api/core/DebugElement#classes): added/removed per current [state](https://angular.io/api/animations/state) of component [properties](https://angular.io/api/core/DebugElement#properties)

this.currentClasses = {

'saveable': this.canSave,

'modified': !this.isUnchanged,

'special': this.isSpecial

};

}

Adding an [ngClass](https://angular.io/api/common/NgClass) property binding to currentClasses sets the element's classes accordingly:

src/app/app.component.html

<div [[ngClass](https://angular.io/api/common/NgClass)]="currentClasses">This div is initially saveable, unchanged, and special.</div>

Remember that in this situation you'd call setCurrentClasses(), both initially and when the dependent properties change.

### [**NgStyle**](https://angular.io/api/common/NgStyle)

Use [NgStyle](https://angular.io/api/common/NgStyle) to set many inline styles simultaneously and dynamically, based on the state of the component.

#### **Without**[**NgStyle**](https://angular.io/api/common/NgStyle)

For context, consider setting a single style value with [style binding](https://angular.io/guide/template-syntax#style-binding), without [NgStyle](https://angular.io/api/common/NgStyle).

src/app/app.component.html

<div [style.font-size]="isSpecial ? 'x-large' : 'smaller'">

This div is x-large or smaller.

</div>

However, to set many inline styles at the same time, use the [NgStyle](https://angular.io/api/common/NgStyle) directive.

The following is a setCurrentStyles() method that sets a component property, currentStyles, with an object that defines three styles, based on the state of three other component properties:

src/app/app.component.ts

currentStyles: {};

setCurrentStyles() {

// CSS styles: set per current [state](https://angular.io/api/animations/state) of component [properties](https://angular.io/api/core/DebugElement#properties)

this.currentStyles = {

'font-style': this.canSave ? 'italic' : 'normal',

'font-weight': !this.isUnchanged ? 'bold' : 'normal',

'font-size': this.isSpecial ? '24px' : '12px'

};

}

Adding an [ngStyle](https://angular.io/api/common/NgStyle) property binding to currentStyles sets the element's styles accordingly:

src/app/app.component.html

<div [[ngStyle](https://angular.io/api/common/NgStyle)]="currentStyles">

This div is initially italic, normal weight, and extra large (24px).

</div>

Remember to call setCurrentStyles(), both initially and when the dependent properties change.

### **[(**[**ngModel**](https://angular.io/api/forms/NgModel)**)]: Two-way binding**

The [NgModel](https://angular.io/api/forms/NgModel) directive allows you to display a data property and update that property when the user makes changes. Here's an example:

src/app/app.component.html (NgModel example)

<label for="example-ngModel">[([ngModel](https://angular.io/api/forms/NgModel))]:</label>

<input [([ngModel](https://angular.io/api/forms/NgModel))]="currentItem.name" id="example-ngModel">

#### **Import**[**FormsModule**](https://angular.io/api/forms/FormsModule)**to use**[**ngModel**](https://angular.io/api/forms/NgModel)

Before using the [ngModel](https://angular.io/api/forms/NgModel) directive in a two-way data binding, you must import the [FormsModule](https://angular.io/api/forms/FormsModule) and add it to the NgModule's imports list. Learn more about the [FormsModule](https://angular.io/api/forms/FormsModule) and [ngModel](https://angular.io/api/forms/NgModel) in [Forms](https://angular.io/guide/forms#ngModel).

Remember to import the [FormsModule](https://angular.io/api/forms/FormsModule) to make [([ngModel](https://angular.io/api/forms/NgModel))] available as follows:

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

import { [FormsModule](https://angular.io/api/forms/FormsModule) } from '@angular/forms'; // <--- JavaScript import from Angular

/\* . . . \*/

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

/\* . . . \*/

imports: [

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

[FormsModule](https://angular.io/api/forms/FormsModule) // <--- import into the [NgModule](https://angular.io/api/core/NgModule)

],

/\* . . . \*/

})

export class AppModule { }

You could achieve the same result with separate bindings to the <input> element's value property and input event:

src/app/app.component.html

<label for="without">without [NgModel](https://angular.io/api/forms/NgModel):</label>

<input [value]="currentItem.name" (input)="currentItem.name=$event.target.value" id="without">

To streamline the syntax, the [ngModel](https://angular.io/api/forms/NgModel) directive hides the details behind its own [ngModel](https://angular.io/api/forms/NgModel) input and ngModelChange output properties:

src/app/app.component.html

<label for="example-change">(ngModelChange)="...name=$event":</label>

<input [[ngModel](https://angular.io/api/forms/NgModel)]="currentItem.name" (ngModelChange)="currentItem.name=$event" id="example-change">

The [ngModel](https://angular.io/api/forms/NgModel) data property sets the element's value property and the ngModelChange event property listens for changes to the element's value.

#### [**NgModel**](https://angular.io/api/forms/NgModel)**and value accessors**

The details are specific to each kind of element and therefore the [NgModel](https://angular.io/api/forms/NgModel) directive only works for an element supported by a [ControlValueAccessor](https://angular.io/api/forms/ControlValueAccessor) that adapts an element to this protocol. Angular provides value accessors for all of the basic HTML form elements and the [Forms](https://angular.io/guide/forms) guide shows how to bind to them.

You can't apply [([ngModel](https://angular.io/api/forms/NgModel))] to a non-form native element or a third-party custom component until you write a suitable value accessor. For more information, see the API documentation on [DefaultValueAccessor](https://angular.io/api/forms/DefaultValueAccessor).

You don't need a value accessor for an Angular component that you write because you can name the value and event properties to suit Angular's basic [two-way binding syntax](https://angular.io/guide/template-syntax#two-way) and skip [NgModel](https://angular.io/api/forms/NgModel) altogether. The sizer in the [Two-way Binding](https://angular.io/guide/template-syntax#two-way) section is an example of this technique.

Separate [ngModel](https://angular.io/api/forms/NgModel) bindings are an improvement over binding to the element's native properties, but you can streamline the binding with a single declaration using the [([ngModel](https://angular.io/api/forms/NgModel))] syntax:

src/app/app.component.html

<label for="example-ngModel">[([ngModel](https://angular.io/api/forms/NgModel))]:</label>

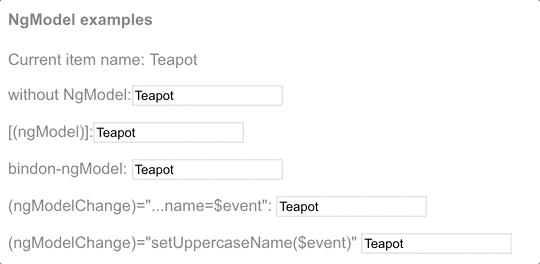
<input [([ngModel](https://angular.io/api/forms/NgModel))]="currentItem.name" id="example-ngModel">

This [([ngModel](https://angular.io/api/forms/NgModel))] syntax can only set a data-bound property. If you need to do something more, you can write the expanded form; for example, the following changes the <input> value to uppercase:

src/app/app.component.html

<input [[ngModel](https://angular.io/api/forms/NgModel)]="currentItem.name" (ngModelChange)="setUppercaseName($event)" id="example-uppercase">

Here are all variations in action, including the uppercase version:



## Built-in structural directives

Structural directives are responsible for HTML layout. They shape or reshape the DOM's structure, typically by adding, removing, and manipulating the host elements to which they are attached.

This section is an introduction to the common built-in structural directives:

* [NgIf](https://angular.io/guide/template-syntax#ngIf)—conditionally creates or destroys subviews from the template.
* [NgFor](https://angular.io/guide/template-syntax#ngFor)—repeat a node for each item in a list.
* [NgSwitch](https://angular.io/guide/template-syntax#ngSwitch)—a set of directives that switch among alternative views.

The deep details of structural directives are covered in the [Structural Directives](https://angular.io/guide/structural-directives) guide, which explains the following:

* Why you [prefix the directive name with an asterisk (\*)](https://angular.io/guide/structural-directives#the-asterisk--prefix).
* Using [<ng-container>](https://angular.io/guide/structural-directives#ngcontainer) to group elements when there is no suitable host element for the directive.
* How to write your own structural directive.
* That you can only apply [one structural directive](https://angular.io/guide/structural-directives#one-per-element) to an element.

### **NgIf**

You can add or remove an element from the DOM by applying an [NgIf](https://angular.io/api/common/NgIf) directive to a host element. Bind the directive to a condition expression like [isActive](https://angular.io/api/router/RouterLinkActive#isActive) in this example.

src/app/app.component.html

<app-item-detail \*[ngIf](https://angular.io/api/common/NgIf)="[isActive](https://angular.io/api/router/RouterLinkActive#isActive)" [[item](https://angular.io/api/core/IterableChangeRecord#item)]="[item](https://angular.io/api/core/IterableChangeRecord#item)"></app-item-detail>

Don't forget the asterisk (\*) in front of [ngIf](https://angular.io/api/common/NgIf). For more information on the asterisk, see the [asterisk (\*) prefix](https://angular.io/guide/structural-directives#the-asterisk--prefix) section of [Structural Directives](https://angular.io/guide/structural-directives).

When the [isActive](https://angular.io/api/router/RouterLinkActive#isActive) expression returns a truthy value, [NgIf](https://angular.io/api/common/NgIf) adds the ItemDetailComponent to the DOM. When the expression is falsy, [NgIf](https://angular.io/api/common/NgIf) removes the ItemDetailComponent from the DOM, destroying that component and all of its sub-components.

#### **Show/hide vs.**[**NgIf**](https://angular.io/api/common/NgIf)

Hiding an element is different from removing it with [NgIf](https://angular.io/api/common/NgIf). For comparison, the following example shows how to control the visibility of an element with a [class](https://angular.io/guide/template-syntax#class-binding) or [style](https://angular.io/guide/template-syntax#style-binding)binding.

src/app/app.component.html

<!-- isSpecial is true -->

<div [class.hidden]="!isSpecial">Show with class</div>

<div [class.hidden]="isSpecial">Hide with class</div>

<p>ItemDetail is in the DOM but hidden</p>

<app-item-detail [class.hidden]="isSpecial"></app-item-detail>

<div [style.display]="isSpecial ? 'block' : 'none'">Show with [style](https://angular.io/api/animations/style)</div>

<div [style.display]="isSpecial ? 'none' : 'block'">Hide with [style](https://angular.io/api/animations/style)</div>

When you hide an element, that element and all of its descendants remain in the DOM. All components for those elements stay in memory and Angular may continue to check for changes. You could be holding onto considerable computing resources and degrading performance unnecessarily.

[NgIf](https://angular.io/api/common/NgIf) works differently. When [NgIf](https://angular.io/api/common/NgIf) is false, Angular removes the element and its descendants from the DOM. It destroys their components, freeing up resources, which results in a better user experience.

If you are hiding large component trees, consider [NgIf](https://angular.io/api/common/NgIf) as a more efficient alternative to showing/hiding.

**Note:** For more information on [NgIf](https://angular.io/api/common/NgIf) and [ngIfElse](https://angular.io/api/common/NgIf#ngIfElse), see the [API documentation about NgIf](https://angular.io/api/common/NgIf).

#### **Guard against null**

Another advantage of [ngIf](https://angular.io/api/common/NgIf) is that you can use it to guard against null. Show/hide is best suited for very simple use cases, so when you need a guard, opt instead for [ngIf](https://angular.io/api/common/NgIf). Angular will throw an error if a nested expression tries to access a property of null.

The following shows [NgIf](https://angular.io/api/common/NgIf) guarding two <div>s. The currentCustomer name appears only when there is a currentCustomer. The nullCustomer will not be displayed as long as it is null.

src/app/app.component.html

<div \*[ngIf](https://angular.io/api/common/NgIf)="currentCustomer">Hello, {{currentCustomer.name}}</div>

src/app/app.component.html

<div \*[ngIf](https://angular.io/api/common/NgIf)="nullCustomer">Hello, <span>{{nullCustomer}}</span></div>

See also the [safe navigation operator](https://angular.io/guide/template-syntax#safe-navigation-operator) below.

### **NgFor**

NgFor is a repeater directive—a way to present a list of items. You define a block of HTML that defines how a single item should be displayed and then you tell Angular to use that block as a template for rendering each item in the list.

Here is an example of NgFor applied to a simple <div>:

src/app/app.component.html

<div \*[ngFor](https://angular.io/api/common/NgForOf)="let [item](https://angular.io/api/core/IterableChangeRecord#item) of items">{{item.name}}</div>

You can also apply an NgFor to a component element, as in this example:

src/app/app.component.html

<app-item-detail \*[ngFor](https://angular.io/api/common/NgForOf)="let [item](https://angular.io/api/core/IterableChangeRecord#item) of items" [[item](https://angular.io/api/core/IterableChangeRecord#item)]="[item](https://angular.io/api/core/IterableChangeRecord#item)"></app-item-detail>

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

The text assigned to \*[ngFor](https://angular.io/api/common/NgForOf) is the instruction that guides the repeater process.

#### **\***[**ngFor**](https://angular.io/api/common/NgForOf)**microsyntax**

The string assigned to \*[ngFor](https://angular.io/api/common/NgForOf) is not a [template expression](https://angular.io/guide/template-syntax#template-expressions). Rather, it's a microsyntax—a little language of its own that Angular interprets. The string "let [item](https://angular.io/api/core/IterableChangeRecord#item) of items"means:

Take each item in the *items* array, store it in the local [*item*](https://angular.io/api/core/IterableChangeRecord#item) looping variable, and make it available to the templated HTML for each iteration.

Angular translates this instruction into an <ng-template> around the host element, then uses this template repeatedly to create a new set of elements and bindings for each [item](https://angular.io/api/core/IterableChangeRecord#item) in the list.

For more information about microsyntax, see the [Structural Directives](https://angular.io/guide/structural-directives#microsyntax) guide.

#### **Template input variables**

The let keyword before [item](https://angular.io/api/core/IterableChangeRecord#item) creates a template input variable called [item](https://angular.io/api/core/IterableChangeRecord#item). The [ngFor](https://angular.io/api/common/NgForOf) directive iterates over the items array returned by the parent component's itemsproperty and sets [item](https://angular.io/api/core/IterableChangeRecord#item) to the current item from the array during each iteration.

Reference [item](https://angular.io/api/core/IterableChangeRecord#item) within the [ngFor](https://angular.io/api/common/NgForOf) host element as well as within its descendants to access the item's properties. The following example references [item](https://angular.io/api/core/IterableChangeRecord#item) first in an interpolation and then passes in a binding to the [item](https://angular.io/api/core/IterableChangeRecord#item) property of the <app-item-detail> component.

src/app/app.component.html

<div \*[ngFor](https://angular.io/api/common/NgForOf)="let [item](https://angular.io/api/core/IterableChangeRecord#item) of items">{{item.name}}</div>

<!-- . . . -->

<app-item-detail \*[ngFor](https://angular.io/api/common/NgForOf)="let [item](https://angular.io/api/core/IterableChangeRecord#item) of items" [[item](https://angular.io/api/core/IterableChangeRecord#item)]="[item](https://angular.io/api/core/IterableChangeRecord#item)"></app-item-detail>

For more information about template input variables, see [Structural Directives](https://angular.io/guide/structural-directives#template-input-variable).

#### **\***[**ngFor**](https://angular.io/api/common/NgForOf)**with index**

The index property of the NgFor directive context returns the zero-based index of the item in each iteration. You can capture the index in a template input variable and use it in the template.

The next example captures the index in a variable named i and displays it with the item name.

src/app/app.component.html

<div \*[ngFor](https://angular.io/api/common/NgForOf)="let [item](https://angular.io/api/core/IterableChangeRecord#item) of items; let i=index">{{i + 1}} - {{item.name}}</div>

NgFor is implemented by the [NgForOf](https://angular.io/api/common/NgForOf) directive. Read more about the other [NgForOf](https://angular.io/api/common/NgForOf) context values such as last, [even](https://angular.io/api/common/NgForOfContext#even), and [odd](https://angular.io/api/common/NgForOfContext#odd) in the [NgForOf API reference](https://angular.io/api/common/NgForOf).

#### **\*ngFor with trackBy**

If you use NgFor with large lists, a small change to one item, such as removing or adding an item, can trigger a cascade of DOM manipulations. For example, re-querying the server could reset a list with all new item objects, even when those items were previously displayed. In this case, Angular sees only a fresh list of new object references and has no choice but to replace the old DOM elements with all new DOM elements.

You can make this more efficient with trackBy. Add a method to the component that returns the value NgFor should track. In this case, that value is the hero's id. If the idhas already been rendered, Angular keeps track of it and doesn't re-query the server for the same id.

src/app/app.component.ts

trackByItems(index: number, [item](https://angular.io/api/core/IterableChangeRecord#item): Item): number { return item.id; }

In the microsyntax expression, set trackBy to the trackByItems() method.

src/app/app.component.html

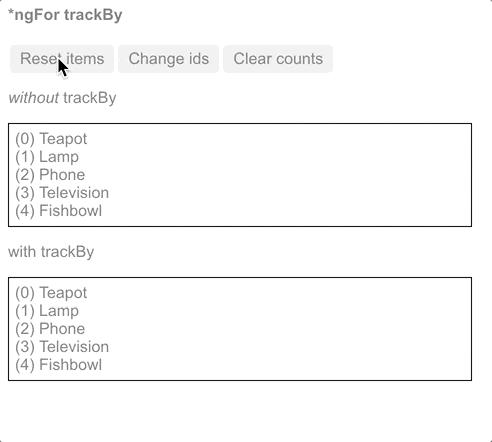
<div \*[ngFor](https://angular.io/api/common/NgForOf)="let [item](https://angular.io/api/core/IterableChangeRecord#item) of items; trackBy: trackByItems">

({{item.id}}) {{item.name}}

</div>

Here is an illustration of the trackBy effect. "Reset items" creates new items with the same item.ids. "Change ids" creates new items with new item.ids.

* With no trackBy, both buttons trigger complete DOM element replacement.
* With trackBy, only changing the id triggers element replacement.



Built-in directives use only public APIs; that is, they do not have special access to any private APIs that other directives can't access.

## The [NgSwitch](https://angular.io/api/common/NgSwitch) directives

NgSwitch is like the JavaScript switch statement. It displays one element from among several possible elements, based on a switch condition. Angular puts only the selected element into the DOM.

[NgSwitch](https://angular.io/api/common/NgSwitch) is actually a set of three, cooperating directives: [NgSwitch](https://angular.io/api/common/NgSwitch), [NgSwitchCase](https://angular.io/api/common/NgSwitchCase), and [NgSwitchDefault](https://angular.io/api/common/NgSwitchDefault) as in the following example.

src/app/app.component.html

<div [[ngSwitch](https://angular.io/api/common/NgSwitch)]="currentItem.feature">

<app-stout-item \*[ngSwitchCase](https://angular.io/api/common/NgSwitchCase)="'stout'" [[item](https://angular.io/api/core/IterableChangeRecord#item)]="currentItem"></app-stout-item>

<app-device-item \*[ngSwitchCase](https://angular.io/api/common/NgSwitchCase)="'slim'" [[item](https://angular.io/api/core/IterableChangeRecord#item)]="currentItem"></app-device-item>

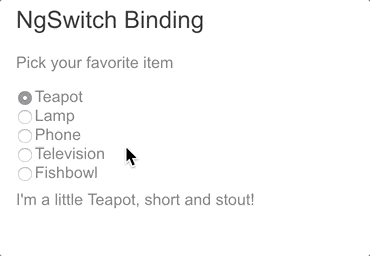
<app-lost-item \*[ngSwitchCase](https://angular.io/api/common/NgSwitchCase)="'vintage'" [[item](https://angular.io/api/core/IterableChangeRecord#item)]="currentItem"></app-lost-item>

<app-best-item \*[ngSwitchCase](https://angular.io/api/common/NgSwitchCase)="'bright'" [[item](https://angular.io/api/core/IterableChangeRecord#item)]="currentItem"></app-best-item>

<!-- . . . -->

<app-unknown-item \*[ngSwitchDefault](https://angular.io/api/common/NgSwitchDefault) [[item](https://angular.io/api/core/IterableChangeRecord#item)]="currentItem"></app-unknown-item>

</div>



[NgSwitch](https://angular.io/api/common/NgSwitch) is the controller directive. Bind it to an expression that returns the switch value, such as feature. Though the feature value in this example is a string, the switch value can be of any type.

**Bind to**[[ngSwitch](https://angular.io/api/common/NgSwitch)]. You'll get an error if you try to set \*[ngSwitch](https://angular.io/api/common/NgSwitch) because [NgSwitch](https://angular.io/api/common/NgSwitch) is an attribute directive, not a structural directive. Rather than touching the DOM directly, it changes the behavior of its companion directives.

**Bind to**\*[ngSwitchCase](https://angular.io/api/common/NgSwitchCase)**and**\*[ngSwitchDefault](https://angular.io/api/common/NgSwitchDefault). The [NgSwitchCase](https://angular.io/api/common/NgSwitchCase) and [NgSwitchDefault](https://angular.io/api/common/NgSwitchDefault) directives are structural directives because they add or remove elements from the DOM.

* [NgSwitchCase](https://angular.io/api/common/NgSwitchCase) adds its element to the DOM when its bound value equals the switch value and removes its bound value when it doesn't equal the switch value.
* [NgSwitchDefault](https://angular.io/api/common/NgSwitchDefault) adds its element to the DOM when there is no selected [NgSwitchCase](https://angular.io/api/common/NgSwitchCase).

The switch directives are particularly useful for adding and removing component elements. This example switches among four [item](https://angular.io/api/core/IterableChangeRecord#item) components defined in the item-switch.components.ts file. Each component has an [item](https://angular.io/api/core/IterableChangeRecord#item) [input property](https://angular.io/guide/template-syntax#inputs-outputs) which is bound to the currentItem of the parent component.

Switch directives work as well with native elements and web components too. For example, you could replace the <app-best-item> switch case with the following.

src/app/app.component.html

<div \*[ngSwitchCase](https://angular.io/api/common/NgSwitchCase)="'bright'"> Are you as bright as {{currentItem.name}}?</div>

## Template reference variables (#var)

A **template reference variable** is often a reference to a DOM element within a template. It can also refer to a directive (which contains a component), an element, [TemplateRef](https://angular.io/api/core/TemplateRef), or a [web component](https://developer.mozilla.org/en-US/docs/Web/Web_Components).

For a demonstration of the syntax and code snippets in this section, see the [template reference variables example](https://angular.io/generated/live-examples/template-reference-variables/stackblitz.html) / [download example](https://angular.io/generated/zips/template-reference-variables/template-reference-variables.zip).

Use the hash symbol (#) to declare a reference variable. The following reference variable, #phone, declares a phone variable on an <input> element.

src/app/app.component.html

<input #phone placeholder="phone number" />

You can refer to a template reference variable anywhere in the component's template. Here, a <button> further down the template refers to the phone variable.

src/app/app.component.html

<input #phone placeholder="phone number" />

<!-- lots of other elements -->

<!-- phone refers to the input element; pass its `value` to an event handler -->

<button (click)="callPhone(phone.value)">Call</button>

### **How a reference variable gets its value**

In most cases, Angular sets the reference variable's value to the element on which it is declared. In the previous example, phone refers to the phone number <input>. The button's click handler passes the <input> value to the component's callPhone() method.

The [NgForm](https://angular.io/api/forms/NgForm) directive can change that behavior and set the value to something else. In the following example, the template reference variable, itemForm, appears three times separated by HTML.

src/app/hero-form.component.html

<form #itemForm="[ngForm](https://angular.io/api/forms/NgForm)" (ngSubmit)="onSubmit(itemForm)">

<label for="name"

>Name <input class="form-control" name="name" [ngModel](https://angular.io/api/forms/NgModel) required />

</label>

<button type="submit">Submit</button>

</form>

<div [hidden]="!itemForm.form.valid">

<p>{{ submitMessage }}</p>

</div>

The reference value of itemForm, without the ngForm attribute value, would be the [HTMLFormElement](https://developer.mozilla.org/en-US/docs/Web/API/HTMLFormElement). There is, however, a difference between a Component and a Directive in that a [Component](https://angular.io/api/core/Component)will be referenced without specifying the attribute value, and a [Directive](https://angular.io/api/core/Directive) will not change the implicit reference (that is, the element).

However, with [NgForm](https://angular.io/api/forms/NgForm), itemForm is a reference to the [NgForm](https://angular.io/api/forms/NgForm) directive with the ability to track the value and validity of every control in the form.

The native <form> element doesn't have a form property, but the [NgForm](https://angular.io/api/forms/NgForm) directive does, which allows disabling the submit button if the itemForm.form.valid is invalid and passing the entire form control tree to the parent component's onSubmit() method.

### **Template reference variable considerations**

A template reference variable (#phone) is not the same as a template input variable (let phone) such as in an [\*ngFor](https://angular.io/guide/template-syntax#template-input-variable). See [Structural Directives](https://angular.io/guide/structural-directives#template-input-variable) for more information.

The scope of a reference variable is the entire template. So, don't define the same variable name more than once in the same template as the runtime value will be unpredictable.

#### **Alternative syntax**

You can use the ref- prefix alternative to #. This example declares the fax variable as ref-fax instead of #fax.

src/app/app.component.html

<input ref-fax placeholder="fax number" />

<button (click)="callFax(fax.value)">Fax</button>

## @[Input](https://angular.io/api/core/Input)() and @[Output](https://angular.io/api/core/Output)() properties

@[Input](https://angular.io/api/core/Input)() and @[Output](https://angular.io/api/core/Output)() allow Angular to share data between the parent context and child directives or components. An @[Input](https://angular.io/api/core/Input)() property is writable while an @[Output](https://angular.io/api/core/Output)()property is observable.

Consider this example of a child/parent relationship:

<parent-component>

<child-component></child-component>

</parent-component>

Here, the <child-component> selector, or child directive, is embedded within a <parent-component>, which serves as the child's context.

@[Input](https://angular.io/api/core/Input)() and @[Output](https://angular.io/api/core/Output)() act as the API, or application programming interface, of the child component in that they allow the child to communicate with the parent. Think of @[Input](https://angular.io/api/core/Input)() and @[Output](https://angular.io/api/core/Output)() like ports or doorways—@[Input](https://angular.io/api/core/Input)() is the doorway into the component allowing data to flow in while @[Output](https://angular.io/api/core/Output)() is the doorway out of the component, allowing the child component to send data out.

This section about @[Input](https://angular.io/api/core/Input)() and @[Output](https://angular.io/api/core/Output)() has its own [live example](https://angular.io/generated/live-examples/inputs-outputs/stackblitz.html) / [download example](https://angular.io/generated/zips/inputs-outputs/inputs-outputs.zip). The following subsections highlight key points in the sample app.

#### **@**[**Input**](https://angular.io/api/core/Input)**() and @**[**Output**](https://angular.io/api/core/Output)**() are independent**

Though @[Input](https://angular.io/api/core/Input)() and @[Output](https://angular.io/api/core/Output)() often appear together in apps, you can use them separately. If the nested component is such that it only needs to send data to its parent, you wouldn't need an @[Input](https://angular.io/api/core/Input)(), only an @[Output](https://angular.io/api/core/Output)(). The reverse is also true in that if the child only needs to receive data from the parent, you'd only need @[Input](https://angular.io/api/core/Input)().

## How to use @[Input](https://angular.io/api/core/Input)()

Use the @[Input](https://angular.io/api/core/Input)() decorator in a child component or directive to let Angular know that a property in that component can receive its value from its parent component. It helps to remember that the data flow is from the perspective of the child component. So an @[Input](https://angular.io/api/core/Input)() allows data to be input into the child component from the parent component.

To illustrate the use of @[Input](https://angular.io/api/core/Input)(), edit these parts of your app:

* The child component class and template
* The parent component class and template

### **In the child**

To use the @[Input](https://angular.io/api/core/Input)() decorator in a child component class, first import [Input](https://angular.io/api/core/Input) and then decorate the property with @[Input](https://angular.io/api/core/Input)():

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

import { [Component](https://angular.io/api/core/Component), [Input](https://angular.io/api/core/Input) } from '@angular/core'; // [First](https://angular.io/), import [Input](https://angular.io/api/core/Input)

export class ItemDetailComponent {

@[Input](https://angular.io/api/core/Input)() [item](https://angular.io/api/core/IterableChangeRecord#item): string; // decorate the property with @[Input](https://angular.io/api/core/Input)()

}

In this case, @[Input](https://angular.io/api/core/Input)() decorates the property item, which has a type of string, however, @[Input](https://angular.io/api/core/Input)() properties can have any type, such as number, string, boolean, or object. The value for [item](https://angular.io/api/core/IterableChangeRecord#item) will come from the parent component, which the next section covers.

Next, in the child component template, add the following:

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

<p>

Today's [item](https://angular.io/api/core/IterableChangeRecord#item): {{[item](https://angular.io/api/core/IterableChangeRecord#item)}}

</p>

### **In the parent**

The next step is to bind the property in the parent component's template. In this example, the parent component template is app.component.html.

First, use the child's selector, here <app-item-detail>, as a directive within the parent component template. Then, use [property binding](https://angular.io/guide/template-syntax#property-binding) to bind the property in the child to the property of the parent.

src/app/app.component.html

<app-item-detail [[item](https://angular.io/api/core/IterableChangeRecord#item)]="currentItem"></app-item-detail>

Next, in the parent component class, app.component.ts, designate a value for currentItem:

src/app/app.component.ts

export class AppComponent {

currentItem = 'Television';

}

With @[Input](https://angular.io/api/core/Input)(), Angular passes the value for currentItem to the child so that [item](https://angular.io/api/core/IterableChangeRecord#item) renders as Television.

The following diagram shows this structure:

The target in the square brackets, [], is the property you decorate with @[Input](https://angular.io/api/core/Input)() in the child component. The binding source, the part to the right of the equal sign, is the data that the parent component passes to the nested component.

The key takeaway is that when binding to a child component's property in a parent component—that is, what's in square brackets—you must decorate the property with @[Input](https://angular.io/api/core/Input)() in the child component.

#### [**OnChanges**](https://angular.io/api/core/OnChanges)**and @**[**Input**](https://angular.io/api/core/Input)**()**

To watch for changes on an @[Input](https://angular.io/api/core/Input)() property, use [OnChanges](https://angular.io/api/core/OnChanges), one of Angular's [lifecycle hooks](https://angular.io/guide/lifecycle-hooks#onchanges). [OnChanges](https://angular.io/api/core/OnChanges) is specifically designed to work with properties that have the @[Input](https://angular.io/api/core/Input)() decorator. See the [OnChanges](https://angular.io/guide/lifecycle-hooks#onchanges) section of the [Lifecycle Hooks](https://angular.io/guide/lifecycle-hooks) guide for more details and examples.

## How to use @[Output](https://angular.io/api/core/Output)()

Use the @[Output](https://angular.io/api/core/Output)() decorator in the child component or directive to allow data to flow from the child out to the parent.

An @[Output](https://angular.io/api/core/Output)() property should normally be initialized to an Angular [EventEmitter](https://angular.io/api/core/EventEmitter) with values flowing out of the component as [events](https://angular.io/guide/template-syntax#event-binding).

Just like with @[Input](https://angular.io/api/core/Input)(), you can use @[Output](https://angular.io/api/core/Output)() on a property of the child component but its type should be [EventEmitter](https://angular.io/api/core/EventEmitter).

@[Output](https://angular.io/api/core/Output)() marks a property in a child component as a doorway through which data can travel from the child to the parent. The child component then has to raise an event so the parent knows something has changed. To raise an event, @[Output](https://angular.io/api/core/Output)() works hand in hand with [EventEmitter](https://angular.io/api/core/EventEmitter), which is a class in @angular/core that you use to emit custom events.

When you use @[Output](https://angular.io/api/core/Output)(), edit these parts of your app:

* The child component class and template
* The parent component class and template

The following example shows how to set up an @[Output](https://angular.io/api/core/Output)() in a child component that pushes data you enter in an HTML <input> to an array in the parent component.

The HTML element <input> and the Angular decorator @[Input](https://angular.io/api/core/Input)() are different. This documentation is about component communication in Angular as it pertains to @[Input](https://angular.io/api/core/Input)() and @[Output](https://angular.io/api/core/Output)(). For more information on the HTML element <input>, see the [W3C Recommendation](https://www.w3.org/TR/html5/sec-forms.html#the-input-element).

### **In the child**

This example features an <input> where a user can enter a value and click a <button> that raises an event. The [EventEmitter](https://angular.io/api/core/EventEmitter) then relays the data to the parent component.

First, be sure to import [Output](https://angular.io/api/core/Output) and [EventEmitter](https://angular.io/api/core/EventEmitter) in the child component class:

import { [Output](https://angular.io/api/core/Output), [EventEmitter](https://angular.io/api/core/EventEmitter) } from '@angular/core';

Next, still in the child, decorate a property with @[Output](https://angular.io/api/core/Output)() in the component class. The following example @[Output](https://angular.io/api/core/Output)() is called newItemEvent and its type is [EventEmitter](https://angular.io/api/core/EventEmitter), which means it's an event.

src/app/item-output/item-output.component.ts

@[Output](https://angular.io/api/core/Output)() newItemEvent = new [EventEmitter](https://angular.io/api/core/EventEmitter)<string>();

The different parts of the above declaration are as follows:

* @[Output](https://angular.io/api/core/Output)()—a decorator function marking the property as a way for data to go from the child to the parent
* newItemEvent—the name of the @[Output](https://angular.io/api/core/Output)()
* [EventEmitter](https://angular.io/api/core/EventEmitter)<string>—the @[Output](https://angular.io/api/core/Output)()'s type
* new [EventEmitter](https://angular.io/api/core/EventEmitter)<string>()—tells Angular to create a new event emitter and that the data it emits is of type string. The type could be any type, such as number, boolean, and so on. For more information on [EventEmitter](https://angular.io/api/core/EventEmitter), see the [EventEmitter API documentation](https://angular.io/api/core/EventEmitter).

Next, create an addNewItem() method in the same component class:

src/app/item-output/item-output.component.ts

export class ItemOutputComponent {

@[Output](https://angular.io/api/core/Output)() newItemEvent = new [EventEmitter](https://angular.io/api/core/EventEmitter)<string>();

addNewItem(value: string) {

this.newItemEvent.emit(value);

}

}

The addNewItem() function uses the @[Output](https://angular.io/api/core/Output)(), newItemEvent, to raise an event in which it emits the value the user types into the <input>. In other words, when the user clicks the add button in the UI, the child lets the parent know about the event and gives that data to the parent.

#### **In the child's template**

The child's template has two controls. The first is an HTML <input> with a [template reference variable](https://angular.io/guide/template-syntax#ref-var) , #newItem, where the user types in an item name. Whatever the user types into the <input> gets stored in the #newItem variable.

src/app/item-output/item-output.component.html

<label>Add an [item](https://angular.io/api/core/IterableChangeRecord#item): <input #newItem></label>

<button (click)="addNewItem(newItem.value)">Add to parent's list</button>

The second element is a <button> with an [event binding](https://angular.io/guide/template-syntax#event-binding). You know it's an event binding because the part to the left of the equal sign is in parentheses, (click).

The (click) event is bound to the addNewItem() method in the child component class which takes as its argument whatever the value of #newItem is.

Now the child component has an @[Output](https://angular.io/api/core/Output)() for sending data to the parent and a method for raising an event. The next step is in the parent.

### **In the parent**

In this example, the parent component is AppComponent, but you could use any component in which you could nest the child.

The AppComponent in this example features a list of items in an array and a method for adding more items to the array.

src/app/app.component.ts

export class AppComponent {

items = ['item1', 'item2', 'item3', 'item4'];

addItem(newItem: string) {

this.items.push(newItem);

}

}

The addItem() method takes an argument in the form of a string and then pushes, or adds, that string to the items array.

#### **In the parent's template**

Next, in the parent's template, bind the parent's method to the child's event. Put the child selector, here <app-item-output>, within the parent component's template, app.component.html.

src/app/app.component.html

<app-item-output (newItemEvent)="addItem($event)"></app-item-output>

The event binding, (newItemEvent)='addItem($event)', tells Angular to connect the event in the child, newItemEvent, to the method in the parent, addItem(), and that the event that the child is notifying the parent about is to be the argument of addItem(). In other words, this is where the actual hand off of data takes place. The $eventcontains the data that the user types into the <input> in the child template UI.

Now, in order to see the @[Output](https://angular.io/api/core/Output)() working, add the following to the parent's template:

<ul>

<li \*[ngFor](https://angular.io/api/common/NgForOf)="let [item](https://angular.io/api/core/IterableChangeRecord#item) of items">{{[item](https://angular.io/api/core/IterableChangeRecord#item)}}</li>

</ul>

The \*[ngFor](https://angular.io/api/common/NgForOf) iterates over the items in the items array. When you enter a value in the child's <input> and click the button, the child emits the event and the parent's addItem() method pushes the value to the items array and it renders in the list.

## @[Input](https://angular.io/api/core/Input)() and @[Output](https://angular.io/api/core/Output)() together

You can use @[Input](https://angular.io/api/core/Input)() and @[Output](https://angular.io/api/core/Output)() on the same child component as in the following:

src/app/app.component.html

<app-input-output [[item](https://angular.io/api/core/IterableChangeRecord#item)]="currentItem" (deleteRequest)="crossOffItem($event)"></app-input-output>

The target, [item](https://angular.io/api/core/IterableChangeRecord#item), which is an @[Input](https://angular.io/api/core/Input)() property in the child component class, receives its value from the parent's property, currentItem. When you click delete, the child component raises an event, deleteRequest, which is the argument for the parent's crossOffItem() method.

The following diagram is of an @[Input](https://angular.io/api/core/Input)() and an @[Output](https://angular.io/api/core/Output)() on the same child component and shows the different parts of each:

As the diagram shows, use inputs and outputs together in the same manner as using them separately. Here, the child selector is <app-input-output> with [item](https://angular.io/api/core/IterableChangeRecord#item) and deleteRequest being @[Input](https://angular.io/api/core/Input)() and @[Output](https://angular.io/api/core/Output)() properties in the child component class. The property currentItem and the method crossOffItem() are both in the parent component class.

To combine property and event bindings using the banana-in-a-box syntax, [()], see [Two-way Binding](https://angular.io/guide/template-syntax#two-way).

For more detail on how these work, see the previous sections on [Input](https://angular.io/guide/template-syntax#input) and [Output](https://angular.io/guide/template-syntax#output). To see it in action, see the [Inputs and Outputs Example](https://angular.io/generated/live-examples/inputs-outputs/stackblitz.html) / [download example](https://angular.io/generated/zips/inputs-outputs/inputs-outputs.zip).

## @[Input](https://angular.io/api/core/Input)() and @[Output](https://angular.io/api/core/Output)() declarations

Instead of using the @[Input](https://angular.io/api/core/Input)() and @[Output](https://angular.io/api/core/Output)() decorators to declare inputs and outputs, you can identify members in the inputs and outputs arrays of the directive metadata, as in this example:

src/app/in-the-metadata/in-the-metadata.component.ts

// tslint:disable: no-inputs-metadata-property no-outputs-metadata-property

inputs: ['clearanceItem'],

outputs: ['buyEvent']

// tslint:enable: no-inputs-metadata-property no-outputs-metadata-property

While declaring inputs and outputs in the @[Directive](https://angular.io/api/core/Directive) and @[Component](https://angular.io/api/core/Component) metadata is possible, it is a better practice to use the @[Input](https://angular.io/api/core/Input)() and @[Output](https://angular.io/api/core/Output)() class decorators instead, as follows:

src/app/input-output/input-output.component.ts

@[Input](https://angular.io/api/core/Input)() [item](https://angular.io/api/core/IterableChangeRecord#item): string;

@[Output](https://angular.io/api/core/Output)() deleteRequest = new [EventEmitter](https://angular.io/api/core/EventEmitter)<string>();

See the [Decorate input and output properties](https://angular.io/guide/styleguide#decorate-input-and-output-properties) section of the [Style Guide](https://angular.io/guide/styleguide) for details.

If you get a template parse error when trying to use inputs or outputs, but you know that the properties do indeed exist, double check that your properties are annotated with @[Input](https://angular.io/api/core/Input)() / @[Output](https://angular.io/api/core/Output)() or that you've declared them in an inputs/outputs array:

Uncaught [Error](https://angular.io/api/core/MissingTranslationStrategy#Error): [Template](https://angular.io/) parse errors:

Can't bind to '[item](https://angular.io/api/core/IterableChangeRecord#item)' since it isn't [a](https://angular.io/api/router/RouterLinkWithHref) known property of 'app-item-detail'

## Aliasing inputs and outputs

Sometimes the public name of an input/output property should be different from the internal name. While it is a best practice to avoid this situation, Angular does offer a solution.

### **Aliasing in the metadata**

Alias inputs and outputs in the metadata using a colon-delimited (:) string with the directive property name on the left and the public alias on the right:

src/app/aliasing/aliasing.component.ts

// tslint:disable: no-inputs-metadata-property no-outputs-metadata-property

inputs: ['input1: saveForLaterItem'], // propertyName:alias

outputs: ['outputEvent1: saveForLaterEvent']

// tslint:disable: no-inputs-metadata-property no-outputs-metadata-property

### **Aliasing with the @**[**Input**](https://angular.io/api/core/Input)**()/@**[**Output**](https://angular.io/api/core/Output)**() decorator**

You can specify the alias for the property name by passing the alias name to the @[Input](https://angular.io/api/core/Input)()/@[Output](https://angular.io/api/core/Output)() decorator. The internal name remains as usual.

src/app/aliasing/aliasing.component.ts

@[Input](https://angular.io/api/core/Input)('wishListItem') input2: string; // @[Input](https://angular.io/api/core/Input)(alias)

@[Output](https://angular.io/api/core/Output)('wishEvent') outputEvent2 = new [EventEmitter](https://angular.io/api/core/EventEmitter)<string>(); // @[Output](https://angular.io/api/core/Output)(alias) propertyName = ...

## Template expression operators

The Angular template expression language employs a subset of JavaScript syntax supplemented with a few special operators for specific scenarios. The next sections cover three of these operators:

* [pipe](https://angular.io/guide/template-syntax#pipe)
* [safe navigation operator](https://angular.io/guide/template-syntax#safe-navigation-operator)
* [non-null assertion operator](https://angular.io/guide/template-syntax#non-null-assertion-operator)

### **The pipe operator (|)**

The result of an expression might require some transformation before you're ready to use it in a binding. For example, you might display a number as a currency, change text to uppercase, or filter a list and sort it.

Pipes are simple functions that accept an input value and return a transformed value. They're easy to apply within template expressions, using the pipe operator (|):

src/app/app.component.html

<p>[Title](https://angular.io/api/platform-browser/Title) through uppercase pipe: {{title | [uppercase](https://angular.io/api/common/UpperCasePipe)}}</p>

The pipe operator passes the result of an expression on the left to a pipe function on the right.

You can chain expressions through multiple pipes:

src/app/app.component.html

<!-- convert title to uppercase, then to lowercase -->

<p>[Title](https://angular.io/api/platform-browser/Title) through [a](https://angular.io/api/router/RouterLinkWithHref) pipe chain: {{title | [uppercase](https://angular.io/api/common/UpperCasePipe) | [lowercase](https://angular.io/api/common/LowerCasePipe)}}</p>

And you can also [apply parameters](https://angular.io/guide/pipes#parameterizing-a-pipe) to a pipe:

src/app/app.component.html

<!-- pipe with configuration argument => "February 25, 1980" -->

<p>Manufacture date with date format pipe: {{item.manufactureDate | date:'longDate'}}</p>

The [json](https://angular.io/api/common/JsonPipe) pipe is particularly helpful for debugging bindings:

src/app/app.component.html

<p>Item json pipe: {{[item](https://angular.io/api/core/IterableChangeRecord#item) | [json](https://angular.io/api/common/JsonPipe)}}</p>

The generated output would look something like this:

{ "name": "Telephone",

"manufactureDate": "1980-02-25T05:00:00.000Z",

"price": 98 }

**Note**: The pipe operator has a higher precedence than the ternary operator (?:), which means [a](https://angular.io/api/router/RouterLinkWithHref) ? b : c | x is parsed as [a](https://angular.io/api/router/RouterLinkWithHref) ? b : (c | x). Nevertheless, for a number of reasons, the pipe operator cannot be used without parentheses in the first and second operands of ?:. A good practice is to use parentheses in the third operand too.

### **The safe navigation operator ( ? ) and null property paths**

The Angular safe navigation operator, ?, guards against null and undefined values in property paths. Here, it protects against a view render failure if [item](https://angular.io/api/core/IterableChangeRecord#item) is null.

src/app/app.component.html

<p>The [item](https://angular.io/api/core/IterableChangeRecord#item) name is: {{[item](https://angular.io/api/core/IterableChangeRecord#item)?.name}}</p>

If [item](https://angular.io/api/core/IterableChangeRecord#item) is null, the view still renders but the displayed value is blank; you see only "The item name is:" with nothing after it.

Consider the next example, with a nullItem.

The null [item](https://angular.io/api/core/IterableChangeRecord#item) name is {{nullItem.name}}

Since there is no safe navigation operator and nullItem is null, JavaScript and Angular would throw a null reference error and break the rendering process of Angular:

TypeError: Cannot [read](https://angular.io/api/core/Query#read) property 'name' of null.

Sometimes however, null values in the property path may be OK under certain circumstances, especially when the value starts out null but the data arrives eventually.

With the safe navigation operator, ?, Angular stops evaluating the expression when it hits the first null value and renders the view without errors.

It works perfectly with long property paths such as [a](https://angular.io/api/router/RouterLinkWithHref)?.b?.c?.d.

### **The non-null assertion operator ( ! )**

As of Typescript 2.0, you can enforce [strict null checking](http://www.typescriptlang.org/docs/handbook/release-notes/typescript-2-0.html) with the --strictNullChecks flag. TypeScript then ensures that no variable is unintentionally null or undefined.

In this mode, typed variables disallow null and undefined by default. The type checker throws an error if you leave a variable unassigned or try to assign null or undefined to a variable whose type disallows null and undefined.

The type checker also throws an error if it can't determine whether a variable will be null or undefined at runtime. You tell the type checker not to throw an error by applying the postfix [non-null assertion operator, !](http://www.typescriptlang.org/docs/handbook/release-notes/typescript-2-0.html#non-null-assertion-operator).

The Angular non-null assertion operator, !, serves the same purpose in an Angular template. For example, after you use [\*ngIf](https://angular.io/guide/template-syntax#ngIf) to check that [item](https://angular.io/api/core/IterableChangeRecord#item) is defined, you can assert that [item](https://angular.io/api/core/IterableChangeRecord#item) properties are also defined.

src/app/app.component.html

<!--No color, no error -->

<p \*[ngIf](https://angular.io/api/common/NgIf)="[item](https://angular.io/api/core/IterableChangeRecord#item)">The [item](https://angular.io/api/core/IterableChangeRecord#item)'s color is: {{[item](https://angular.io/api/core/IterableChangeRecord#item)!.color}}</p>

When the Angular compiler turns your template into TypeScript code, it prevents TypeScript from reporting that [item](https://angular.io/api/core/IterableChangeRecord#item) might be null or undefined.

Unlike the [safe navigation operator](https://angular.io/guide/template-syntax#safe-navigation-operator), the non-null assertion operator does not guard against null or undefined. Rather, it tells the TypeScript type checker to suspend strict null checks for a specific property expression.

The non-null assertion operator, !, is optional with the exception that you must use it when you turn on strict null checks.

[back to top](https://angular.io/guide/template-syntax#top-of-page)

## Built-in template functions

### **The $any() type cast function**

Sometimes a binding expression triggers a type error during [AOT compilation](https://angular.io/guide/aot-compiler) and it is not possible or difficult to fully specify the type. To silence the error, you can use the $any() cast function to cast the expression to [the any type](http://www.typescriptlang.org/docs/handbook/basic-types.html#any) as in the following example:

src/app/app.component.html

<p>The [item](https://angular.io/api/core/IterableChangeRecord#item)'s undeclared best by date is: {{$any([item](https://angular.io/api/core/IterableChangeRecord#item)).bestByDate}}</p>

When the Angular compiler turns this template into TypeScript code, it prevents TypeScript from reporting that bestByDate is not a member of the [item](https://angular.io/api/core/IterableChangeRecord#item) object when it runs type checking on the template.

The $any() cast function also works with this to allow access to undeclared members of the component.

src/app/app.component.html

<p>The [item](https://angular.io/api/core/IterableChangeRecord#item)'s undeclared best by date is: {{$any(this).bestByDate}}</p>

The $any() cast function works anywhere in a binding expression where a method call is valid.

## SVG in templates

It is possible to use SVG as valid templates in Angular. All of the template syntax below is applicable to both SVG and HTML. Learn more in the SVG [1.1](https://www.w3.org/TR/SVG11/) and [2.0](https://www.w3.org/TR/SVG2/)specifications.

Why would you use SVG as template, instead of simply adding it as image to your application?

When you use an SVG as the template, you are able to use directives and bindings just like with HTML templates. This means that you will be able to dynamically generate interactive graphics.

Refer to the sample code snippet below for a syntax example:

src/app/svg.component.ts

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

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

selector: 'app-svg',

templateUrl: './svg.component.svg',

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

})

export class SvgComponent {

fillColor = 'rgb(255, 0, 0)';

changeColor() {

const r = Math.floor(Math.random() \* 256);

const g = Math.floor(Math.random() \* 256);

const b = Math.floor(Math.random() \* 256);

this.fillColor = `rgb(${r}, ${g}, ${b})`;

}

}

Add the below code to your svg.component.svg file:

src/app/svg.component.svg

<svg>

<g>

<rect x="0" y="0" width="100" height="100" [attr.fill]="fillColor" (click)="changeColor()" />

<text x="120" y="50">click the rectangle to change the fill color</text>

</g>

</svg>

Here you can see the use of a click() event binding and the property binding syntax ([attr.fill]="fillColor").