Tutorials

Angular

By- EpubBySudSou

What is Angular?

This topic can help you understand Angular: what Angular is, what advantages it provides, and what you might expect as you start to build your applications.

Angular is a development platform, built on [TypeScript](https://www.typescriptlang.org/). As a platform, Angular includes:

* A component-based framework for building scalable web applications
* A collection of well-integrated libraries that cover a wide variety of features, including routing, forms management, client-server communication, and more
* A suite of developer tools to help you develop, build, test, and update your code

With Angular, you're taking advantage of a platform that can scale from single-developer projects to enterprise-level applications. Angular is designed to make updating as easy as possible, so you can take advantage of the latest developments with a minimum of effort. Best of all, the Angular ecosystem consists of a diverse group of over 1.7 million developers, library authors, and content creators.

# Angular applications: The essentials

This section explains the core ideas behind Angular. Understanding these ideas can help you design and build your applications more effectively.

### **Components**

Components are the building blocks that compose an application. A component includes a TypeScript class with a @[Component](https://angular.io/api/core/Component)() decorator, an HTML template, and styles. The @[Component](https://angular.io/api/core/Component)() decorator specifies the following Angular-specific information:

* A CSS selector that defines how the component is used in a template. HTML elements in your template that match this selector become instances of the component.
* An HTML template that instructs Angular how to render the component.
* An optional set of CSS styles that define the appearance of the template's HTML elements.

The following is a minimal Angular component.

|  |
| --- |
| import { Component } from '@angular/core';  @Component({  selector: 'hello-world',  template: `  <h2>Hello World</h2>  <p>This is my first component!</p>  `,  })  export class HelloWorldComponent {  // The code in this class drives the component's behavior.  } |

To use this component, you write the following in a template:

|  |
| --- |
| <hello-world></hello-world> |

When Angular renders this component, the resulting DOM looks like this:

|  |
| --- |
| <hello-world>  <h2>Hello World</h2>  <p>This is my first component!</p>  </hello-world> |

Angular's component model offers strong encapsulation and an intuitive application structure. Components also make your application easier to unit test and can improve the overall readability of your code.

For more information on what you can do with components, see the [Components](https://angular.io/guide/component-overview) section.

### **Templates**

Every component has an HTML template that declares how that component renders. You define this template either inline or by file path.

Angular extends HTML with additional syntax that lets you insert dynamic values from your component. Angular automatically updates the rendered DOM when your component’s state changes. One application of this feature is inserting dynamic text, as shown in the following example.

|  |
| --- |
| <p>{{ message }}</p> |

The value for message comes from the component class:

|  |
| --- |
| import { Component } from '@angular/core';  @Component ({  selector: 'hello-world-interpolation',  templateUrl: './hello-world-interpolation.component.html'  })  export class HelloWorldInterpolationComponent {  message = 'Hello, World!';  } |

When the application loads the component and its template, the user sees the following:

|  |
| --- |
| <p>Hello, World!</p> |

Notice the use of double curly braces--they instruct Angular to interpolate the contents within them.

Angular also supports property bindings, to help you set values for properties and attributes of HTML elements and pass values to your application's presentation logic.

|  |
| --- |
| <p [id]="sayHelloId" [style.color]="fontColor">You can set my color in the component!</p> |

Notice the use of the square brackets--that syntax indicates that you're binding the property or attribute to a value in the component class.

You can also declare event listeners to listen for and respond to user actions such as keystrokes, mouse movements, clicks, and touches. You declare an event listener by specifying the event name in parentheses:

|  |
| --- |
| <button (click)="sayMessage()" [disabled]="canClick">Trigger alert message</button> |

The preceding example calls a method, which is defined in the component class:

|  |
| --- |
| sayMessage() { alert(this.message); } |

You can add additional functionality to your templates through the use of [directives](https://angular.io/guide/built-in-directives). The most popular directives in Angular are \*[ngIf](https://angular.io/api/common/NgIf) and \*[ngFor](https://angular.io/api/common/NgForOf). You can use directives to perform a variety of tasks, such as dynamically modifying the DOM structure. And you can also create your own custom directives to create great user experiences.

Angular's declarative templates allow you to cleanly separate your application's logic from its presentation. Templates are based on standard HTML, so they're easy to build, maintain, and update.

For more information on what you can do with templates, see the [Templates](https://angular.io/guide/template-syntax) section.

### **Dependency injection**

Dependency injection allows you to declare the dependencies of your TypeScript classes without taking care of their instantiation. Instead, Angular handles the instantiation for you. This design pattern allows you to write more testable and flexible code. Even though understanding dependency injection is not critical to start using Angular, we strongly recommend it as a best practice and many aspects of Angular take advantage of it to some degree.

To illustrate how dependency injection works, consider the following example. The first file, logger.service.ts, defines a Logger class. This class contains a writeCount function that logs a number to the console.

|  |
| --- |
| import { Injectable } from '@angular/core';  @Injectable({providedIn: 'root'})  export class Logger {  writeCount(count: number) {  console.warn(count);  }  } |

Next, the hello-world-di.component.ts file defines an Angular component. This component contains a button that uses the writeCount function of the Logger class. To access that function, the Logger service is injected into the HelloWorldDI class by adding private logger: Logger to the constructor.

|  |
| --- |
| import { Component } from '@angular/core';  import { Logger } from '../logger.service';  @Component({  selector: 'hello-world-di',  templateUrl: './hello-world-di.component.html'  })  export class HelloWorldDependencyInjectionComponent {  count = 0;  constructor(private logger: Logger) {  }  onLogMe() {  this.logger.writeCount(this.count);  this.count++;  }  } |

For more information about dependency injection and Angular, see the [Dependency injection in Angular](https://angular.io/guide/dependency-injection) section.

## **Angular CLI**

The Angular CLI is the fastest, easiest, and recommended way to develop Angular applications. The Angular CLI makes a number of tasks easy. Here are some examples:

|  |  |
| --- | --- |
| [ng build](https://angular.io/cli/build) | Compiles an Angular app into an output directory. |
| [ng serve](https://angular.io/cli/serve) | Builds and serves your application, rebuilding on file changes. |
| [ng generate](https://angular.io/cli/generate) | Generates or modifies files based on a schematic. |
| [ng test](https://angular.io/cli/test) | Runs unit tests on a given project. |
| [ng e2e](https://angular.io/cli/e2e) | Builds and serves an Angular application, then runs end-to-end tests. |

You'll find the Angular CLI a valuable tool for building out your applications.

For more information about the Angular CLI, see the [CLI Reference](https://angular.io/cli) section.

## **First-party libraries**

The section, [Angular applications: The essentials](https://angular.io/guide/what-is-angular#essentials), provides a brief overview of a couple of the key architectural elements you'll use when building Angular applications. But the many benefits of Angular really become apparent when your application grows and you want to add additional functions such as site navigation or user input. That's when you can leverage the Angular platform to incorporate one of the many first-party libraries that Angular provides.

Some of the libraries available to you include:

|  |  |
| --- | --- |
| [Angular Router](https://angular.io/guide/router) | Advanced client-side navigation and routing based on Angular components. Supports lazy-loading, nested routes, custom path matching, and more. |
| [Angular Forms](https://angular.io/guide/forms-overview) | Uniform system for form participation and validation. |
| [Angular HttpClient](https://angular.io/guide/http) | Robust HTTP client that can power more advanced client-server communication. |
| [Angular Animations](https://angular.io/guide/animations) | Rich system for driving animations based on application state. |
| [Angular PWA](https://angular.io/guide/service-worker-intro) | Tools for building Progressive Web Applications (PWAs) including a service worker and Web app manifest. |
| [Angular Schematics](https://angular.io/guide/schematics) | Automated scaffolding, refactoring, and update tools that simplify development at large scale. |

These libraries expand your application's functionality while also allowing you to focus more on the features that make your application unique. And you can add these libraries knowing that they're designed to integrate seamlessly into and update simultaneously with the Angular framework.

These libraries are only required if and when they can help you add functionality to your applications or solve a particular problem.

## **Next steps**

This topic is intended to give you a brief overview of what Angular is, the advantages it provides, and what you can expect as you start to build your applications.

To see Angular in action, see our [Getting Started](https://angular.io/start) tutorial. This tutorial uses [stackblitz.com](https://stackblitz.com/), so you can explore a working example of Angular without any installation requirements.

To explore Angular's capabilities further, we recommend reading through the sections, Understanding Angular and Developer Guides.

# Getting started with Angular

Welcome to Angular!

This tutorial introduces you to the essentials of Angular by walking you through building an e-commerce site with a catalog, shopping cart, and check-out form.

To help you get started right away, this tutorial uses a ready-made application that you can examine and modify interactively on [StackBlitz](https://stackblitz.com/)—without having to [set up a local work environment](https://angular.io/guide/setup-local). StackBlitz is a browser-based development environment where you can create, save, and share projects using a variety of technologies.

## **Prerequisites**

To get the most out of this tutorial you should already have a basic understanding of the following.

* [HTML](https://developer.mozilla.org/en-US/docs/Learn/HTML)
* [JavaScript](https://developer.mozilla.org/en-US/docs/Web/JavaScript)
* [TypeScript](https://www.typescriptlang.org/)

## **Take a tour of the example application**

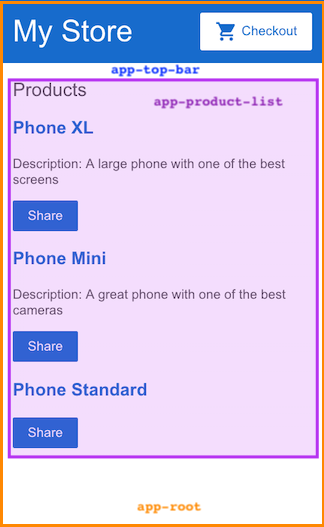
You build Angular applications with components. Components define areas of responsibility in the UI that let you reuse sets of UI functionality.

A component consists of three things:

* **A component class** that handles data and functionality.
* **An HTML template** that determines the UI.
* **Component-specific styles** that define the look and feel.

This guide demonstrates building an application with the following components.

* <app-root>—the first component to load and the container for the other components.
* <app-top-bar>—the store name and checkout button.
* <app-product-list>—the product list.
* <app-product-alerts>—a component that contains the application's alerts.



For more information about components, see [Introduction to Components](https://angular.io/guide/architecture-components).

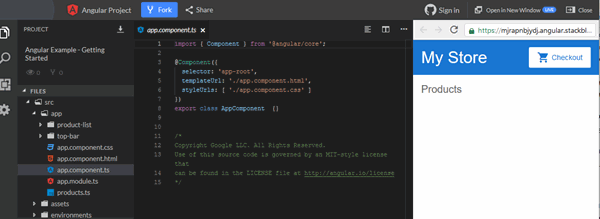
## **Create the sample project**

To create the sample project, generate the [ready-made sample project in StackBlitz](https://angular.io/generated/live-examples/getting-started-v0/stackblitz.html). To save your work:

1. Log into StackBlitz.
2. Fork the project you generated.
3. Save periodically.

In StackBlitz, the preview pane on the right shows the starting state of the example application. The preview features two areas:

* a top bar with the store name, My Store, and a checkout button
* a header for a product list, Products



The project section on the left shows the source files that make up the application, including the infrastructure and configuration files.

When you generate the StackBlitz example applications that accompany the tutorials, StackBlitz creates the starter files and mock data for you. The files you use throughout the tutorial are in the src folder.

For more information on how to use StackBlitz, see the [StackBlitz documentation](https://developer.stackblitz.com/docs/platform/).

## **Create the product list**

In this section, you'll update the application to display a list of products. You'll use predefined product data from the products.ts file and methods from the product-list.component.ts file. This section guides you through editing the HTML, also known as the template.

1. In the product-list folder, open the template file product-list.component.html.
2. Add an \*[ngFor](https://angular.io/api/common/NgForOf) structural directive on a <div>, as follows.

src/app/product-list/product-list.component.html

content\_copy<h2>Products</h2>

<div \*[ngFor](https://angular.io/api/common/NgForOf)="let product of products">

</div>

With \*[ngFor](https://angular.io/api/common/NgForOf), the <div> repeats for each product in the list.

Structural directives shape or reshape the DOM's structure, by adding, removing, and manipulating elements. For more information about structural directives, see [Structural directives](https://angular.io/guide/structural-directives).

1. Inside the <div>, add an <h3> and {{ product.name }}. The {{ product.name }} statement is an example of Angular's interpolation syntax. Interpolation {{ }} lets you render the property value as text.

src/app/product-list/product-list.component.html

content\_copy<h2>Products</h2>

<div \*[ngFor](https://angular.io/api/common/NgForOf)="let product of products">

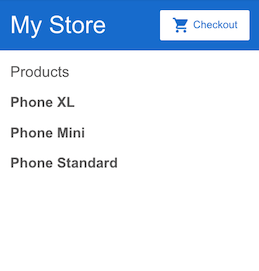
<h3>

{{ product.name }}

</h3>

</div>

The preview pane updates to display the name of each product in the list.



1. To make each product name a link to product details, add the <a> element around {{ product.name }}.
2. Set the title to be the product's name by using the property binding [ ] syntax, as follows:

src/app/product-list/product-list.component.html

content\_copy<h2>Products</h2>

<div \*[ngFor](https://angular.io/api/common/NgForOf)="let product of products">

<h3>

<a [title]="product.name + ' details'">

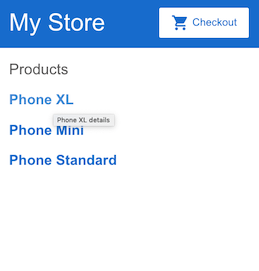
{{ product.name }}

</a>

</h3>

</div>

In the preview pane, hover over a product name to see the bound name property value, which is the product name plus the word "details". Property binding [ ] lets you use the property value in a template expression.



1. Add the product descriptions. On a <p> element, use an \*[ngIf](https://angular.io/api/common/NgIf) directive so that Angular only creates the <p> element if the current product has a description.

src/app/product-list/product-list.component.html

content\_copy<h2>Products</h2>

<div \*[ngFor](https://angular.io/api/common/NgForOf)="let product of products">

<h3>

<a [title]="product.name + ' details'">

{{ product.name }}

</a>

</h3>

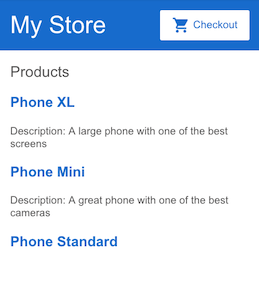
<p \*[ngIf](https://angular.io/api/common/NgIf)="product.description">

Description: {{ product.description }}

</p>

</div>

The application now displays the name and description of each product in the list. Notice that the final product does not have a description paragraph. Angular doesn't create the <p> element because the product's description property is empty.



1. Add a button so users can share a product. Bind the button's click event to the share() method in product-list.component.ts. Event binding uses a set of parentheses, ( ), around the event, as in the (click) event on the <button> element.

src/app/product-list/product-list.component.html

content\_copy<h2>Products</h2>

<div \*[ngFor](https://angular.io/api/common/NgForOf)="let product of products">

<h3>

<a [title]="product.name + ' details'">

{{ product.name }}

</a>

</h3>

<p \*[ngIf](https://angular.io/api/common/NgIf)="product.description">

Description: {{ product.description }}

</p>

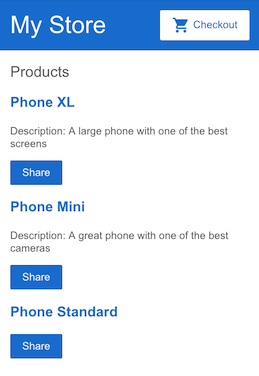
<button (click)="share()">

Share

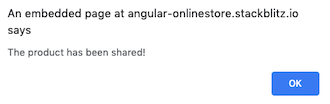
</button>

</div>

Each product now has a **Share** button.



Clicking the **Share** button triggers an alert that states, "The product has been shared!".



In editing the template, you have explored some of the most popular features of Angular templates. For more information, see [Introduction to components and templates](https://angular.io/guide/architecture-components#template-syntax).

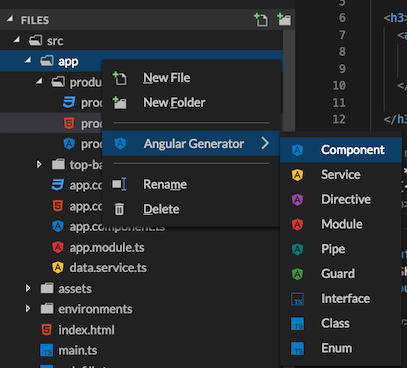
## **Pass data to a child component**

Currently, the product list displays the name and description of each product. The ProductListComponent also defines a products property that contains imported data for each product from the products array in products.ts.

The next step is to create a new alert feature that uses product data from the ProductListComponent. The alert checks the product's price, and, if the price is greater than $700, displays a **Notify Me** button that lets users sign up for notifications when the product goes on sale.

This section walks you through creating a child component, ProductAlertsComponent that can receive data from its parent component, ProductListComponent.

1. Right click on the app folder and use the Angular Generator to generate a new component named product-alerts.



The generator creates starter files for the three parts of the component:

* + product-alerts.component.ts
  + product-alerts.component.html
  + product-alerts.component.css

1. Open product-alerts.component.ts. The @[Component](https://angular.io/api/core/Component)() decorator indicates that the following class is a component. @[Component](https://angular.io/api/core/Component)() also provides metadata about the component, including its selector, templates, and styles.

src/app/product-alerts/product-alerts.component.ts

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

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

selector: 'app-product-alerts',

templateUrl: './product-alerts.component.html',

styleUrls: ['./product-alerts.component.css']

})

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

constructor() { }

ngOnInit() {

}

}

Key features in the @[Component](https://angular.io/api/core/Component)() are as follows:

* + The selector, app-product-alerts, identifies the component. By convention, Angular component selectors begin with the prefix app-, followed by the component name.
  + The template and style filenames reference the component's HTML and CSS.
  + The @[Component](https://angular.io/api/core/Component)() definition also exports the class, ProductAlertsComponent, which handles functionality for the component.

1. To set up ProductAlertsComponent to receive product data, first import [Input](https://angular.io/api/core/Input) from @angular/core.

src/app/product-alerts/product-alerts.component.ts

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

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

import { Product } from '../products';

1. In the ProductAlertsComponent class definition, define a property named product with an @[Input](https://angular.io/api/core/Input)() decorator. The @[Input](https://angular.io/api/core/Input)() decorator indicates that the property value passes in from the component's parent, ProductListComponent.

src/app/product-alerts/product-alerts.component.ts

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

@[Input](https://angular.io/api/core/Input)() product!: Product;

constructor() { }

ngOnInit() {

}

}

1. Open product-alerts.component.html and replace the placeholder paragraph with a **Notify Me** button that appears if the product price is over $700.

src/app/product-alerts/product-alerts.component.html

content\_copy<p \*[ngIf](https://angular.io/api/common/NgIf)="product && product.price > 700">

<button>Notify Me</button>

</p>

1. To display ProductAlertsComponent as a child of ProductListComponent, add the selector, <app-product-alerts> to product-list.component.html. Pass the current product as input to the component using property binding.

src/app/product-list/product-list.component.html

content\_copy<button (click)="share()">

Share

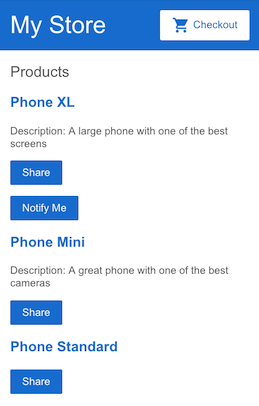
</button>

<app-product-alerts

[product]="product">

</app-product-alerts>

The new product alert component takes a product as input from the product list. With that input, it shows or hides the **Notify Me** button, based on the price of the product. The Phone XL price is over $700, so the **Notify Me** button appears on that product.



## **Pass data to a parent component**

To make the **Notify Me** button work, the child component needs to notify and pass the data to the parent component. The ProductAlertsComponent needs to emit an event when the user clicks **Notify Me** and the ProductListComponent needs to respond to the event.

1. In product-alerts.component.ts, import [Output](https://angular.io/api/core/Output) and [EventEmitter](https://angular.io/api/core/EventEmitter) from @angular/core.

src/app/product-alerts/product-alerts.component.ts

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

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

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

import { Product } from '../products';

1. In the component class, define a property named notify with an @[Output](https://angular.io/api/core/Output)() decorator and an instance of [EventEmitter](https://angular.io/api/core/EventEmitter)(). Configuring ProductAlertsComponent with an @[Output](https://angular.io/api/core/Output)() allows the ProductAlertsComponent to emit an event when the value of the notify property changes.

src/app/product-alerts/product-alerts.component.ts

content\_copyexport class ProductAlertsComponent {

@[Input](https://angular.io/api/core/Input)() product: Product|undefined;

@[Output](https://angular.io/api/core/Output)() notify = new [EventEmitter](https://angular.io/api/core/EventEmitter)();

}

In new components, the Angular Generator includes an empty constructor(), the [OnInit](https://angular.io/api/core/OnInit) interface, and the ngOnInit() method. Since these steps don't use them, the following code example omits them for brevity.

1. In product-alerts.component.html, update the **Notify Me** button with an event binding to call the notify.emit() method.

src/app/product-alerts/product-alerts.component.html

content\_copy<p \*[ngIf](https://angular.io/api/common/NgIf)="product && product.price > 700">

<button (click)="notify.emit()">Notify Me</button>

</p>

1. Define the behavior that happens when the user clicks the button. The parent, ProductListComponent—not the ProductAlertsComponent—acts when the child raises the event. In product-list.component.ts, define an onNotify() method, similar to the share() method.

src/app/product-list/product-list.component.ts

content\_copyexport class ProductListComponent {

products = products;

share() {

window.alert('The product has been shared!');

}

onNotify() {

window.alert('You will be notified when the product goes on sale');

}

}

1. Update the ProductListComponent to receive data from the ProductAlertsComponent.

In product-list.component.html, bind <app-product-alerts> to the onNotify() method of the product list component. <app-product-alerts> is what displays the **Notify Me** button.

src/app/product-list/product-list.component.html

content\_copy<button (click)="share()">

Share

</button>

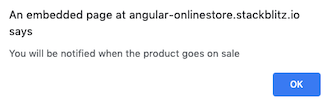
<app-product-alerts

[product]="product"

(notify)="onNotify()">

</app-product-alerts>

1. Click the **Notify Me** button to trigger an alert which reads, "You will be notified when the product goes on sale".



For more information on communication between components, see [Component Interaction](https://angular.io/guide/component-interaction).

# Adding navigation

This guide builds on the first step of the Getting Started tutorial, [Get started with a basic Angular app](https://angular.io/start).

At this stage of development, the online store application has a basic product catalog.

In the following sections, you'll add the following features to the application:

* Type a URL in the address bar to navigate to a corresponding product page.
* Click links on the page to navigate within your single-page application.
* Click the browser's back and forward buttons to navigate the browser history intuitively.

## **Associate a URL path with a component**

The application already uses the Angular [Router](https://angular.io/api/router/Router) to navigate to the ProductListComponent. This section shows you how to define a route to show individual product details.

1. Generate a new component for product details. In the file list, right-click the app folder, choose Angular Generator and [Component](https://angular.io/api/core/Component). Name the component product-details.
2. In app.module.ts, add a route for product details, with a path of products/:productId and ProductDetailsComponent for the component.

src/app/app.module.ts

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

imports: [

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

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

RouterModule.forRoot([

{ path: '', component: ProductListComponent },

{ path: 'products/:productId', component: ProductDetailsComponent },

])

],

1. Open product-list.component.html.
2. Modify the product name anchor to include a [routerLink](https://angular.io/api/router/RouterLink) with the product.id as a parameter.

src/app/product-list/product-list.component.html

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

<h3>

<a [title]="product.name + ' details'" [[routerLink](https://angular.io/api/router/RouterLink)]="['/products', product.id]">

{{ product.name }}

</a>

</h3>

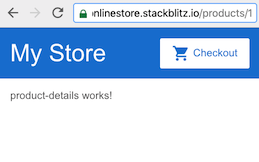
<!-- . . . -->

</div>

The [RouterLink](https://angular.io/api/router/RouterLink) directive helps you customize the anchor element. In this case, the route, or URL, contains one fixed segment, /products. The final segment is variable, inserting the id property of the current product. For example, the URL for a product with an id of 1 would be similar to https://getting-started-myfork.stackblitz.io/products/1.

1. Verify that the router works as intended by clicking the product name. The application should display the ProductDetailsComponent, which currently says "product-details works!"

Notice that the URL in the preview window changes. The final segment is products/# where # is the number of the route you clicked.



## **View product details**

The ProductDetailsComponent handles the display of each product. The Angular Router displays components based on the browser's URL and [your defined routes](https://angular.io/start/start-routing#define-routes).

In this section, you'll use the Angular Router to combine the products data and route information to display the specific details for each product.

1. In product-details.component.ts, import [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) from @angular/router, and the products array from ../products.

src/app/product-details/product-details.component.ts

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

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

import { Product, products } from '../products';

1. Define the product property.

src/app/product-details/product-details.component.ts

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

product: Product|undefined;

/\* ... \*/

}

1. Inject [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) into the constructor() by adding private route: [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) as an argument within the constructor's parentheses.

src/app/product-details/product-details.component.ts

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

product: Product|undefined;

constructor(

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

) { }

}

[ActivatedRoute](https://angular.io/api/router/ActivatedRoute) is specific to each component that the Angular Router loads. [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) contains information about the route and the route's parameters.

By injecting [ActivatedRoute](https://angular.io/api/router/ActivatedRoute), you are configuring the component to use a service. The [Managing Data](https://angular.io/start/start-data) step covers services in more detail.

1. In the ngOnInit() method, extract the productId from the route parameters and find the corresponding product in the products array.

src/app/product-details/product-details.component.ts

content\_copyngOnInit() {

// First get the product id from the current route.

const routeParams = this.route.snapshot.paramMap;

const productIdFromRoute = Number(routeParams.get('productId'));

// Find the product that correspond with the id provided in route.

this.product = products.find(product => product.id === productIdFromRoute);

}

The route parameters correspond to the path variables you define in the route. To access the route parameters, we use route.snapshot, which is the [ActivatedRouteSnapshot](https://angular.io/api/router/ActivatedRouteSnapshot) that contains information about the active route at that particular moment in time. The URL that matches the route provides the productId . Angular uses the productId to display the details for each unique product.

1. Update the ProductDetailsComponent template to display product details with an \*[ngIf](https://angular.io/api/common/NgIf). If a product exists, the <div> renders with a name, price, and description.

src/app/product-details/product-details.component.html

content\_copy<h2>Product Details</h2>

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

<h3>{{ product.name }}</h3>

<h4>{{ product.price | [currency](https://angular.io/api/common/CurrencyPipe) }}</h4>

<p>{{ product.description }}</p>

</div>

The line, <h4>{{ product.price | [currency](https://angular.io/api/common/CurrencyPipe) }}</h4>, uses the [currency](https://angular.io/api/common/CurrencyPipe) pipe to transform product.price from a number to a currency string. A pipe is a way you can transform data in your HTML template. For more information about Angular pipes, see [Pipes](https://angular.io/guide/pipes).

When users click on a name in the product list, the router navigates them to the distinct URL for the product, shows the ProductDetailsComponent, and displays the product details.



For more information about the Angular Router, see [Routing & Navigation](https://angular.io/guide/router).

# Managing data

This guide builds on the second step of the [Getting started with a basic Angular application](https://angular.io/start) tutorial, [Adding navigation](https://angular.io/start/start-routing). At this stage of development, the store application has a product catalog with two views: a product list and product details. Users can click on a product name from the list to see details in a new view, with a distinct URL, or route.

This step of the tutorial guides you through creating a shopping cart in the following phases:

* Update the product details view to include a **Buy** button, which adds the current product to a list of products that a cart service manages.
* Add a cart component, which displays the items in the cart.
* Add a shipping component, which retrieves shipping prices for the items in the cart by using Angular's [HttpClient](https://angular.io/api/common/http/HttpClient) to retrieve shipping data from a .json file.

## **Create the shopping cart service**

In Angular, a service is an instance of a class that you can make available to any part of your application using Angular's [dependency injection system](https://angular.io/guide/glossary#dependency-injection-di).

Currently, users can view product information, and the application can simulate sharing and notifications about product changes.

The next step is to build a way for users to add products to a cart. This section walks you through adding a **Buy** button and setting up a cart service to store information about products in the cart.

### **Define a cart service**

1. To generate a cart service, right click on the app folder, choose **Angular Generator**, and choose **Service**. Name the new service cart.

src/app/cart.service.ts

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

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

providedIn: 'root'

})

export class CartService {

constructor() {}

}

1. In the CartService class, define an items property to store the array of the current products in the cart.

src/app/cart.service.ts

content\_copyexport class CartService {

items: Product[] = [];

}

1. Define methods to add items to the cart, return cart items, and clear the cart items.

src/app/cart.service.ts

content\_copyexport class CartService {

items: Product[] = [];

addToCart(product: Product) {

this.items.push(product);

}

getItems() {

return this.items;

}

clearCart() {

this.items = [];

return this.items;

}

}

* + The addToCart() method appends a product to an array of items.
  + The getItems() method collects the items users add to the cart and returns each item with its associated quantity.
  + The clearCart() method returns an empty array of items, which empties the cart.

### **Use the cart service**

This section walks you through using the CartService to add a product to the cart.

1. In product-details.component.ts, import the cart service.

src/app/product-details/product-details.component.ts

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

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

import { Product, products } from '../products';

import { CartService } from '../cart.service';

1. Inject the cart service by adding it to the constructor().

src/app/product-details/product-details.component.ts

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

constructor(

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

private cartService: CartService

) { }

}

1. Define the addToCart() method, which adds the current product to the cart.

src/app/product-details/product-details.component.ts

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

addToCart(product: Product) {

this.cartService.addToCart(product);

window.alert('Your product has been added to the cart!');

}

}

The addToCart() method does the following:

* + Takes the current product as an argument.
  + Uses the CartService addToCart() method to add the product to the cart.
  + Displays a message that you've added a product to the cart.

1. In product-details.component.html, add a button with the label **Buy**, and bind the click() event to the addToCart() method. This code updates the product details template with a **Buy** button that adds the current product to the cart.

src/app/product-details/product-details.component.html

content\_copy<h2>Product Details</h2>

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

<h3>{{ product.name }}</h3>

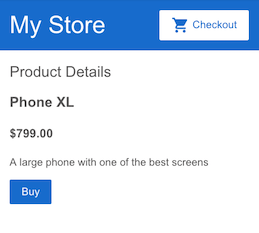
<h4>{{ product.price | [currency](https://angular.io/api/common/CurrencyPipe) }}</h4>

<p>{{ product.description }}</p>

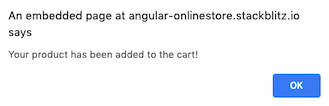
<button (click)="addToCart(product)">Buy</button>

</div>

1. Verify that the new **Buy** button appears as expected by refreshing the application and clicking on a product's name to display its details.



1. Click the **Buy** button to add the product to the stored list of items in the cart and display a confirmation message.



## **Create the cart view**

For customers to see their cart, you can create the cart view in two steps:

1. Create a cart component and configure routing to the new component.
2. Display the cart items.

### **Set up the cart component**

To create the cart view, follow the same steps you did to create the ProductDetailsComponent and configure routing for the new component.

1. Generate a cart component named cart by right-clicking the app folder, choosing **Angular Generator**, and **Component**.

src/app/cart/cart.component.ts

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

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

selector: 'app-cart',

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

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

})

export class CartComponent {

constructor() { }

}

StackBlitz also generates an ngOnInit() by default in components. You can ignore the CartComponent ngOnInit() for this tutorial.

1. Open app.module.ts and add a route for the component CartComponent, with a path of cart.

src/app/app.module.ts

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

imports: [

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

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

RouterModule.forRoot([

{ path: '', component: ProductListComponent },

{ path: 'products/:productId', component: ProductDetailsComponent },

{ path: 'cart', component: CartComponent },

])

],

1. Update the **Checkout** button so that it routes to the /cart URL. In top-bar.component.html, add a [routerLink](https://angular.io/api/router/RouterLink) directive pointing to /cart.

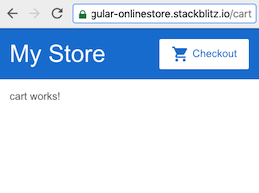
src/app/top-bar/top-bar.component.html

content\_copy<a [routerLink](https://angular.io/api/router/RouterLink)="/cart" class="button fancy-button">

<i class="material-icons">shopping\_cart</i>Checkout

</a>

1. Verify the new CartComponent works as expected by clicking the **Checkout** button. You can see the "cart works!" default text, and the URL has the pattern https://getting-started.stackblitz.io/cart, where getting-started.stackblitz.io may be different for your StackBlitz project.



### **Display the cart items**

This section shows you how to use the cart service to display the products in the cart.

1. In cart.component.ts, import the CartService from the cart.service.ts file.

src/app/cart/cart.component.ts

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

import { CartService } from '../cart.service';

1. Inject the CartService so that the CartComponent can use it by adding it to the constructor().

src/app/cart/cart.component.ts

content\_copyexport class CartComponent {

constructor(

private cartService: CartService

) { }

}

1. Define the items property to store the products in the cart.

src/app/cart/cart.component.ts

content\_copyexport class CartComponent {

items = this.cartService.getItems();

constructor(

private cartService: CartService

) { }

}

This code sets the items using the CartService getItems() method. You defined this method [when you created cart.service.ts](https://angular.io/start/start-data#generate-cart-service).

1. Update the cart template with a header, and use a <div> with an \*[ngFor](https://angular.io/api/common/NgForOf) to display each of the cart items with its name and price. The resulting CartComponent template is as follows.

src/app/cart/cart.component.html

content\_copy<h3>Cart</h3>

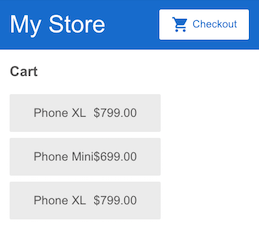
<div class="cart-item" \*[ngFor](https://angular.io/api/common/NgForOf)="let item of items">

<span>{{ item.name }}</span>

<span>{{ item.price | [currency](https://angular.io/api/common/CurrencyPipe) }}</span>

</div>

1. Verify that your cart works as expected:
   * Click **My Store**
   * Click on a product name to display its details.
   * Click **Buy** to add the product to the cart.
   * Click **Checkout** to see the cart.



For more information about services, see [Introduction to Services and Dependency Injection](https://angular.io/guide/architecture-services).

## **Retrieve shipping prices**

Servers often return data in the form of a stream. Streams are useful because they make it easy to transform the returned data and make modifications to the way you request that data. Angular [HttpClient](https://angular.io/api/common/http/HttpClient) is a built-in way to fetch data from external APIs and provide them to your application as a stream.

This section shows you how to use [HttpClient](https://angular.io/api/common/http/HttpClient) to retrieve shipping prices from an external file.

The application that StackBlitz generates for this guide comes with predefined shipping data in assets/shipping.json. Use this data to add shipping prices for items in the cart.

src/assets/shipping.json

content\_copy[

{

"type": "Overnight",

"price": 25.99

},

{

"type": "2-Day",

"price": 9.99

},

{

"type": "Postal",

"price": 2.99

}

]

### **Configure AppModule to use**[**HttpClient**](https://angular.io/api/common/http/HttpClient)

To use Angular's [HttpClient](https://angular.io/api/common/http/HttpClient), you must configure your application to use [HttpClientModule](https://angular.io/api/common/http/HttpClientModule).

Angular's [HttpClientModule](https://angular.io/api/common/http/HttpClientModule) registers the providers your application needs to use the [HttpClient](https://angular.io/api/common/http/HttpClient) service throughout your application.

1. In app.module.ts, import [HttpClientModule](https://angular.io/api/common/http/HttpClientModule) from the @angular/common/[http](https://angular.io/api/common/http) package at the top of the file with the other imports. As there are a number of other imports, this code snippet omits them for brevity. Be sure to leave the existing imports in place.

src/app/app.module.ts

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

1. To register Angular's [HttpClient](https://angular.io/api/common/http/HttpClient) providers globally, add [HttpClientModule](https://angular.io/api/common/http/HttpClientModule) to the AppModule @[NgModule](https://angular.io/api/core/NgModule)() imports array.

src/app/app.module.ts

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

imports: [

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

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

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

RouterModule.forRoot([

{ path: '', component: ProductListComponent },

{ path: 'products/:productId', component: ProductDetailsComponent },

{ path: 'cart', component: CartComponent },

])

],

declarations: [

AppComponent,

TopBarComponent,

ProductListComponent,

ProductAlertsComponent,

ProductDetailsComponent,

CartComponent,

],

bootstrap: [

AppComponent

]

})

export class AppModule { }

### **Configure CartService to use**[**HttpClient**](https://angular.io/api/common/http/HttpClient)

The next step is to inject the [HttpClient](https://angular.io/api/common/http/HttpClient) service into your service so your application can fetch data and interact with external APIs and resources.

1. In cart.service.ts, import [HttpClient](https://angular.io/api/common/http/HttpClient) from the @angular/common/[http](https://angular.io/api/common/http) package.

src/app/cart.service.ts

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

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

import { Product } from './products';

1. Inject [HttpClient](https://angular.io/api/common/http/HttpClient) into the CartService constructor().

src/app/cart.service.ts

content\_copyexport class CartService {

items: Product[] = [];

constructor(

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

) {}

}

### **Configure CartService to get shipping prices**

To get shipping data, from shipping.json, You can use the [HttpClient](https://angular.io/api/common/http/HttpClient) get() method.

1. In cart.service.ts, below the clearCart() method, define a new getShippingPrices() method that uses the [HttpClient](https://angular.io/api/common/http/HttpClient) get() method.

src/app/cart.service.ts

content\_copyexport class CartService {

getShippingPrices() {

return this.http.get<{type: string, price: number}[]>('/assets/shipping.json');

}

}

For more information about Angular's [HttpClient](https://angular.io/api/common/http/HttpClient), see the [Client-Server Interaction](https://angular.io/guide/http) guide.

## **Create a shipping component**

Now that you've configured your application to retrieve shipping data, you can create a place to render that data.

1. Generate a new component named shipping by right-clicking the app folder, choosing **Angular Generator**, and selecting **Component**.

src/app/shipping/shipping.component.ts

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

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

selector: 'app-shipping',

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

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

})

export class ShippingComponent {

constructor() { }

}

1. In app.module.ts, add a route for shipping. Specify a path of shipping and a component of ShippingComponent.

src/app/app.module.ts

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

imports: [

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

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

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

RouterModule.forRoot([

{ path: '', component: ProductListComponent },

{ path: 'products/:productId', component: ProductDetailsComponent },

{ path: 'cart', component: CartComponent },

{ path: 'shipping', component: ShippingComponent },

])

],

declarations: [

AppComponent,

TopBarComponent,

ProductListComponent,

ProductAlertsComponent,

ProductDetailsComponent,

CartComponent,

ShippingComponent

],

bootstrap: [

AppComponent

]

})

export class AppModule { }

There's no link to the new shipping component yet, but you can see its template in the preview pane by entering the URL its route specifies. The URL has the pattern: https://getting-started.stackblitz.io/shipping where the getting-started.stackblitz.io part may be different for your StackBlitz project.

### **Configuring the ShippingComponent to use CartService**

This section guides you through modifying the ShippingComponent to retrieve shipping data via HTTP from the shipping.json file.

1. In shipping.component.ts, import CartService.

src/app/shipping/shipping.component.ts

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

import { CartService } from '../cart.service';

1. Inject the cart service in the ShippingComponent constructor().

src/app/shipping/shipping.component.ts

content\_copyconstructor(private cartService: CartService) {

}

1. Define a shippingCosts property that sets the shippingCosts property using the getShippingPrices() method from the CartService.

src/app/shipping/shipping.component.ts

content\_copyexport class ShippingComponent {

shippingCosts = this.cartService.getShippingPrices();

}

1. Update the ShippingComponent template to display the shipping types and prices using the [async](https://angular.io/api/common/AsyncPipe) pipe.

src/app/shipping/shipping.component.html

content\_copy<h3>Shipping Prices</h3>

<div class="shipping-item" \*[ngFor](https://angular.io/api/common/NgForOf)="let shipping of shippingCosts | [async](https://angular.io/api/common/AsyncPipe)">

<span>{{ shipping.type }}</span>

<span>{{ shipping.price | [currency](https://angular.io/api/common/CurrencyPipe) }}</span>

</div>

The [async](https://angular.io/api/common/AsyncPipe) pipe returns the latest value from a stream of data and continues to do so for the life of a given component. When Angular destroys that component, the [async](https://angular.io/api/common/AsyncPipe) pipe automatically stops. For detailed information about the [async](https://angular.io/api/common/AsyncPipe) pipe, see the [AsyncPipe API documentation](https://angular.io/api/common/AsyncPipe).

1. Add a link from the CartComponent view to the ShippingComponent view.

src/app/cart/cart.component.html

content\_copy<h3>Cart</h3>

<p>

<a [routerLink](https://angular.io/api/router/RouterLink)="/shipping">Shipping Prices</a>

</p>

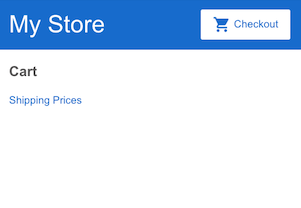
<div class="cart-item" \*[ngFor](https://angular.io/api/common/NgForOf)="let item of items">

<span>{{ item.name }}</span>

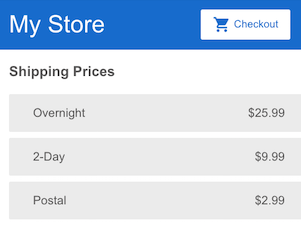
<span>{{ item.price | [currency](https://angular.io/api/common/CurrencyPipe) }}</span>

</div>

1. Click the **Checkout** button to see the updated cart. Remember that changing the application causes the preview to refresh, which empties the cart.



Click on the link to navigate to the shipping prices.



# Using forms for user input

This guide builds on the [Managing Data](https://angular.io/start/start-data) step of the Getting Started tutorial, [Get started with a basic Angular app](https://angular.io/start).

This section walks you through adding a form-based checkout feature to collect user information as part of checkout.

## **Define the checkout form model**

This step shows you how to set up the checkout form model in the component class. The form model determines the status of the form.

1. Open cart.component.ts.
2. Import the [FormBuilder](https://angular.io/api/forms/FormBuilder) service from the @angular/forms package. This service provides convenient methods for generating controls.

src/app/cart/cart.component.ts

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

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

import { CartService } from '../cart.service';

1. Inject the [FormBuilder](https://angular.io/api/forms/FormBuilder) service in the CartComponent constructor(). This service is part of the [ReactiveFormsModule](https://angular.io/api/forms/ReactiveFormsModule) module, which you've already imported.

src/app/cart/cart.component.ts

content\_copyexport class CartComponent {

constructor(

private cartService: CartService,

private formBuilder: [FormBuilder](https://angular.io/api/forms/FormBuilder),

) {}

}

1. To gather the user's name and address, use the [FormBuilder](https://angular.io/api/forms/FormBuilder) group() method to set the checkoutForm property to a form model containing name and address fields.

src/app/cart/cart.component.ts

content\_copyexport class CartComponent {

items = this.cartService.getItems();

checkoutForm = this.formBuilder.group({

name: '',

address: ''

});

constructor(

private cartService: CartService,

private formBuilder: [FormBuilder](https://angular.io/api/forms/FormBuilder),

) {}

}

1. Define an onSubmit() method to process the form. This method allows users to submit their name and address. In addition, this method uses the clearCart() method of the CartService to reset the form and clear the cart.

The entire cart component class is as follows:

src/app/cart/cart.component.ts

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

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

import { CartService } from '../cart.service';

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

selector: 'app-cart',

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

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

})

export class CartComponent {

items = this.cartService.getItems();

checkoutForm = this.formBuilder.group({

name: '',

address: ''

});

constructor(

private cartService: CartService,

private formBuilder: [FormBuilder](https://angular.io/api/forms/FormBuilder),

) {}

onSubmit(): void {

// Process checkout data here

this.items = this.cartService.clearCart();

console.warn('Your order has been submitted', this.checkoutForm.value);

this.checkoutForm.reset();

}

}

## **Create the checkout form**

Use the following steps to add a checkout form at the bottom of the Cart view.

1. At the bottom of cart.component.html, add an HTML <form> element and a **Purchase** button.
2. Use a formGroup property binding to bind checkoutForm to the HTML <form>.

src/app/cart/cart.component.html

content\_copy<form [formGroup]="checkoutForm">

<button class="button" type="submit">Purchase</button>

</form>

1. On the form tag, use an ngSubmit event binding to listen for the form submission and call the onSubmit() method with the checkoutForm value.

src/app/cart/cart.component.html (cart component template detail)

content\_copy<form [formGroup]="checkoutForm" (ngSubmit)="onSubmit()">

</form>

1. Add <input> fields for name and address, each with a [formControlName](https://angular.io/api/forms/FormControlName) attribute that binds to the checkoutForm form controls for name and address to their <input> fields. The complete component is as follows:

src/app/cart/cart.component.html

content\_copy<h3>Cart</h3>

<p>

<a [routerLink](https://angular.io/api/router/RouterLink)="/shipping">Shipping Prices</a>

</p>

<div class="cart-item" \*[ngFor](https://angular.io/api/common/NgForOf)="let item of items">

<span>{{ item.name }} </span>

<span>{{ item.price | [currency](https://angular.io/api/common/CurrencyPipe) }}</span>

</div>

<form [formGroup]="checkoutForm" (ngSubmit)="onSubmit()">

<div>

<label for="name">

Name

</label>

<input id="name" type="text" [formControlName](https://angular.io/api/forms/FormControlName)="name">

</div>

<div>

<label for="address">

Address

</label>

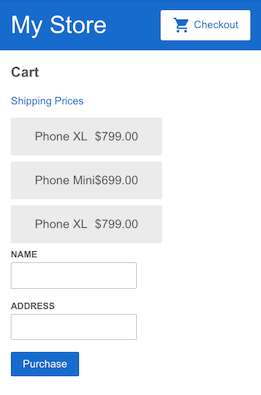
<input id="address" type="text" [formControlName](https://angular.io/api/forms/FormControlName)="address">

</div>

<button class="button" type="submit">Purchase</button>

</form>

After putting a few items in the cart, users can review their items, enter their name and address, and submit their purchase.



To confirm submission, open the console to see an object containing the name and address you submitted.

# Deploying an application

Deploying your application is the process of compiling, or building, your code and hosting the JavaScript, CSS, and HTML on a web server.

This section builds on the previous steps in the [Getting Started](https://angular.io/start) tutorial and shows you how to deploy your application.

## **Prerequisites**

A best practice is to run your project locally before you deploy it. To run your project locally, you need the following installed on your computer:

* [Node.js](https://nodejs.org/en/).
* The [Angular CLI](https://cli.angular.io/). From the terminal, install the Angular CLI globally with:

content\_copynpm install -g @angular/cli

With the Angular CLI, you can use the command ng to create new workspaces, new projects, serve your application during development, or produce builds to share or distribute.

## **Running your application locally**

1. Download the source code from your StackBlitz project by clicking the Download Project icon in the left menu, across from Project, to download your files.
2. Create a new Angular CLI workspace using the [ng new](https://angular.io/cli/new) command, where my-project-name is what you would like to call your project:

content\_copyng new my-project-name

This command displays a series of configuration prompts. For this tutorial, accept the default settings for each prompt.

1. In your newly CLI-generated application, replace the /src folder with the /src folder from your StackBlitz download.
2. Use the following CLI command to run your application locally:

content\_copyng serve

1. To see your application in the browser, go to http://localhost:4200/. If the default port 4200 is not available, you can specify another port with the port flag as in the following example:

content\_copyng serve --port 4201

While serving your application, you can edit your code and see the changes update automatically in the browser. To stop the ng serve command, press Ctrl+c.

## **Building and hosting your application**

1. To build your application for production, use the build command. By default, this command uses the production build configuration.

content\_copyng build

This command creates a dist folder in the application root directory with all the files that a hosting service needs for serving your application.

If the above ng build command throws an error about missing packages, append the missing dependencies in your local project's package.json file to match the one in the downloaded StackBlitz project.

1. Copy the contents of the dist/my-project-name folder to your web server. Because these files are static, you can host them on any web server capable of serving files; such as Node.js, Java, .NET, or any backend such as [Firebase](https://firebase.google.com/docs/hosting), [Google Cloud](https://cloud.google.com/solutions/web-hosting), or [App Engine](https://cloud.google.com/appengine/docs/standard/python/getting-started/hosting-a-static-website). For more information, see [Building & Serving](https://angular.io/guide/build) and [Deployment](https://angular.io/guide/deployment).

# Setting up the local environment and workspace

This guide explains how to set up your environment for Angular development using the [Angular CLI tool](https://angular.io/cli). It includes information about prerequisites, installing the CLI, creating an initial workspace and starter app, and running that app locally to verify your setup.

TRY ANGULAR WITHOUT LOCAL SETUP

If you are new to Angular, you might want to start with [Try it now!](https://angular.io/start), which introduces the essentials of Angular in the context of a ready-made basic online store app that you can examine and modify. This standalone tutorial takes advantage of the interactive [StackBlitz](https://stackblitz.com/) environment for online development. You don't need to set up your local environment until you're ready.

## **Prerequisites**

To use the Angular framework, you should be familiar with the following:

* [JavaScript](https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript)
* [HTML](https://developer.mozilla.org/docs/Learn/HTML/Introduction_to_HTML)
* [CSS](https://developer.mozilla.org/docs/Learn/CSS/First_steps)

Knowledge of [TypeScript](https://www.typescriptlang.org/) is helpful, but not required.

To install Angular on your local system, you need the following:

* **Node.js**

Angular requires an [active LTS or maintenance LTS](https://nodejs.org/about/releases) version of Node.js.

For information about specific version requirements, see the engines key in the [package.json](https://unpkg.com/browse/@angular/core/package.json) file.

For more information on installing Node.js, see [nodejs.org](https://nodejs.org/). If you are unsure what version of Node.js runs on your system, run node -v in a terminal window.

* **npm package manager**

Angular, the Angular CLI, and Angular applications depend on [npm packages](https://docs.npmjs.com/getting-started/what-is-npm) for many features and functions. To download and install npm packages, you need an npm package manager. This guide uses the [npm client](https://docs.npmjs.com/cli/install) command line interface, which is installed with Node.js by default. To check that you have the npm client installed, run npm -v in a terminal window.

## **Install the Angular CLI**

You use the Angular CLI to create projects, generate application and library code, and perform a variety of ongoing development tasks such as testing, bundling, and deployment.

To install the Angular CLI, open a terminal window and run the following command:

content\_copynpm install -g @angular/cli

## **Create a workspace and initial application**

You develop apps in the context of an Angular [**workspace**](https://angular.io/guide/glossary#workspace).

To create a new workspace and initial starter app:

1. Run the CLI command ng new and provide the name my-app, as shown here:

content\_copyng new my-app

1. The ng new command prompts you for information about features to include in the initial app. 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.

The CLI creates a new workspace and a simple Welcome app, ready to run.

## **Run the application**

The Angular CLI includes a server, so that you can build and serve your app locally.

1. Navigate to the workspace folder, such as my-app.
2. Run the following command:

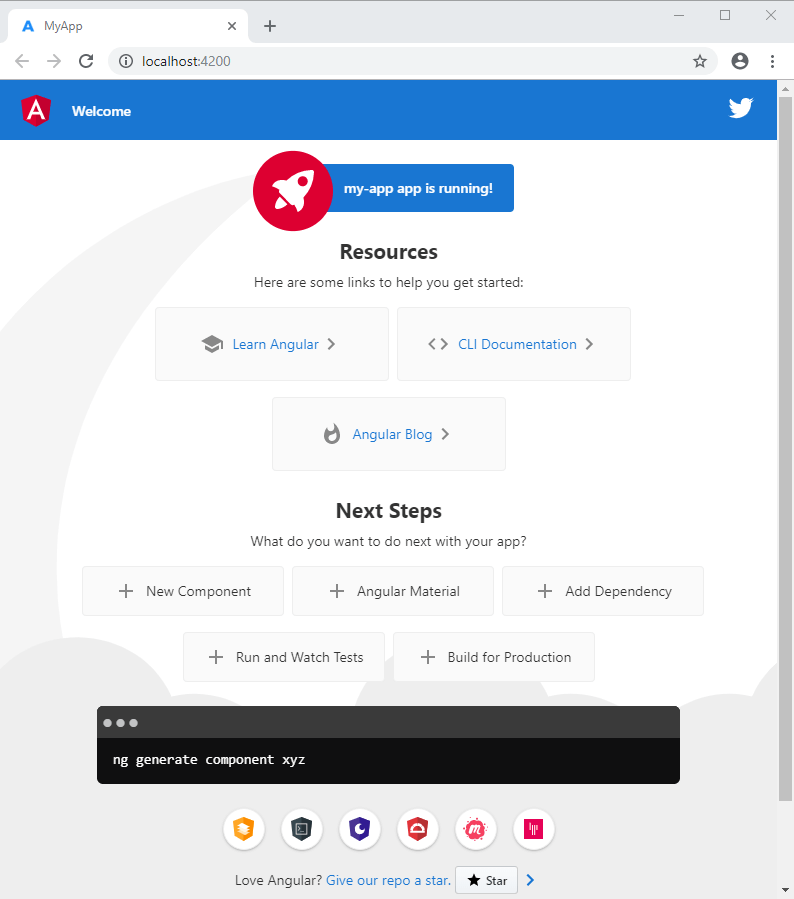
content\_copycd my-app

ng serve --open

The ng serve command launches the server, watches your files, and rebuilds the app as you make changes to those files.

The --open (or just -o) option automatically opens your browser to http://localhost:4200/.

If your installation and setup was successful, you should see a page similar to the following.



## **Next steps**

* For a more thorough introduction to the fundamental concepts and terminology of Angular single-page app architecture and design principles, read the [Angular Concepts](https://angular.io/guide/architecture) section.
* Work through the [Tour of Heroes Tutorial](https://angular.io/tutorial), a complete hands-on exercise that introduces you to the app development process using the Angular CLI and walks through important subsystems.
* To learn more about using the Angular CLI, see the [CLI Overview](https://angular.io/cli). In addition to creating the initial workspace and app scaffolding, you can use the CLI to generate Angular code such as components and services. The CLI supports the full development cycle, including building, testing, bundling, and deployment.
* For more information about the Angular files generated by ng new, see [Workspace and Project File Structure](https://angular.io/guide/file-structure).

# Angular Components Overview

Components are the main building block for Angular applications. Each component consists of:

* An HTML template that declares what renders on the page
* A Typescript class that defines behavior
* A CSS selector that defines how the component is used in a template
* Optionally, CSS styles applied to the template

This topic describes how to create and configure an Angular component.

To view or download the example code used in this topic, see the [live example](https://angular.io/generated/live-examples/component-overview/stackblitz.html) / [download example](https://angular.io/generated/zips/component-overview/component-overview.zip).

## **Prerequisites**

To create a component, verify that you have met the following prerequisites:

1. [Install the Angular CLI.](https://angular.io/guide/setup-local#install-the-angular-cli)
2. [Create an Angular workspace](https://angular.io/guide/setup-local#create-a-workspace-and-initial-application) with initial application. If you don't have a project, you can create one using ng new <project-name>, where <project-name> is the name of your Angular application.

## **Creating a component**

The easiest way to create a component is with the Angular CLI. You can also create a component manually.

### **Creating a component using the Angular CLI**

To create a component using the Angular CLI:

1. From a terminal window, navigate to the directory containing your application.
2. Run the ng generate component <component-name> command, where <component-name> is the name of your new component.

By default, this command creates the following:

* A folder named after the component
* A component file, <component-name>.component.ts
* A template file, <component-name>.component.html
* A CSS file, <component-name>.component.css
* A testing specification file, <component-name>.component.spec.ts

Where <component-name> is the name of your component.

You can change how ng generate component creates new components. For more information, see [ng generate component](https://angular.io/cli/generate#component-command) in the Angular CLI documentation.

### **Creating a component manually**

Although the Angular CLI is the easiest way to create an Angular component, you can also create a component manually. This section describes how to create the core component file within an existing Angular project.

To create a new component manually:

1. Navigate to your Angular project directory.
2. Create a new file, <component-name>.component.ts.
3. At the top of the file, add the following import statement.

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

1. After the import statement, add a @[Component](https://angular.io/api/core/Component) decorator.
2. content\_copy@[Component](https://angular.io/api/core/Component)({

})

1. Choose a CSS selector for the component.
2. content\_copy@[Component](https://angular.io/api/core/Component)({
3. selector: 'app-component-overview',

})

For more information on choosing a selector, see [Specifying a component's selector](https://angular.io/guide/component-overview#specifying-a-components-css-selector).

1. Define the HTML template that the component uses to display information. In most cases, this template is a separate HTML file.
2. content\_copy@[Component](https://angular.io/api/core/Component)({
3. selector: 'app-component-overview',
4. templateUrl: './component-overview.component.html',

})

For more information on defining a component's template, see [Defining a component's template](https://angular.io/guide/component-overview#defining-a-components-template).

1. Select the styles for the component's template. In most cases, you define the styles for your component's template in a separate file.
2. content\_copy@[Component](https://angular.io/api/core/Component)({
3. selector: 'app-component-overview',
4. templateUrl: './component-overview.component.html',
5. styleUrls: ['./component-overview.component.css']

})

1. Add a class statement that includes the code for the component.
2. content\_copyexport class ComponentOverviewComponent {

}

## **Specifying a component's CSS selector**

Every component requires a CSS selector. A selector instructs Angular to instantiate this component wherever it finds the corresponding tag in template HTML. For example, consider a component hello-world.component.ts that defines its selector as app-hello-world. This selector instructs Angular to instantiate this component any time the tag <app-hello-world> appears in a template.

Specify a component's selector by adding a selector statement to the @[Component](https://angular.io/api/core/Component) decorator.

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

selector: 'app-component-overview',

})

## **Defining a component's template**

A template is a block of HTML that tells Angular how to render the component in your application. You can define a template for your component in one of two ways: by referencing an external file, or directly within the component.

To define a template as an external file, add a templateUrl property to the @[Component](https://angular.io/api/core/Component) decorator.

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

selector: 'app-component-overview',

templateUrl: './component-overview.component.html',

})

To define a template within the component, add a template property to the @[Component](https://angular.io/api/core/Component) decorator that contains the HTML you want to use.

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

selector: 'app-component-overview',

template: '<h1>Hello World!</h1>',

})

If you want your template to span multiple lines, you can use backticks ( ` ). For example:

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

selector: 'app-component-overview',

template: `<h1>Hello World!</h1>

<p>This template definition spans

[multiple](https://angular.io/api/forms/SelectMultipleControlValueAccessor) lines.</p>`

})

An Angular component requires a template defined using template or templateUrl. You cannot have both statements in a component.

## **Declaring a component's styles**

You can declare component styles uses for its template in one of two ways: by referencing an external file, or directly within the component.

To declare the styles for a component in a separate file, add a styleUrls property to the @[Component](https://angular.io/api/core/Component) decorator.

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

selector: 'app-component-overview',

templateUrl: './component-overview.component.html',

styleUrls: ['./component-overview.component.css']

})

To declare the styles within the component, add a styles property to the @[Component](https://angular.io/api/core/Component) decorator that contains the styles you want to use.

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

selector: 'app-component-overview',

template: '<h1>Hello World!</h1>',

styles: ['h1 { font-weight: normal; }']

})

The styles property takes an array of strings that contain the CSS rule declarations.

## **Next steps**

* For an architectural overview of components, see [Introduction to components and templates](https://angular.io/guide/architecture-components).
* For additional options you can use when creating a component, see [Component](https://angular.io/api/core/Component) in the API Reference.
* For more information on styling components, see [Component styles](https://angular.io/guide/component-styles).
* For more information on templates, see [Template syntax](https://angular.io/guide/template-syntax).

# Lifecycle hooks

A component instance has a lifecycle that starts when Angular instantiates the component class and renders the component view along with its child views. The lifecycle continues with change detection, as Angular checks to see when data-bound properties change, and updates both the view and the component instance as needed. The lifecycle ends when Angular destroys the component instance and removes its rendered template from the DOM. Directives have a similar lifecycle, as Angular creates, updates, and destroys instances in the course of execution.

Your application can use [lifecycle hook methods](https://angular.io/guide/glossary#lifecycle-hook) to tap into key events in the lifecycle of a component or directive in order to initialize new instances, initiate change detection when needed, respond to updates during change detection, and clean up before deletion of instances.

## **Prerequisites**

Before working with lifecycle hooks, you should have a basic understanding of the following:

* [TypeScript programming](https://www.typescriptlang.org/).
* Angular app-design fundamentals, as described in [Angular Concepts](https://angular.io/guide/architecture).

## **Responding to lifecycle events**

You can respond to events in the lifecycle of a component or directive by implementing one or more of the lifecycle hook interfaces in the Angular core library. The hooks give you the opportunity to act on a component or directive instance at the appropriate moment, as Angular creates, updates, or destroys that instance.

Each interface defines the prototype for a single hook method, whose name is the interface name prefixed with ng. For example, the [OnInit](https://angular.io/api/core/OnInit) interface has a hook method named ngOnInit(). If you implement this method in your component or directive class, Angular calls it shortly after checking the input properties for that component or directive for the first time.

peek-a-boo.directive.ts (excerpt)

content\_copy@[Directive](https://angular.io/api/core/Directive)({selector: '[appPeekABoo]'})

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

constructor(private logger: LoggerService) { }

// implement [OnInit](https://angular.io/api/core/OnInit)'s `ngOnInit` method

ngOnInit() {

this.logIt(`[OnInit](https://angular.io/api/core/OnInit)`);

}

logIt(msg: string) {

this.logger.log(`#${nextId++} ${msg}`);

}

}

You don't have to implement all (or any) of the lifecycle hooks, just the ones you need.

### **Lifecycle event sequence**

After your application instantiates a component or directive by calling its constructor, Angular calls the hook methods you have implemented at the appropriate point in the lifecycle of that instance.

Angular executes hook methods in the following sequence. You can use them to perform the following kinds of operations.

|  |  |  |
| --- | --- | --- |
| **Hook method** | **Purpose** | **Timing** |
| ngOnChanges() | Respond when Angular sets or resets data-bound input properties. The method receives a [SimpleChanges](https://angular.io/api/core/SimpleChanges) object of current and previous property values.  Note that this happens very frequently, so any operation you perform here impacts performance significantly. See details in [Using change detection hooks](https://angular.io/guide/lifecycle-hooks#onchanges) in this document. | Called before ngOnInit() (if the component has bound inputs) and whenever one or more data-bound input properties change.  Note that if your component has no inputs or you use it without providing any inputs, the framework will not call ngOnChanges(). |
| ngOnInit() | Initialize the directive or component after Angular first displays the data-bound properties and sets the directive or component's input properties. See details in [Initializing a component or directive](https://angular.io/guide/lifecycle-hooks#oninit) in this document. | Called once, after the first ngOnChanges(). ngOnInit() is still called even when ngOnChanges() is not (which is the case when there are no template-bound inputs). |
| ngDoCheck() | Detect and act upon changes that Angular can't or won't detect on its own. See details and example in [Defining custom change detection](https://angular.io/guide/lifecycle-hooks#docheck) in this document. | Called immediately after ngOnChanges() on every change detection run, and immediately after ngOnInit() on the first run. |
| ngAfterContentInit() | Respond after Angular projects external content into the component's view, or into the view that a directive is in.  See details and example in [Responding to changes in content](https://angular.io/guide/lifecycle-hooks#aftercontent) in this document. | Called once after the first ngDoCheck(). |
| ngAfterContentChecked() | Respond after Angular checks the content projected into the directive or component.  See details and example in [Responding to projected content changes](https://angular.io/guide/lifecycle-hooks#aftercontent) in this document. | Called after ngAfterContentInit() and every subsequent ngDoCheck(). |
| ngAfterViewInit() | Respond after Angular initializes the component's views and child views, or the view that contains the directive.  See details and example in [Responding to view changes](https://angular.io/guide/lifecycle-hooks#afterview) in this document. | Called once after the first ngAfterContentChecked(). |
| ngAfterViewChecked() | Respond after Angular checks the component's views and child views, or the view that contains the directive. | Called after the ngAfterViewInit() and every subsequent ngAfterContentChecked(). |
| ngOnDestroy() | Cleanup just before Angular destroys the directive or component. Unsubscribe Observables and detach event handlers to avoid memory leaks. See details in [Cleaning up on instance destruction](https://angular.io/guide/lifecycle-hooks#ondestroy) in this document. | Called immediately before Angular destroys the directive or component. |

### **Lifecycle example set**

The [live example](https://angular.io/generated/live-examples/lifecycle-hooks/stackblitz.html) / [download example](https://angular.io/generated/zips/lifecycle-hooks/lifecycle-hooks.zip) demonstrates the use of lifecycle hooks through a series of exercises presented as components under the control of the root AppComponent. In each case a parent component serves as a test rig for a child component that illustrates one or more of the lifecycle hook methods.

The following table lists the exercises with brief descriptions. The sample code is also used to illustrate specific tasks in the following sections.

|  |  |
| --- | --- |
| **Component** | **Description** |
| [Peek-a-boo](https://angular.io/guide/lifecycle-hooks#peek-a-boo) | Demonstrates every lifecycle hook. Each hook method writes to the on-screen log. |
| [Spy](https://angular.io/guide/lifecycle-hooks#spy) | Shows how you can use lifecycle hooks with a custom directive. The SpyDirective implements the ngOnInit() and ngOnDestroy() hooks, and uses them to watch and report when an element goes in or out of the current view. |
| [OnChanges](https://angular.io/guide/lifecycle-hooks#onchanges) | Demonstrates how Angular calls the ngOnChanges() hook every time one of the component input properties changes, and shows how to interpret the changes object passed to the hook method. |
| [DoCheck](https://angular.io/guide/lifecycle-hooks#docheck) | Implements the ngDoCheck() method with custom change detection. Watch the hook post changes to a log to see how often Angular calls this hook. |
| [AfterView](https://angular.io/guide/lifecycle-hooks#afterview) | Shows what Angular means by a [view](https://angular.io/guide/glossary#view). Demonstrates the ngAfterViewInit() and ngAfterViewChecked() hooks. |
| [AfterContent](https://angular.io/guide/lifecycle-hooks#aftercontent) | Shows how to project external content into a component and how to distinguish projected content from a component's view children. Demonstrates the ngAfterContentInit() and ngAfterContentChecked() hooks. |
| [Counter](https://angular.io/guide/lifecycle-hooks#counter) | Demonstrates a combination of a component and a directive, each with its own hooks. |

## **Initializing a component or directive**

Use the ngOnInit() method to perform the following initialization tasks.

* Perform complex initializations outside of the constructor. Components should be cheap and safe to construct. You should not, for example, fetch data in a component constructor. You shouldn't worry that a new component will try to contact a remote server when created under test or before you decide to display it.

An ngOnInit() is a good place for a component to fetch its initial data. For an example, see the [Tour of Heroes tutorial](https://angular.io/tutorial/toh-pt4#oninit).

* Set up the component after Angular sets the input properties. Constructors should do no more than set the initial local variables to simple values.

Keep in mind that a directive's data-bound input properties are not set until after construction. If you need to initialize the directive based on those properties, set them when ngOnInit() runs.

The ngOnChanges() method is your first opportunity to access those properties. Angular calls ngOnChanges() before ngOnInit(), but also many times after that. It only calls ngOnInit() once.

## **Cleaning up on instance destruction**

Put cleanup logic in ngOnDestroy(), the logic that must run before Angular destroys the directive.

This is the place to free resources that won't be garbage-collected automatically. You risk memory leaks if you neglect to do so.

* Unsubscribe from Observables and DOM events.
* Stop interval timers.
* Unregister all callbacks that the directive registered with global or application services.

The ngOnDestroy() method is also the time to notify another part of the application that the component is going away.

## **General examples**

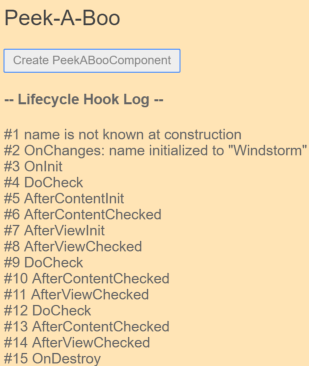
The following examples demonstrate the call sequence and relative frequency of the various lifecycle events, and how the hooks can be used separately or together for components and directives.

### **Sequence and frequency of all lifecycle events**

To show how Angular calls the hooks in the expected order, the PeekABooComponent demonstrates all of the hooks in one component.

In practice you would rarely, if ever, implement all of the interfaces the way this demo does.

The following snapshot reflects the state of the log after the user clicked the Create... button and then the Destroy... button.



The sequence of log messages follows the prescribed hook calling order: [OnChanges](https://angular.io/api/core/OnChanges), [OnInit](https://angular.io/api/core/OnInit), [DoCheck](https://angular.io/api/core/DoCheck) (3x), [AfterContentInit](https://angular.io/api/core/AfterContentInit), [AfterContentChecked](https://angular.io/api/core/AfterContentChecked) (3x), [AfterViewInit](https://angular.io/api/core/AfterViewInit), [AfterViewChecked](https://angular.io/api/core/AfterViewChecked) (3x), and [OnDestroy](https://angular.io/api/core/OnDestroy).

Notice that the log confirms that input properties (the name property in this case) have no assigned values at construction. The input properties are available to the onInit() method for further initialization.

Had the user clicked the Update Hero button, the log would show another [OnChanges](https://angular.io/api/core/OnChanges) and two more triplets of [DoCheck](https://angular.io/api/core/DoCheck), [AfterContentChecked](https://angular.io/api/core/AfterContentChecked) and [AfterViewChecked](https://angular.io/api/core/AfterViewChecked). Notice that these three hooks fire often, so it is important to keep their logic as lean as possible.

### **Use directives to watch the DOM**

The Spy example demonstrates how you can use hook method for directives as well as components. The SpyDirective implements two hooks, ngOnInit() and ngOnDestroy(), in order to discover when a watched element is in the current view.

This template applies the SpyDirective to a <div> in the [ngFor](https://angular.io/api/common/NgForOf) hero repeater managed by the parent SpyComponent.

The example does not perform any initialization or clean-up. It just tracks the appearance and disappearance of an element in the view by recording when the directive itself is instantiated and destroyed.

A spy directive like this can provide insight into a DOM object that you cannot change directly. You can't touch the implementation of a native <div>, or modify a third party component. You can, however watch these elements with a directive.

The directive defines ngOnInit() and ngOnDestroy() hooks that log messages to the parent using an injected LoggerService.

src/app/spy.directive.ts

content\_copylet nextId = 1;

// Spy on any element to which it is applied.

// Usage: <div appSpy>...</div>

@[Directive](https://angular.io/api/core/Directive)({selector: '[appSpy]'})

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

private id = nextId++;

constructor(private logger: LoggerService) { }

ngOnInit() {

this.logger.log(`Spy #${this.id} onInit`);

}

ngOnDestroy() {

this.logger.log(`Spy #${this.id} onDestroy`);

}

}

You can apply the spy to any native or component element, and see that it is initialized and destroyed at the same time as that element. Here it is attached to the repeated hero <div>:

src/app/spy.component.html

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

{{hero}}

</p>

Each spy's creation and destruction marks the appearance and disappearance of the attached hero <div> with an entry in the Hook Log. Adding a hero results in a new hero <div>. The spy's ngOnInit() logs that event.

The Reset button clears the heroes list. Angular removes all hero <div> elements from the DOM and destroys their spy directives at the same time. The spy's ngOnDestroy() method reports its last moments.

### **Use component and directive hooks together**

In this example, a CounterComponent uses the ngOnChanges() method to log a change every time the parent component increments its input counter property.

This example applies the SpyDirective from the previous example to the CounterComponent log, in order to watch the creation and destruction of log entries.

## **Using change detection hooks**

Angular calls the ngOnChanges() method of a component or directive whenever it detects changes to the input properties. The onChanges example demonstrates this by monitoring the [OnChanges](https://angular.io/api/core/OnChanges)() hook.

on-changes.component.ts (excerpt)

content\_copyngOnChanges(changes: [SimpleChanges](https://angular.io/api/core/SimpleChanges)) {

for (const propName in changes) {

const chng = changes[propName];

const cur = JSON.stringify(chng.currentValue);

const prev = JSON.stringify(chng.previousValue);

this.changeLog.push(`${propName}: currentValue = ${cur}, previousValue = ${prev}`);

}

}

The ngOnChanges() method takes an object that maps each changed property name to a [SimpleChange](https://angular.io/api/core/SimpleChange) object holding the current and previous property values. This hook iterates over the changed properties and logs them.

The example component, OnChangesComponent, has two input properties: hero and power.

src/app/on-changes.component.ts

content\_copy@[Input](https://angular.io/api/core/Input)() hero!: Hero;

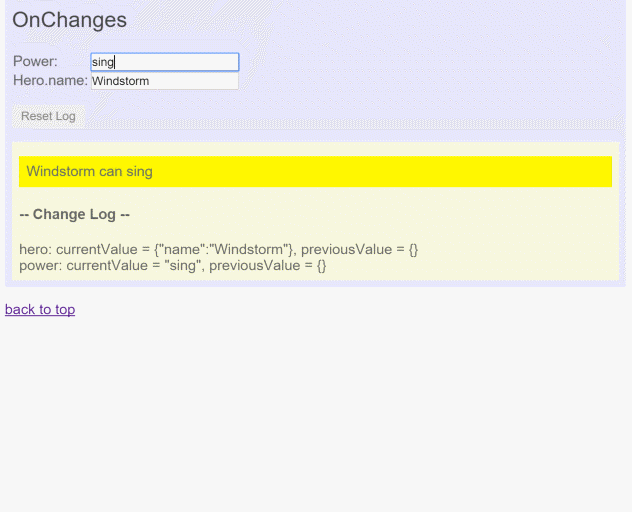
@[Input](https://angular.io/api/core/Input)() power = '';

The host OnChangesParentComponent binds to them as follows.

src/app/on-changes-parent.component.html

content\_copy<on-changes [hero]="hero" [power]="power"></on-changes>

Here's the sample in action as the user makes changes.



The log entries appear as the string value of the power property changes. Notice, however, that the ngOnChanges() method does not catch changes to hero.name. This is because Angular calls the hook only when the value of the input property changes. In this case, hero is the input property, and the value of the hero property is the reference to the hero object. The object reference did not change when the value of its own name property changed.

### **Responding to view changes**

As Angular traverses the [view hierarchy](https://angular.io/guide/glossary#view-hierarchy) during change detection, it needs to be sure that a change in a child does not attempt to cause a change in its own parent. Such a change would not be rendered properly, because of how [unidirectional data flow](https://angular.io/guide/glossary#unidirectional-data-flow) works.

If you need to make a change that inverts the expected data flow, you must trigger a new change detection cycle to allow that change to be rendered. The examples illustrate how to make such changes safely.

The AfterView sample explores the [AfterViewInit](https://angular.io/api/core/AfterViewInit)() and [AfterViewChecked](https://angular.io/api/core/AfterViewChecked)() hooks that Angular calls after it creates a component's child views.

Here's a child view that displays a hero's name in an <input>:

ChildViewComponent

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

selector: 'app-child-view',

template: `<label for="hero-name">Hero name: </label>

<input type="text" id="hero-name" [([ngModel](https://angular.io/api/forms/NgModel))]="hero">`

})

export class ChildViewComponent {

hero = 'Magneta';

}

The AfterViewComponent displays this child view within its template:

AfterViewComponent (template)

content\_copytemplate: `

<div>child view begins</div>

<app-child-view></app-child-view>

<div>child view ends</div>`

The following hooks take action based on changing values within the child view, which can only be reached by querying for the child view using the property decorated with [@ViewChild](https://angular.io/api/core/ViewChild).

AfterViewComponent (class excerpts)

content\_copyexport class AfterViewComponent implements [AfterViewChecked](https://angular.io/api/core/AfterViewChecked), [AfterViewInit](https://angular.io/api/core/AfterViewInit) {

private prevHero = '';

// [Query](https://angular.io/api/core/Query) for a VIEW child of type `ChildViewComponent`

@[ViewChild](https://angular.io/api/core/ViewChild)(ChildViewComponent) viewChild!: ChildViewComponent;

ngAfterViewInit() {

// viewChild is set after the view has been initialized

this.logIt('[AfterViewInit](https://angular.io/api/core/AfterViewInit)');

this.doSomething();

}

ngAfterViewChecked() {

// viewChild is updated after the view has been checked

if (this.prevHero === this.viewChild.hero) {

this.logIt('[AfterViewChecked](https://angular.io/api/core/AfterViewChecked) (no change)');

} else {

this.prevHero = this.viewChild.hero;

this.logIt('[AfterViewChecked](https://angular.io/api/core/AfterViewChecked)');

this.doSomething();

}

}

// ...

}

#### Wait before updating the view

In this example, the doSomething() method updates the screen when the hero name exceeds 10 characters, but waits a tick before updating comment.

AfterViewComponent (doSomething)

content\_copy// This surrogate for real business logic sets the `comment`

private doSomething() {

const c = this.viewChild.hero.length > 10 ? `That's a long name` : '';

if (c !== this.comment) {

// Wait a [tick](https://angular.io/api/core/testing/tick) because the component's view has already been checked

this.logger.tick\_then(() => this.comment = c);

}

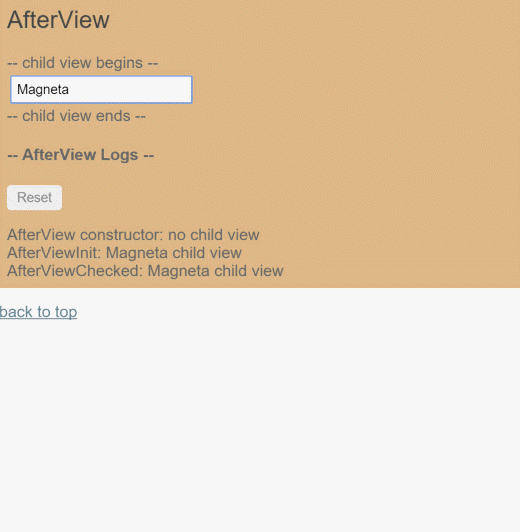
}

Both the [AfterViewInit](https://angular.io/api/core/AfterViewInit)() and [AfterViewChecked](https://angular.io/api/core/AfterViewChecked)() hooks fire after the component's view has been composed. If you modify the code so that the hook updates the component's data-bound comment property immediately, you can see that Angular throws an error.

The LoggerService.tick\_then() statement postpones the log update for one turn of the browser's JavaScript cycle, which triggers a new change-detection cycle.

#### Write lean hook methods to avoid performance problems

When you run the AfterView sample, notice how frequently Angular calls [AfterViewChecked](https://angular.io/api/core/AfterViewChecked)()-often when there are no changes of interest. Be very careful about how much logic or computation you put into one of these methods.



### **Responding to projected content changes**

Content projection is a way to import HTML content from outside the component and insert that content into the component's template in a designated spot. You can identify content projection in a template by looking for the following constructs.

* HTML between component element tags.
* The presence of [<ng-content>](https://angular.io/api/core/ng-content) tags in the component's template.

AngularJS developers know this technique as transclusion.

The AfterContent sample explores the [AfterContentInit](https://angular.io/api/core/AfterContentInit)() and [AfterContentChecked](https://angular.io/api/core/AfterContentChecked)() hooks that Angular calls after Angular projects external content into the component.

Consider this variation on the [previous AfterView](https://angular.io/guide/lifecycle-hooks#afterview) example. This time, instead of including the child view within the template, it imports the content from the AfterContentComponent's parent. The following is the parent's template.

AfterContentParentComponent (template excerpt)

content\_copy`<after-content>

<app-child></app-child>

</after-content>`

Notice that the <app-child> tag is tucked between the <after-content> tags. Never put content between a component's element tags unless you intend to project that content into the component.

Now look at the component's template.

AfterContentComponent (template)

content\_copytemplate: `

<div>projected content begins</div>

<ng-content></ng-content>

<div>projected content ends</div>`

The [<ng-content>](https://angular.io/api/core/ng-content) tag is a placeholder for the external content. It tells Angular where to insert that content. In this case, the projected content is the <app-child> from the parent.



#### Using AfterContent hooks

AfterContent hooks are similar to the AfterView hooks. The key difference is in the child component.

* The AfterView hooks concern [ViewChildren](https://angular.io/api/core/ViewChildren), the child components whose element tags appear within the component's template.
* The AfterContent hooks concern [ContentChildren](https://angular.io/api/core/ContentChildren), the child components that Angular projected into the component.

The following AfterContent hooks take action based on changing values in a content child, which can only be reached by querying for them using the property decorated with [@ContentChild](https://angular.io/api/core/ContentChild).

AfterContentComponent (class excerpts)

content\_copyexport class AfterContentComponent implements [AfterContentChecked](https://angular.io/api/core/AfterContentChecked), [AfterContentInit](https://angular.io/api/core/AfterContentInit) {

private prevHero = '';

comment = '';

// [Query](https://angular.io/api/core/Query) for a CONTENT child of type `ChildComponent`

@[ContentChild](https://angular.io/api/core/ContentChild)(ChildComponent) contentChild!: ChildComponent;

ngAfterContentInit() {

// contentChild is set after the content has been initialized

this.logIt('[AfterContentInit](https://angular.io/api/core/AfterContentInit)');

this.doSomething();

}

ngAfterContentChecked() {

// contentChild is updated after the content has been checked

if (this.prevHero === this.contentChild.hero) {

this.logIt('[AfterContentChecked](https://angular.io/api/core/AfterContentChecked) (no change)');

} else {

this.prevHero = this.contentChild.hero;

this.logIt('[AfterContentChecked](https://angular.io/api/core/AfterContentChecked)');

this.doSomething();

}

}

// ...

}

No need to wait for content updates

This component's doSomething() method updates the component's data-bound comment property immediately. There's no need to [delay the update to ensure proper rendering](https://angular.io/guide/lifecycle-hooks#wait-a-tick).

Angular calls both AfterContent hooks before calling either of the AfterView hooks. Angular completes composition of the projected content before finishing the composition of this component's view. There is a small window between the AfterContent... and AfterView... hooks that allows you to modify the host view.

## **Defining custom change detection**

To monitor changes that occur where ngOnChanges() won't catch them, you can implement your own change check, as shown in the DoCheck example. This example shows how you can use the ngDoCheck() hook to detect and act upon changes that Angular doesn't catch on its own.

The DoCheck sample extends the OnChanges sample with the following ngDoCheck() hook:

DoCheckComponent (ngDoCheck)

content\_copyngDoCheck() {

if (this.hero.name !== this.oldHeroName) {

this.changeDetected = true;

this.changeLog.push(`[DoCheck](https://angular.io/api/core/DoCheck): Hero name changed to "${this.hero.name}" from "${this.oldHeroName}"`);

this.oldHeroName = this.hero.name;

}

if (this.power !== this.oldPower) {

this.changeDetected = true;

this.changeLog.push(`[DoCheck](https://angular.io/api/core/DoCheck): Power changed to "${this.power}" from "${this.oldPower}"`);

this.oldPower = this.power;

}

if (this.changeDetected) {

this.noChangeCount = 0;

} else {

// log that hook was called when there was no relevant change.

const count = this.noChangeCount += 1;

const noChangeMsg = `[DoCheck](https://angular.io/api/core/DoCheck) called ${count}x when no change to hero or power`;

if (count === 1) {

// add new "no change" message

this.changeLog.push(noChangeMsg);

} else {

// update last "no change" message

this.changeLog[this.changeLog.length - 1] = noChangeMsg;

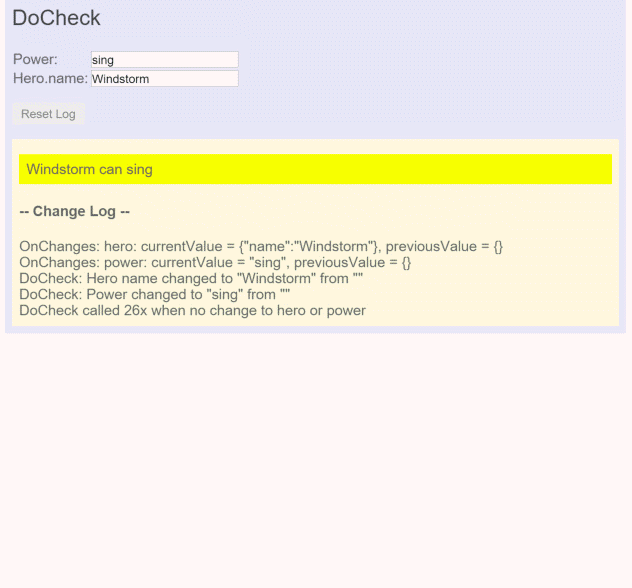
}

}

this.changeDetected = false;

}

This code inspects certain values of interest, capturing and comparing their current state against previous values. It writes a special message to the log when there are no substantive changes to the hero or the power so you can see how often [DoCheck](https://angular.io/api/core/DoCheck)() is called. The results are illuminating.



While the ngDoCheck() hook can detect when the hero's name has changed, it is very expensive. This hook is called with enormous frequency—after every change detection cycle no matter where the change occurred. It's called over twenty times in this example before the user can do anything.

Most of these initial checks are triggered by Angular's first rendering of unrelated data elsewhere on the page. Just moving the cursor into another <input> triggers a call. Relatively few calls reveal actual changes to pertinent data. If you use this hook, your implementation must be extremely lightweight or the user experience suffers.

# View encapsulation

In Angular, component CSS styles are encapsulated into the component's view and don't affect the rest of the application.

To control how this encapsulation happens on a per component basis, you can set the view encapsulation mode in the component metadata. Choose from the following modes:

* ShadowDom view encapsulation uses the browser's native shadow DOM implementation (see [Shadow DOM](https://developer.mozilla.org/en-US/docs/Web/Web_Components/Shadow_DOM)) to attach a shadow DOM to the component's host element, and then puts the component view inside that shadow DOM. The component's styles are included within the shadow DOM.
* Emulated view encapsulation (the default) emulates the behavior of shadow DOM by preprocessing (and renaming) the CSS code to effectively scope the CSS to the component's view. For details, see [Inspecting generated CSS](https://angular.io/guide/view-encapsulation#inspect-generated-css) below.
* None means that Angular does no view encapsulation. Angular adds the CSS to the global styles. The scoping rules, isolations, and protections discussed earlier don't apply. This mode is essentially the same as pasting the component's styles into the HTML.

To set the component's encapsulation mode, use the encapsulation property in the component metadata:

src/app/quest-summary.component.ts

content\_copy// warning: not all browsers support shadow DOM encapsulation at this time

encapsulation: [ViewEncapsulation.ShadowDom](https://angular.io/api/core/ViewEncapsulation#ShadowDom)

ShadowDom view encapsulation only works on browsers that have native support for shadow DOM (see [Can I use - Shadow DOM v1](https://caniuse.com/shadowdomv1)). The support is still limited, which is why Emulated view encapsulation is the default mode and recommended in most cases.

## **Inspecting generated CSS**

When using emulated view encapsulation, Angular preprocesses all component styles so that they approximate the standard shadow CSS scoping rules.

In the DOM of a running Angular application with emulated view encapsulation enabled, each DOM element has some extra attributes attached to it:

content\_copy<hero-details \_nghost-pmm-5>

<h2 \_ngcontent-pmm-5>Mister Fantastic</h2>

<hero-team \_ngcontent-pmm-5 \_nghost-pmm-6>

<h3 \_ngcontent-pmm-6>Team</h3>

</hero-team>

</hero-detail>

There are two kinds of generated attributes:

* An element that would be a shadow DOM host in native encapsulation has a generated \_nghost attribute. This is typically the case for component host elements.
* An element within a component's view has a \_ngcontent attribute that identifies to which host's emulated shadow DOM this element belongs.

The exact values of these attributes aren't important. They are automatically generated and you should never refer to them in application code. But they are targeted by the generated component styles, which are in the <head> section of the DOM:

content\_copy[\_nghost-pmm-5] {

display: block;

border: 1px solid black;

}

h3[\_ngcontent-pmm-6] {

background-color: white;

border: 1px solid #777;

}

These styles are post-processed so that each selector is augmented with \_nghost or \_ngcontent attribute selectors. These extra selectors enable the scoping rules described in this page.

## **Mixing encapsulation modes**

Avoid mixing components that use different view encapsulation. Where it is necessary, you should be aware of how the component styles will interact.

* The styles of components with [ViewEncapsulation.Emulated](https://angular.io/api/core/ViewEncapsulation#Emulated) are added to the <head> of the document, making them available throughout the application, but are "scoped" so they only affect elements within the component's template.
* The styles of components with [ViewEncapsulation.None](https://angular.io/api/core/ViewEncapsulation#None) are added to the <head> of the document, making them available throughout the application, and are not "scoped" so they can affect any element in the application.
* The styles of components with [ViewEncapsulation.ShadowDom](https://angular.io/api/core/ViewEncapsulation#ShadowDom) are only added to the shadow DOM host, ensuring that they only affect elements within the component's template.

**All the styles for**[ViewEncapsulation.Emulated](https://angular.io/api/core/ViewEncapsulation#Emulated)**and**[ViewEncapsulation.None](https://angular.io/api/core/ViewEncapsulation#None)**components are also added to the shadow DOM host of each**[ViewEncapsulation.ShadowDom](https://angular.io/api/core/ViewEncapsulation#ShadowDom)**component.**

The result is that styling for components with [ViewEncapsulation.None](https://angular.io/api/core/ViewEncapsulation#None) will affect matching elements within the shadow DOM.

This approach may seem counter-intuitive at first, but without it a component with [ViewEncapsulation.None](https://angular.io/api/core/ViewEncapsulation#None) could not be used within a component with [ViewEncapsulation.ShadowDom](https://angular.io/api/core/ViewEncapsulation#ShadowDom), since its styles would not be available.

### **Examples**

This section shows examples of how the styling of components with different [ViewEncapsulation](https://angular.io/api/core/ViewEncapsulation) interact.

See the [live example](https://angular.io/generated/live-examples/view-encapsulation/stackblitz.html) to try out these components yourself.

#### No encapsulation

The first example shows a component that has [ViewEncapsulation.None](https://angular.io/api/core/ViewEncapsulation#None). This component colors its template elements red.

src/app/no-encapsulation.component.ts

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

selector: 'app-no-encapsulation',

template: `

<h2>None</h2>

<div class="none-message">No encapsulation</div>

`,

styles: ['h2, .none-message { color: red; }'],

encapsulation: [ViewEncapsulation.None](https://angular.io/api/core/ViewEncapsulation#None),

})

export class NoEncapsulationComponent { }

Angular adds the styles for this component as global styles to the <head> of the document.

**Angular also adds the styles to all shadow DOM hosts.** Therefore, the styles are available throughout the application.



#### Emulated encapsulation

The second example shows a component that has [ViewEncapsulation.Emulated](https://angular.io/api/core/ViewEncapsulation#Emulated). This component colors its template elements green.

src/app/emulated-encapsulation.component.ts

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

selector: 'app-emulated-encapsulation',

template: `

<h2>Emulated</h2>

<div class="emulated-message">Emulated encapsulation</div>

<app-no-encapsulation></app-no-encapsulation>

`,

styles: ['h2, .emulated-message { color: green; }'],

encapsulation: [ViewEncapsulation.Emulated](https://angular.io/api/core/ViewEncapsulation#Emulated),

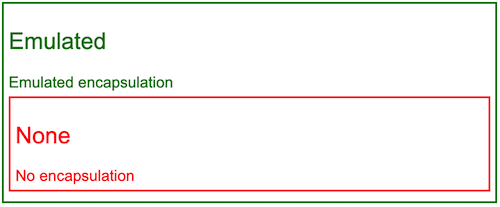
})

export class EmulatedEncapsulationComponent { }

Similar to [ViewEncapsulation.None](https://angular.io/api/core/ViewEncapsulation#None), Angular adds the styles for this component to the <head> of the document, and to all the shadow DOM hosts. But in this case, the styles are "scoped" by the attributes described in ["Inspecting generated CSS"](https://angular.io/guide/view-encapsulation#inspecting-generated-css).

Therefore, only the elements directly within this component's template will match its styles. Since the "scoped" styles from the EmulatedEncapsulationComponent are very specific, they override the global styles from the NoEncapsulationComponent.

In this example, the EmulatedEncapsulationComponent contains a NoEncapsulationComponent. The NoEncapsulationComponent is styled as expected because the scoped styles do not match elements in its template.



#### Shadow DOM encapsulation

The third example shows a component that has [ViewEncapsulation.ShadowDom](https://angular.io/api/core/ViewEncapsulation#ShadowDom). This component colors its template elements blue.

src/app/shadow-dom-encapsulation.component.ts

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

selector: 'app-shadow-dom-encapsulation',

template: `

<h2>ShadowDom</h2>

<div class="shadow-message">Shadow DOM encapsulation</div>

<app-emulated-encapsulation></app-emulated-encapsulation>

<app-no-encapsulation></app-no-encapsulation>

`,

styles: ['h2, .shadow-message { color: blue; }'],

encapsulation: [ViewEncapsulation.ShadowDom](https://angular.io/api/core/ViewEncapsulation#ShadowDom),

})

export class ShadowDomEncapsulationComponent { }

Angular adds styles for this component only to the shadow DOM host, so they are not visible outside the shadow DOM.

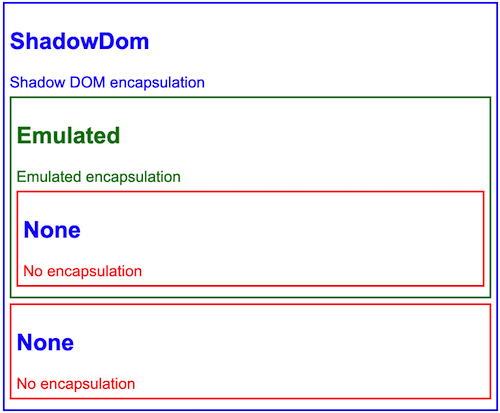
Note that Angular also adds the global styles from the NoEncapsulationComponent and ViewEncapsulationComponent to the shadow DOM host, so those styles are still available to the elements in the template of this component.

In this example, the ShadowDomEncapsulationComponent contains both a NoEncapsulationComponent and ViewEncapsulationComponent.

The styles added by the ShadowDomEncapsulationComponent component are available throughout the shadow DOM of this component, and so to both the NoEncapsulationComponent and ViewEncapsulationComponent.

The EmulatedEncapsulationComponent has specific "scoped" styles, so the styling of this component's template is unaffected.

But since styles from ShadowDomEncapsulationComponent are added to the shadow host after the global styles, the h2 style overrides the style from the NoEncapsulationComponent. The result is that the <h2> element in the NoEncapsulationComponent is colored blue rather than red, which may not be what the component author intended.



# Component interaction

This cookbook contains recipes for common component communication scenarios in which two or more components share information.

**See the**[live example](https://angular.io/generated/live-examples/component-interaction/stackblitz.html) / [download example](https://angular.io/generated/zips/component-interaction/component-interaction.zip).

## **Pass data from parent to child with input binding**

HeroChildComponent has two input properties, typically adorned with [@Input() decorator](https://angular.io/guide/inputs-outputs#input).

component-interaction/src/app/hero-child.component.ts

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

import { Hero } from './hero';

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

selector: 'app-hero-child',

template: `

<h3>{{hero.name}} says:</h3>

<p>I, {{hero.name}}, am at your service, {{masterName}}.</p>

`

})

export class HeroChildComponent {

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

@[Input](https://angular.io/api/core/Input)('master') masterName = ''; // tslint:disable-line: no-input-rename

}

The second @[Input](https://angular.io/api/core/Input) aliases the child component property name masterName as 'master'.

The HeroParentComponent nests the child HeroChildComponent inside an \*[ngFor](https://angular.io/api/common/NgForOf) repeater, binding its master string property to the child's master alias, and each iteration's hero instance to the child's hero property.

component-interaction/src/app/hero-parent.component.ts

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

import { HEROES } from './hero';

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

selector: 'app-hero-parent',

template: `

<h2>{{master}} controls {{heroes.length}} heroes</h2>

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

[hero]="hero"

[master]="master">

</app-hero-child>

`

})

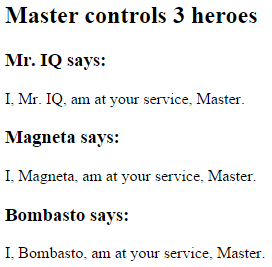
export class HeroParentComponent {

heroes = HEROES;

master = 'Master';

}

The running application displays three heroes:



### **Test it**

E2E test that all children were instantiated and displayed as expected:

component-interaction/e2e/src/app.e2e-spec.ts

content\_copy// ...

const heroNames = ['Dr IQ', 'Magneta', 'Bombasto'];

const masterName = 'Master';

it('should pass properties to children properly', async () => {

const parent = element(by.tagName('app-hero-parent'));

const heroes = parent.all(by.tagName('app-hero-child'));

for (let i = 0; i < heroNames.length; i++) {

const childTitle = await heroes.get(i).element(by.tagName('h3')).getText();

const childDetail = await heroes.get(i).element(by.tagName('p')).getText();

expect(childTitle).toEqual(heroNames[i] + ' says:');

expect(childDetail).toContain(masterName);

}

});

// ...

[Back to top](https://angular.io/guide/component-interaction#top)

## **Intercept input property changes with a setter**

Use an input property setter to intercept and act upon a value from the parent.

The setter of the name input property in the child NameChildComponent trims the whitespace from a name and replaces an empty value with default text.

component-interaction/src/app/name-child.component.ts

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

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

selector: 'app-name-child',

template: '<h3>"{{name}}"</h3>'

})

export class NameChildComponent {

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

get name(): string { return this.\_name; }

set name(name: string) {

this.\_name = (name && name.trim()) || '<no name set>';

}

private \_name = '';

}

Here's the NameParentComponent demonstrating name variations including a name with all spaces:

component-interaction/src/app/name-parent.component.ts

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

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

selector: 'app-name-parent',

template: `

<h2>Master controls {{names.length}} names</h2>

<app-name-child \*[ngFor](https://angular.io/api/common/NgForOf)="let name of names" [name]="name"></app-name-child>

`

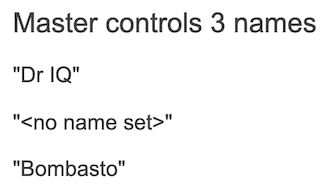
})

export class NameParentComponent {

// Displays 'Dr IQ', '<no name set>', 'Bombasto'

names = ['Dr IQ', ' ', ' Bombasto '];

}



### **Test it**

E2E tests of input property setter with empty and non-empty names:

component-interaction/e2e/src/app.e2e-spec.ts

content\_copy// ...

it('should display trimmed, non-empty names', async () => {

const nonEmptyNameIndex = 0;

const nonEmptyName = '"Dr IQ"';

const parent = element(by.tagName('app-name-parent'));

const hero = parent.all(by.tagName('app-name-child')).get(nonEmptyNameIndex);

const displayName = await hero.element(by.tagName('h3')).getText();

expect(displayName).toEqual(nonEmptyName);

});

it('should replace empty name with default name', async () => {

const emptyNameIndex = 1;

const defaultName = '"<no name set>"';

const parent = element(by.tagName('app-name-parent'));

const hero = parent.all(by.tagName('app-name-child')).get(emptyNameIndex);

const displayName = await hero.element(by.tagName('h3')).getText();

expect(displayName).toEqual(defaultName);

});

// ...

[Back to top](https://angular.io/guide/component-interaction#top)

## **Intercept input property changes with ngOnChanges()**

Detect and act upon changes to input property values with the ngOnChanges() method of the [OnChanges](https://angular.io/api/core/OnChanges) lifecycle hook interface.

You may prefer this approach to the property setter when watching multiple, interacting input properties.

Learn about ngOnChanges() in the [Lifecycle Hooks](https://angular.io/guide/lifecycle-hooks) chapter.

This VersionChildComponent detects changes to the major and minor input properties and composes a log message reporting these changes:

component-interaction/src/app/version-child.component.ts

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

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

selector: 'app-version-child',

template: `

<h3>[Version](https://angular.io/api/core/Version) {{major}}.{{minor}}</h3>

<h4>Change log:</h4>

<ul>

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

</ul>

`

})

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

@[Input](https://angular.io/api/core/Input)() major = 0;

@[Input](https://angular.io/api/core/Input)() minor = 0;

changeLog: string[] = [];

ngOnChanges(changes: [SimpleChanges](https://angular.io/api/core/SimpleChanges)) {

const log: string[] = [];

for (const propName in changes) {

const changedProp = changes[propName];

const to = JSON.stringify(changedProp.currentValue);

if (changedProp.isFirstChange()) {

log.push(`Initial value of ${propName} set to ${to}`);

} else {

const from = JSON.stringify(changedProp.previousValue);

log.push(`${propName} changed from ${from} to ${to}`);

}

}

this.changeLog.push(log.join(', '));

}

}

The VersionParentComponent supplies the minor and major values and binds buttons to methods that change them.

component-interaction/src/app/version-parent.component.ts

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

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

selector: 'app-version-parent',

template: `

<h2>Source code version</h2>

<button (click)="newMinor()">New minor version</button>

<button (click)="newMajor()">New major version</button>

<app-version-child [major]="major" [minor]="minor"></app-version-child>

`

})

export class VersionParentComponent {

major = 1;

minor = 23;

newMinor() {

this.minor++;

}

newMajor() {

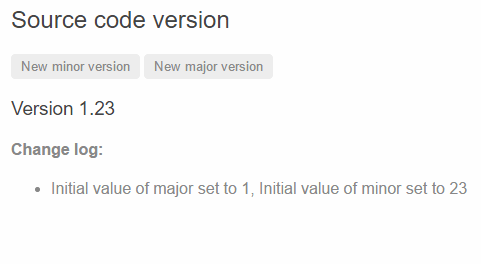
this.major++;

this.minor = 0;

}

}

Here's the output of a button-pushing sequence:



### **Test it**

Test that both input properties are set initially and that button clicks trigger the expected ngOnChanges calls and values:

component-interaction/e2e/src/app.e2e-spec.ts

content\_copy// ...

// Test must all execute in this exact order

it('should set expected initial values', async () => {

const actual = await getActual();

const initialLabel = '[Version](https://angular.io/api/core/Version) 1.23';

const initialLog = 'Initial value of major set to 1, Initial value of minor set to 23';

expect(actual.label).toBe(initialLabel);

expect(actual.count).toBe(1);

expect(await actual.logs.get(0).getText()).toBe(initialLog);

});

it('should set expected values after clicking \'Minor\' twice', async () => {

const repoTag = element(by.tagName('app-version-parent'));

const newMinorButton = repoTag.all(by.tagName('button')).get(0);

await newMinorButton.click();

await newMinorButton.click();

const actual = await getActual();

const labelAfter2Minor = '[Version](https://angular.io/api/core/Version) 1.25';

const logAfter2Minor = 'minor changed from 24 to 25';

expect(actual.label).toBe(labelAfter2Minor);

expect(actual.count).toBe(3);

expect(await actual.logs.get(2).getText()).toBe(logAfter2Minor);

});

it('should set expected values after clicking \'Major\' once', async () => {

const repoTag = element(by.tagName('app-version-parent'));

const newMajorButton = repoTag.all(by.tagName('button')).get(1);

await newMajorButton.click();

const actual = await getActual();

const labelAfterMajor = '[Version](https://angular.io/api/core/Version) 2.0';

const logAfterMajor = 'major changed from 1 to 2, minor changed from 23 to 0';

expect(actual.label).toBe(labelAfterMajor);

expect(actual.count).toBe(2);

expect(await actual.logs.get(1).getText()).toBe(logAfterMajor);

});

async function getActual() {

const versionTag = element(by.tagName('app-version-child'));

const label = await versionTag.element(by.tagName('h3')).getText();

const ul = versionTag.element((by.tagName('ul')));

const logs = ul.all(by.tagName('li'));

return {

label,

logs,

count: await logs.count(),

};

}

// ...

[Back to top](https://angular.io/guide/component-interaction#top)

## **Parent listens for child event**

The child component exposes an [EventEmitter](https://angular.io/api/core/EventEmitter) property with which it emits events when something happens. The parent binds to that event property and reacts to those events.

The child's [EventEmitter](https://angular.io/api/core/EventEmitter) property is an output property, typically adorned with an [@Output() decorator](https://angular.io/guide/inputs-outputs#output) as seen in this VoterComponent:

component-interaction/src/app/voter.component.ts

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

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

selector: 'app-voter',

template: `

<h4>{{name}}</h4>

<button (click)="vote(true)" [disabled]="didVote">Agree</button>

<button (click)="vote(false)" [disabled]="didVote">Disagree</button>

`

})

export class VoterComponent {

@[Input](https://angular.io/api/core/Input)() name = '';

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

didVote = false;

vote(agreed: boolean) {

this.voted.emit(agreed);

this.didVote = true;

}

}

Clicking a button triggers emission of a true or false, the boolean payload.

The parent VoteTakerComponent binds an event handler called onVoted() that responds to the child event payload $event and updates a counter.

component-interaction/src/app/votetaker.component.ts

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

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

selector: 'app-vote-taker',

template: `

<h2>Should mankind colonize the Universe?</h2>

<h3>Agree: {{agreed}}, Disagree: {{disagreed}}</h3>

<app-voter \*[ngFor](https://angular.io/api/common/NgForOf)="let voter of voters"

[name]="voter"

(voted)="onVoted($event)">

</app-voter>

`

})

export class VoteTakerComponent {

agreed = 0;

disagreed = 0;

voters = ['Narco', 'Celeritas', 'Bombasto'];

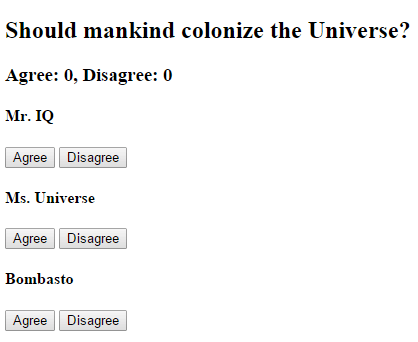
onVoted(agreed: boolean) {

agreed ? this.agreed++ : this.disagreed++;

}

}

The framework passes the event argument—represented by $event—to the handler method, and the method processes it:



### **Test it**

Test that clicking the Agree and Disagree buttons update the appropriate counters:

component-interaction/e2e/src/app.e2e-spec.ts

content\_copy// ...

it('should not emit the event initially', async () => {

const voteLabel = element(by.tagName('app-vote-taker')).element(by.tagName('h3'));

expect(await voteLabel.getText()).toBe('Agree: 0, Disagree: 0');

});

it('should process Agree vote', async () => {

const voteLabel = element(by.tagName('app-vote-taker')).element(by.tagName('h3'));

const agreeButton1 = element.all(by.tagName('app-voter')).get(0)

.all(by.tagName('button')).get(0);

await agreeButton1.click();

expect(await voteLabel.getText()).toBe('Agree: 1, Disagree: 0');

});

it('should process Disagree vote', async () => {

const voteLabel = element(by.tagName('app-vote-taker')).element(by.tagName('h3'));

const agreeButton1 = element.all(by.tagName('app-voter')).get(1)

.all(by.tagName('button')).get(1);

await agreeButton1.click();

expect(await voteLabel.getText()).toBe('Agree: 0, Disagree: 1');

});

// ...

[Back to top](https://angular.io/guide/component-interaction#top)

## **Parent interacts with child using local variable**

A parent component cannot use data binding to read child properties or invoke child methods. You can do both by creating a template reference variable for the child element and then reference that variable within the parent template as seen in the following example.

The following is a child CountdownTimerComponent that repeatedly counts down to zero and launches a rocket. It has start and stop methods that control the clock and it displays a countdown status message in its own template.

component-interaction/src/app/countdown-timer.component.ts

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

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

selector: 'app-countdown-timer',

template: '<p>{{message}}</p>'

})

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

intervalId = 0;

message = '';

seconds = 11;

ngOnDestroy() { this.clearTimer(); }

start() { this.countDown(); }

stop() {

this.clearTimer();

this.message = `Holding at T-${this.seconds} seconds`;

}

private clearTimer() { clearInterval(this.intervalId); }

private countDown() {

this.clearTimer();

this.intervalId = window.setInterval(() => {

this.seconds -= 1;

if (this.seconds === 0) {

this.message = 'Blast off!';

} else {

if (this.seconds < 0) { this.seconds = 10; } // reset

this.message = `T-${this.seconds} seconds and counting`;

}

}, 1000);

}

}

The CountdownLocalVarParentComponent that hosts the timer component is as follows:

component-interaction/src/app/countdown-parent.component.ts

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

import { CountdownTimerComponent } from './countdown-timer.component';

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

selector: 'app-countdown-parent-lv',

template: `

<h3>Countdown to Liftoff (via local variable)</h3>

<button (click)="timer.start()">Start</button>

<button (click)="timer.stop()">Stop</button>

<div class="seconds">{{timer.seconds}}</div>

<app-countdown-timer #timer></app-countdown-timer>

`,

styleUrls: ['../assets/demo.css']

})

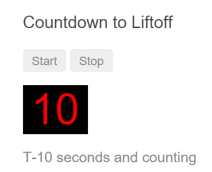
export class CountdownLocalVarParentComponent { }

The parent component cannot data bind to the child's start and stop methods nor to its seconds property.

You can place a local variable, #timer, on the tag <countdown-timer> representing the child component. That gives you a reference to the child component and the ability to access any of its properties or methods from within the parent template.

This example wires parent buttons to the child's start and stop and uses interpolation to display the child's seconds property.

Here we see the parent and child working together.



### **Test it**

Test that the seconds displayed in the parent template match the seconds displayed in the child's status message. Test also that clicking the Stop button pauses the countdown timer:

component-interaction/e2e/src/app.e2e-spec.ts

content\_copy// ...

// The tests [trigger](https://angular.io/api/animations/trigger) periodic asynchronous operations (via `setInterval()`), which will prevent

// the app from stabilizing. See https://angular.io/api/core/[ApplicationRef](https://angular.io/api/core/ApplicationRef)#is-stable-examples

// for more details.

// To allow the tests to complete, we will disable automatically waiting for the Angular app to

// stabilize.

beforeEach(() => browser.waitForAngularEnabled(false));

afterEach(() => browser.waitForAngularEnabled(true));

it('timer and parent seconds should match', async () => {

const parent = element(by.tagName(parentTag));

const startButton = parent.element(by.buttonText('Start'));

const seconds = parent.element(by.className('seconds'));

const timer = parent.element(by.tagName('app-countdown-timer'));

await startButton.click();

// Wait for `<app-countdown-timer>` to be populated with any text.

await browser.wait(() => timer.getText(), 2000);

expect(await timer.getText()).toContain(await seconds.getText());

});

it('should stop the countdown', async () => {

const parent = element(by.tagName(parentTag));

const startButton = parent.element(by.buttonText('Start'));

const stopButton = parent.element(by.buttonText('Stop'));

const timer = parent.element(by.tagName('app-countdown-timer'));

await startButton.click();

expect(await timer.getText()).not.toContain('Holding');

await stopButton.click();

expect(await timer.getText()).toContain('Holding');

});

// ...

[Back to top](https://angular.io/guide/component-interaction#top)

## **Parent calls an @ViewChild()**

The local variable approach is easy. But it is limited because the parent-child wiring must be done entirely within the parent template. The parent component itself has no access to the child.

You can't use the local variable technique if the parent component's class relies on the child component's class. The parent-child relationship of the components is not established within each components respective class with the local variable technique. Since the class instances are not connected to one another, the parent class cannot access the child class properties and methods.

When the parent component class requires that kind of access, inject the child component into the parent as a ViewChild.

The following example illustrates this technique with the same [Countdown Timer](https://angular.io/guide/component-interaction#countdown-timer-example) example. Neither its appearance nor its behavior will change. The child [CountdownTimerComponent](https://angular.io/guide/component-interaction#countdown-timer-example) is the same as well.

The switch from the local variable to the ViewChild technique is solely for the purpose of demonstration.

Here is the parent, CountdownViewChildParentComponent:

component-interaction/src/app/countdown-parent.component.ts

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

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

import { CountdownTimerComponent } from './countdown-timer.component';

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

selector: 'app-countdown-parent-vc',

template: `

<h3>Countdown to Liftoff (via [ViewChild](https://angular.io/api/core/ViewChild))</h3>

<button (click)="start()">Start</button>

<button (click)="stop()">Stop</button>

<div class="seconds">{{ seconds() }}</div>

<app-countdown-timer></app-countdown-timer>

`,

styleUrls: ['../assets/demo.css']

})

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

@[ViewChild](https://angular.io/api/core/ViewChild)(CountdownTimerComponent)

private timerComponent!: CountdownTimerComponent;

seconds() { return 0; }

ngAfterViewInit() {

// Redefine `seconds()` to get from the `CountdownTimerComponent.seconds` ...

// but wait a [tick](https://angular.io/api/core/testing/tick) first to avoid one-time devMode

// unidirectional-data-flow-violation error

setTimeout(() => this.seconds = () => this.timerComponent.seconds, 0);

}

start() { this.timerComponent.start(); }

stop() { this.timerComponent.stop(); }

}

It takes a bit more work to get the child view into the parent component class.

First, you have to import references to the [ViewChild](https://angular.io/api/core/ViewChild) decorator and the [AfterViewInit](https://angular.io/api/core/AfterViewInit) lifecycle hook.

Next, inject the child CountdownTimerComponent into the private timerComponent property using the @[ViewChild](https://angular.io/api/core/ViewChild) property decoration.

The #timer local variable is gone from the component metadata. Instead, bind the buttons to the parent component's own start and stop methods and present the ticking seconds in an interpolation around the parent component's seconds method.

These methods access the injected timer component directly.

The ngAfterViewInit() lifecycle hook is an important wrinkle. The timer component isn't available until after Angular displays the parent view. So it displays 0 seconds initially.

Then Angular calls the ngAfterViewInit lifecycle hook at which time it is too late to update the parent view's display of the countdown seconds. Angular's unidirectional data flow rule prevents updating the parent view's in the same cycle. The application has to wait one turn before it can display the seconds.

Use setTimeout() to wait one tick and then revise the seconds() method so that it takes future values from the timer component.

### **Test it**

Use [the same countdown timer tests](https://angular.io/guide/component-interaction#countdown-tests) as before.

[Back to top](https://angular.io/guide/component-interaction#top)

## **Parent and children communicate using a service**

A parent component and its children share a service whose interface enables bi-directional communication within the family.

The scope of the service instance is the parent component and its children. Components outside this component subtree have no access to the service or their communications.

This MissionService connects the MissionControlComponent to multiple AstronautComponent children.

component-interaction/src/app/mission.service.ts

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

import { Subject } from 'rxjs';

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

export class MissionService {

// Observable string sources

private missionAnnouncedSource = new Subject<string>();

private missionConfirmedSource = new Subject<string>();

// Observable string streams

missionAnnounced$ = this.missionAnnouncedSource.asObservable();

missionConfirmed$ = this.missionConfirmedSource.asObservable();

// Service message commands

announceMission(mission: string) {

this.missionAnnouncedSource.next(mission);

}

confirmMission(astronaut: string) {

this.missionConfirmedSource.next(astronaut);

}

}

The MissionControlComponent both provides the instance of the service that it shares with its children (through the providers metadata array) and injects that instance into itself through its constructor:

component-interaction/src/app/missioncontrol.component.ts

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

import { MissionService } from './mission.service';

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

selector: 'app-mission-control',

template: `

<h2>Mission Control</h2>

<button (click)="announce()">Announce mission</button>

<app-astronaut \*[ngFor](https://angular.io/api/common/NgForOf)="let astronaut of astronauts"

[astronaut]="astronaut">

</app-astronaut>

<h3>History</h3>

<ul>

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

</ul>

`,

providers: [MissionService]

})

export class MissionControlComponent {

astronauts = ['Lovell', 'Swigert', 'Haise'];

history: string[] = [];

missions = ['Fly to the moon!',

'Fly to mars!',

'Fly to Vegas!'];

nextMission = 0;

constructor(private missionService: MissionService) {

missionService.missionConfirmed$.subscribe(

astronaut => {

this.history.push(`${astronaut} confirmed the mission`);

});

}

announce() {

const mission = this.missions[this.nextMission++];

this.missionService.announceMission(mission);

this.history.push(`Mission "${mission}" announced`);

if (this.nextMission >= this.missions.length) { this.nextMission = 0; }

}

}

The AstronautComponent also injects the service in its constructor. Each AstronautComponent is a child of the MissionControlComponent and therefore receives its parent's service instance:

component-interaction/src/app/astronaut.component.ts

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

import { MissionService } from './mission.service';

import { Subscription } from 'rxjs';

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

selector: 'app-astronaut',

template: `

<p>

{{astronaut}}: <strong>{{mission}}</strong>

<button

(click)="confirm()"

[disabled]="!announced || confirmed">

Confirm

</button>

</p>

`

})

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

@[Input](https://angular.io/api/core/Input)() astronaut = '';

mission = '<no mission announced>';

confirmed = false;

announced = false;

subscription: Subscription;

constructor(private missionService: MissionService) {

this.subscription = missionService.missionAnnounced$.subscribe(

mission => {

this.mission = mission;

this.announced = true;

this.confirmed = false;

});

}

confirm() {

this.confirmed = true;

this.missionService.confirmMission(this.astronaut);

}

ngOnDestroy() {

// prevent memory leak when component destroyed

this.subscription.unsubscribe();

}

}

Notice that this example captures the subscription and unsubscribe() when the AstronautComponent is destroyed. This is a memory-leak guard step. There is no actual risk in this application because the lifetime of a AstronautComponent is the same as the lifetime of the application itself. That would not always be true in a more complex application.

You don't add this guard to the MissionControlComponent because, as the parent, it controls the lifetime of the MissionService.

The History log demonstrates that messages travel in both directions between the parent MissionControlComponent and the AstronautComponent children, facilitated by the service:



### **Test it**

Tests click buttons of both the parent MissionControlComponent and the AstronautComponent children and verify that the history meets expectations:

component-interaction/e2e/src/app.e2e-spec.ts

content\_copy// ...

it('should announce a mission', async () => {

const missionControl = element(by.tagName('app-mission-control'));

const announceButton = missionControl.all(by.tagName('button')).get(0);

const history = missionControl.all(by.tagName('li'));

await announceButton.click();

expect(await history.count()).toBe(1);

expect(await history.get(0).getText()).toMatch(/Mission.\* announced/);

});

it('should confirm the mission by Lovell', async () => {

await testConfirmMission(1, 'Lovell');

});

it('should confirm the mission by Haise', async () => {

await testConfirmMission(3, 'Haise');

});

it('should confirm the mission by Swigert', async () => {

await testConfirmMission(2, 'Swigert');

});

async function testConfirmMission(buttonIndex: number, astronaut: string) {

const missionControl = element(by.tagName('app-mission-control'));

const announceButton = missionControl.all(by.tagName('button')).get(0);

const confirmButton = missionControl.all(by.tagName('button')).get(buttonIndex);

const history = missionControl.all(by.tagName('li'));

await announceButton.click();

await confirmButton.click();

expect(await history.count()).toBe(2);

expect(await history.get(1).getText()).toBe(`${astronaut} confirmed the mission`);

}

// ...

# Component styles

Angular applications are styled with standard CSS. That means you can apply everything you know about CSS stylesheets, selectors, rules, and media queries directly to Angular applications.

Additionally, Angular can bundle component styles with components, enabling a more modular design than regular stylesheets.

This page describes how to load and apply these component styles.

You can run the [live example](https://angular.io/generated/live-examples/component-styles/stackblitz.html) / [download example](https://angular.io/generated/zips/component-styles/component-styles.zip) in Stackblitz and download the code from there.

## **Using component styles**

For every Angular component you write, you may define not only an HTML template, but also the CSS styles that go with that template, specifying any selectors, rules, and media queries that you need.

One way to do this is to set the styles property in the component metadata. The styles property takes an array of strings that contain CSS code. Usually you give it one string, as in the following example:

src/app/hero-app.component.ts

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

selector: 'app-root',

template: `

<h1>Tour of Heroes</h1>

<app-hero-main [hero]="hero"></app-hero-main>

`,

styles: ['h1 { font-weight: normal; }']

})

export class HeroAppComponent {

/\* . . . \*/

}

## **Style scope**

The styles specified in @[Component](https://angular.io/api/core/Component) metadata apply only within the template of that component.

They are not inherited by any components nested within the template nor by any content projected into the component.

In this example, the h1 style applies only to the HeroAppComponent, not to the nested HeroMainComponent nor to <h1> tags anywhere else in the application.

This scoping restriction is a styling modularity feature.

* You can use the CSS class names and selectors that make the most sense in the context of each component.
* Class names and selectors are local to the component and don't collide with classes and selectors used elsewhere in the application.
* Changes to styles elsewhere in the application don't affect the component's styles.
* You can co-locate the CSS code of each component with the TypeScript and HTML code of the component, which leads to a neat and tidy project structure.
* You can change or remove component CSS code without searching through the whole application to find where else the code is used.

## **Special selectors**

Component styles have a few special selectors from the world of shadow DOM style scoping (described in the [CSS Scoping Module Level 1](https://www.w3.org/TR/css-scoping-1) page on the [W3C](https://www.w3.org/) site). The following sections describe these selectors.

### **:host**

Use the :host pseudo-class selector to target styles in the element that hosts the component (as opposed to targeting elements inside the component's template).

src/app/hero-details.component.css

content\_copy:host {

display: block;

border: 1px solid black;

}

The :host selector is the only way to target the host element. You can't reach the host element from inside the component with other selectors because it's not part of the component's own template. The host element is in a parent component's template.

Use the function form to apply host styles conditionally by including another selector inside parentheses after :host.

The next example targets the host element again, but only when it also has the active CSS class.

src/app/hero-details.component.css

content\_copy:host(.active) {

border-width: 3px;

}

The :host selector can also be combined with other selectors. Add selectors behind the :host to select child elements, for example using :host h2 to target all <h2> elements inside a component's view.

You should not add selectors (other than :host-context) in front of the :host selector to style a component based on the outer context of the component's view. Such selectors are not scoped to a component's view and will select the outer context, but it's not native behavior. Use :host-context selector for that purpose instead.

### **:host-context**

Sometimes it's useful to apply styles based on some condition outside of a component's view. For example, a CSS theme class could be applied to the document <body> element, and you want to change how your component looks based on that.

Use the :host-context() pseudo-class selector, which works just like the function form of :host(). The :host-context() selector looks for a CSS class in any ancestor of the component host element, up to the document root. The :host-context() selector is useful when combined with another selector.

The following example applies a background-color style to all <h2> elements inside the component, only if some ancestor element has the CSS class theme-light.

src/app/hero-details.component.css

content\_copy:host-context(.theme-light) h2 {

background-color: #eef;

}

### **(deprecated) /deep/, >>>, and ::ng-deep**

Component styles normally apply only to the HTML in the component's own template.

Applying the ::ng-deep pseudo-class to any CSS rule completely disables view-encapsulation for that rule. Any style with ::ng-deep applied becomes a global style. In order to scope the specified style to the current component and all its descendants, be sure to include the :host selector before ::ng-deep. If the ::ng-deep combinator is used without the :host pseudo-class selector, the style can bleed into other components.

The following example targets all <h3> elements, from the host element down through this component to all of its child elements in the DOM.

src/app/hero-details.component.css

content\_copy:host ::ng-deep h3 {

font-style: italic;

}

The /deep/ combinator also has the aliases >>>, and ::ng-deep.

Use /deep/, >>> and ::ng-deep only with emulated view encapsulation. Emulated is the default and most commonly used view encapsulation. For more information, see the [View Encapsulation](https://angular.io/guide/view-encapsulation) section.

The shadow-piercing descendant combinator is deprecated and [support is being removed from major browsers](https://www.chromestatus.com/feature/6750456638341120) and tools. As such we plan to drop support in Angular (for all 3 of /deep/, >>> and ::ng-deep). Until then ::ng-deep should be preferred for a broader compatibility with the tools.

## **Loading component styles**

There are several ways to add styles to a component:

* By setting styles or styleUrls metadata.
* Inline in the template HTML.
* With CSS imports.

The scoping rules outlined earlier apply to each of these loading patterns.

### **Styles in component metadata**

You can add a styles array property to the @[Component](https://angular.io/api/core/Component) decorator.

Each string in the array defines some CSS for this component.

src/app/hero-app.component.ts (CSS inline)

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

selector: 'app-root',

template: `

<h1>Tour of Heroes</h1>

<app-hero-main [hero]="hero"></app-hero-main>

`,

styles: ['h1 { font-weight: normal; }']

})

export class HeroAppComponent {

/\* . . . \*/

}

Reminder: these styles apply only to this component. They are not inherited by any components nested within the template nor by any content projected into the component.

The Angular CLI command [ng generate component](https://angular.io/cli/generate) defines an empty styles array when you create the component with the --inline-style flag.

content\_copyng generate component hero-app --inline-style

### **Style files in component metadata**

You can load styles from external CSS files by adding a styleUrls property to a component's @[Component](https://angular.io/api/core/Component) decorator:

src/app/hero-app.component.ts (CSS in file)

src/app/hero-app.component.css

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

selector: 'app-root',

template: `

<h1>Tour of Heroes</h1>

<app-hero-main [hero]="hero"></app-hero-main>

`,

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

})

export class HeroAppComponent {

/\* . . . \*/

}

Reminder: the styles in the style file apply only to this component. They are not inherited by any components nested within the template nor by any content projected into the component.

You can specify more than one styles file or even a combination of styles and styleUrls.

When you use the Angular CLI command [ng generate component](https://angular.io/cli/generate) without the --inline-style flag, it creates an empty styles file for you and references that file in the component's generated styleUrls.

content\_copyng generate component hero-app

### **Template inline styles**

You can embed CSS styles directly into the HTML template by putting them inside <[style](https://angular.io/api/animations/style)> tags.

src/app/hero-controls.component.ts

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

selector: 'app-hero-controls',

template: `

<[style](https://angular.io/api/animations/style)>

button {

background-color: white;

border: 1px solid #777;

}

</[style](https://angular.io/api/animations/style)>

<h3>Controls</h3>

<button (click)="activate()">Activate</button>

`

})

### **Template link tags**

You can also write <link> tags into the component's HTML template.

src/app/hero-team.component.ts

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

selector: 'app-hero-team',

template: `

<!-- We must use a relative URL so that the AOT compiler can find the stylesheet -->

<link rel="stylesheet" href="../assets/hero-team.component.css">

<h3>Team</h3>

<ul>

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

{{member}}

</li>

</ul>`

})

When building with the CLI, be sure to include the linked style file among the assets to be copied to the server as described in the [Assets configuration guide](https://angular.io/guide/workspace-config#assets-configuration).

Once included, the CLI will include the stylesheet, whether the link tag's href URL is relative to the application root or the component file.

### **CSS @imports**

You can also import CSS files into the CSS files using the standard CSS @import rule. For details, see [@import](https://developer.mozilla.org/en/docs/Web/CSS/@import) on the [MDN](https://developer.mozilla.org/) site.

In this case, the URL is relative to the CSS file into which you're importing.

src/app/hero-details.component.css (excerpt)

content\_copy/\* The AOT compiler needs the `./` to show that this is local \*/

@import './hero-details-box.css';

### **External and global style files**

When building with the CLI, you must configure the angular.json to include all external assets, including external style files.

Register **global** style files in the styles section which, by default, is pre-configured with the global styles.css file.

See the [Styles configuration guide](https://angular.io/guide/workspace-config#styles-and-scripts-configuration) to learn more.

### **Non-CSS style files**

If you're building with the CLI, you can write style files in [sass](https://sass-lang.com/), [less](http://lesscss.org/), or [stylus](https://stylus-lang.com/) and specify those files in the @[Component.styleUrls](https://angular.io/api/core/Component#styleUrls) metadata with the appropriate extensions (.scss, .less, .styl) as in the following example:

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

selector: 'app-root',

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

styleUrls: ['./app.component.scss']

})

...

The CLI build process runs the pertinent CSS preprocessor.

When generating a component file with ng generate component, the CLI emits an empty CSS styles file (.css) by default. You can configure the CLI to default to your preferred CSS preprocessor as explained in the [Workspace configuration guide](https://angular.io/guide/workspace-config#generation-schematics).

Style strings added to the @[Component.styles](https://angular.io/api/core/Component#styles) array must be written in CSS because the CLI cannot apply a preprocessor to inline styles.

# Sharing data between child and parent directives and components

A common pattern in Angular is sharing data between a parent component and one or more child components. You can implement this pattern by using the @[Input](https://angular.io/api/core/Input)() and @[Output](https://angular.io/api/core/Output)() directives.

See the [live example](https://angular.io/generated/live-examples/inputs-outputs/stackblitz.html) / [download example](https://angular.io/generated/zips/inputs-outputs/inputs-outputs.zip) for a working example containing the code snippets in this guide.

Consider the following hierarchy:

content\_copy<parent-component>

<child-component></child-component>

</parent-component>

The <parent-component> serves as the context for the <child-component>.

@[Input](https://angular.io/api/core/Input)() and @[Output](https://angular.io/api/core/Output)() give a child component a way to communicate with its parent component. @[Input](https://angular.io/api/core/Input)() allows a parent component to update data in the child component. Conversely, @[Output](https://angular.io/api/core/Output)() allows the child to send data to a parent component.

## **Sending data to a child component**

The @[Input](https://angular.io/api/core/Input)() decorator in a child component or directive signifies that the property can receive its value from its parent component.

To use @[Input](https://angular.io/api/core/Input)(), you must configure the parent and child.

### **Configuring the child component**

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)(), as in the following example.

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

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

export class ItemDetailComponent {

@[Input](https://angular.io/api/core/Input)() item = ''; // 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](https://angular.io/api/common/DecimalPipe), string, boolean, or object. The value for item comes from the parent component.

Next, in the child component template, add the following:

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

content\_copy<p>

Today's item: {{item}}

</p>

### **Configuring the parent component**

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.

1. Use the child's selector, here <app-item-detail>, as a directive within the parent component template.
2. Use [property binding](https://angular.io/guide/property-binding) to bind the item property in the child to the currentItem property of the parent.

src/app/app.component.html

content\_copy<app-item-detail [item]="currentItem"></app-item-detail>

1. In the parent component class, designate a value for currentItem:

src/app/app.component.ts

content\_copyexport class AppComponent {

currentItem = 'Television';

}

With @[Input](https://angular.io/api/core/Input)(), Angular passes the value for currentItem to the child so that 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.

### **Watching for @**[**Input**](https://angular.io/api/core/Input)**() changes**

To watch for changes on an @[Input](https://angular.io/api/core/Input)() property, you can use [OnChanges](https://angular.io/api/core/OnChanges), one of Angular's [lifecycle hooks](https://angular.io/guide/lifecycle-hooks). 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.

## **Sending data to a parent component**

The @[Output](https://angular.io/api/core/Output)() decorator in a child component or directive allows data to flow from the child to the parent.

@[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 uses the @[Output](https://angular.io/api/core/Output)() property to raise an event to notify the parent of the change. To raise an event, an @[Output](https://angular.io/api/core/Output)() must have the type of [EventEmitter](https://angular.io/api/core/EventEmitter), which is a class in @angular/core that you use to emit custom events.

The following example shows how to set up an @[Output](https://angular.io/api/core/Output)() in a child component that pushes data from an HTML <input> to an array in the parent component.

To use @[Output](https://angular.io/api/core/Output)(), you must configure the parent and child.

### **Configuring the child component**

The following 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.

1. Import [Output](https://angular.io/api/core/Output) and [EventEmitter](https://angular.io/api/core/EventEmitter) in the child component class:

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

1. In the component class, decorate a property with @[Output](https://angular.io/api/core/Output)(). The following example newItemEvent @[Output](https://angular.io/api/core/Output)() has a type of [EventEmitter](https://angular.io/api/core/EventEmitter), which means it's an event.

src/app/item-output/item-output.component.ts

content\_copy@[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.

For more information on [EventEmitter](https://angular.io/api/core/EventEmitter), see the [EventEmitter API documentation](https://angular.io/api/core/EventEmitter).

1. Create an addNewItem() method in the same component class:

src/app/item-output/item-output.component.ts

content\_copyexport 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 with the value the user types into the <input>.

### **Configuring 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-reference-variables) , #newItem, where the user types in an item name. The value property of the #newItem variable stores what the user types into the <input>.

src/app/item-output/item-output.component.html

content\_copy<label for="item-input">Add an item:</label>

<input type="text" id="item-input" #newItem>

<button (click)="addNewItem(newItem.value)">Add to parent's list</button>

The second element is a <button> with a click [event binding](https://angular.io/guide/event-binding).

The (click) event is bound to the addNewItem() method in the child component class. The addNewItem() method takes as its argument the value of the #newItem.value property.

### **Configuring the parent component**

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

content\_copyexport 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 adds that string to the items array.

### **Configuring the parent's template**

1. In the parent's template, bind the parent's method to the child's event.
2. Put the child selector, here <app-item-output>, within the parent component's template, app.component.html.

src/app/app.component.html

content\_copy<app-item-output (newItemEvent)="addItem($event)"></app-item-output>

The event binding, (newItemEvent)='addItem($event)', connects the event in the child, newItemEvent, to the method in the parent, addItem().

The $event contains the data that the user types into the <input> in the child template UI.

To see the @[Output](https://angular.io/api/core/Output)() working, you can add the following to the parent's template:

content\_copy<ul>

<li \*[ngFor](https://angular.io/api/common/NgForOf)="let item of items">{{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 new item renders in the list.

## **Using @**[**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 follows:

src/app/app.component.html

content\_copy<app-input-output [item]="currentItem" (deleteRequest)="crossOffItem($event)"></app-input-output>

The target, 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 shows the different parts of the @[Input](https://angular.io/api/core/Input)() and @[Output](https://angular.io/api/core/Output)() on the <app-input-output> child component.

The child selector is <app-input-output> with 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/two-way-binding).

# Content projection

This topic describes how to use content projection to create flexible, reusable components.

To view or download the example code used in this topic, see the [live example](https://angular.io/generated/live-examples/content-projection/stackblitz.html) / [download example](https://angular.io/generated/zips/content-projection/content-projection.zip).

Content projection is a pattern in which you insert, or project, the content you want to use inside another component. For example, you could have a Card component that accepts content provided by another component.

The following sections describe common implementations of content projection in Angular, including:

* [Single-slot content projection](https://angular.io/guide/content-projection#single-slot). With this type of content projection, a component accepts content from a single source.
* [Multi-slot content projection](https://angular.io/guide/content-projection#multi-slot). In this scenario, a component accepts content from multiple sources.
* [Conditional content projection](https://angular.io/guide/content-projection#conditional). Components that use conditional content projection render content only when specific conditions are met.

## **Single-slot content projection**

The most basic form of content projection is single-slot content projection. Single-slot content projection refers to creating a component into which you can project one component.

To create a component that uses single-slot content projection:

1. [Create](https://angular.io/guide/component-overview) a component.
2. In the template for your component, add an ng-content element where you want the projected content to appear.

For example, the following component uses an ng-content element to display a message.

content-projection/src/app/zippy-basic/zippy-basic.component.ts

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

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

selector: 'app-zippy-basic',

template: `

<h2>Single-slot content projection</h2>

<ng-content></ng-content>

`

})

export class ZippyBasicComponent {}

With the ng-content element in place, users of this component can now project their own message into the component. For example:

content-projection/src/app/app.component.html

content\_copy<app-zippy-basic>

<p>Is content projection cool?</p>

</app-zippy-basic>

The ng-content element is a placeholder that does not create a real DOM element. Custom attributes applied to ng-content are ignored.

## **Multi-slot content projection**

A component can have multiple slots. Each slot can specify a CSS selector that determines which content goes into that slot. This pattern is referred to as multi-slot content projection. With this pattern, you must specify where you want the projected content to appear. You accomplish this task by using the select attribute of ng-content.

To create a component that uses multi-slot content projection:

1. [Create](https://angular.io/guide/component-overview) a component.
2. In the template for your component, add an ng-content element where you want the projected content to appear.
3. Add a select attribute to the ng-content elements. Angular supports [selectors](https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Selectors) for any combination of tag name, attribute, CSS class, and the :not pseudo-class.

For example, the following component uses two ng-content elements.

content-projection/src/app/zippy-multislot/zippy-multislot.component.ts

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

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

selector: 'app-zippy-multislot',

template: `

<h2>Multi-slot content projection</h2>

Default:

<ng-content></ng-content>

Question:

<ng-content select="[question]"></ng-content>

`

})

export class ZippyMultislotComponent {}

Content that uses the question attribute is projected into the ng-content element with the select=[question] attribute.

content-projection/src/app/app.component.html

content\_copy<app-zippy-multislot>

<p question>

Is content projection cool?

</p>

<p>Let's learn about content projection!</p>

</app-zippy-multislot>

NG-CONTENT WITHOUT A SELECT ATTRIBUTE

If your component includes an ng-content element without a select attribute, that instance receives all projected components that do not match any of the other ng-content elements.

In the preceding example, only the second ng-content element defines a select attribute. As a result, the first ng-content element receives any other content projected into the component.

## **Conditional content projection**

If your component needs to conditionally render content, or render content multiple times, you should configure that component to accept an ng-template element that contains the content you want to conditionally render.

Using an ng-content element in these cases is not recommended, because when the consumer of a component supplies the content, that content is always initialized, even if the component does not define an ng-content element or if that ng-content element is inside of an [ngIf](https://angular.io/api/common/NgIf) statement.

With an ng-template element, you can have your component explicitly render content based on any condition you want, as many times as you want. Angular will not initialize the content of an ng-template element until that element is explicitly rendered.

The following steps demonstrate a typical implementation of conditional content projection using ng-template.

1. [Create](https://angular.io/guide/component-overview) a component.
2. In the component that accepts an ng-template element, use an ng-container element to render that template, such as:

content-projection/src/app/example-zippy.template.html

content\_copy<ng-container [[ngTemplateOutlet](https://angular.io/api/common/NgTemplateOutlet)]="content.templateRef"> </ng-container>

This example uses the [ngTemplateOutlet](https://angular.io/api/common/NgTemplateOutlet) directive to render a given ng-template element, which you will define in a later step. You can apply an [ngTemplateOutlet](https://angular.io/api/common/NgTemplateOutlet) directive to any type of element. This example assigns the directive to an ng-container element because the component does not need to render a real DOM element.

1. Wrap the ng-container element in another element, such as a div element, and apply your conditional logic.

content-projection/src/app/example-zippy.template.html

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

<ng-container [[ngTemplateOutlet](https://angular.io/api/common/NgTemplateOutlet)]="content.templateRef"> </ng-container>

</div>

1. In the template where you want to project content, wrap the projected content in an ng-template element, such as:
2. content\_copy<ng-template appExampleZippyContent>
3. It depends on what you do with it.

</ng-template>

The ng-template element defines a block of content that a component can render based on its own logic. A component can get a reference to this template content, or [TemplateRef](https://angular.io/api/core/TemplateRef), by using either the [@ContentChild](https://angular.io/api/core/ContentChild) or [@ContentChildren](https://angular.io/api/core/ContentChildren) decorators. The preceding example creates a custom directive, appExampleZippyContent, as an API to mark the ng-template for the component's content. With the [TemplateRef](https://angular.io/api/core/TemplateRef), the component can render the referenced content by using either the [ngTemplateOutlet](https://angular.io/api/common/NgTemplateOutlet) directive, or with [ViewContainerRef.createEmbeddedView](https://angular.io/api/core/ViewContainerRef#createembeddedview).

1. Create a directive with a selector that matches the custom attribute for your template. In this directive, inject a TemplateRef instance.

content-projection/src/app/app.component.ts

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

selector: '[appExampleZippyContent]'

})

export class ZippyContentDirective {

constructor(public templateRef: [TemplateRef](https://angular.io/api/core/TemplateRef)<unknown>) {}

}

In the previous step, you added an ng-template element with a custom attribute, appExampleZippyDirective. This code provides the logic that Angular will use when it encounters that custom attribute. In this case, that logic instructs Angular to instantiate a template reference.

1. In the component you want to project content into, use @[ContentChild](https://angular.io/api/core/ContentChild) to get the template of the projected content.

content-projection/src/app/app.component.ts

content\_copy@[ContentChild](https://angular.io/api/core/ContentChild)(ZippyContentDirective) content!: ZippyContentDirective;

Prior to this step, your application has a component that instantiates a template when certain conditions are met. You've also created a directive that provides a reference to that template. In this last step, the @[ContentChild](https://angular.io/api/core/ContentChild) decorator instructs Angular to instantiate the template in the designated component.

In the case of multi-slot content projection, you can use @[ContentChildren](https://angular.io/api/core/ContentChildren) to get a QueryList of projected elements.

## **Projecting content in more complex environments**

As described in [Multi-slot Content Projection](https://angular.io/guide/content-projection#multi-slot), you typically use either an attribute, element, CSS Class, or some combination of all three to identify where to project your content. For example, in the following HTML template, a paragraph tag uses a custom attribute, question, to project content into the app-zippy-multislot component.

content-projection/src/app/app.component.html

content\_copy<app-zippy-multislot>

<p question>

Is content projection cool?

</p>

<p>Let's learn about content projection!</p>

</app-zippy-multislot>

In some cases, you might want to project content as a different element. For example, the content you want to project might be a child of another element. You can accomplish this by using the ngProjectAs attribute.

For instance, consider the following HTML snippet:

content-projection/src/app/app.component.html

content\_copy<ng-container ngProjectAs="[question]">

<p>Is content projection cool?</p>

</ng-container>

This example uses an ng-container attribute to simulate projecting a component into a more complex structure.

REMINDER!

The ng-container element is a logical construct that you can use to group other DOM elements; however, the ng-container itself is not rendered in the DOM tree.

In this example, the content we want to project resides inside another element. To project this content as intended, the template uses the ngProjectAs attribute. With ngProjectAs, the entire ng-container element is projected into a component using the [question] selector.

# Dynamic component loader

Component templates are not always fixed. An application may need to load new components at runtime.

This cookbook shows you how to use [ComponentFactoryResolver](https://angular.io/api/core/ComponentFactoryResolver) to add components dynamically.

See the [live example](https://angular.io/generated/live-examples/dynamic-component-loader/stackblitz.html) / [download example](https://angular.io/generated/zips/dynamic-component-loader/dynamic-component-loader.zip) of the code in this cookbook.

## **Dynamic component loading**

The following example shows how to build a dynamic ad banner.

The hero agency is planning an ad campaign with several different ads cycling through the banner. New ad components are added frequently by several different teams. This makes it impractical to use a template with a static component structure.

Instead, you need a way to load a new component without a fixed reference to the component in the ad banner's template.

Angular comes with its own API for loading components dynamically.

## **The anchor directive**

Before you can add components you have to define an anchor point to tell Angular where to insert components.

The ad banner uses a helper directive called AdDirective to mark valid insertion points in the template.

src/app/ad.directive.ts

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

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

selector: '[adHost]',

})

export class AdDirective {

constructor(public viewContainerRef: [ViewContainerRef](https://angular.io/api/core/ViewContainerRef)) { }

}

AdDirective injects [ViewContainerRef](https://angular.io/api/core/ViewContainerRef) to gain access to the view container of the element that will host the dynamically added component.

In the @[Directive](https://angular.io/api/core/Directive) decorator, notice the selector name, adHost; that's what you use to apply the directive to the element. The next section shows you how.

## **Loading components**

Most of the ad banner implementation is in ad-banner.component.ts. To keep things simple in this example, the HTML is in the @[Component](https://angular.io/api/core/Component) decorator's template property as a template string.

The [<ng-template>](https://angular.io/api/core/ng-template) element is where you apply the directive you just made. To apply the AdDirective, recall the selector from ad.directive.ts, [adHost]. Apply that to [<ng-template>](https://angular.io/api/core/ng-template) without the square brackets. Now Angular knows where to dynamically load components.

src/app/ad-banner.component.ts (template)

content\_copytemplate: `

<div class="ad-banner-example">

<h3>Advertisements</h3>

<ng-template adHost></ng-template>

</div>

`

The [<ng-template>](https://angular.io/api/core/ng-template) element is a good choice for dynamic components because it doesn't render any additional output.

## **Resolving components**

Take a closer look at the methods in ad-banner.component.ts.

AdBannerComponent takes an array of AdItem objects as input, which ultimately comes from AdService. AdItem objects specify the type of component to load and any data to bind to the component.AdService returns the actual ads making up the ad campaign.

Passing an array of components to AdBannerComponent allows for a dynamic list of ads without static elements in the template.

With its getAds() method, AdBannerComponent cycles through the array of AdItems and loads a new component every 3 seconds by calling loadComponent().

src/app/ad-banner.component.ts (excerpt)

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

@[Input](https://angular.io/api/core/Input)() ads: AdItem[] = [];

currentAdIndex = -1;

@[ViewChild](https://angular.io/api/core/ViewChild)(AdDirective, {[static](https://angular.io/api/upgrade/static): true}) adHost!: AdDirective;

interval: any;

constructor(private componentFactoryResolver: [ComponentFactoryResolver](https://angular.io/api/core/ComponentFactoryResolver)) { }

ngOnInit() {

this.loadComponent();

this.getAds();

}

ngOnDestroy() {

clearInterval(this.interval);

}

loadComponent() {

this.currentAdIndex = (this.currentAdIndex + 1) % this.ads.length;

const adItem = this.ads[this.currentAdIndex];

const componentFactory = this.componentFactoryResolver.resolveComponentFactory(adItem.component);

const viewContainerRef = this.adHost.viewContainerRef;

viewContainerRef.clear();

const componentRef = viewContainerRef.createComponent<AdComponent>(componentFactory);

componentRef.instance.data = adItem.data;

}

getAds() {

this.interval = setInterval(() => {

this.loadComponent();

}, 3000);

}

}

The loadComponent() method is doing a lot of the heavy lifting here. Take it step by step. First, it picks an ad.

**How**loadComponent()**chooses an ad**

The loadComponent() method chooses an ad using some math.

First, it sets the currentAdIndex by taking whatever it currently is plus one, dividing that by the length of the AdItem array, and using the remainder as the new currentAdIndex value. Then, it uses that value to select an adItem from the array.

After loadComponent() selects an ad, it uses [ComponentFactoryResolver](https://angular.io/api/core/ComponentFactoryResolver) to resolve a [ComponentFactory](https://angular.io/api/core/ComponentFactory) for each specific component. The [ComponentFactory](https://angular.io/api/core/ComponentFactory) then creates an instance of each component.

Next, you're targeting the viewContainerRef that exists on this specific instance of the component. How do you know it's this specific instance? Because it's referring to adHost and adHost is the directive you set up earlier to tell Angular where to insert dynamic components.

As you may recall, AdDirective injects [ViewContainerRef](https://angular.io/api/core/ViewContainerRef) into its constructor. This is how the directive accesses the element that you want to use to host the dynamic component.

To add the component to the template, you call createComponent() on [ViewContainerRef](https://angular.io/api/core/ViewContainerRef).

The createComponent() method returns a reference to the loaded component. Use that reference to interact with the component by assigning to its properties or calling its methods.

## **The AdComponent interface**

In the ad banner, all components implement a common AdComponent interface to standardize the API for passing data to the components.

Here are two sample components and the AdComponent interface for reference:

hero-job-ad.component.ts

hero-profile.component.ts

ad.component.ts

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

import { AdComponent } from './ad.component';

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

template: `

<div class="job-ad">

<h4>{{data.headline}}</h4>

{{data.body}}

</div>

`

})

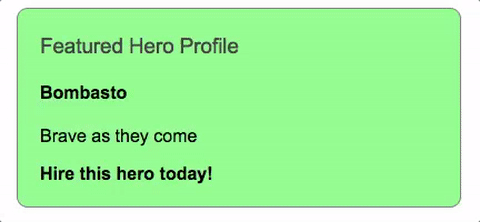
export class HeroJobAdComponent implements AdComponent {

@[Input](https://angular.io/api/core/Input)() data: any;

}

## **Final ad banner**

The final ad banner looks like this:



See the [live example](https://angular.io/generated/live-examples/dynamic-component-loader/stackblitz.html) / [download example](https://angular.io/generated/zips/dynamic-component-loader/dynamic-component-loader.zip).

# Angular elements overview

Angular elements are Angular components packaged as custom elements (also called Web Components), a web standard for defining new HTML elements in a framework-agnostic way.

For the sample application that this page describes, see the [live example](https://angular.io/generated/live-examples/elements/stackblitz.html) / [download example](https://angular.io/generated/zips/elements/elements.zip).

[Custom elements](https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_custom_elements) are a Web Platform feature currently supported by Chrome, Edge (Chromium-based), Firefox, Opera, and Safari, and available in other browsers through polyfills (see [Browser Support](https://angular.io/guide/elements#browser-support)). A custom element extends HTML by allowing you to define a tag whose content is created and controlled by JavaScript code. The browser maintains a CustomElementRegistry of defined custom elements, which maps an instantiable JavaScript class to an HTML tag.

The @angular/elements package exports a [createCustomElement](https://angular.io/api/elements/createCustomElement)() API that provides a bridge from Angular's component interface and change detection functionality to the built-in DOM API.

Transforming a component to a custom element makes all of the required Angular infrastructure available to the browser. Creating a custom element is simple and straightforward, and automatically connects your component-defined view with change detection and data binding, mapping Angular functionality to the corresponding native HTML equivalents.

We are working on custom elements that can be used by web apps built on other frameworks. A minimal, self-contained version of the Angular framework will be injected as a service to support the component's change-detection and data-binding functionality. For more about the direction of development, check out this [video presentation](https://www.youtube.com/watch?v=Z1gLFPLVJjY&t=4s).

## **Using custom elements**

Custom elements bootstrap themselves - they start automatically when they are added to the DOM, and are automatically destroyed when removed from the DOM. Once a custom element is added to the DOM for any page, it looks and behaves like any other HTML element, and does not require any special knowledge of Angular terms or usage conventions.

* **Easy dynamic content in an Angular application**

Transforming a component to a custom element provides an easy path to creating dynamic HTML content in your Angular application. HTML content that you add directly to the DOM in an Angular application is normally displayed without Angular processing, unless you define a dynamic component, adding your own code to connect the HTML tag to your application data, and participate in change detection. With a custom element, all of that wiring is taken care of automatically.

* **Content-rich applications**

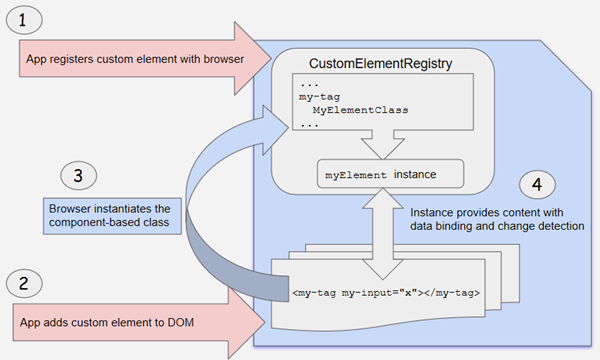
If you have a content-rich application, such as the Angular app that presents this documentation, custom elements let you give your content providers sophisticated Angular functionality without requiring knowledge of Angular. For example, an Angular guide like this one is added directly to the DOM by the Angular navigation tools, but can include special elements like <code-snippet> that perform complex operations. All you need to tell your content provider is the syntax of your custom element. They don't need to know anything about Angular, or anything about your component's data structures or implementation.

### **How it works**

Use the [createCustomElement](https://angular.io/api/elements/createCustomElement)() function to convert a component into a class that can be registered with the browser as a custom element. After you register your configured class with the browser's custom-element registry, you can use the new element just like a built-in HTML element in content that you add directly into the DOM:

content\_copy<my-popup message="Use Angular!"></my-popup>

When your custom element is placed on a page, the browser creates an instance of the registered class and adds it to the DOM. The content is provided by the component's template, which uses Angular template syntax, and is rendered using the component and DOM data. Input properties in the component correspond to input attributes for the element.

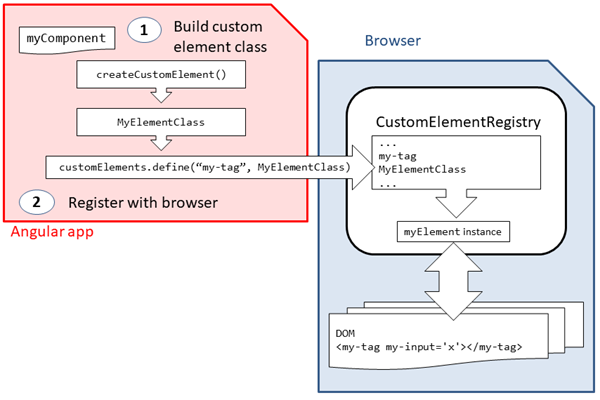


## **Transforming components to custom elements**

Angular provides the [createCustomElement](https://angular.io/api/elements/createCustomElement)() function for converting an Angular component, together with its dependencies, to a custom element. The function collects the component's observable properties, along with the Angular functionality the browser needs to create and destroy instances, and to detect and respond to changes.

The conversion process implements the [NgElementConstructor](https://angular.io/api/elements/NgElementConstructor) interface, and creates a constructor class that is configured to produce a self-bootstrapping instance of your component.

Use the built-in [customElements.define()](https://developer.mozilla.org/en-US/docs/Web/API/CustomElementRegistry/define) function to register the configured constructor and its associated custom-element tag with the browser's [CustomElementRegistry](https://developer.mozilla.org/en-US/docs/Web/API/CustomElementRegistry). When the browser encounters the tag for the registered element, it uses the constructor to create a custom-element instance.



Avoid using the [@Component](https://angular.io/api/core/Component) [selector](https://angular.io/api/core/Directive#selector) as the custom-element tag name. This can lead to unexpected behavior, due to Angular creating two component instances for a single DOM element: One regular Angular component and a second one using the custom element.

### **Mapping**

A custom element hosts an Angular component, providing a bridge between the data and logic defined in the component and standard DOM APIs. Component properties and logic maps directly into HTML attributes and the browser's event system.

* The creation API parses the component looking for input properties, and defines corresponding attributes for the custom element. It transforms the property names to make them compatible with custom elements, which do not recognize case distinctions. The resulting attribute names use dash-separated lowercase. For example, for a component with @[Input](https://angular.io/api/core/Input)('myInputProp') inputProp, the corresponding custom element defines an attribute my-input-prop.
* Component outputs are dispatched as HTML [Custom Events](https://developer.mozilla.org/en-US/docs/Web/API/CustomEvent), with the name of the custom event matching the output name. For example, for a component with @[Output](https://angular.io/api/core/Output)() valueChanged = new [EventEmitter](https://angular.io/api/core/EventEmitter)(), the corresponding custom element will dispatch events with the name "valueChanged", and the emitted data will be stored on the event’s detail property. If you provide an alias, that value is used; for example, @[Output](https://angular.io/api/core/Output)('myClick') clicks = new [EventEmitter](https://angular.io/api/core/EventEmitter)<string>(); results in dispatch events with the name "myClick".

For more information, see Web Component documentation for [Creating custom events](https://developer.mozilla.org/en-US/docs/Web/Guide/Events/Creating_and_triggering_events#Creating_custom_events).

## **Browser support for custom elements**

The recently-developed [custom elements](https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_custom_elements) Web Platform feature is currently supported natively in a number of browsers.

|  |  |
| --- | --- |
| **Browser** | **Custom Element Support** |
| Chrome | Supported natively. |
| Edge (Chromium-based) | Supported natively. |
| Firefox | Supported natively. |
| Opera | Supported natively. |
| Safari | Supported natively. |

In browsers that support Custom Elements natively, the specification requires developers use ES2015 classes to define Custom Elements - developers can opt-in to this by setting the target: "es2015" property in their project's [TypeScript configuration file](https://angular.io/guide/typescript-configuration). As Custom Element and ES2015 support may not be available in all browsers, developers can instead choose to use a polyfill to support older browsers and ES5 code.

Use the [Angular CLI](https://angular.io/cli) to automatically set up your project with the correct polyfill:

content\_copyng add @angular/elements --project=\*your\_project\_name\*

* For more information about polyfills, see [polyfill documentation](https://www.webcomponents.org/polyfills).
* For more information about Angular browser support, see [Browser Support](https://angular.io/guide/browser-support).

## **Example: A Popup Service**

Previously, when you wanted to add a component to an application at runtime, you had to define a dynamic component, and then you would have to load it, attach it to an element in the DOM, and wire up all of the dependencies, change detection, and event handling, as described in [Dynamic Component Loader](https://angular.io/guide/dynamic-component-loader).

Using an Angular custom element makes the process much simpler and more transparent, by providing all of the infrastructure and framework automatically—all you have to do is define the kind of event handling you want. (You do still have to exclude the component from compilation, if you are not going to use it in your application.)

The Popup Service example application (shown below) defines a component that you can either load dynamically or convert to a custom element.

* popup.component.ts defines a simple pop-up element that displays an input message, with some animation and styling.
* popup.service.ts creates an injectable service that provides two different ways to invoke the PopupComponent; as a dynamic component, or as a custom element. Notice how much more setup is required for the dynamic-loading method.
* app.module.ts adds the PopupComponent in the module's declarations list.
* app.component.ts defines the application's root component, which uses the PopupService to add the pop-up to the DOM at run time. When the application runs, the root component's constructor converts PopupComponent to a custom element.

For comparison, the demo shows both methods. One button adds the popup using the dynamic-loading method, and the other uses the custom element. You can see that the result is the same; only the preparation is different.

popup.component.ts

popup.service.ts

app.module.ts

app.component.ts

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

import { [animate](https://angular.io/api/animations/animate), state, [style](https://angular.io/api/animations/style), [transition](https://angular.io/api/animations/transition), [trigger](https://angular.io/api/animations/trigger) } from '@angular/animations';

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

selector: 'my-popup',

template: `

<span>Popup: {{message}}</span>

<button (click)="closed.next()">&#x2716;</button>

`,

animations: [

[trigger](https://angular.io/api/animations/trigger)('state', [

state('opened', [style](https://angular.io/api/animations/style)({transform: 'translateY(0%)'})),

state('void, closed', [style](https://angular.io/api/animations/style)({transform: 'translateY(100%)', opacity: 0})),

[transition](https://angular.io/api/animations/transition)('\* => \*', [animate](https://angular.io/api/animations/animate)('100ms ease-in')),

])

],

styles: [`

:host {

position: absolute;

bottom: 0;

left: 0;

right: 0;

background: #009cff;

height: 48px;

padding: 16px;

display: flex;

justify-content: space-between;

align-items: center;

border-top: 1px solid black;

font-size: 24px;

}

button {

border-radius: 50%;

}

`]

})

export class PopupComponent {

@[HostBinding](https://angular.io/api/core/HostBinding)('@state')

state: 'opened' | 'closed' = 'closed';

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

get message(): string { return this.\_message; }

set message(message: string) {

this.\_message = message;

this.state = 'opened';

}

private \_message = '';

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

closed = new [EventEmitter](https://angular.io/api/core/EventEmitter)<void>();

}

## **Typings for custom elements**

Generic DOM APIs, such as document.createElement() or document.querySelector(), return an element type that is appropriate for the specified arguments. For example, calling document.createElement('a') will return an HTMLAnchorElement, which TypeScript knows has an href property. Similarly, document.createElement('div') will return an HTMLDivElement, which TypeScript knows has no href property.

When called with unknown elements, such as a custom element name (popup-element in our example), the methods will return a generic type, such as HTMLElement, since TypeScript can't infer the correct type of the returned element.

Custom elements created with Angular extend [NgElement](https://angular.io/api/elements/NgElement) (which in turn extends HTMLElement). Additionally, these custom elements will have a property for each input of the corresponding component. For example, our popup-element will have a message property of type string.

There are a few options if you want to get correct types for your custom elements. Let's assume you create a my-dialog custom element based on the following component:

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

class MyDialog {

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

}

The most straightforward way to get accurate typings is to cast the return value of the relevant DOM methods to the correct type. For that, you can use the [NgElement](https://angular.io/api/elements/NgElement) and [WithProperties](https://angular.io/api/elements/WithProperties) types (both exported from @angular/elements):

content\_copyconst aDialog = document.createElement('my-dialog') as [NgElement](https://angular.io/api/elements/NgElement) & [WithProperties](https://angular.io/api/elements/WithProperties)<{content: string}>;

aDialog.content = 'Hello, world!';

aDialog.content = 123; // <-- ERROR: TypeScript knows this should be a string.

aDialog.body = 'News'; // <-- ERROR: TypeScript knows there is no `body` property on `aDialog`.

This is a good way to quickly get TypeScript features, such as type checking and autocomplete support, for your custom element. But it can get cumbersome if you need it in several places, because you have to cast the return type on every occurrence.

An alternative way, that only requires defining each custom element's type once, is augmenting the HTMLElementTagNameMap, which TypeScript uses to infer the type of a returned element based on its tag name (for DOM methods such as document.createElement(), document.querySelector(), etc.):

content\_copydeclare [global](https://angular.io/api/core/global) {

interface HTMLElementTagNameMap {

'my-dialog': [NgElement](https://angular.io/api/elements/NgElement) & [WithProperties](https://angular.io/api/elements/WithProperties)<{content: string}>;

'my-other-element': [NgElement](https://angular.io/api/elements/NgElement) & [WithProperties](https://angular.io/api/elements/WithProperties)<{foo: 'bar'}>;

...

}

}

Now, TypeScript can infer the correct type the same way it does for built-in elements:

content\_copydocument.createElement('div') //--> HTMLDivElement (built-in element)

document.querySelector('foo') //--> Element (unknown element)

document.createElement('my-dialog') //--> [NgElement](https://angular.io/api/elements/NgElement) & [WithProperties](https://angular.io/api/elements/WithProperties)<{content: string}> (custom element)

document.querySelector('my-other-element') //--> [NgElement](https://angular.io/api/elements/NgElement) & [WithProperties](https://angular.io/api/elements/WithProperties)<{foo: 'bar'}> (custom element)

# Text interpolation

Text interpolation allows you to incorporate dynamic string values into your HTML templates. With interpolation, you can dynamically change what appears in an application view, such as displaying a custom greeting that includes the user's name.

See the [live example](https://angular.io/generated/live-examples/interpolation/stackblitz.html) / [download example](https://angular.io/generated/zips/interpolation/interpolation.zip) for all of the syntax and code snippets in this guide.

## **Displaying values with interpolation**

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

To illustrate how interpolation works, consider an Angular component that contains a currentCustomer variable:

src/app/app.component.ts

content\_copycurrentCustomer = 'Maria';

You can use interpolation to display the value of this variable in the corresponding component template:

src/app/app.component.html

content\_copy<h3>Current customer: {{ currentCustomer }}</h3>

Angular replaces currentCustomer with the string value of the corresponding component property. In this case, the value is Maria.

In the following example, Angular evaluates the title and itemImageUrl properties to display some title text and an image.

src/app/app.component.html

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

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

## **Template expressions**

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

### **Resolving expressions with interpolation**

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

content\_copy<!-- "The sum of 1 + 1 is 2" -->

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

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

src/app/app.component.html

content\_copy<!-- "The sum of 1 + 1 is not 4" -->

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

With interpolation, Angular performs the following tasks:

1. Evaluates all expressions in double curly braces.
2. Converts the expression results to strings.
3. Links the results to any adjacent literal strings.
4. Assigns the composite to an element or directive property.

You can configure the interpolation delimiter with the [interpolation](https://angular.io/api/core/Component#interpolation) option in the @[Component](https://angular.io/api/core/Component)() metadata.

### **Syntax**

Template expressions are similar to JavaScript. Many JavaScript expressions are legal template expressions, with the following exceptions.

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

* Assignments (=, +=, -=, ...)
* Operators such as new, typeof, or instanceof
* 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](https://angular.io/guide/template-expression-operators), such as |, ?. and !

## **Expression context**

Interpolated expressions have a context—a particular part of the application to which the expression belongs. Typically, this context is the component instance.

In the following snippet, the expression recommended and the expression itemImageUrl2 refer to properties of the AppComponent.

src/app/app.component.html

content\_copy<h4>{{recommended}}</h4>

<img [src]="itemImageUrl2">

An expression can also refer to properties of the template's context such as a [template input variable](https://angular.io/guide/structural-directives#shorthand) or a [template reference variable](https://angular.io/guide/template-reference-variables).

The following example uses a template input variable of customer.

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

content\_copy<ul>

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

</ul>

This next example features a template reference variable, #customerInput.

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

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

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

</label>

Template expressions cannot refer to anything in the global namespace, except undefined. They can't refer to window or document. Additionally, they can't call console.log() or Math.max() and they are restricted to referencing members of the expression context.

### **Preventing name collisions**

The context against which an expression evaluates is the union 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, Angular applies the following logic to determine the context:

1. The template variable name.
2. A name in the directive's context.
3. The component's member names.

To avoid variables shadowing variables in another context, keep variable names unique. In the following example, the AppComponent template greets the customer, Padma.

An [ngFor](https://angular.io/api/common/NgForOf) then lists each customer in the customers array.

src/app/app.component.ts

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

template: `

<div>

<!-- Hello, Padma -->

<h1>Hello, {{customer}}</h1>

<ul>

<!-- Ebony and Chiho in a list-->

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

</ul>

</div>

`

})

class AppComponent {

customers = [{value: 'Ebony'}, {value: 'Chiho'}];

customer = 'Padma';

}

The customer within the [ngFor](https://angular.io/api/common/NgForOf) is in the context of an [<ng-template>](https://angular.io/api/core/ng-template) and so refers to the customer in the customers array, in this case Ebony and Chiho. This list does not feature Padma because customer outside of the [ngFor](https://angular.io/api/common/NgForOf) is in a different context. Conversely, customer in the <h1> doesn't include Ebony or Chiho because the context for this customer is the class and the class value for customer is Padma.

## **Expression best practices**

When using template expressions, follow these best practices:

* **Use short expressions**

Use property names or method calls whenever possible. Keep application and business logic in the component, where it is easier to develop and test.

* **Quick execution**

Angular executes template expressions after every [change detection](https://angular.io/guide/glossary#change-detection) cycle. Many asynchronous activities trigger change detection cycles, such as promise resolutions, HTTP results, timer events, key presses and mouse moves.

Expressions should finish quickly to keep the user experience as efficient as possible, especially on slower devices. Consider caching values when their computation requires greater resources.

* **No visible side effects**

According to Angular's [unidirectional data flow model](https://angular.io/guide/glossary#unidirectional-data-flow), a template expression should not change any application state other than the value of the target property. Reading a component value should not change some other displayed value. The view should be stable throughout a single rendering pass.

IDEMPOTENT EXPRESSIONS REDUCE SIDE EFFECTS

An [idempotent](https://en.wikipedia.org/wiki/Idempotence) expression 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 if you call it twice consecutively. If the expression returns an object, including an array, it returns the same object reference if you call it twice consecutively.

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 changing values in objects when iterating over them. See [\*ngFor with trackBy](https://angular.io/guide/built-in-directives#ngfor-with-trackby) for details.

# Template statements

Template statements are methods or properties that you can use in your HTML to respond to user events. With template statements, your application can engage users through actions such as displaying dynamic content or submitting forms.

See the [Template syntax](https://angular.io/generated/live-examples/template-syntax/stackblitz.html) / [download example](https://angular.io/generated/zips/template-syntax/template-syntax.zip) for the syntax and code snippets in this guide.

In the following example, the template statement deleteHero() appears in quotes to the right of the = symbol as in (event)="statement".

src/app/app.component.html

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

When the user clicks the **Delete hero** button, Angular calls the deleteHero() method in the component class.

You can use template statements with elements, components, or directives in response to events.

Responding to events is an aspect of Angular's [unidirectional data flow](https://angular.io/guide/glossary#unidirectional-data-flow). You can change anything in your application during a single event loop.

## **Syntax**

Like [template expressions](https://angular.io/guide/interpolation), template statements use a language that looks like JavaScript. However, the parser for template statements differs from the parser for template expressions. In addition, the template statements parser specifically supports both basic assignment, =, and chaining expressions with semicolons, ;.

The following JavaScript and template expression syntax is not allowed:

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

## **Statement context**

Statements have a context—a particular part of the application to which the statement belongs.

Statements can refer only to what's in the statement context, which is typically the component instance. For example, deleteHero() of (click)="deleteHero()" is a method of the component in the following snippet.

src/app/app.component.html

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

The statement context may also refer to properties of the template's own context. In the following example, the component's event handling method, onSave() takes the template's own $event object as an argument. On the next two lines, the deleteHero() method takes a [template input variable](https://angular.io/guide/structural-directives#shorthand), hero, and onSubmit() takes a [template reference variable](https://angular.io/guide/template-reference-variables), #heroForm.

src/app/app.component.html

content\_copy<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>

In this example, the context of the $event object, hero, and #heroForm is the template.

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

## **Statement best practices**

* **Conciseness**

Keep template statements minimal by using method calls or basic property assignments.

* **Work within the context**

The context of a template statement can be the component class instance or the template. Because of this, template statements cannot refer to anything in the global namespace such as window or document. For example, template statements can't call console.log() or Math.max().

# Transforming Data Using Pipes

Use [pipes](https://angular.io/guide/glossary#pipe) to transform strings, currency amounts, dates, and other data for display. Pipes are simple functions you can use in [template expressions](https://angular.io/guide/glossary#template-expression) to accept an input value and return a transformed value. Pipes are useful because you can use them throughout your application, while only declaring each pipe once. For example, you would use a pipe to show a date as **April 15, 1988** rather than the raw string format.

For the sample application used in this topic, see the [live example](https://angular.io/generated/live-examples/pipes/stackblitz.html) / [download example](https://angular.io/generated/zips/pipes/pipes.zip).

Angular provides built-in pipes for typical data transformations, including transformations for internationalization (i18n), which use locale information to format data. The following are commonly used built-in pipes for data formatting:

* [DatePipe](https://angular.io/api/common/DatePipe): Formats a date value according to locale rules.
* [UpperCasePipe](https://angular.io/api/common/UpperCasePipe): Transforms text to all upper case.
* [LowerCasePipe](https://angular.io/api/common/LowerCasePipe): Transforms text to all lower case.
* [CurrencyPipe](https://angular.io/api/common/CurrencyPipe): Transforms a number to a currency string, formatted according to locale rules.
* [DecimalPipe](https://angular.io/api/common/DecimalPipe): Transforms a number into a string with a decimal point, formatted according to locale rules.
* [PercentPipe](https://angular.io/api/common/PercentPipe): Transforms a number to a percentage string, formatted according to locale rules.
* For a complete list of built-in pipes, see the [pipes API documentation](https://angular.io/api/common#pipes).
* To learn more about using pipes for internationalization (i18n) efforts, see [formatting data based on locale](https://angular.io/guide/i18n#i18n-pipes).

You can also create pipes to encapsulate custom transformations and use your custom pipes in template expressions.

## **Prerequisites**

To use pipes you should have a basic understanding of the following:

* [Typescript](https://angular.io/guide/glossary#typescript) and HTML5 programming
* [Templates](https://angular.io/guide/glossary#template) in HTML with CSS styles
* [Components](https://angular.io/guide/glossary#component)

## **Using a pipe in a template**

To apply a pipe, use the pipe operator (|) within a template expression as shown in the following code example, along with the name of the pipe, which is [date](https://angular.io/api/common/DatePipe) for the built-in [DatePipe](https://angular.io/api/common/DatePipe). The tabs in the example show the following:

* app.component.html uses [date](https://angular.io/api/common/DatePipe) in a separate template to display a birthday.
* hero-birthday1.component.ts uses the same pipe as part of an in-line template in a component that also sets the birthday value.

src/app/app.component.html

src/app/hero-birthday1.component.ts

content\_copy<p>The hero's birthday is {{ birthday | [date](https://angular.io/api/common/DatePipe) }}</p>

The component's birthday value flows through the pipe operator, | to the [date](https://angular.io/api/common/DatePipe) function.

## **Transforming data with parameters and chained pipes**

Use optional parameters to fine-tune a pipe's output. For example, you can use the [CurrencyPipe](https://angular.io/api/common/CurrencyPipe) with a country code such as EUR as a parameter. The template expression {{ amount | [currency](https://angular.io/api/common/CurrencyPipe):'EUR' }} transforms the amount to currency in euros. Follow the pipe name ([currency](https://angular.io/api/common/CurrencyPipe)) with a colon (:) and the parameter value ('EUR').

If the pipe accepts multiple parameters, separate the values with colons. For example, {{ amount | [currency](https://angular.io/api/common/CurrencyPipe):'EUR':'Euros '}} adds the second parameter, the string literal 'Euros ', to the output string. You can use any valid template expression as a parameter, such as a string literal or a component property.

Some pipes require at least one parameter and allow more optional parameters, such as [SlicePipe](https://angular.io/api/common/SlicePipe). For example, {{ slice:1:5 }} creates a new array or string containing a subset of the elements starting with element 1 and ending with element 5.

### **Example: Formatting a date**

The tabs in the following example demonstrates toggling between two different formats ('shortDate' and 'fullDate'):

* The app.component.html template uses a format parameter for the [DatePipe](https://angular.io/api/common/DatePipe) (named [date](https://angular.io/api/common/DatePipe)) to show the date as **04/15/88**.
* The hero-birthday2.component.ts component binds the pipe's format parameter to the component's format property in the template section, and adds a button for a click event bound to the component's toggleFormat() method.
* The hero-birthday2.component.ts component's toggleFormat() method toggles the component's format property between a short form ('shortDate') and a longer form ('fullDate').

src/app/app.component.html

src/app/hero-birthday2.component.ts (template)

src/app/hero-birthday2.component.ts (class)

content\_copy<p>The hero's birthday is {{ birthday | [date](https://angular.io/api/common/DatePipe):"MM/dd/yy" }} </p>

Clicking the **Toggle Format** button alternates the date format between **04/15/1988** and **Friday, April 15, 1988**.

For [date](https://angular.io/api/common/DatePipe) pipe format options, see [DatePipe](https://angular.io/api/common/DatePipe).

### **Example: Applying two formats by chaining pipes**

You can chain pipes so that the output of one pipe becomes the input to the next.

In the following example, chained pipes first apply a format to a date value, then convert the formatted date to uppercase characters. The first tab for the src/app/app.component.html template chains [DatePipe](https://angular.io/api/common/DatePipe) and [UpperCasePipe](https://angular.io/api/common/UpperCasePipe) to display the birthday as **APR 15, 1988**. The second tab for the src/app/app.component.html template passes the fullDate parameter to [date](https://angular.io/api/common/DatePipe) before chaining to [uppercase](https://angular.io/api/common/UpperCasePipe), which produces **FRIDAY, APRIL 15, 1988**.

src/app/app.component.html (1)

src/app/app.component.html (2)

content\_copyThe chained hero's birthday is

{{ birthday | [date](https://angular.io/api/common/DatePipe) | [uppercase](https://angular.io/api/common/UpperCasePipe)}}

## **Creating pipes for custom data transformations**

Create custom pipes to encapsulate transformations that are not provided with the built-in pipes. You can then use your custom pipe in template expressions, the same way you use built-in pipes—to transform input values to output values for display.

### **Marking a class as a pipe**

To mark a class as a pipe and supply configuration metadata, apply the [@Pipe](https://angular.io/api/core/Pipe) [decorator](https://angular.io/guide/glossary#decorator--decoration) to the class. Use [UpperCamelCase](https://angular.io/guide/glossary#case-types) (the general convention for class names) for the pipe class name, and [camelCase](https://angular.io/guide/glossary#case-types) for the corresponding name string. Do not use hyphens in the name. For details and more examples, see [Pipe names](https://angular.io/guide/styleguide#pipe-names).

Use name in template expressions as you would for a built-in pipe.

* Include your pipe in the declarations field of the [NgModule](https://angular.io/api/core/NgModule) metadata in order for it to be available to a template. See the app.module.ts file in the example application ([live example](https://angular.io/generated/live-examples/pipes/stackblitz.html) / [download example](https://angular.io/generated/zips/pipes/pipes.zip)). For details, see [NgModules](https://angular.io/guide/ngmodules).
* Register your custom pipes. The [Angular CLI](https://angular.io/cli) [ng generate pipe](https://angular.io/cli/generate#pipe) command registers the pipe automatically.

### **Using the PipeTransform interface**

Implement the [PipeTransform](https://angular.io/api/core/PipeTransform) interface in your custom pipe class to perform the transformation.

Angular invokes the transform method with the value of a binding as the first argument, and any parameters as the second argument in list form, and returns the transformed value.

### **Example: Transforming a value exponentially**

In a game, you may want to implement a transformation that raises a value exponentially to increase a hero's power. For example, if the hero's score is 2, boosting the hero's power exponentially by 10 produces a score of 1024. You can use a custom pipe for this transformation.

The following code example shows two component definitions:

* The exponential-strength.pipe.ts component defines a custom pipe named exponentialStrength with the transform method that performs the transformation. It defines an argument to the transform method (exponent) for a parameter passed to the pipe.
* The power-booster.component.ts component demonstrates how to use the pipe, specifying a value (2) and the exponent parameter (10).

src/app/exponential-strength.pipe.ts

src/app/power-booster.component.ts

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

/\*

\* Raise the value exponentially

\* Takes an exponent argument that defaults to 1.

\* Usage:

\* value | exponentialStrength:exponent

\* Example:

\* {{ 2 | exponentialStrength:10 }}

\* formats to: 1024

\*/

@[Pipe](https://angular.io/api/core/Pipe)({name: 'exponentialStrength'})

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

transform(value: number, exponent = 1): number {

return Math.pow(value, exponent);

}

}

The browser displays the following:

content\_copyPower Booster

Superpower boost: 1024

To examine the behavior the exponentialStrength pipe in the [live example](https://angular.io/generated/live-examples/pipes/stackblitz.html) / [download example](https://angular.io/generated/zips/pipes/pipes.zip), change the value and optional exponent in the template.

## **Detecting changes with data binding in pipes**

You use [data binding](https://angular.io/guide/glossary#data-binding) with a pipe to display values and respond to user actions. If the data is a primitive input value, such as String or Number, or an object reference as input, such as Date or Array, Angular executes the pipe whenever it detects a change for the input value or reference.

For example, you could change the previous custom pipe example to use two-way data binding with [ngModel](https://angular.io/api/forms/NgModel) to input the amount and boost factor, as shown in the following code example.

src/app/power-boost-calculator.component.ts

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

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

selector: 'app-power-boost-calculator',

template: `

<h2>Power Boost Calculator</h2>

<label for="power-input">Normal power: </label>

<input id="power-input" type="text" [([ngModel](https://angular.io/api/forms/NgModel))]="power">

<label for="boost-input">Boost factor: </label>

<input id="boost-input" type="text" [([ngModel](https://angular.io/api/forms/NgModel))]="factor">

<p>

Super Hero Power: {{power | exponentialStrength: factor}}

</p>

`,

styles: ['input {margin: .5rem 0;}']

})

export class PowerBoostCalculatorComponent {

power = 5;

factor = 1;

}

The exponentialStrength pipe executes every time the user changes the "normal power" value or the "boost factor".

Angular detects each change and immediately runs the pipe. This is fine for primitive input values. However, if you change something inside a composite object (such as the month of a date, an element of an array, or an object property), you need to understand how change detection works, and how to use an impure pipe.

### **How change detection works**

Angular looks for changes to data-bound values in a [change detection](https://angular.io/guide/glossary#change-detection) process that runs after every DOM event: every keystroke, mouse move, timer tick, and server response. The following example, which doesn't use a pipe, demonstrates how Angular uses its default change detection strategy to monitor and update its display of every hero in the heroes array. The example tabs show the following:

* In the flying-heroes.component.html (v1) template, the \*[ngFor](https://angular.io/api/common/NgForOf) repeater displays the hero names.
* Its companion component class flying-heroes.component.ts (v1) provides heroes, adds heroes into the array, and resets the array.

src/app/flying-heroes.component.html (v1)

src/app/flying-heroes.component.ts (v1)

content\_copy<label for="hero-name">New hero name: </label>

<input type="text" #box

id="hero-name"

(keyup.enter)="addHero(box.value); box.value=''"

placeholder="hero name">

<button (click)="reset()">Reset list of heroes</button>

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

{{hero.name}}

</div>

Angular updates the display every time the user adds a hero. If the user clicks the **Reset** button, Angular replaces heroes with a new array of the original heroes and updates the display. If you add the ability to remove or change a hero, Angular would detect those changes and update the display as well.

However, executing a pipe to update the display with every change would slow down your application's performance. So Angular uses a faster change-detection algorithm for executing a pipe, as described in the next section.

### **Detecting pure changes to primitives and object references**

By default, pipes are defined as pure so that Angular executes the pipe only when it detects a pure change to the input value. A pure change is either a change to a primitive input value (such as String, Number, Boolean, or Symbol), or a changed object reference (such as Date, Array, Function, or Object).

A pure pipe must use a pure function, which is one that processes inputs and returns values without side effects. In other words, given the same input, a pure function should always return the same output.

With a pure pipe, Angular ignores changes within composite objects, such as a newly added element of an existing array, because checking a primitive value or object reference is much faster than performing a deep check for differences within objects. Angular can quickly determine if it can skip executing the pipe and updating the view.

However, a pure pipe with an array as input may not work the way you want. To demonstrate this issue, change the previous example to filter the list of heroes to just those heroes who can fly. Use the FlyingHeroesPipe in the \*[ngFor](https://angular.io/api/common/NgForOf) repeater as shown in the following code. The tabs for the example show the following:

* The template (flying-heroes.component.html (flyers)) with the new pipe.
* The FlyingHeroesPipe custom pipe implementation (flying-heroes.pipe.ts).

src/app/flying-heroes.component.html (flyers)

src/app/flying-heroes.pipe.ts

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

{{hero.name}}

</div>

The application now shows unexpected behavior: When the user adds flying heroes, none of them appear under "Heroes who fly." This happens because the code that adds a hero does so by pushing it onto the heroes array:

src/app/flying-heroes.component.ts

content\_copythis.heroes.push(hero);

The change detector ignores changes to elements of an array, so the pipe doesn't run.

The reason Angular ignores the changed array element is that the reference to the array hasn't changed. Since the array is the same, Angular does not update the display.

One way to get the behavior you want is to change the object reference itself. You can replace the array with a new array containing the newly changed elements, and then input the new array to the pipe. In the above example, you can create an array with the new hero appended, and assign that to heroes. Angular detects the change in the array reference and executes the pipe.

To summarize, if you mutate the input array, the pure pipe doesn't execute. If you replace the input array, the pipe executes and the display is updated.

The above example demonstrates changing a component's code to accommodate a pipe.

To keep your component independent of HTML templates that use pipes, you can, as an alternative, use an impure pipe to detect changes within composite objects such as arrays, as described in the next section.

### **Detecting impure changes within composite objects**

To execute a custom pipe after a change within a composite object, such as a change to an element of an array, you need to define your pipe as impure to detect impure changes. Angular executes an impure pipe every time it detects a change with every keystroke or mouse movement.

While an impure pipe can be useful, be careful using one. A long-running impure pipe could dramatically slow down your application.

Make a pipe impure by setting its pure flag to false:

src/app/flying-heroes.pipe.ts

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

name: 'flyingHeroesImpure',

pure: false

})

The following code shows the complete implementation of FlyingHeroesImpurePipe, which extends FlyingHeroesPipe to inherit its characteristics. The example shows that you don't have to change anything else—the only difference is setting the pure flag as false in the pipe metadata.

src/app/flying-heroes.pipe.ts (FlyingHeroesImpurePipe)

src/app/flying-heroes.pipe.ts (FlyingHeroesPipe)

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

name: 'flyingHeroesImpure',

pure: false

})

export class FlyingHeroesImpurePipe extends FlyingHeroesPipe {}

FlyingHeroesImpurePipe is a good candidate for an impure pipe because the transform function is trivial and fast:

src/app/flying-heroes.pipe.ts (filter)

content\_copyreturn allHeroes.filter(hero => hero.canFly);

You can derive a FlyingHeroesImpureComponent from FlyingHeroesComponent. As shown in the code below, only the pipe in the template changes.

src/app/flying-heroes-impure.component.html (excerpt)

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

{{hero.name}}

</div>

To confirm that the display updates as the user adds heroes, see the [live example](https://angular.io/generated/live-examples/pipes/stackblitz.html) / [download example](https://angular.io/generated/zips/pipes/pipes.zip).

## **Unwrapping data from an observable**

[Observables](https://angular.io/guide/glossary#observable) let you pass messages between parts of your application. Observables are recommended for event handling, asynchronous programming, and handling multiple values. Observables can deliver single or multiple values of any type, either synchronously (as a function delivers a value to its caller) or asynchronously on a schedule.

For details and examples of observables, see the [Observables Overview](https://angular.io/guide/observables#using-observables-to-pass-values).

Use the built-in [AsyncPipe](https://angular.io/api/common/AsyncPipe) to accept an observable as input and subscribe to the input automatically. Without this pipe, your component code would have to subscribe to the observable to consume its values, extract the resolved values, expose them for binding, and unsubscribe when the observable is destroyed in order to prevent memory leaks. [AsyncPipe](https://angular.io/api/common/AsyncPipe) is an impure pipe that saves boilerplate code in your component to maintain the subscription and keep delivering values from that observable as they arrive.

The following code example binds an observable of message strings (message$) to a view with the [async](https://angular.io/api/common/AsyncPipe) pipe.

src/app/hero-async-message.component.ts

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

import { Observable, interval } from 'rxjs';

import { map, take } from 'rxjs/operators';

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

selector: 'app-hero-async-message',

template: `

<h2>Async Hero Message and [AsyncPipe](https://angular.io/api/common/AsyncPipe)</h2>

<p>Message: {{ message$ | async }}</p>

<button (click)="resend()">Resend</button>`,

})

export class HeroAsyncMessageComponent {

message$: Observable<string>;

private messages = [

'You are my hero!',

'You are the best hero!',

'Will you be my hero?'

];

constructor() {

this.message$ = this.getResendObservable();

}

resend() {

this.message$ = this.getResendObservable();

}

private getResendObservable() {

return interval(500).pipe(

map(i => this.messages[i]),

take(this.messages.length)

);

}

}

## **Caching HTTP requests**

To [communicate with backend services using HTTP](https://angular.io/guide/http), the [HttpClient](https://angular.io/api/common/http/HttpClient) service uses observables and offers the [HttpClient.get()](https://angular.io/api/common/http/HttpClient#get) method to fetch data from a server. The asynchronous method sends an HTTP request, and returns an observable that emits the requested data for the response.

As shown in the previous section, you can use the impure [AsyncPipe](https://angular.io/api/common/AsyncPipe) to accept an observable as input and subscribe to the input automatically. You can also create an impure pipe to make and cache an HTTP request.

Impure pipes are called whenever change detection runs for a component, which could be as often as every few milliseconds. To avoid performance problems, call the server only when the requested URL changes, as shown in the following example, and use the pipe to cache the server response. The tabs show the following:

* The fetch pipe (fetch-json.pipe.ts).
* A harness component (hero-list.component.ts) for demonstrating the request, using a template that defines two bindings to the pipe requesting the heroes from the heroes.json file. The second binding chains the fetch pipe with the built-in [JsonPipe](https://angular.io/api/common/JsonPipe) to display the same hero data in JSON format.

src/app/fetch-json.pipe.ts

src/app/hero-list.component.ts

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

import { [Pipe](https://angular.io/api/core/Pipe), [PipeTransform](https://angular.io/api/core/PipeTransform) } from '@angular/core';

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

name: 'fetch',

pure: false

})

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

private cachedData: any = null;

private cachedUrl = '';

constructor(private [http](https://angular.io/api/common/http): [HttpClient](https://angular.io/api/common/http/HttpClient)) { }

transform(url: string): any {

if (url !== this.cachedUrl) {

this.cachedData = null;

this.cachedUrl = url;

this.http.get(url).subscribe(result => this.cachedData = result);

}

return this.cachedData;

}

}

In the above example, a breakpoint on the pipe's request for data shows the following:

* Each binding gets its own pipe instance.
* Each pipe instance caches its own URL and data and calls the server only once.

The fetch and fetch-json pipes display the heroes in the browser as follows:

content\_copyHeroes from JSON File

Windstorm

Bombasto

Magneto

Tornado

Heroes as JSON: [ { "name": "Windstorm", "canFly": true }, { "name": "Bombasto", "canFly": false }, { "name": "Magneto", "canFly": false }, { "name": "Tornado", "canFly": true } ]

The built-in [JsonPipe](https://angular.io/api/common/JsonPipe) provides a way to diagnose a mysteriously failing data binding or to inspect an object for future binding.

## **Pipes and precedence**

The pipe operator has a higher precedence than the ternary operator (?:), which means a ? b : c | x is parsed as a ? b : (c | x). The pipe operator cannot be used without parentheses in the first and second operands of ?:.

Due to precedence, if you want a pipe to apply to the result of a ternary, wrap the entire expression in parentheses; for example, (a ? b : c) | x.

src/app/precedence.component.html

content\_copy<!-- use parentheses in the third operand so the pipe applies to the whole expression -->

{{ (true ? 'true' : 'false') | uppercase }}

# Property binding

Property binding in Angular helps you set values for properties of HTML elements or directives. With property binding, you can do things such as toggle button functionality, set paths programmatically, and share values between components.

See the [live example](https://angular.io/generated/live-examples/property-binding/stackblitz.html) / [download example](https://angular.io/generated/zips/property-binding/property-binding.zip) for a working example containing the code snippets in this guide.

## **Prerequisites**

To get the most out of property binding, you should be familiar with the following:

* [Basics of components](https://angular.io/guide/architecture-components)
* [Basics of templates](https://angular.io/guide/glossary#template)
* [Binding syntax](https://angular.io/guide/binding-syntax)

## **Understanding the flow of data**

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

For more information on listening for events, see [Event binding](https://angular.io/guide/event-binding).

To 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).

## **Binding to a property**

To bind to an element's property, enclose it in square brackets, [], which identifies the property as a target property. A target property is the DOM property to which you want to assign a value. For example, the target property in the following code is the image element's src property.

src/app/app.component.html

content\_copy<img [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. In this example, src is the name of the <img> element property.

The brackets, [], cause Angular to evaluate the right-hand side of the assignment as a dynamic expression. Without the brackets, Angular treats the right-hand side as a string literal and sets the property to that static value.

src/app.component.html

content\_copy<app-item-detail childItem="parentItem"></app-item-detail>

Omitting the brackets renders the string parentItem, not the value of parentItem.

## **Setting an element property to a component property value**

To bind the src property of an <img> element to a component's property, place the target, src, in square brackets followed by an equal sign and then the property. The property here is itemImageUrl.

src/app/app.component.html

content\_copy<img [src]="itemImageUrl">

Declare the itemImageUrl property in the class, in this case AppComponent.

src/app/app.component.ts

content\_copyitemImageUrl = '../assets/phone.png';

#### colspan and colSpan

A common point of confusion is between the attribute, colspan, and the property, colSpan. Notice that these two names differ by only a single letter.

If you wrote something like this:

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

You'd get this error:

content\_copyTemplate parse errors:

Can't bind to 'colspan' since it isn't a known native 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

content\_copy<!-- Notice the colSpan property is camel case -->

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

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

src/app/app.component.html

content\_copy<!-- Bind button disabled state to `isUnchanged` property -->

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

Another is setting a property of a directive:

src/app/app.component.html

content\_copy<p [[ngClass](https://angular.io/api/common/NgClass)]="classes">[[ngClass](https://angular.io/api/common/NgClass)] binding to the 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

content\_copy<app-item-detail [childItem]="parentItem"></app-item-detail>

## **Toggling button functionality**

To disable a button's functionality depending on a Boolean value, bind the DOM disabled property to a property in the class that is true or false.

src/app/app.component.html

content\_copy<!-- Bind button disabled state to `isUnchanged` property -->

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

Because the value of the property isUnchanged is true in the AppComponent, Angular disables the button.

src/app/app.component.ts

content\_copyisUnchanged = true;

## **Setting a directive property**

To set a property of a directive, place the directive within square brackets , such as [[ngClass](https://angular.io/api/common/NgClass)], followed by an equal sign and the property. Here, the property is classes.

src/app/app.component.html

content\_copy<p [[ngClass](https://angular.io/api/common/NgClass)]="classes">[[ngClass](https://angular.io/api/common/NgClass)] binding to the classes property making this blue</p>

To use the property, you must declare it in the class, which in this example is AppComponent. The value of classes is special.

src/app/app.component.ts

content\_copyclasses = 'special';

Angular applies the class special to the <p> element so that you can use special to apply CSS styles.

## **Bind values between components**

To set the model property of a custom component, place the target, here childItem, between square brackets [] followed by an equal sign and the property. Here, the property is parentItem.

src/app/app.component.html

content\_copy<app-item-detail [childItem]="parentItem"></app-item-detail>

To use the target and the property, you must declare them in their respective classes.

Declare the target of childItem in its component class, in this case ItemDetailComponent.

For example, the following code declares the target of childItem in its component class, in this case ItemDetailComponent.

Then, the code contains an @[Input](https://angular.io/api/core/Input)() decorator with the childItem property so data can flow into it.

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

content\_copy@[Input](https://angular.io/api/core/Input)() childItem = '';

Next, the code declares the property of parentItem in its component class, in this case AppComponent. In this example the type of childItem is string, so parentItem needs to be a string. Here, parentItem has the string value of lamp.

src/app/app.component.ts

content\_copyparentItem = 'lamp';

With this configuration, the view of <app-item-detail> uses the value of lamp for childItem.

## **Property binding and security**

Property binding can help keep content secure. For example, consider the following malicious content.

src/app/app.component.ts

content\_copyevilTitle = 'Template <script>alert("evil never sleeps")</script> Syntax';

The component template interpolates the content as follows:

src/app/app.component.html

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

The browser doesn't process the HTML and instead displays it raw, as follows.

content\_copy"Template <script>alert("evil never sleeps")</script> Syntax" is the interpolated evil title.

Angular does not allow HTML with <script> tags, neither with [interpolation](https://angular.io/guide/interpolation) nor property binding, which prevents the JavaScript from running.

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

content\_copy<!--

Angular generates a warning for the following line as it sanitizes them

WARNING: sanitizing HTML stripped some content (see https://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 sanitized evilTitle example.

content\_copy"Template Syntax" is the property bound evil title.

## **Property binding and interpolation**

Often [interpolation](https://angular.io/guide/interpolation) and property binding can achieve the same results. The following binding pairs do the same thing.

src/app/app.component.html

content\_copy<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>

You can use either form when rendering data values as strings, though interpolation is preferable for readability. However, when setting an element property to a non-string data value, you must use property binding.

## **What's next**

* [Property binding best practices](https://angular.io/guide/property-binding-best-practices)

# Attribute, class, and style bindings

Attribute binding in Angular helps you set values for attributes directly. With attribute binding, you can improve accessibility, style your application dynamically, and manage multiple CSS classes or styles simultaneously.

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) for a working example containing the code snippets in this guide.

## **Binding to an attribute**

It is recommended that you set an element property with a [property binding](https://angular.io/guide/property-binding) whenever possible. However, sometimes you don't have an element property to bind. In those situations, you can use attribute binding.

For example, [ARIA](https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA) and [SVG](https://developer.mozilla.org/en-US/docs/Web/SVG) are purely attributes. Neither ARIA nor SVG correspond to element properties and don't set element properties. In these cases, you must use attribute binding because there are no corresponding property targets.

## **Syntax**

Attribute binding syntax resembles [property binding](https://angular.io/guide/property-binding), but instead of an element property between brackets, you precede the name of the attribute with the prefix attr, followed by a dot. Then, you set the attribute value with an expression that resolves to a string.

content\_copy<p [attr.attribute-you-are-targeting]="expression"></p>

When the expression resolves to null or undefined, Angular removes the attribute altogether.

## **Binding ARIA attributes**

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

src/app/app.component.html

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

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

## **Binding to colspan**

Another common use case for attribute binding is with the colspan attribute in tables. Binding to the colspan attribute helps you keep your tables programmatically dynamic. Depending on the amount of data that your application populates a table with, the number of columns that a row spans could change.

To use attribute binding with the <td> attribute colspan:

1. Specify the colspan attribute by using the following syntax: [attr.colspan].
2. Set [attr.colspan] equal to an expression.

In the following example, we bind the colspan attribute to the expression 1 + 1.

src/app/app.component.html

content\_copy<!-- expression calculates colspan=2 -->

<tr><td [attr.colspan]="1 + 1">One-Two</td></tr>

This binding causes the <tr> to span two columns.

Sometimes there are differences between the name of property and an attribute.

colspan is an attribute of <tr>, while colSpan with a capital "S" is a property. When using attribute binding, use colspan with a lowercase "s". For more information on how to bind to the colSpan property, see the [colspan and colSpan](https://angular.io/guide/property-binding#colspan) section of [Property Binding](https://angular.io/guide/property-binding).

## **Binding to the class attribute**

You can use class binding to add and remove CSS class names from an element's class attribute.

### **Binding to a single CSS class**

To create a single class binding, use the prefix class followed by a dot and the name of the CSS class—for example, [class.sale]="onSale". Angular adds the class when the bound expression, onSale is truthy, and it removes the class when the expression is falsy—with the exception of undefined. See [styling delegation](https://angular.io/guide/style-precedence#styling-delegation) for more information.

### **Binding to multiple CSS classes**

To bind to multiple classes, use [class] set to an expression—for example, [class]="classExpression". The expression can be one of:

* A space-delimited string of class names.
* An object with class names as the keys and truthy or falsy expressions as the values.
* An array of class names.

With the object format, Angular adds a class only if its associated value is truthy.

With any object-like expression—such as object, Array, Map, or Set—the identity of the object must change for Angular to update the class list. Updating the property without changing object identity has no effect.

If there are multiple bindings to the same class name, Angular uses [styling precedence](https://angular.io/guide/style-precedence) to determine which binding to use.

The following table summarizes class binding syntax.

|  |  |  |  |
| --- | --- | --- | --- |
| **Binding Type** | **Syntax** | **Input Type** | **Example Input Values** |
| Single class binding | [class.sale]="onSale" | boolean | undefined | null | true, false |
| Multi-class binding | [class]="classExpression" | string | "my-class-1 my-class-2 my-class-3" |
| Record<string, boolean | undefined | null> | {foo: true, bar: false} |
| Array<string> | ['foo', 'bar'] |

## **Binding to the style attribute**

You can use style binding to set styles dynamically.

### **Binding to a single style**

To create a single style binding, use the prefix [style](https://angular.io/api/animations/style) followed by a dot and the name of the CSS style property—for example, [style.width]="width". Angular sets the property to the value of the bound expression, which is usually a string. Optionally, you can add a unit extension like em or %, which requires a number type.

You can write a style property name in either [dash-case](https://angular.io/guide/glossary#dash-case), or [camelCase](https://angular.io/guide/glossary#camelcase).

content\_copy<nav [style.background-color]="expression"></nav>

<nav [style.backgroundColor]="expression"></nav>

### **Binding to multiple styles**

To toggle multiple styles, bind to the [[style](https://angular.io/api/animations/style)] attribute—for example, [[style](https://angular.io/api/animations/style)]="styleExpression". The styleExpression can be one of:

* A string list of styles such as "width: 100px; height: 100px; background-color: cornflowerblue;".
* An object with style names as the keys and style values as the values, such as {width: '100px', height: '100px', backgroundColor: 'cornflowerblue'}.

Note that binding an array to [[style](https://angular.io/api/animations/style)] is not supported.

When binding [[style](https://angular.io/api/animations/style)] to an object expression, the identity of the object must change for Angular to update the class list. Updating the property without changing object identity has no effect.

#### Single and multiple-style binding example

nav-bar.component.ts

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

selector: 'app-nav-bar',

template: `

<nav [[style](https://angular.io/api/animations/style)]='navStyle'>

<a [style.text-decoration]="activeLinkStyle">Home Page</a>

<a [style.text-decoration]="linkStyle">Login</a>

</nav>`

})

export class NavBarComponent {

navStyle = 'font-size: 1.2rem; color: cornflowerblue;';

linkStyle = 'underline';

activeLinkStyle = 'overline';

/\* . . . \*/

}

If there are multiple bindings to the same style attribute, Angular uses [styling precedence](https://angular.io/guide/style-precedence) to determine which binding to use.

The following table summarizes style binding syntax.

|  |  |  |  |
| --- | --- | --- | --- |
| **Binding Type** | **Syntax** | **Input Type** | **Example Input Values** |
| Single style binding | [style.width]="width" | string | undefined | null | "100px" |
|  |  |  |  |
| Single style binding with units | [style.width.px]="width" | [number](https://angular.io/api/common/DecimalPipe) | undefined | null | 100 |
| Multi-style binding | [[style](https://angular.io/api/animations/style)]="styleExpression" | string | "width: 100px; height: 100px" |
| Record<string, string | undefined | null> | {width: '100px', height: '100px'} |

The [NgStyle](https://angular.io/guide/built-in-directives/#ngstyle) directive can be used as an alternative to direct [[style](https://angular.io/api/animations/style)] bindings. However, using the above style binding syntax without [NgStyle](https://angular.io/api/common/NgStyle) is preferred because due to improvements in style binding in Angular, [NgStyle](https://angular.io/api/common/NgStyle) no longer provides significant value, and might eventually be removed in the future.

## **Styling Precedence**

A single HTML element can have its CSS class list and style values bound to multiple sources (for example, host bindings from multiple directives).

When there are multiple bindings to the same class name or style property, Angular uses a set of precedence rules to resolve conflicts and determine which classes or styles are ultimately applied to the element.

#### Styling precedence (highest to lowest)

1. Template bindings
   1. Property binding (for example, <div [class.foo]="hasFoo"> or <div [style.color]="color">)
   2. Map binding (for example, <div [class]="classExpr"> or <div [[style](https://angular.io/api/animations/style)]="styleExpr">)
   3. Static value (for example, <div class="foo"> or <div [style](https://angular.io/api/animations/style)="color: blue">)
2. Directive host bindings
   1. Property binding (for example, host: {'[class.foo]': 'hasFoo'} or host: {'[style.color]': 'color'})
   2. Map binding (for example, host: {'[class]': 'classExpr'} or host: {'[[style](https://angular.io/api/animations/style)]': 'styleExpr'})
   3. Static value (for example, host: {'class': 'foo'} or host: {'[style](https://angular.io/api/animations/style)': 'color: blue'})
3. Component host bindings
   1. Property binding (for example, host: {'[class.foo]': 'hasFoo'} or host: {'[style.color]': 'color'})
   2. Map binding (for example, host: {'[class]': 'classExpr'} or host: {'[[style](https://angular.io/api/animations/style)]': 'styleExpr'})
   3. Static value (for example, host: {'class': 'foo'} or host: {'[style](https://angular.io/api/animations/style)': 'color: blue'})

The more specific a class or style binding is, the higher its precedence.

A binding to a specific class (for example, [class.foo]) will take precedence over a generic [class] binding, and a binding to a specific style (for example, [style.bar]) will take precedence over a generic [[style](https://angular.io/api/animations/style)] binding.

src/app/app.component.html

content\_copy<h3>Basic specificity</h3>

<!-- The `class.special` binding overrides any value for the `special` class in `classExpression`. -->

<div [class.special]="isSpecial" [class]="classExpression">Some text.</div>

<!-- The `style.border` binding overrides any value for the `border` property in `styleExpression`. -->

<div [style.border]="border" [[style](https://angular.io/api/animations/style)]="styleExpression">Some text.</div>

Specificity rules also apply when it comes to bindings that originate from different sources. It's possible for an element to have bindings in the template where it's declared, from host bindings on matched directives, and from host bindings on matched components.

Template bindings are the most specific because they apply to the element directly and exclusively, so they have the highest precedence.

Directive host bindings are considered less specific because directives can be used in multiple locations, so they have a lower precedence than template bindings.

Directives often augment component behavior, so host bindings from components have the lowest precedence.

src/app/app.component.html

content\_copy<h3>Source specificity</h3>

<!-- The `class.special` template binding overrides any host binding to the `special` class set by `dirWithClassBinding` or `comp-with-host-binding`.-->

<comp-with-host-binding [class.special]="isSpecial" dirWithClassBinding></comp-with-host-binding>

<!-- The `style.color` template binding overrides any host binding to the `color` property set by `dirWithStyleBinding` or `comp-with-host-binding`. -->

<div>

<comp-with-host-binding [style.color]="color" dirWithStyleBinding></comp-with-host-binding>

</div>

<h3>Dynamic vs [static](https://angular.io/api/upgrade/static)</h3>

<!-- If `classExpression` has a value for the `special` class, this value overrides the `class="special"` below -->

<div class="special" [class]="classExpression">Some text.</div>

<!-- If `styleExpression` has a value for the `border` property, this value overrides the `[style](https://angular.io/api/animations/style)="border: dotted darkblue 3px"` below -->

<div [style](https://angular.io/api/animations/style)="border: dotted darkblue 3px" [[style](https://angular.io/api/animations/style)]="styleExpression">Some text.</div>

<div class="readability">

<comp-with-host-binding dirWithHostBinding></comp-with-host-binding>

</div>

<app-my-input-with-attribute-decorator type="number"></app-my-input-with-attribute-decorator>

In addition, bindings take precedence over static attributes.

In the following case, class and [class] have similar specificity, but the [class] binding will take precedence because it is dynamic.

src/app/app.component.html

content\_copy<h3>Dynamic vs [static](https://angular.io/api/upgrade/static)</h3>

<!-- If `classExpression` has a value for the `special` class, this value overrides the `class="special"` below -->

<div class="special" [class]="classExpression">Some text.</div>

<!-- If `styleExpression` has a value for the `border` property, this value overrides the `[style](https://angular.io/api/animations/style)="border: dotted darkblue 3px"` below -->

<div [style](https://angular.io/api/animations/style)="border: dotted darkblue 3px" [[style](https://angular.io/api/animations/style)]="styleExpression">Some text.</div>

### **Delegating to styles with lower precedence**

It is possible for higher precedence styles to "delegate" to lower precedence styles using undefined values. Whereas setting a style property to null ensures the style is removed, setting it to undefined will cause Angular to fall back to the next-highest precedence binding to that style.

For example, consider the following template:

src/app/app.component.html

content\_copy<comp-with-host-binding dirWithHostBinding></comp-with-host-binding>

Imagine that the dirWithHostBinding directive and the comp-with-host-binding component both have a [style.width] host binding. In that case, if dirWithHostBinding sets its binding to undefined, the width property will fall back to the value of the comp-with-host-binding host binding. However, if dirWithHostBinding sets its binding to null, the width property will be removed entirely.

## **Injecting attribute values**

There are cases where you need to differentiate the behavior of a [Component](https://angular.io/api/core/Component) or [Directive](https://angular.io/api/core/Directive) based on a static value set on the host element as an HTML attribute. For example, you might have a directive that needs to know the type of a <button> or <input> element.

The [Attribute](https://angular.io/api/core/Attribute) parameter decorator is great for passing the value of an HTML attribute to a component/directive constructor using [dependency injection](https://angular.io/guide/dependency-injection).

The injected value captures the value of the specified HTML attribute at that moment. Future updates to the attribute value are not reflected in the injected value.

src/app/my-input-with-attribute-decorator.component.ts

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

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

selector: 'app-my-input-with-attribute-decorator',

template: '<p>The type of the input is: {{ type }}</p>'

})

export class MyInputWithAttributeDecoratorComponent {

constructor(@[Attribute](https://angular.io/api/core/Attribute)('type') public type: string) { }

}

src/app/app.component.html

content\_copy<app-my-input-with-attribute-decorator type="number"></app-my-input-with-attribute-decorator>

In the preceding example, the result of app.component.html is **The type of the input is: number**.

Another example is the [RouterOutlet](https://angular.io/api/router/RouterOutlet) directive, which makes use of the [Attribute](https://angular.io/api/core/Attribute) decorator to retrieve the unique [name](https://angular.io/api/router/RouterOutlet#description) on each outlet.

@ATTRIBUTE() VS @INPUT()

Remember, use [@Input()](https://angular.io/api/core/Input) when you want to keep track of the attribute value and update the associated property. Use [@Attribute()](https://angular.io/api/core/Attribute) when you want to inject the value of an HTML attribute to a component or directive constructor.

# Event binding

Event binding allows you to listen for and respond to user actions such as keystrokes, mouse movements, clicks, and touches.

See the [live example](https://angular.io/generated/live-examples/event-binding/stackblitz.html) / [download example](https://angular.io/generated/zips/event-binding/event-binding.zip) for a working example containing the code snippets in this guide.

## **Binding to events**

To bind to an event you use the Angular event binding syntax. This syntax consists of a target event name within parentheses to the left of an equal sign, and a quoted template statement to the right. In the following example, the target event name is click and the template statement is onSave().

Event binding syntax

content\_copy<button (click)="onSave()">Save</button>

The event binding listens for the button's click events and calls the component's onSave() method whenever a click occurs.

## **Binding to passive events**

Angular also supports passive event listeners. For example, you can use the following steps to make a scroll event passive.

1. Create a file zone-flags.ts under src directory.
2. Add the following line into this file.

content\_copy(window as any)['\_\_zone\_symbol\_\_PASSIVE\_EVENTS'] = ['scroll'];

1. In the src/polyfills.ts file, before importing zone.js, import the newly created zone-flags.

content\_copyimport './zone-flags';

import 'zone.js'; // Included with Angular CLI.

After those steps, if you add event listeners for the scroll event, the listeners will be passive.

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

[Directives](https://angular.io/guide/built-in-directives) typically raise custom events with an Angular [EventEmitter](https://angular.io/api/core/EventEmitter) as follows.

1. The directive creates an [EventEmitter](https://angular.io/api/core/EventEmitter) and exposes it as a property.
2. The directive then calls EventEmitter.emit(data) to emit an event, passing in message data, which can be anything.
3. Parent directives listen for the event by binding to this property and accessing the data 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 contain the functionality to delete the hero. It can only raise an event reporting the user's delete request.

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

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

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

</span>

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

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.

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

content\_copy// This component makes a request but it can't actually delete a 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 hosting parent component binds to the deleteRequest event of the ItemDetailComponent as follows.

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

content\_copy<app-item-detail (deleteRequest)="deleteItem($event)" [item]="currentItem"></app-item-detail>

When the deleteRequest event fires, Angular calls the parent component's deleteItem() method with the item.

### **Determining an event target**

To determine an event target, Angular checks if the name of the target event matches an event property of a known directive. In the following example, Angular checks to see if myClick is an event on the custom ClickDirective.

src/app/app.component.html

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

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

{{clickMessage}}

If the target event name, myClick fails to match an element event or an output property of ClickDirective, Angular reports an "unknown directive" error.

## **What's next**

For more information on how event binding works, see [How event binding works](https://angular.io/guide/event-binding-concepts).

# Two-way binding

Two-way binding gives components in your application a way to share data. Use two-way binding to listen for events and update values simultaneously between parent and child components.

See the [live 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) for a working example containing the code snippets in this guide.

## **Prerequisites**

To get the most out of two-way binding, you should have a basic understanding of the following concepts:

* [Property binding](https://angular.io/guide/property-binding)
* [Event binding](https://angular.io/guide/event-binding)
* [Inputs and Outputs](https://angular.io/guide/inputs-outputs)

Two-way binding combines property binding with event binding:

* [Property binding](https://angular.io/guide/property-binding) sets a specific element property.
* [Event binding](https://angular.io/guide/event-binding) listens for an element change event.

## **Adding two-way data binding**

Angular's two-way binding syntax is a combination of square brackets and parentheses, [()]. The [()] syntax combines the brackets of property binding, [], with the parentheses of event binding, (), as follows.

src/app/app.component.html

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

## **How two-way binding works**

For two-way data binding to work, the @[Output](https://angular.io/api/core/Output)() property must use the pattern, inputChange, where input is the name of the @[Input](https://angular.io/api/core/Input)() property. For example, if the @[Input](https://angular.io/api/core/Input)() property is size, the @[Output](https://angular.io/api/core/Output)() property must be sizeChange.

The following sizerComponent has a size value property and a sizeChange event. The size property is an @[Input](https://angular.io/api/core/Input)(), so data can flow into the sizerComponent. The sizeChange event is an @[Output](https://angular.io/api/core/Output)(), which allows data to flow out of the sizerComponent to the parent component.

Next, there are two methods, dec() to decrease the font size and inc() to increase the font size. These two methods use resize() to change the value of the size property within min/max value constraints, and to emit an event that conveys the new size value.

src/app/sizer.component.ts

content\_copyexport 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 sizerComponent template has two buttons that each bind the click event to the inc() and dec() methods. When the user clicks one of the buttons, the sizerComponent calls the corresponding method. Both methods, inc() and dec(), call the resize() method with a +1 or -1, which in turn raises the sizeChange event with the new size value.

src/app/sizer.component.html

content\_copy<div>

<button (click)="dec()" title="smaller">-</button>

<button (click)="inc()" title="bigger">+</button>

<label [style.font-size.px]="size">FontSize: {{size}}px</label>

</div>

In the AppComponent template, fontSizePx is two-way bound to the SizerComponent.

src/app/app.component.html

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

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

In the AppComponent, fontSizePx establishes the initial SizerComponent.size value by setting the value to 16.

src/app/app.component.ts

content\_copyfontSizePx = 16;

Clicking the buttons updates the AppComponent.fontSizePx. The revised AppComponent.fontSizePx value updates the style binding, which makes the displayed text bigger or smaller.

The two-way binding syntax is shorthand for a combination of property binding and event binding. The SizerComponent binding as separate property binding and event binding is as follows.

src/app/app.component.html (expanded)

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

The $event variable contains the data 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

Because no native HTML element follows the x value and xChange event pattern, two-way binding with form elements requires [NgModel](https://angular.io/api/forms/NgModel). For more information on how to use two-way binding in forms, see Angular [NgModel](https://angular.io/guide/built-in-directives#ngModel).

# Template variables

Template variables help you use data from one part of a template in another part of the template. With template variables, you can perform tasks such as respond to user input or finely tune your application's forms.

A template variable can refer to the following:

* a DOM element within a template
* a directive
* an element
* [TemplateRef](https://angular.io/api/core/TemplateRef)
* a [web component](https://developer.mozilla.org/en-US/docs/Web/Web_Components)

See the [live 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) for a working example containing the code snippets in this guide.

## **Syntax**

In the template, you use the hash symbol, #, to declare a template variable. The following template variable, #phone, declares a phone variable on an <input> element.

src/app/app.component.html

content\_copy<input #phone placeholder="phone number" />

You can refer to a template variable anywhere in the component's template. Here, a <button> further down the template refers to the phone variable.

src/app/app.component.html

content\_copy<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 Angular assigns values to template variables**

Angular assigns a template variable a value based on where you declare the variable:

* If you declare the variable on a component, the variable refers to the component instance.
* If you declare the variable on a standard HTML tag, the variable refers to the element.
* If you declare the variable on an [<ng-template>](https://angular.io/api/core/ng-template) element, the variable refers to a [TemplateRef](https://angular.io/api/core/TemplateRef) instance, which represents the template. For more information on [<ng-template>](https://angular.io/api/core/ng-template), see [How Angular uses the asterisk, \*, syntax](https://angular.io/guide/structural-directives#asterisk) in [Structural directives](https://angular.io/guide/structural-directives).
* If the variable specifies a name on the right-hand side, such as #var="[ngModel](https://angular.io/api/forms/NgModel)", the variable refers to the directive or component on the element with a matching exportAs name.

### **Using**[**NgForm**](https://angular.io/api/forms/NgForm)**with template variables**

In most cases, Angular sets the template variable's value to the element on which it occurs. 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 demonstrates getting a reference to a different value by reference a directive's exportAs name. In the following example, the template variable, itemForm, appears three times separated by HTML.

src/app/hero-form.component.html

content\_copy<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>

Without the [ngForm](https://angular.io/api/forms/NgForm) attribute value, the reference value of itemForm would be the [HTMLFormElement](https://developer.mozilla.org/en-US/docs/Web/API/HTMLFormElement), <form>. There is, however, a difference between a [Component](https://angular.io/api/core/Component) and a [Directive](https://angular.io/api/core/Directive) in that Angular references a [Component](https://angular.io/api/core/Component) without specifying the attribute value, and a [Directive](https://angular.io/api/core/Directive) does not change the implicit reference, or the element.

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.

Unlike the native <form> element, the [NgForm](https://angular.io/api/forms/NgForm) directive has a form property. The [NgForm](https://angular.io/api/forms/NgForm) form property allows you to disable the submit button if the itemForm.form.valid is invalid.

## **Template variable scope**

You can refer to a template variable anywhere within its surrounding template. [Structural directives](https://angular.io/guide/built-in-directives), such as \*[ngIf](https://angular.io/api/common/NgIf) and \*[ngFor](https://angular.io/api/common/NgForOf), or [<ng-template>](https://angular.io/api/core/ng-template) act as a template boundary. You cannot access template variables outside of these boundaries.

Define a variable only once in the template so the runtime value remains predictable.

### **Accessing in a nested template**

An inner template can access template variables that the outer template defines.

In the following example, changing the text in the <input> changes the value in the <span> because Angular immediately updates changes through the template variable, ref1.

src/app/app.component.html

content\_copy<input #ref1 type="text" [([ngModel](https://angular.io/api/forms/NgModel))]="firstExample" />

<span \*[ngIf](https://angular.io/api/common/NgIf)="true">Value: {{ ref1.value }}</span>

In this case, there is an implied [<ng-template>](https://angular.io/api/core/ng-template) around the <span> and the definition of the variable is outside of it. Accessing a template variable from the parent template works because the child template inherits the context from the parent template.

Rewriting the above code in a more verbose form explicitly shows the [<ng-template>](https://angular.io/api/core/ng-template).

content\_copy<input #ref1 type="text" [([ngModel](https://angular.io/api/forms/NgModel))]="firstExample" />

<!-- New template -->

<ng-template [[ngIf](https://angular.io/api/common/NgIf)]="true">

<!-- Since the context is inherited, the value is available to the new template -->

<span>Value: {{ ref1.value }}</span>

</ng-template>

However, accessing a template variable from outside the parent template doesn't work.

content\_copy<input \*[ngIf](https://angular.io/api/common/NgIf)="true" #ref2 type="text" [([ngModel](https://angular.io/api/forms/NgModel))]="secondExample" />

<span>Value: {{ ref2?.value }}</span> <!-- doesn't work -->

The verbose form shows that ref2 is outside the parent template.

content\_copy<ng-template [[ngIf](https://angular.io/api/common/NgIf)]="true">

<!-- The reference is defined within a template -->

<input #ref2 type="text" [([ngModel](https://angular.io/api/forms/NgModel))]="secondExample" />

</ng-template>

<!-- ref2 accessed from outside that template doesn't work -->

<span>Value: {{ ref2?.value }}</span>

Consider the following example that uses \*[ngFor](https://angular.io/api/common/NgForOf).

content\_copy<ng-container \*[ngFor](https://angular.io/api/common/NgForOf)="let i of [1,2]">

<input #ref type="text" [value]="i" />

</ng-container>

{{ ref.value }}

Here, ref.value doesn't work. The structural directive, \*[ngFor](https://angular.io/api/common/NgForOf) instantiates the template twice because \*[ngFor](https://angular.io/api/common/NgForOf) iterates over the two items in the array. It is impossible to define what the ref.value reference signifies.

With structural directives, such as \*[ngFor](https://angular.io/api/common/NgForOf) or \*[ngIf](https://angular.io/api/common/NgIf), there is no way for Angular to know if a template is ever instantiated.

As a result, Angular isn't able to access the value and returns an error.

### **Accessing a template variable within**[**<ng-template>**](https://angular.io/api/core/ng-template)

When you declare the variable on an [<ng-template>](https://angular.io/api/core/ng-template), the variable refers to a [TemplateRef](https://angular.io/api/core/TemplateRef) instance, which represents the template.

src/app/app.component.html

content\_copy<ng-template #ref3></ng-template>

<button (click)="log(ref3)">Log type of #ref</button>

In this example, clicking the button calls the log() function, which outputs the value of #ref3 to the console. Because the #ref variable is on an [<ng-template>](https://angular.io/api/core/ng-template), the value is [TemplateRef](https://angular.io/api/core/TemplateRef).

The following is the expanded browser console output of the [TemplateRef](https://angular.io/api/core/TemplateRef)() function with the name of [TemplateRef](https://angular.io/api/core/TemplateRef).

content\_copy▼ ƒ TemplateRef()

name: "TemplateRef"

\_\_proto\_\_: Function

## **Template input variable**

A template input variable is a variable you can reference within a single instance of the template. You declare a template input variable using the let keyword as in let hero.

There are several such variables in this example: hero, i, and odd.

The variable's scope is limited to a single instance of the repeated template. You can use the same variable name again in the definition of other structural directives.

In contrast, you declare a template variable by prefixing the variable name with #, as in #var. A template variable refers to its attached element, component, or directive.

Template input variables and template variables names have their own namespaces. The template input variable hero in let hero is distinct from the template variable hero in #hero.

# SVG as templates

You can use SVG files as templates in your Angular applications. When you use an SVG as the template, you are able to use directives and bindings just like with HTML templates. With these features, you can dynamically generate interactive graphics.

See the [live example](https://angular.io/generated/live-examples/template-syntax/stackblitz.html) / [download example](https://angular.io/generated/zips/template-syntax/template-syntax.zip) for a working example containing the code snippets in this guide.

## **SVG syntax example**

The following example shows the syntax for using an SVG as a template.

src/app/svg.component.ts

content\_copyimport { [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: ['./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})`;

}

}

To see property and event binding in action, add the following code to your svg.component.svg file:

src/app/svg.component.svg

content\_copy<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>

The example given uses a click() event binding and the property binding syntax ([attr.fill]="fillColor").

# Built-in directives

Directives are classes that add additional behavior to elements in your Angular applications. With Angular's built-in directives, you can manage forms, lists, styles, and what users see.

See the [live 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 a working example containing the code snippets in this guide.

The different types of Angular directives are as follows:

1. [Components](https://angular.io/guide/component-overview)—directives with a template. This type of directive is the most common directive type.
2. [Attribute directives](https://angular.io/guide/built-in-directives#built-in-attribute-directives)—directives that change the appearance or behavior of an element, component, or another directive.
3. [Structural directives](https://angular.io/guide/built-in-directives#built-in-structural-directives)—directives that change the DOM layout by adding and removing DOM elements.

This guide covers built-in [attribute directives](https://angular.io/guide/built-in-directives#built-in-attribute-directives) and [structural directives](https://angular.io/guide/built-in-directives#built-in-structural-directives).

## **Built-in attribute directives**

Attribute directives listen to and modify the behavior of other HTML elements, attributes, properties, and components.

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

Built-in directives use only public APIs. They do not have special access to any private APIs that other directives can't access.

## **Adding and removing classes with**[**NgClass**](https://angular.io/api/common/NgClass)

You can add or remove multiple CSS classes simultaneously with [ngClass](https://angular.io/api/common/NgClass).

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

### **Using**[**NgClass**](https://angular.io/api/common/NgClass)**with an expression**

On the element you'd like to style, add [[ngClass](https://angular.io/api/common/NgClass)] and set it equal to an expression. In this case, isSpecial is a boolean set to true in app.component.ts. Because isSpecial is true, [ngClass](https://angular.io/api/common/NgClass) applies the class of special to the <div>.

src/app/app.component.html

content\_copy<!-- toggle the "special" class on/off with a property -->

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

### **Using**[**NgClass**](https://angular.io/api/common/NgClass)**with a method**

1. To use [NgClass](https://angular.io/api/common/NgClass) with a method, add the method to the component class. In the following example, setCurrentClasses() sets the property currentClasses with an object that adds or removes three classes based on the true or false state of three other component properties.

Each key of the object is a CSS class name. If a key is true, [ngClass](https://angular.io/api/common/NgClass) adds the class. If a key is false, [ngClass](https://angular.io/api/common/NgClass) removes the class.

src/app/app.component.ts

content\_copycurrentClasses: Record<string, boolean> = {};

/\* . . . \*/

setCurrentClasses() {

// CSS classes: added/removed per current state of component properties

this.currentClasses = {

saveable: this.canSave,

modified: !this.isUnchanged,

special: this.isSpecial

};

}

1. In the template, add the [ngClass](https://angular.io/api/common/NgClass) property binding to currentClasses to set the element's classes:

src/app/app.component.html

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

For this use case, Angular applies the classes on initialization and in case of changes. The full example calls setCurrentClasses() initially with ngOnInit() and when the dependent properties change through a button click. These steps are not necessary to implement [ngClass](https://angular.io/api/common/NgClass). For more information, see the [live 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) app.component.ts and app.component.html.

## **Setting inline styles with**[**NgStyle**](https://angular.io/api/common/NgStyle)

You can use [NgStyle](https://angular.io/api/common/NgStyle) to set multiple inline styles simultaneously, based on the state of the component.

1. To use [NgStyle](https://angular.io/api/common/NgStyle), add a method to the component class.

In the following example, setCurrentStyles() sets the property currentStyles with an object that defines three styles, based on the state of three other component properties.

src/app/app.component.ts

content\_copycurrentStyles: Record<string, string> = {};

/\* . . . \*/

setCurrentStyles() {

// CSS styles: set per current state of component properties

this.currentStyles = {

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

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

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

};

}

1. To set the element's styles, add an [ngStyle](https://angular.io/api/common/NgStyle) property binding to currentStyles.

src/app/app.component.html

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

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

</div>

For this use case, Angular applies the styles upon initialization and in case of changes. To do this, the full example calls setCurrentStyles() initially with ngOnInit() and when the dependent properties change through a button click. However, these steps are not necessary to implement [ngStyle](https://angular.io/api/common/NgStyle) on its own. See the [live 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) app.component.ts and app.component.html for this optional implementation.

## **Displaying and updating properties with**[**ngModel**](https://angular.io/api/forms/NgModel)

You can use the [NgModel](https://angular.io/api/forms/NgModel) directive to display a data property and update that property when the user makes changes.

1. Import [FormsModule](https://angular.io/api/forms/FormsModule) and add it to the NgModule's imports list.

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

content\_copyimport { [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 { }

1. Add an [([ngModel](https://angular.io/api/forms/NgModel))] binding on an HTML <form> element and set it equal to the property, here name.

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

content\_copy<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">

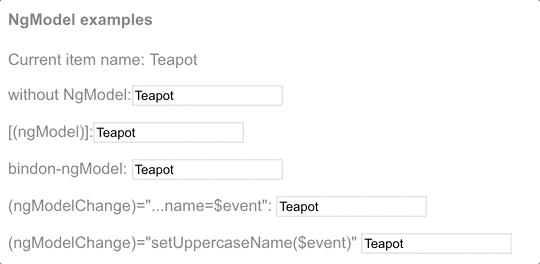
This [([ngModel](https://angular.io/api/forms/NgModel))] syntax can only set a data-bound property.

To customize your configuration, you can write the expanded form, which separates the property and event binding. Use [property binding](https://angular.io/guide/property-binding) to set the property and [event binding](https://angular.io/guide/event-binding) to respond to changes. The following example changes the <input> value to uppercase:

src/app/app.component.html

content\_copy<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:



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

The [NgModel](https://angular.io/api/forms/NgModel) directive works for an element supported by a [ControlValueAccessor](https://angular.io/api/forms/ControlValueAccessor). Angular provides value accessors for all of the basic HTML form elements. For more information, see [Forms](https://angular.io/guide/forms).

To apply [([ngModel](https://angular.io/api/forms/NgModel))] to a non-form native element or a third-party custom component, you have to write a value accessor. For more information, see the API documentation on [DefaultValueAccessor](https://angular.io/api/forms/DefaultValueAccessor).

When you write an Angular component, you don't need a value accessor or [NgModel](https://angular.io/api/forms/NgModel) if you name the value and event properties according to Angular's [two-way binding syntax](https://angular.io/guide/two-way-binding#how-two-way-binding-works).

## **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 introduces the most common built-in structural directives:

* [NgIf](https://angular.io/guide/built-in-directives#ngIf)—conditionally creates or disposes of subviews from the template.
* [NgFor](https://angular.io/guide/built-in-directives#ngFor)—repeat a node for each item in a list.
* [NgSwitch](https://angular.io/guide/built-in-directives#ngSwitch)—a set of directives that switch among alternative views.

For more information, see [Structural Directives](https://angular.io/guide/structural-directives).

## **Adding or removing an element with**[**NgIf**](https://angular.io/api/common/NgIf)

You can add or remove an element by applying an [NgIf](https://angular.io/api/common/NgIf) directive to a host element.

When [NgIf](https://angular.io/api/common/NgIf) is false, Angular removes an element and its descendants from the DOM. Angular then disposes of their components, which frees up memory and resources.

To add or remove an element, bind \*[ngIf](https://angular.io/api/common/NgIf) to a condition expression such as isActive in the following example.

src/app/app.component.html

content\_copy<app-item-detail \*[ngIf](https://angular.io/api/common/NgIf)="isActive" [item]="item"></app-item-detail>

When the 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 and disposes of the component and all of its sub-components.

For more information on [NgIf](https://angular.io/api/common/NgIf) and NgIfElse, see the [NgIf API documentation](https://angular.io/api/common/NgIf).

### **Guarding against null**

By default, [NgIf](https://angular.io/api/common/NgIf) prevents display of an element bound to a null value.

To use [NgIf](https://angular.io/api/common/NgIf) to guard a <div>, add \*[ngIf](https://angular.io/api/common/NgIf)="yourProperty" to the <div>. In the following example, the currentCustomer name appears because there is a currentCustomer.

src/app/app.component.html

content\_copy<div \*[ngIf](https://angular.io/api/common/NgIf)="currentCustomer">Hello, {{currentCustomer.name}}</div>

However, if the property is null, Angular does not display the <div>. In this example, Angular does not display the nullCustomer because it is null.

src/app/app.component.html

content\_copy<div \*[ngIf](https://angular.io/api/common/NgIf)="nullCustomer">Hello, <span>{{nullCustomer}}</span></div>

## **Listing items with NgFor**

You can use the NgFor directive to present a list of items.

1. Define a block of HTML that determines how Angular renders a single item.
2. To list your items, assign the short hand let item of items to \*[ngFor](https://angular.io/api/common/NgForOf).

src/app/app.component.html

content\_copy<div \*[ngFor](https://angular.io/api/common/NgForOf)="let item of items">{{item.name}}</div>

The string "let item of items" instructs Angular to do the following:

* Store each item in the items array in the local item looping variable
* Make each item available to the templated HTML for each iteration
* Translate "let item of items" into an [<ng-template>](https://angular.io/api/core/ng-template) around the host element
* Repeat the [<ng-template>](https://angular.io/api/core/ng-template) for each item in the list

For more information see the [Structural directive shorthand](https://angular.io/guide/structural-directives#shorthand) section of [Structural directives](https://angular.io/guide/structural-directives).

### **Repeating a component view**

To repeat a component element, apply \*[ngFor](https://angular.io/api/common/NgForOf) to the selector. In the following example, the selector is <app-item-detail>.

src/app/app.component.html

content\_copy<app-item-detail \*[ngFor](https://angular.io/api/common/NgForOf)="let item of items" [item]="item"></app-item-detail>

You can reference a template input variable, such as item, in the following locations:

* within the [ngFor](https://angular.io/api/common/NgForOf) host element
* within the host element descendants to access the item's properties

The following example references item first in an interpolation and then passes in a binding to the item property of the <app-item-detail> component.

src/app/app.component.html

content\_copy<div \*[ngFor](https://angular.io/api/common/NgForOf)="let item of items">{{item.name}}</div>

<!-- . . . -->

<app-item-detail \*[ngFor](https://angular.io/api/common/NgForOf)="let item of items" [item]="item"></app-item-detail>

For more information about template input variables, see [Structural directive shorthand](https://angular.io/guide/structural-directives#shorthand).

### **Getting the index of \***[**ngFor**](https://angular.io/api/common/NgForOf)

You can get the index of \*[ngFor](https://angular.io/api/common/NgForOf) in a template input variable and use it in the template.

In the \*[ngFor](https://angular.io/api/common/NgForOf), add a semicolon and let i=index to the short hand. The following example gets the index in a variable named i and displays it with the item name.

src/app/app.component.html

content\_copy<div \*[ngFor](https://angular.io/api/common/NgForOf)="let item of items; let i=index">{{i + 1}} - {{item.name}}</div>

The index property of the NgFor directive context returns the zero-based index of the item in each iteration.

Angular translates this instruction into an [<ng-template>](https://angular.io/api/core/ng-template) around the host element, then uses this template repeatedly to create a new set of elements and bindings for each item in the list. For more information about shorthand, see the [Structural Directives](https://angular.io/guide/structural-directives#shorthand) guide.

## **Repeating elements when a condition is true**

To repeat a block of HTML when a particular condition is true, put the \*[ngIf](https://angular.io/api/common/NgIf) on a container element that wraps an \*[ngFor](https://angular.io/api/common/NgForOf) element. One or both elements can be an [<ng-container>](https://angular.io/api/core/ng-container) so you don't have to introduce extra levels of HTML.

Because structural directives add and remove nodes from the DOM, apply only one structural directive per element.

For more information about NgFor see the [NgForOf API reference](https://angular.io/api/common/NgForOf).

### **Tracking items with \***[**ngFor**](https://angular.io/api/common/NgForOf)**trackBy**

By tracking changes to an item list, you can reduce the number of calls your application makes to the server. With the \*[ngFor](https://angular.io/api/common/NgForOf) trackBy property, Angular can change and re-render only those items that have changed, rather than reloading the entire list of items.

1. Add a method to the component that returns the value NgFor should track. In this example, the value to track is the item's id. If the browser has already rendered id, Angular keeps track of it and doesn't re-query the server for the same id.

src/app/app.component.ts

content\_copytrackByItems(index: number, item: Item): number { return item.id; }

1. In the short hand expression, set trackBy to the trackByItems() method.

src/app/app.component.html

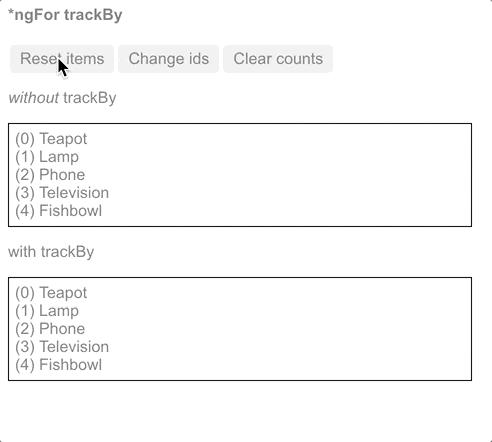
content\_copy<div \*[ngFor](https://angular.io/api/common/NgForOf)="let item of items; trackBy: trackByItems">

({{item.id}}) {{item.name}}

</div>

**Change ids** creates new items with new item.ids. In the following illustration of the trackBy effect, **Reset items** creates new items with the same item.ids.

* With no trackBy, both buttons trigger complete DOM element replacement.
* With trackBy, only changing the id triggers element replacement.



## **Hosting a directive without a DOM element**

The Angular [<ng-container>](https://angular.io/api/core/ng-container) is a grouping element that doesn't interfere with styles or layout because Angular doesn't put it in the DOM.

You can use [<ng-container>](https://angular.io/api/core/ng-container) when there's no single element to host the directive.

Here's a conditional paragraph using [<ng-container>](https://angular.io/api/core/ng-container).

src/app/app.component.html (ngif-ngcontainer)

content\_copy<p>

I turned the corner

<ng-container \*[ngIf](https://angular.io/api/common/NgIf)="hero">

and saw {{hero.name}}. I waved

</ng-container>

and continued on my way.

</p>

ngcontainer paragraph with proper style

1. Import the [ngModel](https://angular.io/api/forms/NgModel) directive from [FormsModule](https://angular.io/api/forms/FormsModule).
2. Add [FormsModule](https://angular.io/api/forms/FormsModule) to the imports section of the relevant Angular module.
3. To conditionally exclude an <option>, wrap the <option> in an [<ng-container>](https://angular.io/api/core/ng-container).

src/app/app.component.html (select-ngcontainer)

content\_copy<div>

Pick your favorite hero

(<label><input type="checkbox" checked (change)="showSad = !showSad">show sad</label>)

</div>

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

<ng-container \*[ngFor](https://angular.io/api/common/NgForOf)="let h of heroes">

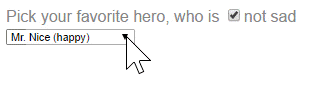
<ng-container \*[ngIf](https://angular.io/api/common/NgIf)="showSad || h.emotion !== 'sad'">

<option [ngValue]="h">{{h.name}} ({{h.emotion}})</option>

</ng-container>

</ng-container>

</select>



## **Switching cases with**[**NgSwitch**](https://angular.io/api/common/NgSwitch)

Like the JavaScript switch statement, [NgSwitch](https://angular.io/api/common/NgSwitch) 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 a set of three directives:

* [NgSwitch](https://angular.io/api/common/NgSwitch)—an attribute directive that changes the behavior of its companion directives.
* [NgSwitchCase](https://angular.io/api/common/NgSwitchCase)—structural directive that 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)—structural directive that adds its element to the DOM when there is no selected [NgSwitchCase](https://angular.io/api/common/NgSwitchCase).

1. On an element, such as a <div>, add [[ngSwitch](https://angular.io/api/common/NgSwitch)] bound 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.
2. Bind to \*[ngSwitchCase](https://angular.io/api/common/NgSwitchCase) and \*[ngSwitchDefault](https://angular.io/api/common/NgSwitchDefault) on the elements for the cases.

src/app/app.component.html

content\_copy<div [[ngSwitch](https://angular.io/api/common/NgSwitch)]="currentItem.feature">

<app-stout-item \*[ngSwitchCase](https://angular.io/api/common/NgSwitchCase)="'stout'" [item]="currentItem"></app-stout-item>

<app-device-item \*[ngSwitchCase](https://angular.io/api/common/NgSwitchCase)="'slim'" [item]="currentItem"></app-device-item>

<app-lost-item \*[ngSwitchCase](https://angular.io/api/common/NgSwitchCase)="'vintage'" [item]="currentItem"></app-lost-item>

<app-best-item \*[ngSwitchCase](https://angular.io/api/common/NgSwitchCase)="'bright'" [item]="currentItem"></app-best-item>

<!-- . . . -->

<app-unknown-item \*[ngSwitchDefault](https://angular.io/api/common/NgSwitchDefault) [item]="currentItem"></app-unknown-item>

</div>

1. In the parent component, define currentItem so you can use it in the [[ngSwitch](https://angular.io/api/common/NgSwitch)] expression.

src/app/app.component.ts

content\_copycurrentItem!: Item;

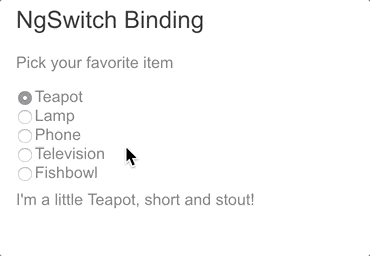
1. In each child component, add an item [input property](https://angular.io/guide/inputs-outputs#input) which is bound to the currentItem of the parent component. The following two snippets show the parent component and one of the child components. The other child components are identical to StoutItemComponent.

In each child component, here StoutItemComponent

content\_copyexport class StoutItemComponent {

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

}



Switch directives also work with native HTML elements and web components. For example, you could replace the <app-best-item> switch case with a <div> as follows.

src/app/app.component.html

content\_copy<div \*[ngSwitchCase](https://angular.io/api/common/NgSwitchCase)="'bright'"> Are you as bright as {{currentItem.name}}?</div>

## **What's next**

For information on 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).

# Attribute directives

With attribute directives, you can change the appearance or behavior of DOM elements and Angular components.

See the [live example](https://angular.io/generated/live-examples/attribute-directives/stackblitz.html) / [download example](https://angular.io/generated/zips/attribute-directives/attribute-directives.zip) for a working example containing the code snippets in this guide.

## **Building an attribute directive**

This section walks you through creating a highlight directive that sets the background color of the host element to yellow.

1. To create a directive, use the CLI command [ng generate directive](https://angular.io/cli/generate).

content\_copyng generate directive highlight

The CLI creates src/app/highlight.directive.ts, a corresponding test file src/app/highlight.directive.spec.ts, and declares the directive class in the AppModule.

The CLI generates the default src/app/highlight.directive.ts as follows:

src/app/highlight.directive.ts

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

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

selector: '[appHighlight]'

})

export class HighlightDirective {

constructor() { }

}

The @[Directive](https://angular.io/api/core/Directive)() decorator's configuration property specifies the directive's CSS attribute selector, [appHighlight].

1. Import [ElementRef](https://angular.io/api/core/ElementRef) from @angular/core. [ElementRef](https://angular.io/api/core/ElementRef) grants direct access to the host DOM element through its nativeElement property.
2. Add [ElementRef](https://angular.io/api/core/ElementRef) in the directive's constructor() to [inject](https://angular.io/guide/dependency-injection) a reference to the host DOM element, the element to which you apply appHighlight.
3. Add logic to the HighlightDirective class that sets the background to yellow.

src/app/highlight.directive.ts

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

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

selector: '[appHighlight]'

})

export class HighlightDirective {

constructor(el: [ElementRef](https://angular.io/api/core/ElementRef)) {

el.nativeElement.style.backgroundColor = 'yellow';

}

}

Directives do not support namespaces.

src/app/app.component.avoid.html (unsupported)

content\_copy<p app:Highlight>This is invalid</p>

## **Applying an attribute directive**

1. To use the HighlightDirective, add a <p> element to the HTML template with the directive as an attribute.

src/app/app.component.html

content\_copy<p appHighlight>Highlight me!</p>

Angular creates an instance of the HighlightDirective class and injects a reference to the <p> element into the directive's constructor, which sets the <p> element's background style to yellow.

## **Handling user events**

This section shows you how to detect when a user mouses into or out of the element and to respond by setting or clearing the highlight color.

1. Import [HostListener](https://angular.io/api/core/HostListener) from '@angular/core'.

src/app/highlight.directive.ts (imports)

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

1. Add two event handlers that respond when the mouse enters or leaves, each with the @[HostListener](https://angular.io/api/core/HostListener)() decorator.

src/app/highlight.directive.ts (mouse-methods)

content\_copy@[HostListener](https://angular.io/api/core/HostListener)('mouseenter') onMouseEnter() {

this.highlight('yellow');

}

@[HostListener](https://angular.io/api/core/HostListener)('mouseleave') onMouseLeave() {

this.highlight('');

}

private highlight(color: string) {

this.el.nativeElement.style.backgroundColor = color;

}

With the @[HostListener](https://angular.io/api/core/HostListener)() decorator, you can subscribe to events of the DOM element that hosts an attribute directive, the <p> in this case.

The handlers delegate to a helper method, highlight(), that sets the color on the host DOM element, el.

The complete directive is as follows:

src/app/highlight.directive.ts

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

selector: '[appHighlight]'

})

export class HighlightDirective {

constructor(private el: [ElementRef](https://angular.io/api/core/ElementRef)) { }

@[HostListener](https://angular.io/api/core/HostListener)('mouseenter') onMouseEnter() {

this.highlight('yellow');

}

@[HostListener](https://angular.io/api/core/HostListener)('mouseleave') onMouseLeave() {

this.highlight('');

}

private highlight(color: string) {

this.el.nativeElement.style.backgroundColor = color;

}

}

The background color appears when the pointer hovers over the paragraph element and disappears as the pointer moves out.



## **Passing values into an attribute directive**

This section walks you through setting the highlight color while applying the HighlightDirective.

1. In highlight.directive.ts, import [Input](https://angular.io/api/core/Input) from @angular/core.

src/app/highlight.directive.ts (imports)

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

1. Add an appHighlight @[Input](https://angular.io/api/core/Input)() property.

src/app/highlight.directive.ts

content\_copy@[Input](https://angular.io/api/core/Input)() appHighlight = '';

The @[Input](https://angular.io/api/core/Input)() decorator adds metadata to the class that makes the directive's appHighlight property available for binding.

1. In app.component.ts, add a color property to the AppComponent.

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

content\_copyexport class AppComponent {

color = 'yellow';

}

1. To simultaneously apply the directive and the color, use property binding with the appHighlight directive selector, setting it equal to color.

src/app/app.component.html (color)

content\_copy<p [appHighlight]="color">Highlight me!</p>

The [appHighlight] attribute binding performs two tasks:

* + applies the highlighting directive to the <p> element
  + sets the directive's highlight color with a property binding

### **Setting the value with user input**

This section guides you through adding radio buttons to bind your color choice to the appHighlight directive.

1. Add markup to app.component.html for choosing a color as follows:

src/app/app.component.html (v2)

content\_copy<h1>My First [Attribute](https://angular.io/api/core/Attribute) [Directive](https://angular.io/api/core/Directive)</h1>

<h2>Pick a highlight color</h2>

<div>

<input type="radio" name="colors" (click)="color='lightgreen'">Green

<input type="radio" name="colors" (click)="color='yellow'">Yellow

<input type="radio" name="colors" (click)="color='cyan'">Cyan

</div>

<p [appHighlight]="color">Highlight me!</p>

1. Revise the AppComponent.color so that it has no initial value.

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

content\_copyexport class AppComponent {

color = '';

}

1. Serve your application to verify that the user can choose the color with the radio buttons.



## **Binding to a second property**

This section guides you through configuring your application so the developer can set the default color.

1. Add a second [Input](https://angular.io/api/core/Input)() property to HighlightDirective called defaultColor.

src/app/highlight.directive.ts (defaultColor)

content\_copy@[Input](https://angular.io/api/core/Input)() defaultColor = '';

1. Revise the directive's onMouseEnter so that it first tries to highlight with the highlightColor, then with the defaultColor, and falls back to red if both properties are undefined.

src/app/highlight.directive.ts (mouse-enter)

content\_copy@[HostListener](https://angular.io/api/core/HostListener)('mouseenter') onMouseEnter() {

this.highlight(this.highlightColor || this.defaultColor || 'red');

}

1. To bind to the AppComponent.color and fall back to "violet" as the default color, add the following HTML. In this case, the defaultColor binding doesn't use square brackets, [], because it is static.

src/app/app.component.html (defaultColor)

content\_copy<p [appHighlight]="color" defaultColor="violet">

Highlight me too!

</p>

As with components, you can add multiple directive property bindings to a host element.

The default color is red if there is no default color binding. When the user chooses a color the selected color becomes the active highlight color.



## **Deactivating Angular processing with NgNonBindable**

To prevent expression evaluation in the browser, add ngNonBindable to the host element. ngNonBindable deactivates interpolation, directives, and binding in templates.

In the following example, the expression {{ 1 + 1 }} renders just as it does in your code editor, and does not display 2.

src/app/app.component.html

content\_copy<p>Use ngNonBindable to stop evaluation.</p>

<p ngNonBindable>This should not evaluate: {{ 1 + 1 }}</p>

Applying ngNonBindable to an element stops binding for that element's child elements. However, ngNonBindable still allows directives to work on the element where you apply ngNonBindable. In the following example, the appHighlight directive is still active but Angular does not evaluate the expression {{ 1 + 1 }}.

src/app/app.component.html

content\_copy<h3>ngNonBindable with a directive</h3>

<div ngNonBindable [appHighlight]="'yellow'">This should not evaluate: {{ 1 +1 }}, but will highlight yellow.

</div>

If you apply ngNonBindable to a parent element, Angular disables interpolation and binding of any sort, such as property binding or event binding, for the element's children.

# Writing structural directives

This topic demonstrates how to create a structural directive and provides conceptual information on how directives work, how Angular interprets shorthand, and how to add template guard properties to catch template type errors.

For the example application that this page describes, see the [live example](https://angular.io/generated/live-examples/structural-directives/stackblitz.html) / [download example](https://angular.io/generated/zips/structural-directives/structural-directives.zip).

For more information on Angular's built-in structural directives, such as [NgIf](https://angular.io/api/common/NgIf), [NgForOf](https://angular.io/api/common/NgForOf), and [NgSwitch](https://angular.io/api/common/NgSwitch), see [Built-in directives](https://angular.io/guide/built-in-directives).

## **Creating a structural directive**

This section guides you through creating an UnlessDirective and how to set condition values. The UnlessDirective does the opposite of [NgIf](https://angular.io/api/common/NgIf), and condition values can be set to true or false. [NgIf](https://angular.io/api/common/NgIf) displays the template content when the condition is true. UnlessDirective displays the content when the condition is false.

Following is the UnlessDirective selector, appUnless, applied to the paragraph element. When condition is false, the browser displays the sentence.

src/app/app.component.html (appUnless-1)

content\_copy<p \*appUnless="condition">Show this sentence unless the condition is true.</p>

1. Using the Angular CLI, run the following command, where unless is the name of the directive:

content\_copyng generate directive unless

Angular creates the directive class and specifies the CSS selector, appUnless, that identifies the directive in a template.

1. Import [Input](https://angular.io/api/core/Input), [TemplateRef](https://angular.io/api/core/TemplateRef), and [ViewContainerRef](https://angular.io/api/core/ViewContainerRef).

src/app/unless.directive.ts (skeleton)

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

@[Directive](https://angular.io/api/core/Directive)({ selector: '[appUnless]'})

export class UnlessDirective {

}

1. Inject [TemplateRef](https://angular.io/api/core/TemplateRef) and [ViewContainerRef](https://angular.io/api/core/ViewContainerRef) in the directive constructor as private variables.

src/app/unless.directive.ts (ctor)

content\_copyconstructor(

private templateRef: [TemplateRef](https://angular.io/api/core/TemplateRef)<any>,

private viewContainer: [ViewContainerRef](https://angular.io/api/core/ViewContainerRef)) { }

The UnlessDirective creates an [embedded view](https://angular.io/api/core/EmbeddedViewRef) from the Angular-generated [<ng-template>](https://angular.io/api/core/ng-template) and inserts that view in a [view container](https://angular.io/api/core/ViewContainerRef) adjacent to the directive's original <p> host element.

[TemplateRef](https://angular.io/api/core/TemplateRef) helps you get to the [<ng-template>](https://angular.io/api/core/ng-template) contents and [ViewContainerRef](https://angular.io/api/core/ViewContainerRef) accesses the view container.

1. Add an appUnless @[Input](https://angular.io/api/core/Input)() property with a setter.

src/app/unless.directive.ts (set)

content\_copy@[Input](https://angular.io/api/core/Input)() set appUnless(condition: boolean) {

if (!condition && !this.hasView) {

this.viewContainer.createEmbeddedView(this.templateRef);

this.hasView = true;

} else if (condition && this.hasView) {

this.viewContainer.clear();

this.hasView = false;

}

}

Angular sets the appUnless property whenever the value of the condition changes.

* + If the condition is falsy and Angular hasn't created the view previously, the setter causes the view container to create the embedded view from the template.
  + If the condition is truthy and the view is currently displayed, the setter clears the container, which disposes of the view.

The complete directive is as follows:

src/app/unless.directive.ts (excerpt)

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

/\*\*

\* Add the template content to the DOM unless the condition is true.

\*/

@[Directive](https://angular.io/api/core/Directive)({ selector: '[appUnless]'})

export class UnlessDirective {

private hasView = false;

constructor(

private templateRef: [TemplateRef](https://angular.io/api/core/TemplateRef)<any>,

private viewContainer: [ViewContainerRef](https://angular.io/api/core/ViewContainerRef)) { }

@[Input](https://angular.io/api/core/Input)() set appUnless(condition: boolean) {

if (!condition && !this.hasView) {

this.viewContainer.createEmbeddedView(this.templateRef);

this.hasView = true;

} else if (condition && this.hasView) {

this.viewContainer.clear();

this.hasView = false;

}

}

}

### **Testing the directive**

In this section, you'll update your application to test the UnlessDirective.

1. Add a condition set to false in the AppComponent.

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

content\_copycondition = false;

1. Update the template to use the directive. Here, \*appUnless is on two <p> tags with opposite condition values, one true and one false.

src/app/app.component.html (appUnless)

content\_copy<p \*appUnless="condition" class="unless a">

(A) This paragraph is displayed because the condition is false.

</p>

<p \*appUnless="!condition" class="unless b">

(B) Although the condition is true,

this paragraph is displayed because appUnless is set to false.

</p>

The asterisk is shorthand that marks appUnless as a structural directive. When the condition is falsy, the top (A) paragraph appears and the bottom (B) paragraph disappears. When the condition is truthy, the top (A) paragraph disappears and the bottom (B) paragraph appears.

1. To change and display the value of condition in the browser, add markup that displays the status and a button.

src/app/app.component.html

content\_copy<p>

The condition is currently

<span [[ngClass](https://angular.io/api/common/NgClass)]="{ 'a': !condition, 'b': condition, 'unless': true }">{{condition}}</span>.

<button

(click)="condition = !condition"

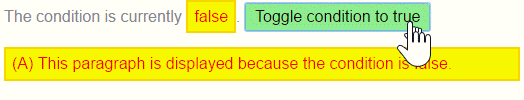
[[ngClass](https://angular.io/api/common/NgClass)] = "{ 'a': condition, 'b': !condition }" >

Toggle condition to {{condition ? 'false' : 'true'}}

</button>

</p>

To verify that the directive works, click the button to change the value of condition.



## **Structural directive shorthand**

The asterisk, \*, syntax on a structural directive, such as \*[ngIf](https://angular.io/api/common/NgIf), is shorthand that Angular interprets into a longer form. Angular transforms the asterisk in front of a structural directive into an [<ng-template>](https://angular.io/api/core/ng-template) that surrounds the host element and its descendants.

The following is an example of \*[ngIf](https://angular.io/api/common/NgIf) that displays the hero's name if hero exists:

src/app/app.component.html (asterisk)

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

The \*[ngIf](https://angular.io/api/common/NgIf) directive moves to the [<ng-template>](https://angular.io/api/core/ng-template) where it becomes a property binding in square brackets, [[ngIf](https://angular.io/api/common/NgIf)]. The rest of the <div>, including its class attribute, moves inside the [<ng-template>](https://angular.io/api/core/ng-template).

src/app/app.component.html (ngif-template)

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

<div class="name">{{hero.name}}</div>

</ng-template>

Angular does not create a real [<ng-template>](https://angular.io/api/core/ng-template) element, instead rendering only the <div> and a comment node placeholder to the DOM.

content\_copy<!--bindings={

"ng-reflect-ng-if": "[object Object]"

}-->

<div \_ngcontent-c0>Mr. Nice</div>

The following example compares the shorthand use of the asterisk in \*[ngFor](https://angular.io/api/common/NgForOf) with the longhand [<ng-template>](https://angular.io/api/core/ng-template) form:

src/app/app.component.html (inside-ngfor)

content\_copy<div \*[ngFor](https://angular.io/api/common/NgForOf)="let hero of heroes; let i=index; let odd=odd; trackBy: trackById" [class.odd]="odd">

({{i}}) {{hero.name}}

</div>

<ng-template [ngFor](https://angular.io/api/common/NgForOf) let-hero [[ngForOf](https://angular.io/api/common/NgForOf)]="heroes" let-i="index" let-odd="odd" [ngForTrackBy]="trackById">

<div [class.odd]="odd">({{i}}) {{hero.name}}</div>

</ng-template>

Here, everything related to the [ngFor](https://angular.io/api/common/NgForOf) structural directive applies to the [<ng-template>](https://angular.io/api/core/ng-template). All other bindings and attributes on the element apply to the <div> element within the [<ng-template>](https://angular.io/api/core/ng-template). Other modifiers on the host element, in addition to the [ngFor](https://angular.io/api/common/NgForOf) string, remain in place as the element moves inside the [<ng-template>](https://angular.io/api/core/ng-template). In this example, the [class.odd]="odd" stays on the <div>.

The let keyword declares a template input variable that you can reference within the template. The input variables in this example are hero, i, and odd. The parser translates let hero, let i, and let odd into variables named let-hero, let-i, and let-odd. The let-i and let-odd variables become let i=index and let odd=odd. Angular sets i and odd to the current value of the context's index and odd properties.

The parser applies PascalCase to all directives and prefixes them with the directive's attribute name, such as ngFor. For example, the [ngFor](https://angular.io/api/common/NgForOf) input properties, of and trackBy, map to [ngForOf](https://angular.io/api/common/NgForOf) and ngForTrackBy. As the NgFor directive loops through the list, it sets and resets properties of its own context object. These properties can include, but aren't limited to, index, odd, and a special property named $implicit.

Angular sets let-hero to the value of the context's $implicit property, which NgFor has initialized with the hero for the current iteration.

For more information, see the [NgFor API](https://angular.io/api/common/NgForOf) and [NgForOf API](https://angular.io/api/common/NgForOf) documentation.

### **Creating template fragments with**[**<ng-template>**](https://angular.io/api/core/ng-template)

Angular's [<ng-template>](https://angular.io/api/core/ng-template) element defines a template that doesn't render anything by default. With [<ng-template>](https://angular.io/api/core/ng-template), you can render the content manually for full control over how the content displays.

If there is no structural directive and you wrap some elements in an [<ng-template>](https://angular.io/api/core/ng-template), those elements disappear. In the following example, Angular does not render the middle "Hip!" in the phrase "Hip! Hip! Hooray!" because of the surrounding [<ng-template>](https://angular.io/api/core/ng-template).

src/app/app.component.html (template-tag)

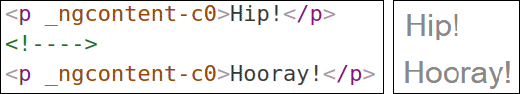
content\_copy<p>Hip!</p>

<ng-template>

<p>Hip!</p>

</ng-template>

<p>Hooray!</p>



## **Structural directive syntax reference**

When you write your own structural directives, use the following syntax:

content\_copy\*:prefix="( :let | :expression ) (';' | ',')? ( :let | :as | :keyExp )\*"

The following tables describe each portion of the structural directive grammar:

|  |  |
| --- | --- |
| prefix | HTML attribute key |
| key | HTML attribute key |
| local | local variable name used in the template |
| export | value exported by the directive under a given name |
| expression | standard Angular expression |

|  |
| --- |
|  |
| keyExp = :key ":"? :expression ("as" :local)? ";"? |
| let = "let" :local "=" :export ";"? |
| as = :export "as" :local ";"? |

### **How Angular translates shorthand**

Angular translates structural directive shorthand into the normal binding syntax as follows:

|  |  |
| --- | --- |
| **Shorthand** | **Translation** |
| prefix and naked expression | [prefix]="expression" |
| keyExp | [prefixKey] "expression" (let-prefixKey="export") Notice that the prefix is added to the key |
| let | let-local="export" |

### **Shorthand examples**

The following table provides shorthand examples:

|  |  |
| --- | --- |
| **Shorthand** | **How Angular interprets the syntax** |
| \*[ngFor](https://angular.io/api/common/NgForOf)="let item of [1,2,3]" | <ng-template [ngFor](https://angular.io/api/common/NgForOf) let-item [[ngForOf](https://angular.io/api/common/NgForOf)]="[1,2,3]"> |
| \*[ngFor](https://angular.io/api/common/NgForOf)="let item of [1,2,3] as items; trackBy: myTrack; index as i" | <ng-template [ngFor](https://angular.io/api/common/NgForOf) let-item [[ngForOf](https://angular.io/api/common/NgForOf)]="[1,2,3]" let-items="[ngForOf](https://angular.io/api/common/NgForOf)" [ngForTrackBy]="myTrack" let-i="index"> |
| \*[ngIf](https://angular.io/api/common/NgIf)="exp" | <ng-template [[ngIf](https://angular.io/api/common/NgIf)]="exp"> |
| \*[ngIf](https://angular.io/api/common/NgIf)="exp as value" | <ng-template [[ngIf](https://angular.io/api/common/NgIf)]="exp" let-value="[ngIf](https://angular.io/api/common/NgIf)"> |

## **Improving template type checking for custom directives**

You can improve template type checking for custom directives by adding template guard properties to your directive definition. These properties help the Angular template type checker find mistakes in the template at compile time, which can avoid runtime errors. These properties are as follows:

* A property ngTemplateGuard\_(someInputProperty) lets you specify a more accurate type for an input expression within the template.
* The ngTemplateContextGuard static property declares the type of the template context.

This section provides examples of both kinds of type-guard property. For more information, see [Template type checking](https://angular.io/guide/template-typecheck).

### **Making in-template type requirements more specific with template guards**

A structural directive in a template controls whether that template is rendered at run time, based on its input expression. To help the compiler catch template type errors, you should specify as closely as possible the required type of a directive's input expression when it occurs inside the template.

A type guard function narrows the expected type of an input expression to a subset of types that might be passed to the directive within the template at run time. You can provide such a function to help the type-checker infer the proper type for the expression at compile time.

For example, the [NgIf](https://angular.io/api/common/NgIf) implementation uses type-narrowing to ensure that the template is only instantiated if the input expression to \*[ngIf](https://angular.io/api/common/NgIf) is truthy. To provide the specific type requirement, the [NgIf](https://angular.io/api/common/NgIf) directive defines a [static property ngTemplateGuard\_ngIf: 'binding'](https://angular.io/api/common/NgIf#static-properties). The binding value is a special case for a common kind of type-narrowing where the input expression is evaluated in order to satisfy the type requirement.

To provide a more specific type for an input expression to a directive within the template, add an ngTemplateGuard\_xx property to the directive, where the suffix to the static property name, xx, is the @[Input](https://angular.io/api/core/Input)() field name. The value of the property can be either a general type-narrowing function based on its return type, or the string "binding", as in the case of [NgIf](https://angular.io/api/common/NgIf).

For example, consider the following structural directive that takes the result of a template expression as an input:

IfLoadedDirective

content\_copyexport type Loaded = { type: 'loaded', data: T };

export type Loading = { type: 'loading' };

export type LoadingState = Loaded | Loading;

export class IfLoadedDirective {

@[Input](https://angular.io/api/core/Input)('ifLoaded') set state(state: LoadingState) {}

[static](https://angular.io/api/upgrade/static) ngTemplateGuard\_state(dir: IfLoadedDirective, expr: LoadingState): expr is Loaded { return true; };

}

export interface Person {

name: string;

}

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

template: `<div \*ifLoaded="state">{{ state.data }}</div>`,

})

export class AppComponent {

state: LoadingState;

}

In this example, the LoadingState<T> type permits either of two states, Loaded<T> or Loading. The expression used as the directive’s state input is of the umbrella type LoadingState, as it’s unknown what the loading state is at that point.

The IfLoadedDirective definition declares the static field ngTemplateGuard\_state, which expresses the narrowing behavior. Within the AppComponent template, the \*ifLoaded structural directive should render this template only when state is actually Loaded<Person>. The type guard allows the type checker to infer that the acceptable type of state within the template is a Loaded<T>, and further infer that T must be an instance of Person.

### **Typing the directive's context**

If your structural directive provides a context to the instantiated template, you can properly type it inside the template by providing a static ngTemplateContextGuard function. The following snippet shows an example of such a function.

myDirective.ts

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

export class ExampleDirective {

// Make sure the template checker knows the type of the context with which the

// template of this directive will be rendered

[static](https://angular.io/api/upgrade/static) ngTemplateContextGuard(dir: ExampleDirective, ctx: unknown): ctx is ExampleContext { return true; };

// …

}

# Dependency injection in Angular

Dependencies are services or objects that a class needs to perform its function. Dependency injection, or DI, is a design pattern in which a class requests dependencies from external sources rather than creating them.

Angular's DI framework provides dependencies to a class upon instantiation. You can use Angular DI to increase flexibility and modularity in your applications.

See the [live example](https://angular.io/generated/live-examples/dependency-injection/stackblitz.html) / [download example](https://angular.io/generated/zips/dependency-injection/dependency-injection.zip) for a working example containing the code snippets in this guide.

## **Creating an injectable service**

To generate a new HeroService class in the src/app/heroes folder use the following [Angular CLI](https://angular.io/cli) command.

content\_copyng generate service heroes/hero

This command creates the following default HeroService.

src/app/heroes/hero.service.ts (CLI-generated)

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

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

providedIn: 'root',

})

export class HeroService {

constructor() { }

}

The @[Injectable](https://angular.io/api/core/Injectable)() decorator specifies that Angular can use this class in the DI system. The metadata, providedIn: 'root', means that the HeroService is visible throughout the application.

Next, to get the hero mock data, add a getHeroes() method that returns the heroes from mock.heroes.ts.

src/app/heroes/hero.service.ts

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

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

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

// declares that this service should be created

// by the root application injector.

providedIn: 'root',

})

export class HeroService {

getHeroes() { return HEROES; }

}

For clarity and maintainability, it is recommended that you define components and services in separate files.

If you do combine a component and service in the same file, it is important to define the service first, and then the component. If you define the component before the service, Angular returns a run-time null reference error.

## **Injecting services**

Injecting services results in making them visible to a component.

To inject a dependency in a component's constructor(), supply a constructor argument with the dependency type. The following example specifies the HeroService in the HeroListComponent constructor. The type of heroService is HeroService.

src/app/heroes/hero-list.component (constructor signature)

content\_copyconstructor(heroService: HeroService)

For more information, see [Providing dependencies in modules](https://angular.io/guide/providers) and [Hierarchical injectors](https://angular.io/guide/hierarchical-dependency-injection).

## **Using services in other services**

When a service depends on another service, follow the same pattern as injecting into a component. In the following example HeroService depends on a Logger service to report its activities.

First, import the Logger service. Next, inject the Logger service in the HeroService constructor() by specifying private logger: Logger within the parentheses.

When you create a class whose constructor() has parameters, specify the type and metadata about those parameters so that Angular can inject the correct service.

Here, the constructor() specifies a type of Logger and stores the instance of Logger in a private field called logger.

The following code tabs feature the Logger service and two versions of HeroService. The first version of HeroService does not depend on the Logger service. The revised second version does depend on Logger service.

src/app/heroes/hero.service (v2)

src/app/heroes/hero.service (v1)

src/app/logger.service

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

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

import { Logger } from '../logger.service';

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

providedIn: 'root',

})

export class HeroService {

constructor(private logger: Logger) { }

getHeroes() {

this.logger.log('Getting heroes ...');

return HEROES;

}

}

In this example, the getHeroes() method uses the Logger service by logging a message when fetching heroes.

## **What's next**

* [Dependency providers](https://angular.io/guide/dependency-injection-providers)
* [DI tokens and providers](https://angular.io/guide/dependency-injection-providers)
* [Dependency Injection in Action](https://angular.io/guide/dependency-injection-in-action)

# Dependency providers[link](https://angular.io/guide/dependency-injection-providers#dependency-providers)

By configuring providers, you can make services available to the parts of your application that need them.

A dependency [provider](https://angular.io/guide/glossary#provider) configures an injector with a [DI token](https://angular.io/guide/glossary#di-token), which that injector uses to provide the runtime version of a dependency value.

## **Specifying a provider token**

If you specify the service class as the provider token, the default behavior is for the injector to instantiate that class with new.

In the following example, the Logger class provides a Logger instance.

content\_copyproviders: [Logger]

You can, however, configure an injector with an alternative provider in order to deliver some other object that provides the needed logging functionality.

You can configure an injector with a service class, you can provide a substitute class, an object, or a factory function.

## **Dependency injection tokens**

When you configure an [injector](https://angular.io/guide/glossary#injector) with a [provider](https://angular.io/guide/glossary#provider), you are associating that provider with a [dependency injection token](https://angular.io/guide/glossary#di-token), or DI token. The injector allows Angular to create a map of any internal dependencies. The DI token acts as a key to that map.

The dependency value is an instance, and the class type serves as a lookup key. Here, the injector uses the HeroService type as the token for looking up heroService.

src/app/injector.component.ts

content\_copyheroService: HeroService;

When you define a constructor parameter with the HeroService class type, Angular knows to inject the service associated with that HeroService class token:

src/app/heroes/hero-list.component.ts

content\_copyconstructor(heroService: HeroService)

Though classes provide many dependency values, the expanded provide object lets you associate different kinds of providers with a DI token.

## **Defining providers**

The class provider syntax is a shorthand expression that expands into a provider configuration, defined by the [Provider interface](https://angular.io/api/core/Provider). The following example is the class provider syntax for providing a Logger class in the providers array.

content\_copyproviders: [Logger]

Angular expands the providers value into a full provider object as follows.

content\_copy[{ provide: Logger, useClass: Logger }]

The expanded provider configuration is an object literal with two properties:

* The provide property holds the [token](https://angular.io/guide/dependency-injection-providers#token) that serves as the key for both locating a dependency value and configuring the injector.
* The second property is a provider definition object, which tells the injector how to create the dependency value. The provider-definition key can be useClass, as in the example. It can also be useExisting, useValue, or useFactory. Each of these keys provides a different type of dependency, as discussed below.

## **Specifying an alternative class provider**

Different classes can provide the same service. For example, the following code tells the injector to return a BetterLogger instance when the component asks for a logger using the Logger token.

content\_copy[{ provide: Logger, useClass: BetterLogger }]

### **Configuring class providers with dependencies**

If the alternative class providers have their own dependencies, specify both providers in the providers metadata property of the parent module or component.

content\_copy[ UserService,

{ provide: Logger, useClass: EvenBetterLogger }]

In this example, EvenBetterLogger displays the user name in the log message. This logger gets the user from an injected UserService instance.

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

export class EvenBetterLogger extends Logger {

constructor(private userService: UserService) { super(); }

log(message: string) {

const name = this.userService.user.name;

super.log(`Message to ${name}: ${message}`);

}

}

The injector needs providers for both this new logging service and its dependent UserService.

### **Aliasing class providers**

To alias a class provider, specify the alias and the class provider in the providers array with the useExisting property.

In the following example, the injector injects the singleton instance of NewLogger when the component asks for either the new or the old logger. In this way, OldLogger is an alias for NewLogger.

content\_copy[ NewLogger,

// Alias OldLogger w/ reference to NewLogger

{ provide: OldLogger, useExisting: NewLogger}]

Be sure you don't alias OldLogger to NewLogger with useClass, as this creates two different NewLogger instances.

## **Aliasing a class interface**

Generally, writing variations of the same parent alias provider uses [forwardRef](https://angular.io/guide/dependency-injection-in-action#forwardref) as follows.

dependency-injection-in-action/src/app/parent-finder.component.ts

content\_copyproviders: [{ provide: Parent, useExisting: [forwardRef](https://angular.io/api/core/forwardRef)(() => AlexComponent) }],

To streamline your code, you can extract that logic into a helper function using the provideParent() helper function.

dependency-injection-in-action/src/app/parent-finder.component.ts

content\_copy// Helper method to provide the current component instance in the name of a `parentType`.

export function provideParent

(component: any) {

return { provide: Parent, useExisting: [forwardRef](https://angular.io/api/core/forwardRef)(() => component) };

}

Now you can add a parent provider to your components that's easier to read and understand.

dependency-injection-in-action/src/app/parent-finder.component.ts

content\_copyproviders: [ provideParent(AliceComponent) ]

### **Aliasing multiple class interfaces**

To alias multiple parent types, each with its own class interface token, configure provideParent() to accept more arguments.

Here's a revised version that defaults to parent but also accepts an optional second parameter for a different parent class interface.

dependency-injection-in-action/src/app/parent-finder.component.ts

content\_copy// Helper method to provide the current component instance in the name of a `parentType`.

// The `parentType` defaults to `Parent` when omitting the second parameter.

export function provideParent

(component: any, parentType?: any) {

return { provide: parentType || Parent, useExisting: [forwardRef](https://angular.io/api/core/forwardRef)(() => component) };

}

Next, to use provideParent() with a different parent type, provide a second argument, here DifferentParent.

dependency-injection-in-action/src/app/parent-finder.component.ts

content\_copyproviders: [ provideParent(BethComponent, DifferentParent) ]

## **Injecting an object**

To inject an object, configure the injector with the useValue option. The following provider object uses the useValue key to associate the variable with the Logger token.

content\_copy[{ provide: Logger, useValue: SilentLogger }]

In this example, SilentLogger is an object that fulfills the logger role.

content\_copy// An object in the shape of the logger service

function silentLoggerFn() {}

export const SilentLogger = {

logs: ['Silent logger says "Shhhhh!". Provided via "useValue"'],

log: silentLoggerFn

};

### **Injecting a configuration object**

A common use case for object literals is a configuration object. The following configuration object includes the title of the application and the address of a web API endpoint.

src/app/app.config.ts (excerpt)

content\_copyexport const HERO\_DI\_CONFIG: AppConfig = {

apiEndpoint: 'api.heroes.com',

title: 'Dependency Injection'

};

To provide and inject the configuration object, specify the object in the @[NgModule](https://angular.io/api/core/NgModule)() providers array.

src/app/app.module.ts (providers)

content\_copyproviders: [

UserService,

{ provide: APP\_CONFIG, useValue: HERO\_DI\_CONFIG }

],

### **Using an**[**InjectionToken**](https://angular.io/api/core/InjectionToken)**object**

You can define and use an [InjectionToken](https://angular.io/api/core/InjectionToken) object for choosing a provider token for non-class dependencies. The following example defines a token, APP\_CONFIG of the type [InjectionToken](https://angular.io/api/core/InjectionToken).

src/app/app.config.ts

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

export const APP\_CONFIG = new [InjectionToken](https://angular.io/api/core/InjectionToken)<AppConfig>('app.config');

The optional type parameter, <AppConfig>, and the token description, app.config, specify the token's purpose.

Next, register the dependency provider in the component using the [InjectionToken](https://angular.io/api/core/InjectionToken) object of APP\_CONFIG.

src/app/providers.component.ts

content\_copyproviders: [{ provide: APP\_CONFIG, useValue: HERO\_DI\_CONFIG }]

Now you can inject the configuration object into the constructor with @[Inject](https://angular.io/api/core/Inject)() parameter decorator.

src/app/app.component.ts

content\_copyconstructor(@[Inject](https://angular.io/api/core/Inject)(APP\_CONFIG) config: AppConfig) {

this.title = config.title;

}

#### Interfaces and dependency injection

Though the TypeScript AppConfig interface supports typing within the class, the AppConfig interface plays no role in dependency injection. In TypeScript, an interface is a design-time artifact, and doesn't have a runtime representation, or token, that the DI framework can use.

When the transpiler changes TypeScript to JavaScript, the interface disappears because JavaScript doesn't have interfaces.

Since there is no interface for Angular to find at runtime, the interface cannot be a token, nor can you inject it.

content\_copy// Can't use interface as provider token

[{ provide: AppConfig, useValue: HERO\_DI\_CONFIG })]

content\_copy// Can't inject using the interface as the parameter type

constructor(private config: AppConfig){ }

## **Using factory providers**

To create a changeable, dependent value based on information unavailable before run time, you can use a factory provider.

In the following example, only authorized users should see secret heroes in the HeroService. Authorization can change during the course of a single application session, as when a different user logs in .

To keep security-sensitive information in UserService and out of HeroService, give the HeroService constructor a boolean flag to control display of secret heroes.

src/app/heroes/hero.service.ts (excerpt)

content\_copyconstructor(

private logger: Logger,

private isAuthorized: boolean) { }

getHeroes() {

const auth = this.isAuthorized ? 'authorized ' : 'unauthorized';

this.logger.log(`Getting heroes for ${auth} user.`);

return HEROES.filter(hero => this.isAuthorized || !hero.isSecret);

}

To implement the isAuthorized flag, use a factory provider to create a new logger instance for HeroService.

src/app/heroes/hero.service.provider.ts (excerpt)

content\_copyconst heroServiceFactory = (logger: Logger, userService: UserService) => {

return new HeroService(logger, userService.user.isAuthorized);

};

The factory function has access to UserService. You inject both Logger and UserService into the factory provider so the injector can pass them along to the factory function.

src/app/heroes/hero.service.provider.ts (excerpt)

content\_copyexport let heroServiceProvider =

{ provide: HeroService,

useFactory: heroServiceFactory,

deps: [Logger, UserService]

};

* The useFactory field specifies that the provider is a factory function whose implementation is heroServiceFactory.
* The deps property is an array of [provider tokens](https://angular.io/guide/dependency-injection-providers#token). The Logger and UserService classes serve as tokens for their own class providers. The injector resolves these tokens and injects the corresponding services into the matching heroServiceFactory factory function parameters.

Capturing the factory provider in the exported variable, heroServiceProvider, makes the factory provider reusable.

The following side-by-side example shows how heroServiceProvider replaces HeroService in the providers array.

src/app/heroes/heroes.component (v3)

src/app/heroes/heroes.component (v2)

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

import { heroServiceProvider } from './hero.service.provider';

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

selector: 'app-heroes',

providers: [ heroServiceProvider ],

template: `

<h2>Heroes</h2>

<app-hero-list></app-hero-list>

`

})

export class HeroesComponent { }

# Common Routing Tasks

This topic describes how to implement many of the common tasks associated with adding the Angular router to your application.

## **Generate an application with routing enabled**

The following command uses the Angular CLI to generate a basic Angular application with an application routing module, called AppRoutingModule, which is an NgModule where you can configure your routes. The application name in the following example is routing-app.

content\_copyng new routing-app --routing

When generating a new application, the CLI prompts you to select CSS or a CSS preprocessor. For this example, accept the default of CSS.

### **Adding components for routing**

To use the Angular router, an application needs to have at least two components so that it can navigate from one to the other. To create a component using the CLI, enter the following at the command line where first is the name of your component:

content\_copyng generate component first

Repeat this step for a second component but give it a different name. Here, the new name is second.

content\_copyng generate component second

The CLI automatically appends [Component](https://angular.io/api/core/Component), so if you were to write first-component, your component would be FirstComponentComponent.

#### <base href>

This guide works with a CLI-generated Angular application. If you are working manually, make sure that you have <base href="/"> in the <head> of your index.html file. This assumes that the app folder is the application root, and uses "/".

### **Importing your new components**

To use your new components, import them into AppRoutingModule at the top of the file, as follows:

AppRoutingModule (excerpt)

content\_copyimport { FirstComponent } from './first/first.component';

import { SecondComponent } from './second/second.component';

## **Defining a basic route**

There are three fundamental building blocks to creating a route.

Import the AppRoutingModule into AppModule and add it to the imports array.

The Angular CLI performs this step for you. However, if you are creating an application manually or working with an existing, non-CLI application, verify that the imports and configuration are correct. The following is the default AppModule using the CLI with the --routing flag.

Default CLI AppModule with routing

content\_copyimport { [BrowserModule](https://angular.io/api/platform-browser/BrowserModule) } from '@angular/platform-browser';

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

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

import { AppComponent } from './app.component';

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

declarations: [

AppComponent

],

imports: [

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

AppRoutingModule // CLI adds AppRoutingModule to the AppModule's imports array

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

1. Import [RouterModule](https://angular.io/api/router/RouterModule) and [Routes](https://angular.io/api/router/Routes) into your routing module.

The Angular CLI performs this step automatically. The CLI also sets up a [Routes](https://angular.io/api/router/Routes) array for your routes and configures the imports and exports arrays for @[NgModule](https://angular.io/api/core/NgModule)().

CLI application routing module

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

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

const routes: [Routes](https://angular.io/api/router/Routes) = []; // sets up routes constant where you define your routes

// configures [NgModule](https://angular.io/api/core/NgModule) imports and exports

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

imports: [RouterModule.forRoot(routes)],

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

})

export class AppRoutingModule { }

1. Define your routes in your [Routes](https://angular.io/api/router/Routes) array.

Each route in this array is a JavaScript object that contains two properties. The first property, path, defines the URL path for the route. The second property, component, defines the component Angular should use for the corresponding path.

AppRoutingModule (excerpt)

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

{ path: 'first-component', component: FirstComponent },

{ path: 'second-component', component: SecondComponent },

];

1. Add your routes to your application.

Now that you have defined your routes, you can add them to your application. First, add links to the two components. Assign the anchor tag that you want to add the route to the [routerLink](https://angular.io/api/router/RouterLink) attribute. Set the value of the attribute to the component to show when a user clicks on each link. Next, update your component template to include <[router-outlet](https://angular.io/api/router/RouterOutlet)>. This element informs Angular to update the application view with the component for the selected route.

Template with routerLink and router-outlet

content\_copy<h1>Angular [Router](https://angular.io/api/router/Router) App</h1>

<!-- This nav gives you links to click, which tells the router which route to use (defined in the routes constant in AppRoutingModule) -->

<nav>

<ul>

<li><a [routerLink](https://angular.io/api/router/RouterLink)="/first-component" [routerLinkActive](https://angular.io/api/router/RouterLinkActive)="active">First [Component](https://angular.io/api/core/Component)</a></li>

<li><a [routerLink](https://angular.io/api/router/RouterLink)="/second-component" [routerLinkActive](https://angular.io/api/router/RouterLinkActive)="active">Second [Component](https://angular.io/api/core/Component)</a></li>

</ul>

</nav>

<!-- The routed views render in the <[router-outlet](https://angular.io/api/router/RouterOutlet)>-->

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

### **Route order**

The order of routes is important because the [Router](https://angular.io/api/router/Router) uses a first-match wins strategy when matching routes, so more specific routes should be placed above less specific routes. List routes with a static path first, followed by an empty path route, which matches the default route. The [wildcard route](https://angular.io/guide/router#setting-up-wildcard-routes) comes last because it matches every URL and the [Router](https://angular.io/api/router/Router) selects it only if no other routes match first.

## **Getting route information**

Often, as a user navigates your application, you want to pass information from one component to another. For example, consider an application that displays a shopping list of grocery items. Each item in the list has a unique id. To edit an item, users click an Edit button, which opens an EditGroceryItem component. You want that component to retrieve the id for the grocery item so it can display the right information to the user.

You can use a route to pass this type of information to your application components. To do so, you use the [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) interface.

To get information from a route:

1. Import [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) and [ParamMap](https://angular.io/api/router/ParamMap) to your component.

In the component class (excerpt)

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

These import statements add several important elements that your component needs. To learn more about each, see the following API pages:

* + [Router](https://angular.io/api/router)
  + [ActivatedRoute](https://angular.io/api/router/ActivatedRoute)
  + [ParamMap](https://angular.io/api/router/ParamMap)

1. Inject an instance of [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) by adding it to your application's constructor:

In the component class (excerpt)

content\_copyconstructor(

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

) {}

1. Update the ngOnInit() method to access the [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) and track the id parameter:

In the component (excerpt)

content\_copyngOnInit() {

this.route.queryParams.subscribe(params => {

this.name = params['name'];

});

}

Note: The preceding example uses a variable, name, and assigns it the value based on the name parameter.

## **Setting up wildcard routes**

A well-functioning application should gracefully handle when users attempt to navigate to a part of your application that does not exist. To add this functionality to your application, you set up a wildcard route. The Angular router selects this route any time the requested URL doesn't match any router paths.

To set up a wildcard route, add the following code to your routes definition.

AppRoutingModule (excerpt)

content\_copy{ path: '\*\*', component: }

The two asterisks, \*\*, indicate to Angular that this routes definition is a wildcard route. For the component property, you can define any component in your application. Common choices include an application-specific PageNotFoundComponent, which you can define to [display a 404 page](https://angular.io/guide/router#404-page-how-to) to your users; or a redirect to your application's main component. A wildcard route is the last route because it matches any URL. For more detail on why order matters for routes, see [Route order](https://angular.io/guide/router#route-order).

## **Displaying a 404 page**

To display a 404 page, set up a [wildcard route](https://angular.io/guide/router#wildcard-route-how-to) with the component property set to the component you'd like to use for your 404 page as follows:

AppRoutingModule (excerpt)

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

{ path: 'first-component', component: FirstComponent },

{ path: 'second-component', component: SecondComponent },

{ path: '\*\*', component: PageNotFoundComponent }, // Wildcard route for a 404 page

];

The last route with the path of \*\* is a wildcard route. The router selects this route if the requested URL doesn't match any of the paths earlier in the list and sends the user to the PageNotFoundComponent.

## **Setting up redirects**

To set up a redirect, configure a route with the path you want to redirect from, the component you want to redirect to, and a pathMatch value that tells the router how to match the URL.

AppRoutingModule (excerpt)

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

{ path: 'first-component', component: FirstComponent },

{ path: 'second-component', component: SecondComponent },

{ path: '', redirectTo: '/first-component', pathMatch: 'full' }, // redirect to `first-component`

{ path: '\*\*', component: PageNotFoundComponent }, // Wildcard route for a 404 page

];

In this example, the third route is a redirect so that the router defaults to the first-component route. Notice that this redirect precedes the wildcard route. Here, path: '' means to use the initial relative URL ('').

For more details on pathMatch see [Spotlight on pathMatch](https://angular.io/guide/router-tutorial-toh#pathmatch).

## **Nesting routes**

As your application grows more complex, you may want to create routes that are relative to a component other than your root component. These types of nested routes are called child routes. This means you're adding a second <[router-outlet](https://angular.io/api/router/RouterOutlet)> to your app, because it is in addition to the <[router-outlet](https://angular.io/api/router/RouterOutlet)> in AppComponent.

In this example, there are two additional child components, child-a, and child-b. Here, FirstComponent has its own <nav> and a second <[router-outlet](https://angular.io/api/router/RouterOutlet)> in addition to the one in AppComponent.

In the template

content\_copy<h2>First [Component](https://angular.io/api/core/Component)</h2>

<nav>

<ul>

<li><a [routerLink](https://angular.io/api/router/RouterLink)="child-a">Child A</a></li>

<li><a [routerLink](https://angular.io/api/router/RouterLink)="child-b">Child B</a></li>

</ul>

</nav>

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

A child route is like any other route, in that it needs both a path and a component. The one difference is that you place child routes in a children array within the parent route.

AppRoutingModule (excerpt)

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

{

path: 'first-component',

component: FirstComponent, // this is the component with the <[router-outlet](https://angular.io/api/router/RouterOutlet)> in the template

children: [

{

path: 'child-a', // child route path

component: ChildAComponent, // child route component that the router renders

},

{

path: 'child-b',

component: ChildBComponent, // another child route component that the router renders

},

],

},

];

## **Using relative paths**

Relative paths allow you to define paths that are relative to the current URL segment. The following example shows a relative route to another component, second-component. FirstComponent and SecondComponent are at the same level in the tree, however, the link to SecondComponent is situated within the FirstComponent, meaning that the router has to go up a level and then into the second directory to find the SecondComponent. Rather than writing out the whole path to get to SecondComponent, you can use the ../ notation to go up a level.

In the template

content\_copy<h2>First [Component](https://angular.io/api/core/Component)</h2>

<nav>

<ul>

<li><a [routerLink](https://angular.io/api/router/RouterLink)="../second-component">Relative [Route](https://angular.io/api/router/Route) to second component</a></li>

</ul>

</nav>

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

In addition to ../, you can use ./ or no leading slash to specify the current level.

### **Specifying a relative route**

To specify a relative route, use the [NavigationExtras](https://angular.io/api/router/NavigationExtras) relativeTo property. In the component class, import [NavigationExtras](https://angular.io/api/router/NavigationExtras) from the @angular/router.

Then use relativeTo in your navigation method. After the link parameters array, which here contains items, add an object with the relativeTo property set to the [ActivatedRoute](https://angular.io/api/router/ActivatedRoute), which is this.route.

RelativeTo

content\_copygoToItems() {

this.router.navigate(['items'], { relativeTo: this.route });

}

The goToItems() method interprets the destination URI as relative to the activated route and navigates to the items route.

## **Accessing query parameters and fragments**

Sometimes, a feature of your application requires accessing a part of a route, such as a query parameter or a fragment. The Tour of Heroes application at this stage in the tutorial uses a list view in which you can click on a hero to see details. The router uses an id to show the correct hero's details.

First, import the following members in the component you want to navigate from.

Component import statements (excerpt)

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

import { Observable } from 'rxjs';

import { switchMap } from 'rxjs/operators';

Next inject the activated route service:

Component (excerpt)

content\_copyconstructor(private route: [ActivatedRoute](https://angular.io/api/router/ActivatedRoute)) {}

Configure the class so that you have an observable, heroes$, a selectedId to hold the id number of the hero, and the heroes in the ngOnInit(), add the following code to get the id of the selected hero. This code snippet assumes that you have a heroes list, a hero service, a function to get your heroes, and the HTML to render your list and details, just as in the Tour of Heroes example.

Component 1 (excerpt)

content\_copyheroes$: Observable;

selectedId: number;

heroes = HEROES;

ngOnInit() {

this.heroes$ = this.route.paramMap.pipe(

switchMap(params => {

this.selectedId = Number(params.get('id'));

return this.service.getHeroes();

})

);

}

Next, in the component that you want to navigate to, import the following members.

Component 2 (excerpt)

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

import { Observable } from 'rxjs';

Inject [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) and [Router](https://angular.io/api/router/Router) in the constructor of the component class so they are available to this component:

Component 2 (excerpt)

content\_copyhero$: Observable;

constructor(

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

private router: [Router](https://angular.io/api/router/Router) ) {}

ngOnInit() {

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

this.hero$ = this.service.getHero(heroId);

}

gotoItems(hero: Hero) {

const heroId = hero ? hero.id : null;

// Pass along the hero id if available

// so that the HeroList component can select that item.

this.router.navigate(['/heroes', { id: heroId }]);

}

## **Lazy loading**

You can configure your routes to lazy load modules, which means that Angular only loads modules as needed, rather than loading all modules when the application launches. Additionally, you can preload parts of your application in the background to improve the user experience.

For more information on lazy loading and preloading see the dedicated guide [Lazy loading NgModules](https://angular.io/guide/lazy-loading-ngmodules).

## **Preventing unauthorized access**

Use route guards to prevent users from navigating to parts of an application without authorization. The following route guards are available in Angular:

* [CanActivate](https://angular.io/api/router/CanActivate)
* [CanActivateChild](https://angular.io/api/router/CanActivateChild)
* [CanDeactivate](https://angular.io/api/router/CanDeactivate)
* [Resolve](https://angular.io/api/router/Resolve)
* [CanLoad](https://angular.io/api/router/CanLoad)

To use route guards, consider using [component-less routes](https://angular.io/api/router/Route#componentless-routes) as this facilitates guarding child routes.

Create a service for your guard:

content\_copyng generate guard your-guard

In your guard class, implement the guard you want to use. The following example uses [CanActivate](https://angular.io/api/router/CanActivate) to guard the route.

Component (excerpt)

content\_copyexport class YourGuard implements [CanActivate](https://angular.io/api/router/CanActivate) {

canActivate(

next: [ActivatedRouteSnapshot](https://angular.io/api/router/ActivatedRouteSnapshot),

state: [RouterStateSnapshot](https://angular.io/api/router/RouterStateSnapshot)): boolean {

// your logic goes here

}

}

In your routing module, use the appropriate property in your routes configuration. Here, canActivate tells the router to mediate navigation to this particular route.

Routing module (excerpt)

content\_copy{

path: '/your-path',

component: YourComponent,

canActivate: [YourGuard],

}

For more information with a working example, see the [routing tutorial section on route guards](https://angular.io/guide/router-tutorial-toh#milestone-5-route-guards).

## **Link parameters array**

A link parameters array holds the following ingredients for router navigation:

* The path of the route to the destination component.
* Required and optional route parameters that go into the route URL.

You can bind the [RouterLink](https://angular.io/api/router/RouterLink) directive to such an array like this:

src/app/app.component.ts (h-anchor)

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

The following is a two-element array when specifying a route parameter:

src/app/heroes/hero-list/hero-list.component.html (nav-to-detail)

content\_copy<a [[routerLink](https://angular.io/api/router/RouterLink)]="['/hero', hero.id]">

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

</a>

You can provide optional route parameters in an object, as in { foo: 'foo' }:

src/app/app.component.ts (cc-query-params)

content\_copy<a [[routerLink](https://angular.io/api/router/RouterLink)]="['/crisis-center', { foo: 'foo' }]">Crisis Center</a>

These three examples cover the needs of an application with one level of routing. However, with a child router, such as in the crisis center, you create new link array possibilities.

The following minimal [RouterLink](https://angular.io/api/router/RouterLink) example builds upon a specified [default child route](https://angular.io/guide/router-tutorial-toh#a-crisis-center-with-child-routes) for the crisis center.

src/app/app.component.ts (cc-anchor-w-default)

content\_copy<a [[routerLink](https://angular.io/api/router/RouterLink)]="['/crisis-center']">Crisis Center</a>

Note the following:

* The first item in the array identifies the parent route (/crisis-center).
* There are no parameters for this parent route.
* There is no default for the child route so you need to pick one.
* You're navigating to the CrisisListComponent, whose route path is /, but you don't need to explicitly add the slash.

Consider the following router link that navigates from the root of the application down to the Dragon Crisis:

src/app/app.component.ts (Dragon-anchor)

content\_copy<a [[routerLink](https://angular.io/api/router/RouterLink)]="['/crisis-center', 1]">Dragon Crisis</a>

* The first item in the array identifies the parent route (/crisis-center).
* There are no parameters for this parent route.
* The second item identifies the child route details about a particular crisis (/:id).
* The details child route requires an id route parameter.
* You added the id of the Dragon Crisis as the second item in the array (1).
* The resulting path is /crisis-center/1.

You could also redefine the AppComponent template with Crisis Center routes exclusively:

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

content\_copytemplate: `

<h1 class="title">Angular [Router](https://angular.io/api/router/Router)</h1>

<nav>

<a [[routerLink](https://angular.io/api/router/RouterLink)]="['/crisis-center']">Crisis Center</a>

<a [[routerLink](https://angular.io/api/router/RouterLink)]="['/crisis-center/1', { foo: 'foo' }]">Dragon Crisis</a>

<a [[routerLink](https://angular.io/api/router/RouterLink)]="['/crisis-center/2']">Shark Crisis</a>

</nav>

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

`

In summary, you can write applications with one, two or more levels of routing. The link parameters array affords the flexibility to represent any routing depth and any legal sequence of route paths, (required) router parameters, and (optional) route parameter objects.

## [**LocationStrategy**](https://angular.io/api/common/LocationStrategy)**and browser URL styles**

When the router navigates to a new component view, it updates the browser's location and history with a URL for that view.

Modern HTML5 browsers support [history.pushState](https://developer.mozilla.org/en-US/docs/Web/API/History_API#Adding_and_modifying_history_entries), a technique that changes a browser's location and history without triggering a server page request. The router can compose a "natural" URL that is indistinguishable from one that would otherwise require a page load.

Here's the Crisis Center URL in this "HTML5 pushState" style:

content\_copylocalhost:3002/crisis-center/

Older browsers send page requests to the server when the location URL changes unless the change occurs after a "#" (called the "hash"). Routers can take advantage of this exception by composing in-application route URLs with hashes. Here's a "hash URL" that routes to the Crisis Center.

content\_copylocalhost:3002/src/#/crisis-center/

The router supports both styles with two [LocationStrategy](https://angular.io/api/common/LocationStrategy) providers:

1. [PathLocationStrategy](https://angular.io/api/common/PathLocationStrategy)—the default "HTML5 pushState" style.
2. [HashLocationStrategy](https://angular.io/api/common/HashLocationStrategy)—the "hash URL" style.

The [RouterModule.forRoot()](https://angular.io/api/router/RouterModule#forRoot) function sets the [LocationStrategy](https://angular.io/api/common/LocationStrategy) to the [PathLocationStrategy](https://angular.io/api/common/PathLocationStrategy), which makes it the default strategy. You also have the option of switching to the [HashLocationStrategy](https://angular.io/api/common/HashLocationStrategy) with an override during the bootstrapping process.

For more information on providers and the bootstrap process, see [Dependency Injection](https://angular.io/guide/dependency-injection#bootstrap).

## **Choosing a routing strategy**

You must choose a routing strategy early in the development of your project because once the application is in production, visitors to your site use and depend on application URL references.

Almost all Angular projects should use the default HTML5 style. It produces URLs that are easier for users to understand and it preserves the option to do server-side rendering.

Rendering critical pages on the server is a technique that can greatly improve perceived responsiveness when the application first loads. An application that would otherwise take ten or more seconds to start could be rendered on the server and delivered to the user's device in less than a second.

This option is only available if application URLs look like normal web URLs without hashes (#) in the middle.

## **<base href>**

The router uses the browser's [history.pushState](https://developer.mozilla.org/en-US/docs/Web/API/History_API#Adding_and_modifying_history_entries) for navigation. pushState allows you to customize in-application URL paths; for example, localhost:4200/crisis-center. The in-application URLs can be indistinguishable from server URLs.

Modern HTML5 browsers were the first to support pushState which is why many people refer to these URLs as "HTML5 style" URLs.

HTML5 style navigation is the router default. In the [LocationStrategy and browser URL styles](https://angular.io/guide/router#browser-url-styles) section, learn why HTML5 style is preferable, how to adjust its behavior, and how to switch to the older hash (#) style, if necessary.

You must add a [<base href> element](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/base) to the application's index.html for pushState routing to work. The browser uses the <base href> value to prefix relative URLs when referencing CSS files, scripts, and images.

Add the <base> element just after the <head> tag. If the app folder is the application root, as it is for this application, set the href value in index.html as shown here.

src/index.html (base-href)

content\_copy<base href="/">

### **HTML5 URLs and the <base href>**

The guidelines that follow will refer to different parts of a URL. This diagram outlines what those parts refer to:

content\_copyfoo://example.com:8042/over/there?name=ferret#nose

\\_/ \\_\_\_\_\_\_\_\_\_\_\_\_\_\_/\\_\_\_\_\_\_\_\_\_/ \\_\_\_\_\_\_\_\_\_/ \\_\_/

| | | | |

scheme authority path [query](https://angular.io/api/animations/query) fragment

While the router uses the [HTML5 pushState](https://developer.mozilla.org/en-US/docs/Web/API/History_API#Adding_and_modifying_history_entries) style by default, you must configure that strategy with a <base href>.

The preferred way to configure the strategy is to add a [<base href> element](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/base) tag in the <head> of the index.html.

src/index.html (base-href)

content\_copy<base href="/">

Without that tag, the browser may not be able to load resources (images, CSS, scripts) when "deep linking" into the application.

Some developers may not be able to add the <base> element, perhaps because they don't have access to <head> or the index.html.

Those developers may still use HTML5 URLs by taking the following two steps:

1. Provide the router with an appropriate [APP\_BASE\_HREF](https://angular.io/api/common/APP_BASE_HREF) value.
2. Use root URLs (URLs with an authority) for all web resources: CSS, images, scripts, and template HTML files.

* The <base href> path should end with a "/", as browsers ignore characters in the path that follow the right-most "/".
* If the <base href> includes a [query](https://angular.io/api/animations/query) part, the [query](https://angular.io/api/animations/query) is only used if the path of a link in the page is empty and has no [query](https://angular.io/api/animations/query). This means that a [query](https://angular.io/api/animations/query) in the <base href> is only included when using [HashLocationStrategy](https://angular.io/api/common/HashLocationStrategy).
* If a link in the page is a root URL (has an authority), the <base href> is not used. In this way, an [APP\_BASE\_HREF](https://angular.io/api/common/APP_BASE_HREF) with an authority will cause all links created by Angular to ignore the <base href> value.
* A fragment in the <base href> is never persisted.

For more complete information on how <base href> is used to construct target URIs, see the [RFC](https://tools.ietf.org/html/rfc3986#section-5.2.2) section on transforming references.

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

You can use [HashLocationStrategy](https://angular.io/api/common/HashLocationStrategy) by providing the useHash: true in an object as the second argument of the [RouterModule.forRoot()](https://angular.io/api/router/RouterModule#forRoot) in the AppModule.

src/app/app.module.ts (hash URL strategy)

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

import { [BrowserModule](https://angular.io/api/platform-browser/BrowserModule) } from '@angular/platform-browser';

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

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

import { AppComponent } from './app.component';

import { PageNotFoundComponent } from './page-not-found/page-not-found.component';

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

];

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

imports: [

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

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

RouterModule.forRoot(routes, { useHash: true }) // .../#/crisis-center/

],

declarations: [

AppComponent,

PageNotFoundComponent

],

providers: [

],

bootstrap: [ AppComponent ]

})

export class AppModule { }

* ck whether users can access a route.
* Control whether the application can discard unsaved changes.
* Improve performance by pre-fetching route data and lazy loading feature modules.
* Require specific criteria to load components.

## **Prerequisites**

To complete this tutorial, you should have a basic understanding of the following concepts:

* JavaScript
* HTML
* CSS
* [Angular CLI](https://angular.io/cli)

You might find the [Tour of Heroes tutorial](https://angular.io/tutorial) helpful, but it is not required.

## **Create a sample application**

Using the Angular CLI, create a new application, angular-router-sample. This application will have two components: crisis-list and heroes-list.

1. Create a new Angular project, angular-router-sample.

content\_copyng new angular-router-sample

When prompted with Would you like to add Angular routing?, select N.

When prompted with Which stylesheet format would you like to use?, select CSS.

After a few moments, a new project, angular-router-sample, is ready.

1. From your terminal, navigate to the angular-router-sample directory.
2. Create a component, crisis-list.

content\_copyng generate component crisis-list

1. In your code editor, locate the file, crisis-list.component.html and replace the placeholder content with the following HTML.

src/app/crisis-list/crisis-list.component.html

content\_copy<h3>CRISIS CENTER</h3>

<p>Get your crisis here</p>

1. Create a second component, heroes-list.

content\_copyng generate component heroes-list

1. In your code editor, locate the file, heroes-list.component.html and replace the placeholder content with the following HTML.

src/app/heroes-list/heroes-list.component.html

content\_copy<h3>HEROES</h3>

<p>Get your heroes here</p>

1. In your code editor, open the file, app.component.html and replace its contents with the following HTML.

src/app/app.component.html

content\_copy<h1>Angular [Router](https://angular.io/api/router/Router) Sample</h1>

<app-crisis-list></app-crisis-list>

<app-heroes-list></app-heroes-list>

1. Verify that your new application runs as expected by running the ng serve command.

content\_copyng serve

1. Open a browser to http://localhost:4200.

You should see a single web page, consisting of a title and the HTML of your two components.

## **Import**[**RouterModule**](https://angular.io/api/router/RouterModule)**from @angular/router**

Routing allows you to display specific views of your application depending on the URL path. To add this functionality to your sample application, you need to update the app.module.ts file to use the module, [RouterModule](https://angular.io/api/router/RouterModule). You import this module from @angular/router.

1. From your code editor, open the app.module.ts file.
2. Add the following import statement.

src/app/app.module.ts

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

## **Define your routes**

In this section, you'll define two routes:

* The route /crisis-center opens the crisis-center component.
* The route /heroes-list opens the heroes-list component.

A route definition is a JavaScript object. Each route typically has two properties. The first property, path, is a string that specifies the URL path for the route. The second property, component, is a string that specifies what component your application should display for that path.

1. From your code editor, open the app.module.ts file.
2. Locate the @[NgModule](https://angular.io/api/core/NgModule)() section.
3. Replace the imports array in that section with the following.

src/app/app.module.ts

content\_copyimports: [

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

RouterModule.forRoot([

{path: 'crisis-list', component: CrisisListComponent},

{path: 'heroes-list', component: HeroesListComponent},

]),

],

This code adds the [RouterModule](https://angular.io/api/router/RouterModule) to the imports array. Next, the code uses the forRoot() method of the [RouterModule](https://angular.io/api/router/RouterModule) to define your two routes. This method takes an array of JavaScript objects, with each object defining the properties of a route. The forRoot() method ensures that your application only instantiates one [RouterModule](https://angular.io/api/router/RouterModule). For more information, see [Singleton Services](https://angular.io/guide/singleton-services#forroot-and-the-router).

## **Update your component with**[**router-outlet**](https://angular.io/api/router/RouterOutlet)

At this point, you have defined two routes for your application. However, your application still has both the crisis-list and heroes-list components hard-coded in your app.component.html template. For your routes to work, you need to update your template to dynamically load a component based on the URL path.

To implement this functionality, you add the [router-outlet](https://angular.io/api/router/RouterOutlet) directive to your template file.

1. From your code editor, open the app.component.html file.
2. Delete the following lines.

src/app/app.component.html

content\_copy<app-crisis-list></app-crisis-list>

<app-heroes-list></app-heroes-list>

1. Add the [router-outlet](https://angular.io/api/router/RouterOutlet) directive.

src/app/app.component.html

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

View your updated application in your browser. You should see only the application title. To view the crisis-list component, add crisis-list to the end of the path in your browser's address bar. For example:

content\_copyhttp://localhost:4200/crisis-list

Notice that the crisis-list component displays. Angular is using the route you defined to dynamically load the component. You can load the heroes-list component the same way:

content\_copyhttp://localhost:4200/heroes-list

## **Control navigation with UI elements**

Currently, your application supports two routes. However, the only way to use those routes is for the user to manually type the path in the browser's address bar. In this section, you'll add two links that users can click to navigate between the heroes-list and crisis-list components. You'll also add some CSS styles. While these styles are not required, they make it easier to identify the link for the currently-displayed component. You'll add that functionality in the next section.

1. Open the app.component.html file and add the following HTML below the title.

src/app/app.component.html

content\_copy<nav>

<a class="button" [routerLink](https://angular.io/api/router/RouterLink)="/crisis-list">Crisis Center</a> |

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

</nav>

This HTML uses an Angular directive, [routerLink](https://angular.io/api/router/RouterLink). This directive connects the routes you defined to your template files.

1. Open the app.component.css file and add the following styles.

src/app/app.component.css

content\_copy.button {

box-shadow: inset 0 1px 0 0 #ffffff;

background: #ffffff linear-gradient(to bottom, #ffffff 5%, #f6f6f6 100%);

border-radius: 6px;

border: 1px solid #dcdcdc;

display: inline-block;

cursor: pointer;

color: #666666;

font-family: Arial, sans-serif;

font-size: 15px;

font-weight: bold;

padding: 6px 24px;

text-decoration: none;

text-shadow: 0 1px 0 #ffffff;

outline: 0;

}

.activebutton {

box-shadow: inset 0 1px 0 0 #dcecfb;

background: #bddbfa linear-gradient(to bottom, #bddbfa 5%, #80b5ea 100%);

border-radius: 6px;

border: 1px solid #84bbf3;

display: inline-block;

cursor: pointer;

color: #ffffff;

font-family: Arial, sans-serif;

font-size: 15px;

font-weight: bold;

padding: 6px 24px;

text-decoration: none;

text-shadow: 0 1px 0 #528ecc;

outline: 0;

}

If you view your application in the browser, you should see these two links. When you click on a link, the corresponding component appears.

## **Identify the active route**

While users can navigate your application using the links you added in the previous section, they don't have an easy way to identify what the active route is. You can add this functionality using Angular's [routerLinkActive](https://angular.io/api/router/RouterLinkActive) directive.

1. From your code editor, open the app.component.html file.
2. Update the anchor tags to include the [routerLinkActive](https://angular.io/api/router/RouterLinkActive) directive.

src/app/app.component.html

content\_copy<nav>

<a class="button" [routerLink](https://angular.io/api/router/RouterLink)="/crisis-list" [routerLinkActive](https://angular.io/api/router/RouterLinkActive)="activebutton">Crisis Center</a> |

<a class="button" [routerLink](https://angular.io/api/router/RouterLink)="/heroes-list" [routerLinkActive](https://angular.io/api/router/RouterLinkActive)="activebutton">Heroes</a>

</nav>

View your application again. As you click one of the buttons, the style for that button updates automatically, identifying the active component to the user. By adding the [routerLinkActive](https://angular.io/api/router/RouterLinkActive) directive, you inform your application to apply a specific CSS class to the active route. In this tutorial, that CSS class is activebutton, but you could use any class that you want.

## **Adding a redirect**

In this step of the tutorial, you add a route that redirects the user to display the /heroes-list component.

1. From your code editor, open the app.module.ts file.
2. In the imports array, update the [RouterModule](https://angular.io/api/router/RouterModule) section as follows.

src/app/app.module.ts

content\_copyimports: [

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

RouterModule.forRoot([

{path: 'crisis-list', component: CrisisListComponent},

{path: 'heroes-list', component: HeroesListComponent},

{path: '', redirectTo: '/heroes-list', pathMatch: 'full'},

]),

],

Notice that this new route uses an empty string as its path. In addition, it replaces the component property with two new ones:

* + redirectTo. This property instructs Angular to redirect from an empty path to the heroes-list path.
  + pathMatch. This property instructs Angular on how much of the URL to match. For this tutorial, you should set this property to full. This strategy is recommended when you have an empty string for a path. For more information about this property, see the [Route API documentation](https://angular.io/api/router/Route).

Now when you open your application, it displays the heroes-list component by default.

## **Adding a 404 page**

It is possible for a user to try to access a route that you have not defined. To account for this behavior, the best practice is to display a 404 page. In this section, you'll create a 404 page and update your route configuration to show that page for any unspecified routes.

1. From the terminal, create a new component, PageNotFound.

content\_copyng generate component page-not-found

1. From your code editor, open the page-not-found.component.html file and replace its contents with the following HTML.

src/app/page-not-found/page-not-found.component.html

content\_copy<h2>Page Not Found</h2>

<p>We couldn't find that page! Not even with x-ray vision.</p>

1. Open the app.module.ts file. In the imports array, update the [RouterModule](https://angular.io/api/router/RouterModule) section as follows.

src/app/app.module.ts

content\_copyimports: [

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

RouterModule.forRoot([

{path: 'crisis-list', component: CrisisListComponent},

{path: 'heroes-list', component: HeroesListComponent},

{path: '', redirectTo: '/heroes-list', pathMatch: 'full'},

{path: '\*\*', component: PageNotFoundComponent}

]),

],

The new route uses a path, \*\*. This path is how Angular identifies a wildcard route. Any route that does not match an existing route in your configuration will use this route.

Notice that the wildcard route is placed at the end of the array. The order of your routes is important, as Angular applies routes in order and uses the first match it finds.

Try navigating to a non-existing route on your application, such as http://localhost:4200/powers. This route doesn't match anything defined in your app.module.ts file. However, because you defined a wildcard route, the application automatically displays your PageNotFound component.

## **Next steps**

At this point, you have a basic application that uses Angular's routing feature to change what components the user can see based on the URL address. You have extended these features to include a redirect, as well as a wildcard route to display a custom 404 page.

For more information about routing, see the following topics:

* [In-app Routing and Navigation](https://angular.io/guide/router)
* [Router API](https://angular.io/api/router)

# Tutorial: Creating custom route matches

The Angular Router supports a powerful matching strategy that you can use to help users navigate your application. This matching strategy supports static routes, variable routes with parameters, wildcard routes, and so on. You can also build your own custom pattern matching for situations in which the URLs are more complicated.

In this tutorial, you'll build a custom route matcher using Angular's [UrlMatcher](https://angular.io/api/router/UrlMatcher). This matcher looks for a Twitter handle in the URL.

For a working example of the final version of this tutorial, see the [live example](https://angular.io/generated/live-examples/routing-with-urlmatcher/stackblitz.html) / [download example](https://angular.io/generated/zips/routing-with-urlmatcher/routing-with-urlmatcher.zip).

## **Objectives**

Implement Angular's [UrlMatcher](https://angular.io/api/router/UrlMatcher) to create a custom route matcher.

## **Prerequisites**

To complete this tutorial, you should have a basic understanding of the following concepts:

* JavaScript
* HTML
* CSS
* [Angular CLI](https://angular.io/cli)

If you are unfamiliar with how Angular's router works, we recommend you review [Using Angular routes in a single-page application](https://angular.io/guide/router-tutorial).

## **Create a sample application**

Using the Angular CLI, create a new application, angular-custom-route-match. In addition to the default Angular application framework, you will also create a profile component.

1. Create a new Angular project, angular-custom-route-match.

content\_copyng new angular-custom-route-match

When prompted with Would you like to add Angular routing?, select Y.

When prompted with Which stylesheet format would you like to use?, select CSS.

After a few moments, a new project, angular-custom-route-match, is ready.

1. From your terminal, navigate to the angular-custom-route-match directory.
2. Create a component, profile.

content\_copyng generate component profile

1. In your code editor, locate the file, profile.component.html and replace the placeholder content with the following HTML.

src/app/profile/profile.component.html

content\_copy<p>

Hello {{ username$ | async }}!

</p>

1. In your code editor, locate the file, app.component.html and replace the placeholder content with the following HTML.

src/app/app.component.html

content\_copy<h2>Routing with Custom Matching</h2>

Navigate to <a [routerLink](https://angular.io/api/router/RouterLink)="/@Angular">my profile</a>

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

## **Configure your routes for your application**

With your application framework in place, you next need to add routing capabilities to the app.module.ts file. As a part of this process, you will create a custom URL matcher that looks for a Twitter handle in the URL. This handle is identified by a preceding @ symbol.

1. In your code editor, open your app.module.ts file.
2. Add an import statement for Angular's [RouterModule](https://angular.io/api/router/RouterModule) and [UrlMatcher](https://angular.io/api/router/UrlMatcher).

src/app/app.module.ts

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

1. In the imports array, add a RouterModule.forRoot([]) statement.

src/app/app.module.ts

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

imports: [

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

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

RouterModule.forRoot([

{

/\* . . . \*/

])],

declarations: [ AppComponent, ProfileComponent ],

bootstrap: [ AppComponent ]

})

1. Define the custom route matcher by adding the following code to the [RouterModule.forRoot()](https://angular.io/api/router/RouterModule#forRoot) statement.

src/app/app.module.ts

content\_copymatcher: (url) => {

if (url.length === 1 && url[0].path.match(/^@[\w]+$/gm)) {

return {

consumed: url,

posParams: {

username: new [UrlSegment](https://angular.io/api/router/UrlSegment)(url[0].path.substr(1), {})

}

};

}

return null;

},

component: ProfileComponent

}

This custom matcher is a function that performs the following tasks:

* The matcher verifies that the array contains only one segment.
* The matcher employs a regular expression to ensure that the format of the username is a match.
* If there is a match, the function returns the entire URL, defining a username route parameter as a substring of the path.
* If there isn't a match, the function returns null and the router continues to look for other routes that match the URL.

A custom URL matcher behaves like any other route definition. You can define child routes or lazy loaded routes as you would with any other route.

## **Subscribe to the route parameters**

With the custom matcher in place, you now need to subscribe to the route parameters in the profile component.

1. In your code editor, open your profile.component.ts file.
2. Add an import statement for Angular's [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) and [ParamMap](https://angular.io/api/router/ParamMap).

src/app/profile/profile.component.ts

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

1. Add an import statement for RxJS map.

src/app/profile/profile.component.ts

content\_copyimport { map } from 'rxjs/operators';

1. Subscribe to the username route parameter.

src/app/profile/profile.component.ts

content\_copyusername$ = this.route.paramMap

.pipe(

map((params: [ParamMap](https://angular.io/api/router/ParamMap)) => params.get('username'))

);

1. Inject the [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) into the component's constructor.

src/app/profile/profile.component.ts

content\_copyconstructor(private route: [ActivatedRoute](https://angular.io/api/router/ActivatedRoute)) { }

## **Test your custom URL matcher**

With your code in place, you can now test your custom URL matcher.

1. From a terminal window, run the ng serve command.

content\_copyng serve

1. Open a browser to http://localhost:4200.

You should see a single web page, consisting of a sentence that reads Navigate to my profile.

1. Click the **my profile** hyperlink.

A new sentence, reading Hello, Angular! appears on the page.

## **Next steps**

Pattern matching with the Angular Router provides you with a lot of flexibility when you have dynamic URLs in your application. To learn more about the Angular Router, see the following topics:

* [In-app Routing and Navigation](https://angular.io/guide/router)
* [Router API](https://angular.io/api/router)

This content is based on [Custom Route Matching with the Angular Router](https://medium.com/@brandontroberts/custom-route-matching-with-the-angular-router-fbdd48665483), by [Brandon Rober](https://twitter.com/brandontroberts)ters

# Router tutorial: tour of heroes

This tutorial provides an extensive overview of the Angular router. In this tutorial, you will build upon a basic router configuration to explore features such as child routes, route parameters, lazy load NgModules, guard routes, and preloading data to improve the user experience.

For a working example of the final version of the app, see the [live example](https://angular.io/generated/live-examples/router/stackblitz.html) / [download example](https://angular.io/generated/zips/router/router.zip).

## **Objectives**

This guide describes development of a multi-page routed sample application. Along the way, it highlights key features of the router such as:

* Organizing the application features into modules.
* Navigating to a component (Heroes link to "Heroes List").
* Including a route parameter (passing the Hero id while routing to the "Hero Detail").
* Child routes (the Crisis Center has its own routes).
* The [CanActivate](https://angular.io/api/router/CanActivate) guard (checking route access).
* The [CanActivateChild](https://angular.io/api/router/CanActivateChild) guard (checking child route access).
* The [CanDeactivate](https://angular.io/api/router/CanDeactivate) guard (ask permission to discard unsaved changes).
* The [Resolve](https://angular.io/api/router/Resolve) guard (pre-fetching route data).
* Lazy loading an [NgModule](https://angular.io/api/core/NgModule).
* The [CanLoad](https://angular.io/api/router/CanLoad) guard (check before loading feature module assets).

This guide proceeds as a sequence of milestones as if you were building the application step-by-step, but assumes you are familiar with basic [Angular concepts](https://angular.io/guide/architecture). For a general introduction to angular, see the [Getting Started](https://angular.io/start). For a more in-depth overview, see the [Tour of Heroes](https://angular.io/tutorial) tutorial.

## **Prerequisites**

To complete this tutorial, you should have a basic understanding of the following concepts:

* JavaScript
* HTML
* CSS
* [Angular CLI](https://angular.io/cli)

You might find the [Tour of Heroes tutorial](https://angular.io/tutorial) helpful, but it is not required.

## **The sample application in action**

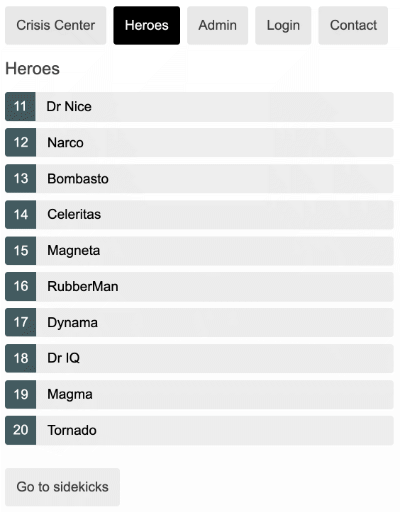
The sample application for this tutorial helps the Hero Employment Agency find crises for heroes to solve.

The application has three main feature areas:

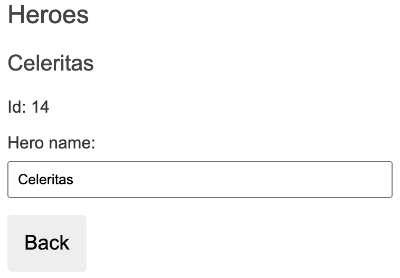
1. A Crisis Center for maintaining the list of crises for assignment to heroes.
2. A Heroes area for maintaining the list of heroes employed by the agency.
3. An Admin area to manage the list of crises and heroes.

Try it by clicking on this [live example link](https://angular.io/generated/live-examples/router/stackblitz.html) / [download example](https://angular.io/generated/zips/router/router.zip).

The application renders with a row of navigation buttons and the Heroes view with its list of heroes.



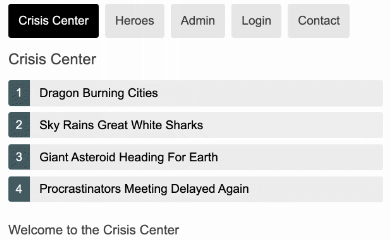
Select one hero and the application takes you to a hero editing screen.



Alter the name. Click the "Back" button and the application returns to the heroes list which displays the changed hero name. Notice that the name change took effect immediately.

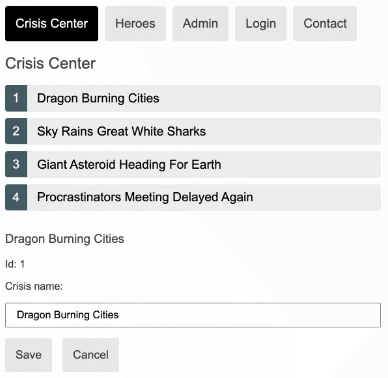
Had you clicked the browser's back button instead of the application's "Back" button, the application would have returned you to the heroes list as well. Angular application navigation updates the browser history as normal web navigation does.

Now click the Crisis Center link for a list of ongoing crises.



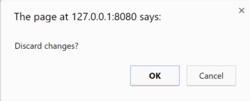
Select a crisis and the application takes you to a crisis editing screen. The Crisis Detail appears in a child component on the same page, beneath the list.

Alter the name of a crisis. Notice that the corresponding name in the crisis list does not change.



Unlike Hero Detail, which updates as you type, Crisis Detail changes are temporary until you either save or discard them by pressing the "Save" or "Cancel" buttons. Both buttons navigate back to the Crisis Center and its list of crises.

Click the browser back button or the "Heroes" link to activate a dialog.



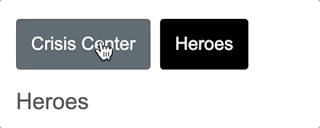
You can say "OK" and lose your changes or click "Cancel" and continue editing.

Behind this behavior is the router's [CanDeactivate](https://angular.io/api/router/CanDeactivate) guard. The guard gives you a chance to clean-up or ask the user's permission before navigating away from the current view.

The Admin and Login buttons illustrate other router capabilities covered later in the guide.

## **Milestone 1: Getting started**

Begin with a basic version of the application that navigates between two empty views.



Generate a sample application with the Angular CLI.

content\_copyng new angular-router-sample

### **Define Routes**

A router must be configured with a list of route definitions.

Each definition translates to a [Route](https://angular.io/api/router/Route) object which has two things: a path, the URL path segment for this route; and a component, the component associated with this route.

The router draws upon its registry of definitions when the browser URL changes or when application code tells the router to navigate along a route path.

The first route does the following:

* When the browser's location URL changes to match the path segment /crisis-center, then the router activates an instance of the CrisisListComponent and displays its view.
* When the application requests navigation to the path /crisis-center, the router activates an instance of CrisisListComponent, displays its view, and updates the browser's address location and history with the URL for that path.

The first configuration defines an array of two routes with minimal paths leading to the CrisisListComponent and HeroListComponent.

Generate the CrisisList and HeroList components so that the router has something to render.

content\_copyng generate component crisis-list

content\_copyng generate component hero-list

Replace the contents of each component with the sample HTML below.

src/app/crisis-list/crisis-list.component.html

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

content\_copy<h2>CRISIS CENTER</h2>

<p>Get your crisis here</p>

### **Register**[**Router**](https://angular.io/api/router/Router)**and**[**Routes**](https://angular.io/api/router/Routes)

In order to use the [Router](https://angular.io/api/router/Router), you must first register the [RouterModule](https://angular.io/api/router/RouterModule) from the @angular/router package. Define an array of routes, appRoutes, and pass them to the [RouterModule.forRoot()](https://angular.io/api/router/RouterModule#forRoot) method. The [RouterModule.forRoot()](https://angular.io/api/router/RouterModule#forRoot) method returns a module that contains the configured [Router](https://angular.io/api/router/Router) service provider, plus other providers that the routing library requires. Once the application is bootstrapped, the [Router](https://angular.io/api/router/Router) performs the initial navigation based on the current browser URL.

**Note:** The [RouterModule.forRoot()](https://angular.io/api/router/RouterModule#forRoot) method is a pattern used to register application-wide providers. Read more about application-wide providers in the [Singleton services](https://angular.io/guide/singleton-services#forRoot-router) guide.

src/app/app.module.ts (first-config)

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

import { [BrowserModule](https://angular.io/api/platform-browser/BrowserModule) } from '@angular/platform-browser';

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

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

import { AppComponent } from './app.component';

import { CrisisListComponent } from './crisis-list/crisis-list.component';

import { HeroListComponent } from './hero-list/hero-list.component';

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

{ path: 'crisis-center', component: CrisisListComponent },

{ path: 'heroes', component: HeroListComponent },

];

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

imports: [

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

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

RouterModule.forRoot(

appRoutes,

{ enableTracing: true } // <-- debugging purposes only

)

],

declarations: [

AppComponent,

HeroListComponent,

CrisisListComponent,

],

bootstrap: [ AppComponent ]

})

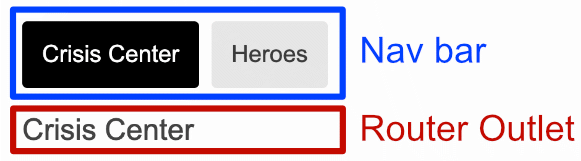
export class AppModule { }

Adding the configured [RouterModule](https://angular.io/api/router/RouterModule) to the AppModule is sufficient for minimal route configurations. However, as the application grows, [refactor the routing configuration](https://angular.io/guide/router-tutorial-toh#refactor-the-routing-configuration-into-a-routing-module) into a separate file and create a [Routing Module](https://angular.io/guide/router-tutorial-toh#routing-module). A routing module is a special type of Service Module dedicated to routing.

Registering the [RouterModule.forRoot()](https://angular.io/api/router/RouterModule#forRoot) in the AppModule imports array makes the [Router](https://angular.io/api/router/Router) service available everywhere in the application.

### **Add the Router Outlet**

The root AppComponent is the application shell. It has a title, a navigation bar with two links, and a router outlet where the router renders components.



The router outlet serves as a placeholder where the routed components are rendered.

The corresponding component template looks like this:

src/app/app.component.html

content\_copy<h1>Angular [Router](https://angular.io/api/router/Router)</h1>

<nav>

<a [routerLink](https://angular.io/api/router/RouterLink)="/crisis-center" [routerLinkActive](https://angular.io/api/router/RouterLinkActive)="active">Crisis Center</a>

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

</nav>

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

### **Define a Wildcard route**

You've created two routes in the application so far, one to /crisis-center and the other to /heroes. Any other URL causes the router to throw an error and crash the app.

Add a wildcard route to intercept invalid URLs and handle them gracefully. A wildcard route has a path consisting of two asterisks. It matches every URL. Thus, the router selects this wildcard route if it can't match a route earlier in the configuration. A wildcard route can navigate to a custom "404 Not Found" component or [redirect](https://angular.io/guide/router-tutorial-toh#redirect) to an existing route.

The router selects the route with a [first match wins](https://angular.io/guide/router-reference#example-config) strategy. Because a wildcard route is the least specific route, place it last in the route configuration.

To test this feature, add a button with a [RouterLink](https://angular.io/api/router/RouterLink) to the HeroListComponent template and set the link to a non-existant route called "/sidekicks".

src/app/hero-list/hero-list.component.html (excerpt)

content\_copy<h2>HEROES</h2>

<p>Get your heroes here</p>

<button [routerLink](https://angular.io/api/router/RouterLink)="/sidekicks">Go to sidekicks</button>

The application fails if the user clicks that button because you haven't defined a "/sidekicks" route yet.

Instead of adding the "/sidekicks" route, define a wildcard route and have it navigate to a PageNotFoundComponent.

src/app/app.module.ts (wildcard)

content\_copy{ path: '\*\*', component: PageNotFoundComponent }

Create the PageNotFoundComponent to display when users visit invalid URLs.

content\_copyng generate component page-not-found

src/app/page-not-found.component.html (404 component)

content\_copy<h2>Page not found</h2>

Now when the user visits /sidekicks, or any other invalid URL, the browser displays "Page not found". The browser address bar continues to point to the invalid URL.

### **Set up redirects**

When the application launches, the initial URL in the browser bar is by default:

content\_copylocalhost:4200

That doesn't match any of the hard-coded routes which means the router falls through to the wildcard route and displays the PageNotFoundComponent.

The application needs a default route to a valid page. The default page for this application is the list of heroes. The application should navigate there as if the user clicked the "Heroes" link or pasted localhost:4200/heroes into the address bar.

Add a redirect route that translates the initial relative URL ('') to the desired default path (/heroes).

Add the default route somewhere above the wildcard route. It's just above the wildcard route in the following excerpt showing the complete appRoutes for this milestone.

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

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

{ path: 'crisis-center', component: CrisisListComponent },

{ path: 'heroes', component: HeroListComponent },

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

{ path: '\*\*', component: PageNotFoundComponent }

];

The browser address bar shows .../heroes as if you'd navigated there directly.

A redirect route requires a pathMatch property to tell the router how to match a URL to the path of a route. In this app, the router should select the route to the HeroListComponent only when the entire URL matches '', so set the pathMatch value to 'full'.

SPOTLIGHT ON PATHMATCH

Technically, pathMatch = 'full' results in a route hit when the remaining, unmatched segments of the URL match ''. In this example, the redirect is in a top level route so the remaining URL and the entire URL are the same thing.

The other possible pathMatch value is 'prefix' which tells the router to match the redirect route when the remaining URL begins with the redirect route's prefix path. This doesn't apply to this sample application because if the pathMatch value were 'prefix', every URL would match ''.

Try setting it to 'prefix' and clicking the Go to sidekicks button. Since that's a bad URL, you should see the "Page not found" page. Instead, you're still on the "Heroes" page. Enter a bad URL in the browser address bar. You're instantly re-routed to /heroes. Every URL, good or bad, that falls through to this route definition is a match.

The default route should redirect to the HeroListComponent only when the entire url is ''. Remember to restore the redirect to pathMatch = 'full'.

Learn more in Victor Savkin's [post on redirects](https://vsavkin.tumblr.com/post/146722301646/angular-router-empty-paths-componentless-routes).

### **Milestone 1 wrap up**

Your sample application can switch between two views when the user clicks a link.

Milestone 1 has covered how to do the following:

* Load the router library.
* Add a nav bar to the shell template with anchor tags, [routerLink](https://angular.io/api/router/RouterLink) and [routerLinkActive](https://angular.io/api/router/RouterLinkActive) directives.
* Add a [router-outlet](https://angular.io/api/router/RouterOutlet) to the shell template where views are displayed.
* Configure the router module with [RouterModule.forRoot()](https://angular.io/api/router/RouterModule#forRoot).
* Set the router to compose HTML5 browser URLs.
* Handle invalid routes with a wildcard route.
* Navigate to the default route when the application launches with an empty path.

The starter application's structure looks like this:

angular-router-sample

src

app

crisis-list

crisis-list.component.css

crisis-list.component.html

crisis-list.component.ts

hero-list

hero-list.component.css

hero-list.component.html

hero-list.component.ts

page-not-found

page-not-found.component.css

page-not-found.component.html

page-not-found.component.ts

app.component.css

app.component.html

app.component.ts

app.module.ts

main.ts

index.html

styles.css

tsconfig.json

node\_modules ...

package.json

Here are the files in this milestone.

app.component.html

app.module.ts

hero-list/hero-list.component.html

crisis-list/crisis-list.component.html

page-not-found/page-not-found.component.html

index.html

content\_copy<h1>Angular [Router](https://angular.io/api/router/Router)</h1>

<nav>

<a [routerLink](https://angular.io/api/router/RouterLink)="/crisis-center" [routerLinkActive](https://angular.io/api/router/RouterLinkActive)="active">Crisis Center</a>

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

</nav>

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

## **Milestone 2: Routing module**

This milestone shows you how to configure a special-purpose module called a Routing Module, which holds your application's routing configuration.

The Routing Module has several characteristics:

* Separates routing concerns from other application concerns.
* Provides a module to replace or remove when testing the application.
* Provides a well-known location for routing service providers such as guards and resolvers.
* Does not declare components.

### **Integrate routing with your app**

The sample routing application does not include routing by default. When you use the [Angular CLI](https://angular.io/cli) to create a project that does use routing, set the --routing option for the project or application, and for each NgModule. When you create or initialize a new project (using the CLI [ng new](https://angular.io/cli/new) command) or a new application (using the [ng generate app](https://angular.io/cli/generate) command), specify the --routing option. This tells the CLI to include the @angular/router npm package and create a file named app-routing.module.ts. You can then use routing in any NgModule that you add to the project or application.

For example, the following command generates an NgModule that can use routing.

content\_copyng generate module my-module --routing

This creates a separate file named my-module-routing.module.ts to store the NgModule's routes. The file includes an empty [Routes](https://angular.io/api/router/Routes) object that you can fill with routes to different components and NgModules.

### **Refactor the routing configuration into a routing module**

Create an AppRouting module in the /app folder to contain the routing configuration.

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

Import the CrisisListComponent, HeroListComponent, and PageNotFoundComponent symbols like you did in the app.module.ts. Then move the [Router](https://angular.io/api/router/Router) imports and routing configuration, including [RouterModule.forRoot()](https://angular.io/api/router/RouterModule#forRoot), into this routing module.

Re-export the Angular [RouterModule](https://angular.io/api/router/RouterModule) by adding it to the module exports array. By re-exporting the [RouterModule](https://angular.io/api/router/RouterModule) here, the components declared in AppModule have access to router directives such as [RouterLink](https://angular.io/api/router/RouterLink) and [RouterOutlet](https://angular.io/api/router/RouterOutlet).

After these steps, the file should look like this.

src/app/app-routing.module.ts

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

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

import { CrisisListComponent } from './crisis-list/crisis-list.component';

import { HeroListComponent } from './hero-list/hero-list.component';

import { PageNotFoundComponent } from './page-not-found/page-not-found.component';

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

{ path: 'crisis-center', component: CrisisListComponent },

{ path: 'heroes', component: HeroListComponent },

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

{ path: '\*\*', component: PageNotFoundComponent }

];

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

imports: [

RouterModule.forRoot(

appRoutes,

{ enableTracing: true } // <-- debugging purposes only

)

],

exports: [

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

]

})

export class AppRoutingModule {}

Next, update the app.module.ts file by removing RouterModule.forRoot in the imports array.

src/app/app.module.ts

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

import { [BrowserModule](https://angular.io/api/platform-browser/BrowserModule) } from '@angular/platform-browser';

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

import { AppComponent } from './app.component';

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

import { CrisisListComponent } from './crisis-list/crisis-list.component';

import { HeroListComponent } from './hero-list/hero-list.component';

import { PageNotFoundComponent } from './page-not-found/page-not-found.component';

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

imports: [

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

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

AppRoutingModule

],

declarations: [

AppComponent,

HeroListComponent,

CrisisListComponent,

PageNotFoundComponent

],

bootstrap: [ AppComponent ]

})

export class AppModule { }

Later, this guide shows you how to create [multiple routing modules](https://angular.io/guide/router-tutorial-toh#heroes-functionality) and import those routing modules [in the correct order](https://angular.io/guide/router-tutorial-toh#routing-module-order).

The application continues to work just the same, and you can use AppRoutingModule as the central place to maintain future routing configuration.

### **Benefits of a routing module**

The routing module, often called the AppRoutingModule, replaces the routing configuration in the root or feature module.

The routing module is helpful as your application grows and when the configuration includes specialized guard and resolver services.

Some developers skip the routing module when the configuration is minimal and merge the routing configuration directly into the companion module (for example, AppModule).

Most applications should implement a routing module for consistency. It keeps the code clean when configuration becomes complex. It makes testing the feature module easier. Its existence calls attention to the fact that a module is routed. It is where developers expect to find and expand routing configuration.

## **Milestone 3: Heroes feature**

This milestone covers the following:

* Organizing the application and routes into feature areas using modules.
* Navigating imperatively from one component to another.
* Passing required and optional information in route parameters.

This sample application recreates the heroes feature in the "Services" section of the [Tour of Heroes tutorial](https://angular.io/tutorial/toh-pt4), and reuses much of the code from the [Tour of Heroes: Services example code](https://angular.io/generated/live-examples/toh-pt4/stackblitz.html) / [download example](https://angular.io/generated/zips/toh-pt4/toh-pt4.zip).

A typical application has multiple feature areas, each dedicated to a particular business purpose with its own folder.

This section shows you how refactor the application into different feature modules, import them into the main module and navigate among them.

### **Add heroes functionality**

Follow these steps:

* To manage the heroes, create a HeroesModule with routing in the heroes folder and register it with the root AppModule.

content\_copyng generate module heroes/heroes --module app --flat --routing

* Move the placeholder hero-list folder that's in the app folder into the heroes folder.
* Copy the contents of the heroes/heroes.component.html from the ["Services" tutorial](https://angular.io/generated/live-examples/toh-pt4/stackblitz.html) / [download example](https://angular.io/generated/zips/toh-pt4/toh-pt4.zip) into the hero-list.component.html template.
  + Re-label the <h2> to <h2>HEROES</h2>.
  + Delete the <app-hero-detail> component at the bottom of the template.
* Copy the contents of the heroes/heroes.component.css from the live example into the hero-list.component.css file.
* Copy the contents of the heroes/heroes.component.ts from the live example into the hero-list.component.ts file.
  + Change the component class name to HeroListComponent.
  + Change the selector to app-hero-list.

Selectors are not required for routed components because components are dynamically inserted when the page is rendered. However, they are useful for identifying and targeting them in your HTML element tree.

* Copy the hero-detail folder, the hero.ts, hero.service.ts, and mock-heroes.ts files into the heroes subfolder.
* Copy the message.service.ts into the src/app folder.
* Update the relative path import to the message.service in the hero.service.ts file.

Next, update the HeroesModule metadata.

* Import and add the HeroDetailComponent and HeroListComponent to the declarations array in the HeroesModule.

src/app/heroes/heroes.module.ts

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

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

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

import { HeroListComponent } from './hero-list/hero-list.component';

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

import { HeroesRoutingModule } from './heroes-routing.module';

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

imports: [

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

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

HeroesRoutingModule

],

declarations: [

HeroListComponent,

HeroDetailComponent

]

})

export class HeroesModule {}

The hero management file structure is as follows:

src/app/heroes

hero-detail

hero-detail.component.css

hero-detail.component.html

hero-detail.component.ts

hero-list

hero-list.component.css

hero-list.component.html

hero-list.component.ts

hero.service.ts

hero.ts

heroes-routing.module.ts

heroes.module.ts

mock-heroes.ts

#### Hero feature routing requirements

The heroes feature has two interacting components, the hero list and the hero detail. When you navigate to list view, it gets a list of heroes and displays them. When you click on a hero, the detail view has to display that particular hero.

You tell the detail view which hero to display by including the selected hero's id in the route URL.

Import the hero components from their new locations in the src/app/heroes/ folder and define the two hero routes.

Now that you have routes for the Heroes module, register them with the [Router](https://angular.io/api/router/Router) using the [RouterModule](https://angular.io/api/router/RouterModule) as you did in the AppRoutingModule, with an important difference.

In the AppRoutingModule, you used the static [RouterModule.forRoot()](https://angular.io/api/router/RouterModule#forRoot) method to register the routes and application level service providers. In a feature module you use the static forChild() method.

Only call [RouterModule.forRoot()](https://angular.io/api/router/RouterModule#forRoot) in the root AppRoutingModule (or the AppModule if that's where you register top level application routes). In any other module, you must call the [RouterModule.forChild()](https://angular.io/api/router/RouterModule#forChild) method to register additional routes.

The updated HeroesRoutingModule looks like this:

src/app/heroes/heroes-routing.module.ts

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

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

import { HeroListComponent } from './hero-list/hero-list.component';

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

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

{ path: 'heroes', component: HeroListComponent },

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

];

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

imports: [

RouterModule.forChild(heroesRoutes)

],

exports: [

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

]

})

export class HeroesRoutingModule { }

Consider giving each feature module its own route configuration file. Though the feature routes are currently minimal, routes have a tendency to grow more complex even in small applications.

#### Remove duplicate hero routes

The hero routes are currently defined in two places: in the HeroesRoutingModule, by way of the HeroesModule, and in the AppRoutingModule.

Routes provided by feature modules are combined together into their imported module's routes by the router. This allows you to continue defining the feature module routes without modifying the main route configuration.

Remove the HeroListComponent import and the /heroes route from the app-routing.module.ts.

Leave the default and the wildcard routes as these are still in use at the top level of the application.

src/app/app-routing.module.ts (v2)

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

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

import { CrisisListComponent } from './crisis-list/crisis-list.component';

// import { HeroListComponent } from './hero-list/hero-list.component'; // <-- delete this line

import { PageNotFoundComponent } from './page-not-found/page-not-found.component';

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

{ path: 'crisis-center', component: CrisisListComponent },

// { path: 'heroes', component: HeroListComponent }, // <-- delete this line

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

{ path: '\*\*', component: PageNotFoundComponent }

];

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

imports: [

RouterModule.forRoot(

appRoutes,

{ enableTracing: true } // <-- debugging purposes only

)

],

exports: [

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

]

})

export class AppRoutingModule {}

#### Remove heroes declarations

Because the HeroesModule now provides the HeroListComponent, remove it from the AppModule's declarations array. Now that you have a separate HeroesModule, you can evolve the hero feature with more components and different routes.

After these steps, the AppModule should look like this:

src/app/app.module.ts

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

import { [BrowserModule](https://angular.io/api/platform-browser/BrowserModule) } from '@angular/platform-browser';

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

import { AppComponent } from './app.component';

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

import { HeroesModule } from './heroes/heroes.module';

import { CrisisListComponent } from './crisis-list/crisis-list.component';

import { PageNotFoundComponent } from './page-not-found/page-not-found.component';

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

imports: [

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

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

HeroesModule,

AppRoutingModule

],

declarations: [

AppComponent,

CrisisListComponent,

PageNotFoundComponent

],

bootstrap: [ AppComponent ]

})

export class AppModule { }

### **Module import order**

Notice that in the module imports array, the AppRoutingModule is last and comes after the HeroesModule.

src/app/app.module.ts (module-imports)

content\_copyimports: [

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

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

HeroesModule,

AppRoutingModule

],

The order of route configuration is important because the router accepts the first route that matches a navigation request path.

When all routes were in one AppRoutingModule, you put the default and [wildcard](https://angular.io/guide/router-tutorial-toh#wildcard) routes last, after the /heroes route, so that the router had a chance to match a URL to the /heroes route before hitting the wildcard route and navigating to "Page not found".

Each routing module augments the route configuration in the order of import. If you listed AppRoutingModule first, the wildcard route would be registered before the hero routes. The wildcard route—which matches every URL—would intercept the attempt to navigate to a hero route.

Reverse the routing modules to see a click of the heroes link resulting in "Page not found". Learn about inspecting the runtime router configuration [below](https://angular.io/guide/router-tutorial-toh#inspect-config).

### **Route Parameters**

#### Route definition with a parameter

Return to the HeroesRoutingModule and look at the route definitions again. The route to HeroDetailComponent has an :id token in the path.

src/app/heroes/heroes-routing.module.ts (excerpt)

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

The :id token creates a slot in the path for a Route Parameter. In this case, this configuration causes the router to insert the id of a hero into that slot.

If you tell the router to navigate to the detail component and display "Magneta", you expect a hero id to appear in the browser URL like this:

content\_copylocalhost:4200/hero/15

If a user enters that URL into the browser address bar, the router should recognize the pattern and go to the same "Magneta" detail view.

ROUTE PARAMETER: REQUIRED OR OPTIONAL?

Embedding the route parameter token, :id, in the route definition path is a good choice for this scenario because the id is required by the HeroDetailComponent and because the value 15 in the path clearly distinguishes the route to "Magneta" from a route for some other hero.

#### Setting the route parameters in the list view

After navigating to the HeroDetailComponent, you expect to see the details of the selected hero. You need two pieces of information: the routing path to the component and the hero's id.

Accordingly, the link parameters array has two items: the routing path and a route parameter that specifies the id of the selected hero.

src/app/heroes/hero-list/hero-list.component.html (link-parameters-array)

content\_copy<a [[routerLink](https://angular.io/api/router/RouterLink)]="['/hero', hero.id]">

The router composes the destination URL from the array like this: localhost:4200/hero/15.

The router extracts the route parameter (id:15) from the URL and supplies it to the HeroDetailComponent using the [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) service.

### **Activated**[**Route**](https://angular.io/api/router/Route)**in action**

Import the [Router](https://angular.io/api/router/Router), [ActivatedRoute](https://angular.io/api/router/ActivatedRoute), and [ParamMap](https://angular.io/api/router/ParamMap) tokens from the router package.

src/app/heroes/hero-detail/hero-detail.component.ts (activated route)

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

Import the switchMap operator because you need it later to process the Observable route parameters.

src/app/heroes/hero-detail/hero-detail.component.ts (switchMap operator import)

content\_copyimport { switchMap } from 'rxjs/operators';

Add the services as private variables to the constructor so that Angular injects them (makes them visible to the component).

src/app/heroes/hero-detail/hero-detail.component.ts (constructor)

content\_copyconstructor(

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

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

private service: HeroService

) {}

In the ngOnInit() method, use the [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) service to retrieve the parameters for the route, pull the hero id from the parameters, and retrieve the hero to display.

src/app/heroes/hero-detail/hero-detail.component.ts (ngOnInit)

content\_copyngOnInit() {

this.hero$ = this.route.paramMap.pipe(

switchMap((params: [ParamMap](https://angular.io/api/router/ParamMap)) =>

this.service.getHero(params.get('id')!))

);

}

When the map changes, paramMap gets the id parameter from the changed parameters.

Then you tell the HeroService to fetch the hero with that id and return the result of the HeroService request.

The switchMap operator does two things. It flattens the Observable<Hero> that HeroService returns and cancels previous pending requests. If the user re-navigates to this route with a new id while the HeroService is still retrieving the old id, switchMap discards that old request and returns the hero for the new id.

[AsyncPipe](https://angular.io/api/common/AsyncPipe) handles the observable subscription and the component's hero property will be (re)set with the retrieved hero.

#### ParamMap API

The [ParamMap](https://angular.io/api/router/ParamMap) API is inspired by the [URLSearchParams interface](https://developer.mozilla.org/en-US/docs/Web/API/URLSearchParams). It provides methods to handle parameter access for both route parameters (paramMap) and query parameters (queryParamMap).

|  |  |
| --- | --- |
| **Member** | **Description** |
| has(name) | Returns true if the parameter name is in the map of parameters. |
| get(name) | Returns the parameter name value (a string) if present, or null if the parameter name is not in the map. Returns the first element if the parameter value is actually an array of values. |
| getAll(name) | Returns a string array of the parameter name value if found, or an empty array if the parameter name value is not in the map. Use getAll when a single parameter could have multiple values. |
| keys | Returns a string array of all parameter names in the map. |

#### Observable *paramMap* and component reuse

In this example, you retrieve the route parameter map from an Observable. That implies that the route parameter map can change during the lifetime of this component.

By default, the router re-uses a component instance when it re-navigates to the same component type without visiting a different component first. The route parameters could change each time.

Suppose a parent component navigation bar had "forward" and "back" buttons that scrolled through the list of heroes. Each click navigated imperatively to the HeroDetailComponent with the next or previous id.

You wouldn't want the router to remove the current HeroDetailComponent instance from the DOM only to re-create it for the next id as this would re-render the view. For better UX, the router re-uses the same component instance and updates the parameter.

Since ngOnInit() is only called once per component instantiation, you can detect when the route parameters change from within the same instance using the observable paramMap property.

When subscribing to an observable in a component, you almost always unsubscribe when the component is destroyed.

However, [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) observables are among the exceptions because [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) and its observables are insulated from the [Router](https://angular.io/api/router/Router) itself. The [Router](https://angular.io/api/router/Router) destroys a routed component when it is no longer needed. This means all the component's members will also be destroyed, including the injected [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) and the subscriptions to its Observable properties.

The [Router](https://angular.io/api/router/Router) does not complete any Observable of the [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) so any finalize or complete blocks will not run. If you need to handle something in a finalize, you will still need to unsubscribe in ngOnDestroy. You will also have to unsubscribe if your observable pipe has a delay with code you do not want to run after the component is destroyed.

#### snapshot: the no-observable alternative

This application won't re-use the HeroDetailComponent. The user always returns to the hero list to select another hero to view. There's no way to navigate from one hero detail to another hero detail without visiting the list component in between. Therefore, the router creates a new HeroDetailComponent instance every time.

When you know for certain that a HeroDetailComponent instance will never be re-used, you can use snapshot.

route.snapshot provides the initial value of the route parameter map. You can access the parameters directly without subscribing or adding observable operators as in the following:

src/app/heroes/hero-detail/hero-detail.component.ts (ngOnInit snapshot)

content\_copyngOnInit() {

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

this.hero$ = this.service.getHero(id);

}

snapshot only gets the initial value of the parameter map with this technique. Use the observable paramMap approach if there's a possibility that the router could re-use the component. This tutorial sample application uses with the observable paramMap.

### **Navigating back to the list component**

The HeroDetailComponent "Back" button uses the gotoHeroes() method that navigates imperatively back to the HeroListComponent.

The router navigate() method takes the same one-item link parameters array that you can bind to a [[routerLink](https://angular.io/api/router/RouterLink)] directive. It holds the path to the HeroListComponent:

src/app/heroes/hero-detail/hero-detail.component.ts (excerpt)

content\_copygotoHeroes() {

this.router.navigate(['/heroes']);

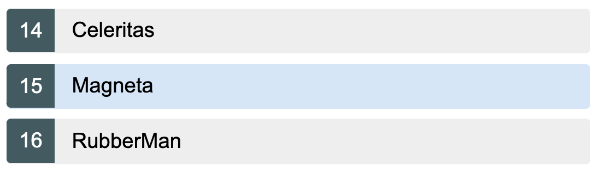
}

#### Route Parameters: Required or optional?

Use [route parameters](https://angular.io/guide/router-tutorial-toh#route-parameters) to specify a required parameter value within the route URL as you do when navigating to the HeroDetailComponent in order to view the hero with id 15:

content\_copylocalhost:4200/hero/15

You can also add optional information to a route request. For example, when returning to the hero-detail.component.ts list from the hero detail view, it would be nice if the viewed hero were preselected in the list.



You implement this feature by including the viewed hero's id in the URL as an optional parameter when returning from the HeroDetailComponent.

Optional information can also include other forms such as:

* Loosely structured search criteria; for example, name='wind\*'.
* Multiple values; for example, after='12/31/2015' & before='1/1/2017'—in no particular order—before='1/1/2017' & after='12/31/2015'— in a variety of formats—during='currentYear'.

As these kinds of parameters don't fit easily in a URL path, you can use optional parameters for conveying arbitrarily complex information during navigation. Optional parameters aren't involved in pattern matching and afford flexibility of expression.

The router supports navigation with optional parameters as well as required route parameters. Define optional parameters in a separate object after you define the required route parameters.

In general, use a required route parameter when the value is mandatory (for example, if necessary to distinguish one route path from another); and an optional parameter when the value is optional, complex, and/or multivariate.

#### Heroes list: optionally selecting a hero

When navigating to the HeroDetailComponent you specified the required id of the hero-to-edit in the route parameter and made it the second item of the [link parameters array](https://angular.io/guide/router-tutorial-toh#link-parameters-array).

src/app/heroes/hero-list/hero-list.component.html (link-parameters-array)

content\_copy<a [[routerLink](https://angular.io/api/router/RouterLink)]="['/hero', hero.id]">

The router embedded the id value in the navigation URL because you had defined it as a route parameter with an :id placeholder token in the route path:

src/app/heroes/heroes-routing.module.ts (hero-detail-route)

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

When the user clicks the back button, the HeroDetailComponent constructs another link parameters array which it uses to navigate back to the HeroListComponent.

src/app/heroes/hero-detail/hero-detail.component.ts (gotoHeroes)

content\_copygotoHeroes() {

this.router.navigate(['/heroes']);

}

This array lacks a route parameter because previously you didn't need to send information to the HeroListComponent.

Now, send the id of the current hero with the navigation request so that the HeroListComponent can highlight that hero in its list.

Send the id with an object that contains an optional id parameter. For demonstration purposes, there's an extra junk parameter (foo) in the object that the HeroListComponent should ignore. Here's the revised navigation statement:

src/app/heroes/hero-detail/hero-detail.component.ts (go to heroes)

content\_copygotoHeroes(hero: Hero) {

const heroId = hero ? hero.id : null;

// Pass along the hero id if available

// so that the HeroList component can select that hero.

// Include a junk 'foo' property for fun.

this.router.navigate(['/heroes', { id: heroId, foo: 'foo' }]);

}

The application still works. Clicking "back" returns to the hero list view.

Look at the browser address bar.

It should look something like this, depending on where you run it:

content\_copylocalhost:4200/heroes;id=15;foo=foo

The id value appears in the URL as (;id=15;foo=foo), not in the URL path. The path for the "Heroes" route doesn't have an :id token.

The optional route parameters are not separated by "?" and "&" as they would be in the URL query string. They are separated by semicolons ";". This is matrix URL notation.

Matrix URL notation is an idea first introduced in a [1996 proposal](https://www.w3.org/DesignIssues/MatrixURIs.html) by the founder of the web, Tim Berners-Lee.

Although matrix notation never made it into the HTML standard, it is legal and it became popular among browser routing systems as a way to isolate parameters belonging to parent and child routes. As such, the Router provides support for the matrix notation across browsers.

### **Route parameters in the**[**ActivatedRoute**](https://angular.io/api/router/ActivatedRoute)**service**

In its current state of development, the list of heroes is unchanged. No hero row is highlighted.

The HeroListComponent needs code that expects parameters.

Previously, when navigating from the HeroListComponent to the HeroDetailComponent, you subscribed to the route parameter map Observable and made it available to the HeroDetailComponent in the [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) service. You injected that service in the constructor of the HeroDetailComponent.

This time you'll be navigating in the opposite direction, from the HeroDetailComponent to the HeroListComponent.

First, extend the router import statement to include the [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) service symbol:

src/app/heroes/hero-list/hero-list.component.ts (import)

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

Import the switchMap operator to perform an operation on the Observable of route parameter map.

src/app/heroes/hero-list/hero-list.component.ts (rxjs imports)

content\_copyimport { Observable } from 'rxjs';

import { switchMap } from 'rxjs/operators';

Inject the [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) in the HeroListComponent constructor.

src/app/heroes/hero-list/hero-list.component.ts (constructor and ngOnInit)

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

heroes$!: Observable<Hero[]>;

selectedId = 0;

constructor(

private service: HeroService,

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

) {}

ngOnInit() {

this.heroes$ = this.route.paramMap.pipe(

switchMap(params => {

this.selectedId = parseInt(params.get('id')!, 10);

return this.service.getHeroes();

})

);

}

}

The [ActivatedRoute.paramMap](https://angular.io/api/router/ActivatedRoute#paramMap) property is an Observable map of route parameters. The paramMap emits a new map of values that includes id when the user navigates to the component. In ngOnInit() you subscribe to those values, set the selectedId, and get the heroes.

Update the template with a [class binding](https://angular.io/guide/attribute-binding#class-binding). The binding adds the selected CSS class when the comparison returns true and removes it when false. Look for it within the repeated <li> tag as shown here:

src/app/heroes/hero-list/hero-list.component.html

content\_copy<h2>Heroes</h2>

<ul class="heroes">

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

<a [[routerLink](https://angular.io/api/router/RouterLink)]="['/hero', hero.id]">

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

</a>

</li>

</ul>

<button [routerLink](https://angular.io/api/router/RouterLink)="/sidekicks">Go to sidekicks</button>

Add some styles to apply when the hero is selected.

src/app/heroes/hero-list/hero-list.component.css

content\_copy.heroes .selected a {

background-color: #d6e6f7;

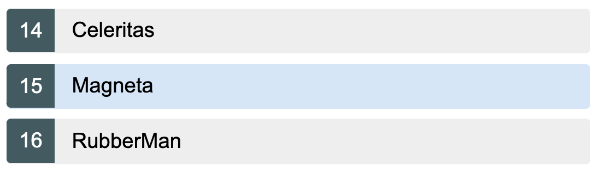
}

.heroes .selected a:hover {

background-color: #bdd7f5;

}

When the user navigates from the heroes list to the "Magneta" hero and back, "Magneta" appears selected:



The optional foo route parameter is harmless and the router continues to ignore it.

### **Adding routable animations**

This section shows you how to add some [animations](https://angular.io/guide/animations) to the HeroDetailComponent.

First, import the [BrowserAnimationsModule](https://angular.io/api/platform-browser/animations/BrowserAnimationsModule) and add it to the imports array:

src/app/app.module.ts (animations-module)

content\_copyimport { [BrowserAnimationsModule](https://angular.io/api/platform-browser/animations/BrowserAnimationsModule) } from '@angular/platform-browser/animations';

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

imports: [

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

],

})

Next, add a data object to the routes for HeroListComponent and HeroDetailComponent. Transitions are based on states and you use the [animation](https://angular.io/api/animations/animation) data from the route to provide a named animation state for the transitions.

src/app/heroes/heroes-routing.module.ts (animation data)

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

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

import { HeroListComponent } from './hero-list/hero-list.component';

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

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

{ path: 'heroes', component: HeroListComponent, data: { [animation](https://angular.io/api/animations/animation): 'heroes' } },

{ path: 'hero/:id', component: HeroDetailComponent, data: { [animation](https://angular.io/api/animations/animation): 'hero' } }

];

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

imports: [

RouterModule.forChild(heroesRoutes)

],

exports: [

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

]

})

export class HeroesRoutingModule { }

Create an animations.ts file in the root src/app/ folder. The contents look like this:

src/app/animations.ts (excerpt)

content\_copyimport {

[trigger](https://angular.io/api/animations/trigger), [animateChild](https://angular.io/api/animations/animateChild), group,

[transition](https://angular.io/api/animations/transition), [animate](https://angular.io/api/animations/animate), [style](https://angular.io/api/animations/style), [query](https://angular.io/api/animations/query)

} from '@angular/animations';

// Routable animations

export const slideInAnimation =

[trigger](https://angular.io/api/animations/trigger)('routeAnimation', [

[transition](https://angular.io/api/animations/transition)('heroes <=> hero', [

[style](https://angular.io/api/animations/style)({ position: 'relative' }),

[query](https://angular.io/api/animations/query)(':enter, :leave', [

[style](https://angular.io/api/animations/style)({

position: 'absolute',

top: 0,

left: 0,

width: '100%'

})

]),

[query](https://angular.io/api/animations/query)(':enter', [

[style](https://angular.io/api/animations/style)({ left: '-100%'})

]),

[query](https://angular.io/api/animations/query)(':leave', [animateChild](https://angular.io/api/animations/animateChild)()),

group([

[query](https://angular.io/api/animations/query)(':leave', [

[animate](https://angular.io/api/animations/animate)('300ms ease-out', [style](https://angular.io/api/animations/style)({ left: '100%'}))

]),

[query](https://angular.io/api/animations/query)(':enter', [

[animate](https://angular.io/api/animations/animate)('300ms ease-out', [style](https://angular.io/api/animations/style)({ left: '0%'}))

])

]),

[query](https://angular.io/api/animations/query)(':enter', [animateChild](https://angular.io/api/animations/animateChild)()),

])

]);

This file does the following:

* Imports the animation symbols that build the animation triggers, control state, and manage transitions between states.
* Exports a constant named slideInAnimation set to an animation trigger named routeAnimation.
* Defines one transition when switching back and forth from the heroes and hero routes to ease the component in from the left of the screen as it enters the application view (:enter), the other to animate the component to the right as it leaves the application view (:leave).

Back in the AppComponent, import the [RouterOutlet](https://angular.io/api/router/RouterOutlet) token from the @angular/router package and the slideInAnimation from './animations.ts.

Add an animations array to the @[Component](https://angular.io/api/core/Component) metadata that contains the slideInAnimation.

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

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

import { slideInAnimation } from './animations';

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

selector: 'app-root',

templateUrl: 'app.component.html',

styleUrls: ['app.component.css'],

animations: [ slideInAnimation ]

})

In order to use the routable animations, wrap the [RouterOutlet](https://angular.io/api/router/RouterOutlet) inside an element, use the @routeAnimation trigger, and bind it to the element.

For the @routeAnimation transitions to key off states, provide it with the data from the [ActivatedRoute](https://angular.io/api/router/ActivatedRoute). The [RouterOutlet](https://angular.io/api/router/RouterOutlet) is exposed as an outlet template variable, so you bind a reference to the router outlet. This example uses a variable of routerOutlet.

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

content\_copy<h1>Angular [Router](https://angular.io/api/router/Router)</h1>

<nav>

<a [routerLink](https://angular.io/api/router/RouterLink)="/crisis-center" [routerLinkActive](https://angular.io/api/router/RouterLinkActive)="active">Crisis Center</a>

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

</nav>

<div [@routeAnimation]="getAnimationData(routerOutlet)">

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

</div>

The @routeAnimation property is bound to the getAnimationData() with the provided routerOutlet reference, so the next step is to define that function in the AppComponent. The getAnimationData() function returns the animation property from the data provided through the [ActivatedRoute](https://angular.io/api/router/ActivatedRoute). The [animation](https://angular.io/api/animations/animation) property matches the [transition](https://angular.io/api/animations/transition) names you used in the slideInAnimation defined in animations.ts.

src/app/app.component.ts (router outlet)

content\_copyexport class AppComponent {

getAnimationData(outlet: [RouterOutlet](https://angular.io/api/router/RouterOutlet)) {

return outlet && outlet.activatedRouteData && outlet.activatedRouteData.animation;

}

}

When switching between the two routes, the HeroDetailComponent and HeroListComponent now ease in from the left when routed to and will slide to the right when navigating away.

### **Milestone 3 wrap up**

This section has covered the following:

* Organizing the application into feature areas.
* Navigating imperatively from one component to another.
* Passing information along in route parameters and subscribe to them in the component.
* Importing the feature area NgModule into the AppModule.
* Applying routable animations based on the page.

After these changes, the folder structure is as follows:

angular-router-sample

src

app

crisis-list

crisis-list.component.css

crisis-list.component.html

crisis-list.component.ts

heroes

hero-detail

hero-detail.component.css

hero-detail.component.html

hero-detail.component.ts

hero-list

hero-list.component.css

hero-list.component.html

hero-list.component.ts

hero.service.ts

hero.ts

heroes-routing.module.ts

heroes.module.ts

mock-heroes.ts

page-not-found

page-not-found.component.css

page-not-found.component.html

page-not-found.component.ts

animations.ts

app.component.css

app.component.html

app.component.ts

app.module.ts

app-routing.module.ts

main.ts

message.service.ts

index.html

styles.css

tsconfig.json

node\_modules ...

package.json

Here are the relevant files for this version of the sample application.

animations.ts

app.component.html

app.component.ts

app.module.ts

app-routing.module.ts

hero-list.component.css

hero-list.component.html

hero-list.component.ts

hero-detail.component.html

hero-detail.component.ts

hero.service.ts

heroes.module.ts

heroes-routing.module.ts

message.service.ts

content\_copyimport {

[trigger](https://angular.io/api/animations/trigger), [animateChild](https://angular.io/api/animations/animateChild), group,

[transition](https://angular.io/api/animations/transition), [animate](https://angular.io/api/animations/animate), [style](https://angular.io/api/animations/style), [query](https://angular.io/api/animations/query)

} from '@angular/animations';

// Routable animations

export const slideInAnimation =

[trigger](https://angular.io/api/animations/trigger)('routeAnimation', [

[transition](https://angular.io/api/animations/transition)('heroes <=> hero', [

[style](https://angular.io/api/animations/style)({ position: 'relative' }),

[query](https://angular.io/api/animations/query)(':enter, :leave', [

[style](https://angular.io/api/animations/style)({

position: 'absolute',

top: 0,

left: 0,

width: '100%'

})

]),

[query](https://angular.io/api/animations/query)(':enter', [

[style](https://angular.io/api/animations/style)({ left: '-100%'})

]),

[query](https://angular.io/api/animations/query)(':leave', [animateChild](https://angular.io/api/animations/animateChild)()),

group([

[query](https://angular.io/api/animations/query)(':leave', [

[animate](https://angular.io/api/animations/animate)('300ms ease-out', [style](https://angular.io/api/animations/style)({ left: '100%'}))

]),

[query](https://angular.io/api/animations/query)(':enter', [

[animate](https://angular.io/api/animations/animate)('300ms ease-out', [style](https://angular.io/api/animations/style)({ left: '0%'}))

])

]),

[query](https://angular.io/api/animations/query)(':enter', [animateChild](https://angular.io/api/animations/animateChild)()),

])

]);

## **Milestone 4: Crisis center feature**

This section shows you how to add child routes and use relative routing in your app.

To add more features to the application's current crisis center, take similar steps as for the heroes feature:

* Create a crisis-center subfolder in the src/app folder.
* Copy the files and folders from app/heroes into the new crisis-center folder.
* In the new files, change every mention of "hero" to "crisis", and "heroes" to "crises".
* Rename the NgModule files to crisis-center.module.ts and crisis-center-routing.module.ts.

Use mock crises instead of mock heroes:

src/app/crisis-center/mock-crises.ts

content\_copyimport { Crisis } from './crisis';

export const CRISES: Crisis[] = [

{ id: 1, name: 'Dragon Burning Cities' },

{ id: 2, name: 'Sky Rains Great White Sharks' },

{ id: 3, name: 'Giant Asteroid Heading For Earth' },

{ id: 4, name: 'Procrastinators Meeting Delayed Again' },

];

The resulting crisis center is a foundation for introducing a new concept—child routing. You can leave Heroes in its current state as a contrast with the Crisis Center.

In keeping with the [Separation of Concerns principle](https://blog.8thlight.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html), changes to the Crisis Center don't affect the AppModule or any other feature's component.

### **A crisis center with child routes**

This section shows you how to organize the crisis center to conform to the following recommended pattern for Angular applications:

* Each feature area resides in its own folder.
* Each feature has its own Angular feature module.
* Each area has its own area root component.
* Each area root component has its own router outlet and child routes.
* Feature area routes rarely (if ever) cross with routes of other features.

If your application had many feature areas, the component trees might consist of multiple components for those features, each with branches of other, related, components.

### **Child routing component**

Generate a CrisisCenter component in the crisis-center folder:

content\_copyng generate component crisis-center/crisis-center

Update the component template with the following markup:

src/app/crisis-center/crisis-center/crisis-center.component.html

content\_copy<h2>Crisis Center</h2>

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

The CrisisCenterComponent has the following in common with the AppComponent:

* It is the root of the crisis center area, just as AppComponent is the root of the entire application.
* It is a shell for the crisis management feature area, just as the AppComponent is a shell to manage the high-level workflow.

Like most shells, the CrisisCenterComponent class is minimal because it has no business logic, and its template has no links, just a title and <[router-outlet](https://angular.io/api/router/RouterOutlet)> for the crisis center child component.

### **Child route configuration**

As a host page for the "Crisis Center" feature, generate a CrisisCenterHome component in the crisis-center folder.

content\_copyng generate component crisis-center/crisis-center-home

Update the template with a welcome message to the Crisis Center.

src/app/crisis-center/crisis-center-home/crisis-center-home.component.html

content\_copy<h3>Welcome to the Crisis Center</h3>

Update the crisis-center-routing.module.ts you renamed after copying it from heroes-routing.module.ts file. This time, you define child routes within the parent crisis-center route.

src/app/crisis-center/crisis-center-routing.module.ts (Routes)

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

{

path: 'crisis-center',

component: CrisisCenterComponent,

children: [

{

path: '',

component: CrisisListComponent,

children: [

{

path: ':id',

component: CrisisDetailComponent

},

{

path: '',

component: CrisisCenterHomeComponent

}

]

}

]

}

];

Notice that the parent crisis-center route has a children property with a single route containing the CrisisListComponent. The CrisisListComponent route also has a children array with two routes.

These two routes navigate to the crisis center child components, CrisisCenterHomeComponent and CrisisDetailComponent, respectively.

There are important differences in the way the router treats child routes.

The router displays the components of these routes in the [RouterOutlet](https://angular.io/api/router/RouterOutlet) of the CrisisCenterComponent, not in the [RouterOutlet](https://angular.io/api/router/RouterOutlet) of the AppComponent shell.

The CrisisListComponent contains the crisis list and a [RouterOutlet](https://angular.io/api/router/RouterOutlet) to display the Crisis Center Home and Crisis Detail route components.

The Crisis Detail route is a child of the Crisis List. The router [reuses components](https://angular.io/guide/router-tutorial-toh#reuse) by default, so the Crisis Detail component will be re-used as you select different crises. In contrast, back in the Hero Detail route, [the component was recreated](https://angular.io/guide/router-tutorial-toh#snapshot-the-no-observable-alternative) each time you selected a different hero from the list of heroes.

At the top level, paths that begin with / refer to the root of the application. But child routes extend the path of the parent route. With each step down the route tree, you add a slash followed by the route path, unless the path is empty.

Apply that logic to navigation within the crisis center for which the parent path is /crisis-center.

* To navigate to the CrisisCenterHomeComponent, the full URL is /crisis-center (/crisis-center + '' + '').
* To navigate to the CrisisDetailComponent for a crisis with id=2, the full URL is /crisis-center/2 (/crisis-center + '' + '/2').

The absolute URL for the latter example, including the localhost origin, is as follows:

content\_copylocalhost:4200/crisis-center/2

Here's the complete crisis-center-routing.module.ts file with its imports.

src/app/crisis-center/crisis-center-routing.module.ts (excerpt)

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

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

import { CrisisCenterHomeComponent } from './crisis-center-home/crisis-center-home.component';

import { CrisisListComponent } from './crisis-list/crisis-list.component';

import { CrisisCenterComponent } from './crisis-center/crisis-center.component';

import { CrisisDetailComponent } from './crisis-detail/crisis-detail.component';

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

{

path: 'crisis-center',

component: CrisisCenterComponent,

children: [

{

path: '',

component: CrisisListComponent,

children: [

{

path: ':id',

component: CrisisDetailComponent

},

{

path: '',

component: CrisisCenterHomeComponent

}

]

}

]

}

];

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

imports: [

RouterModule.forChild(crisisCenterRoutes)

],

exports: [

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

]

})

export class CrisisCenterRoutingModule { }

### **Import crisis center module into the AppModule routes**

As with the HeroesModule, you must add the CrisisCenterModule to the imports array of the AppModule before the AppRoutingModule:

src/app/crisis-center/crisis-center.module.ts

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

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

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

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

import { CrisisCenterHomeComponent } from './crisis-center-home/crisis-center-home.component';

import { CrisisListComponent } from './crisis-list/crisis-list.component';

import { CrisisCenterComponent } from './crisis-center/crisis-center.component';

import { CrisisDetailComponent } from './crisis-detail/crisis-detail.component';

import { CrisisCenterRoutingModule } from './crisis-center-routing.module';

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

imports: [

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

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

CrisisCenterRoutingModule

],

declarations: [

CrisisCenterComponent,

CrisisListComponent,

CrisisCenterHomeComponent,

CrisisDetailComponent

]

})

export class CrisisCenterModule {}

The import order of the modules is important because the order of the routes defined in the modules affects route matching. If the `AppModule` were imported first, its wildcard route (`path: '\*\*'`) would take precedence over the routes defined in `CrisisCenterModule`. For more information, see the section on [route order](guide/router#route-order).

Remove the initial crisis center route from the app-routing.module.ts because now the HeroesModule and the CrisisCenter modules provide the feature routes.

The app-routing.module.ts file retains the top-level application routes such as the default and wildcard routes.

src/app/app-routing.module.ts (v3)

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

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

import { PageNotFoundComponent } from './page-not-found/page-not-found.component';

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

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

{ path: '\*\*', component: PageNotFoundComponent }

];

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

imports: [

RouterModule.forRoot(

appRoutes,

{ enableTracing: true } // <-- debugging purposes only

)

],

exports: [

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

]

})

export class AppRoutingModule {}

### **Relative navigation**

While building out the crisis center feature, you navigated to the crisis detail route using an absolute path that begins with a slash.

The router matches such absolute paths to routes starting from the top of the route configuration.

You could continue to use absolute paths like this to navigate inside the Crisis Center feature, but that pins the links to the parent routing structure. If you changed the parent /crisis-center path, you would have to change the link parameters array.

You can free the links from this dependency by defining paths that are relative to the current URL segment. Navigation within the feature area remains intact even if you change the parent route path to the feature.

The router supports directory-like syntax in a link parameters list to help guide route name lookup:

./ or no leading slash is relative to the current level.

../ to go up one level in the route path.

You can combine relative navigation syntax with an ancestor path. If you must navigate to a sibling route, you could use the ../<sibling> convention to go up one level, then over and down the sibling route path.

To navigate a relative path with the Router.navigate method, you must supply the [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) to give the router knowledge of where you are in the current route tree.

After the link parameters array, add an object with a relativeTo property set to the [ActivatedRoute](https://angular.io/api/router/ActivatedRoute). The router then calculates the target URL based on the active route's location.

Always specify the complete absolute path when calling router's navigateByUrl() method.

### **Navigate to crisis list with a relative URL**

You've already injected the [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) that you need to compose the relative navigation path.

When using a [RouterLink](https://angular.io/api/router/RouterLink) to navigate instead of the [Router](https://angular.io/api/router/Router) service, you'd use the same link parameters array, but you wouldn't provide the object with the relativeTo property. The [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) is implicit in a [RouterLink](https://angular.io/api/router/RouterLink) directive.

Update the gotoCrises() method of the CrisisDetailComponent to navigate back to the Crisis Center list using relative path navigation.

src/app/crisis-center/crisis-detail/crisis-detail.component.ts (relative navigation)

content\_copy// Relative navigation back to the crises

this.router.navigate(['../', { id: crisisId, foo: 'foo' }], { relativeTo: this.route });

Notice that the path goes up a level using the ../ syntax. If the current crisis id is 3, the resulting path back to the crisis list is /crisis-center/;id=3;foo=foo.

### **Displaying multiple routes in named outlets**

You decide to give users a way to contact the crisis center. When a user clicks a "Contact" button, you want to display a message in a popup view.

The popup should stay open, even when switching between pages in the application, until the user closes it by sending the message or canceling. Clearly you can't put the popup in the same outlet as the other pages.

Until now, you've defined a single outlet and you've nested child routes under that outlet to group routes together. The router only supports one primary unnamed outlet per template.

A template can also have any number of named outlets. Each named outlet has its own set of routes with their own components. Multiple outlets can display different content, determined by different routes, all at the same time.

Add an outlet named "popup" in the AppComponent, directly below the unnamed outlet.

src/app/app.component.html (outlets)

content\_copy<div [@routeAnimation]="getAnimationData(routerOutlet)">

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

</div>

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

That's where a popup will go, once you learn how to route a popup component to it.

#### Secondary routes

Named outlets are the targets of secondary routes.

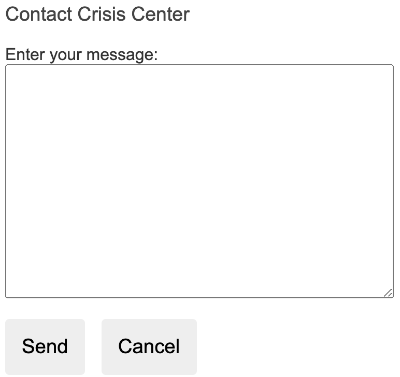
Secondary routes look like primary routes and you configure them the same way. They differ in a few key respects.

* They are independent of each other.
* They work in combination with other routes.
* They are displayed in named outlets.

Generate a new component to compose the message.

content\_copyng generate component compose-message

It displays a short form with a header, an input box for the message, and two buttons, "Send" and "Cancel".



Here's the component, its template, and styles:

src/app/compose-message/compose-message.component.html

src/app/compose-message/compose-message.component.ts

src/app/compose-message/compose-message.component.css

content\_copy<h3>Contact Crisis Center</h3>

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

{{ details }}

</div>

<div>

<div>

<label for="message">Enter your message: </label>

</div>

<div>

<[textarea](https://angular.io/api/forms/DefaultValueAccessor) id="message" [([ngModel](https://angular.io/api/forms/NgModel))]="message" rows="10" cols="35" [disabled]="sending"></[textarea](https://angular.io/api/forms/DefaultValueAccessor)>

</div>

</div>

<p \*[ngIf](https://angular.io/api/common/NgIf)="!sending">

<button (click)="send()">Send</button>

<button (click)="cancel()">Cancel</button>

</p>

It looks similar to any other component in this guide, but there are two key differences.

Note that the send() method simulates latency by waiting a second before "sending" the message and closing the popup.

The closePopup() method closes the popup view by navigating to the popup outlet with a null which the section on [clearing secondary routes](https://angular.io/guide/router-tutorial-toh#clear-secondary-routes) covers.

#### Add a secondary route

Open the AppRoutingModule and add a new compose route to the appRoutes.

src/app/app-routing.module.ts (compose route)

content\_copy{

path: 'compose',

component: ComposeMessageComponent,

outlet: 'popup'

},

In addition to the path and component properties, there's a new property called outlet, which is set to 'popup'. This route now targets the popup outlet and the ComposeMessageComponent will display there.

To give users a way to open the popup, add a "Contact" link to the AppComponent template.

src/app/app.component.html (contact-link)

content\_copy<a [[routerLink](https://angular.io/api/router/RouterLink)]="[{ outlets: { popup: ['compose'] } }]">Contact</a>

Although the compose route is configured to the "popup" outlet, that's not sufficient for connecting the route to a [RouterLink](https://angular.io/api/router/RouterLink) directive. You have to specify the named outlet in a link parameters array and bind it to the [RouterLink](https://angular.io/api/router/RouterLink) with a property binding.

The link parameters array contains an object with a single outlets property whose value is another object keyed by one (or more) outlet names. In this case there is only the "popup" outlet property and its value is another link parameters array that specifies the compose route.

In other words, when the user clicks this link, the router displays the component associated with the compose route in the popup outlet.

This outlets object within an outer object was unnecessary when there was only one route and one unnamed outlet.

The router assumed that your route specification targeted the unnamed primary outlet and created these objects for you.

Routing to a named outlet has revealed a router feature: you can target multiple outlets with multiple routes in the same [RouterLink](https://angular.io/api/router/RouterLink) directive.

#### Secondary route navigation: merging routes during navigation

Navigate to the Crisis Center and click "Contact". you should see something like the following URL in the browser address bar.

content\_copyhttp://.../crisis-center(popup:compose)

The relevant part of the URL follows the ...:

* The crisis-center is the primary navigation.
* Parentheses surround the secondary route.
* The secondary route consists of an outlet name (popup), a colon separator, and the secondary route path (compose).

Click the Heroes link and look at the URL again.

content\_copyhttp://.../heroes(popup:compose)

The primary navigation part has changed; the secondary route is the same.

The router is keeping track of two separate branches in a navigation tree and generating a representation of that tree in the URL.

You can add many more outlets and routes, at the top level and in nested levels, creating a navigation tree with many branches and the router will generate the URLs to go with it.

You can tell the router to navigate an entire tree at once by filling out the outlets object and then pass that object inside a link parameters array to the router.navigate method.

#### Clearing secondary routes

Like regular outlets, secondary outlets persists until you navigate away to a new component.

Each secondary outlet has its own navigation, independent of the navigation driving the primary outlet. Changing a current route that displays in the primary outlet has no effect on the popup outlet. That's why the popup stays visible as you navigate among the crises and heroes.

The closePopup() method again:

src/app/compose-message/compose-message.component.ts (closePopup)

content\_copyclosePopup() {

// Providing a `null` value to the named outlet

// clears the contents of the named outlet

this.router.navigate([{ outlets: { popup: null }}]);

}

Clicking the "send" or "cancel" buttons clears the popup view. The closePopup() function navigates imperatively with the [Router.navigate()](https://angular.io/api/router/Router#navigate) method, passing in a [link parameters array](https://angular.io/guide/router-tutorial-toh#link-parameters-array).

Like the array bound to the Contact [RouterLink](https://angular.io/api/router/RouterLink) in the AppComponent, this one includes an object with an outlets property. The outlets property value is another object with outlet names for keys. The only named outlet is 'popup'.

This time, the value of 'popup' is null. That's not a route, but it is a legitimate value. Setting the popup [RouterOutlet](https://angular.io/api/router/RouterOutlet) to null clears the outlet and removes the secondary popup route from the current URL.

## **Milestone 5: Route guards**

At the moment, any user can navigate anywhere in the application any time, but sometimes you need to control access to different parts of your application for various reasons. Some of which may include the following:

* Perhaps the user is not authorized to navigate to the target component.
* Maybe the user must login (authenticate) first.
* Maybe you should fetch some data before you display the target component.
* You might want to save pending changes before leaving a component.
* You might ask the user if it's OK to discard pending changes rather than save them.

You add guards to the route configuration to handle these scenarios.

A guard's return value controls the router's behavior:

* If it returns true, the navigation process continues.
* If it returns false, the navigation process stops and the user stays put.
* If it returns a [UrlTree](https://angular.io/api/router/UrlTree), the current navigation cancels and a new navigation is initiated to the [UrlTree](https://angular.io/api/router/UrlTree) returned.

**Note:** The guard can also tell the router to navigate elsewhere, effectively canceling the current navigation. When doing so inside a guard, the guard should return false;

The guard might return its boolean answer synchronously. But in many cases, the guard can't produce an answer synchronously. The guard could ask the user a question, save changes to the server, or fetch fresh data. These are all asynchronous operations.

Accordingly, a routing guard can return an Observable<boolean> or a Promise<boolean> and the router will wait for the observable to resolve to true or false.

**Note:** The observable provided to the [Router](https://angular.io/api/router/Router) must also complete. If the observable does not complete, the navigation does not continue.

The router supports multiple guard interfaces:

* [CanActivate](https://angular.io/api/router/CanActivate) to mediate navigation to a route.
* [CanActivateChild](https://angular.io/api/router/CanActivateChild) to mediate navigation to a child route.
* [CanDeactivate](https://angular.io/api/router/CanDeactivate) to mediate navigation away from the current route.
* [Resolve](https://angular.io/api/router/Resolve) to perform route data retrieval before route activation.
* [CanLoad](https://angular.io/api/router/CanLoad) to mediate navigation to a feature module loaded asynchronously.

You can have multiple guards at every level of a routing hierarchy. The router checks the [CanDeactivate](https://angular.io/api/router/CanDeactivate) guards first, from the deepest child route to the top. Then it checks the [CanActivate](https://angular.io/api/router/CanActivate) and [CanActivateChild](https://angular.io/api/router/CanActivateChild) guards from the top down to the deepest child route. If the feature module is loaded asynchronously, the [CanLoad](https://angular.io/api/router/CanLoad) guard is checked before the module is loaded. If any guard returns false, pending guards that have not completed will be canceled, and the entire navigation is canceled.

There are several examples over the next few sections.

### [**CanActivate**](https://angular.io/api/router/CanActivate)**: requiring authentication**

Applications often restrict access to a feature area based on who the user is. You could permit access only to authenticated users or to users with a specific role. You might block or limit access until the user's account is activated.

The [CanActivate](https://angular.io/api/router/CanActivate) guard is the tool to manage these navigation business rules.

#### Add an admin feature module

This section guides you through extending the crisis center with some new administrative features. Start by adding a new feature module named AdminModule.

Generate an admin folder with a feature module file and a routing configuration file.

content\_copyng generate module admin --routing

Next, generate the supporting components.

content\_copyng generate component admin/admin-dashboard

content\_copyng generate component admin/admin

content\_copyng generate component admin/manage-crises

content\_copyng generate component admin/manage-heroes

The admin feature file structure looks like this:

src/app/admin

admin

admin.component.css

admin.component.html

admin.component.ts

admin-dashboard

admin-dashboard.component.css

admin-dashboard.component.html

admin-dashboard.component.ts

manage-crises

manage-crises.component.css

manage-crises.component.html

manage-crises.component.ts

manage-heroes

manage-heroes.component.css

manage-heroes.component.html

manage-heroes.component.ts

admin.module.ts

admin-routing.module.ts

The admin feature module contains the AdminComponent used for routing within the feature module, a dashboard route and two unfinished components to manage crises and heroes.

src/app/admin/admin/admin.component.html

src/app/admin/admin-dashboard/admin-dashboard.component.html

src/app/admin/admin.module.ts

src/app/admin/manage-crises/manage-crises.component.html

src/app/admin/manage-heroes/manage-heroes.component.html

content\_copy<h2>Admin</h2>

<nav>

<a [routerLink](https://angular.io/api/router/RouterLink)="./" [routerLinkActive](https://angular.io/api/router/RouterLinkActive)="active"

[routerLinkActiveOptions]="{ exact: true }">Dashboard</a>

<a [routerLink](https://angular.io/api/router/RouterLink)="./crises" [routerLinkActive](https://angular.io/api/router/RouterLinkActive)="active">Manage Crises</a>

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

</nav>

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

Although the admin dashboard [RouterLink](https://angular.io/api/router/RouterLink) only contains a relative slash without an additional URL segment, it is a match to any route within the admin feature area. You only want the Dashboard link to be active when the user visits that route. Adding an additional binding to the Dashboard routerLink,[routerLinkActiveOptions]="{ exact: true }", marks the ./ link as active when the user navigates to the /admin URL and not when navigating to any of the child routes.

##### Component-less route: grouping routes without a component

The initial admin routing configuration:

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

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

{

path: 'admin',

component: AdminComponent,

children: [

{

path: '',

children: [

{ path: 'crises', component: ManageCrisesComponent },

{ path: 'heroes', component: ManageHeroesComponent },

{ path: '', component: AdminDashboardComponent }

]

}

]

}

];

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

imports: [

RouterModule.forChild(adminRoutes)

],

exports: [

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

]

})

export class AdminRoutingModule {}

The child route under the AdminComponent has a path and a children property but it's not using a component. This defines a component-less route.

To group the Crisis Center management routes under the admin path a component is unnecessary. Additionally, a component-less route makes it easier to [guard child routes](https://angular.io/guide/router-tutorial-toh#can-activate-child-guard).

Next, import the AdminModule into app.module.ts and add it to the imports array to register the admin routes.

src/app/app.module.ts (admin module)

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

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

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

import { AppComponent } from './app.component';

import { PageNotFoundComponent } from './page-not-found/page-not-found.component';

import { ComposeMessageComponent } from './compose-message/compose-message.component';

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

import { HeroesModule } from './heroes/heroes.module';

import { CrisisCenterModule } from './crisis-center/crisis-center.module';

import { AdminModule } from './admin/admin.module';

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

imports: [

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

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

HeroesModule,

CrisisCenterModule,

AdminModule,

AppRoutingModule

],

declarations: [

AppComponent,

ComposeMessageComponent,

PageNotFoundComponent

],

bootstrap: [ AppComponent ]

})

export class AppModule { }

Add an "Admin" link to the AppComponent shell so that users can get to this feature.

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

content\_copy<h1 class="title">Angular [Router](https://angular.io/api/router/Router)</h1>

<nav>

<a [routerLink](https://angular.io/api/router/RouterLink)="/crisis-center" [routerLinkActive](https://angular.io/api/router/RouterLinkActive)="active">Crisis Center</a>

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

<a [routerLink](https://angular.io/api/router/RouterLink)="/admin" [routerLinkActive](https://angular.io/api/router/RouterLinkActive)="active">Admin</a>

<a [[routerLink](https://angular.io/api/router/RouterLink)]="[{ outlets: { popup: ['compose'] } }]">Contact</a>

</nav>

<div [@routeAnimation]="getAnimationData(routerOutlet)">

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

</div>

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

#### Guard the admin feature

Currently, every route within the Crisis Center is open to everyone. The new admin feature should be accessible only to authenticated users.

Write a canActivate() guard method to redirect anonymous users to the login page when they try to enter the admin area.

Generate an AuthGuard in the auth folder.

content\_copyng generate guard auth/auth

To demonstrate the fundamentals, this example only logs to the console, returns true immediately, and allows navigation to proceed:

src/app/auth/auth.guard.ts (excerpt)

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

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

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

providedIn: 'root',

})

export class AuthGuard implements [CanActivate](https://angular.io/api/router/CanActivate) {

canActivate(

next: [ActivatedRouteSnapshot](https://angular.io/api/router/ActivatedRouteSnapshot),

state: [RouterStateSnapshot](https://angular.io/api/router/RouterStateSnapshot)): boolean {

console.log('AuthGuard#canActivate called');

return true;

}

}

Next, open admin-routing.module.ts , import the AuthGuard class, and update the admin route with a canActivate guard property that references it:

src/app/admin/admin-routing.module.ts (guarded admin route)

content\_copyimport { AuthGuard } from '../auth/auth.guard';

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

{

path: 'admin',

component: AdminComponent,

canActivate: [AuthGuard],

children: [

{

path: '',

children: [

{ path: 'crises', component: ManageCrisesComponent },

{ path: 'heroes', component: ManageHeroesComponent },

{ path: '', component: AdminDashboardComponent }

],

}

]

}

];

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

imports: [

RouterModule.forChild(adminRoutes)

],

exports: [

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

]

})

export class AdminRoutingModule {}

The admin feature is now protected by the guard, but the guard requires more customization to work fully.

#### Authenticate with AuthGuard

Make the AuthGuard mimic authentication.

The AuthGuard should call an application service that can login a user and retain information about the current user. Generate a new AuthService in the auth folder:

content\_copyng generate service auth/auth

Update the AuthService to log in the user:

src/app/auth/auth.service.ts (excerpt)

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

import { Observable, of } from 'rxjs';

import { tap, delay } from 'rxjs/operators';

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

providedIn: 'root',

})

export class AuthService {

isLoggedIn = false;

// store the URL so we can redirect after logging in

redirectUrl: string | null = null;

login(): Observable<boolean> {

return of(true).pipe(

delay(1000),

tap(() => this.isLoggedIn = true)

);

}

logout(): void {

this.isLoggedIn = false;

}

}

Although it doesn't actually log in, it has an isLoggedIn flag to tell you whether the user is authenticated. Its login() method simulates an API call to an external service by returning an observable that resolves successfully after a short pause. The redirectUrl property stores the URL that the user wanted to access so you can navigate to it after authentication.

To keep things minimal, this example redirects unauthenticated users to /admin.

Revise the AuthGuard to call the AuthService.

src/app/auth/auth.guard.ts (v2)

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

import { [CanActivate](https://angular.io/api/router/CanActivate), [ActivatedRouteSnapshot](https://angular.io/api/router/ActivatedRouteSnapshot), [RouterStateSnapshot](https://angular.io/api/router/RouterStateSnapshot), [Router](https://angular.io/api/router/Router), [UrlTree](https://angular.io/api/router/UrlTree) } from '@angular/router';

import { AuthService } from './auth.service';

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

providedIn: 'root',

})

export class AuthGuard implements [CanActivate](https://angular.io/api/router/CanActivate) {

constructor(private authService: AuthService, private router: [Router](https://angular.io/api/router/Router)) {}

canActivate(

next: [ActivatedRouteSnapshot](https://angular.io/api/router/ActivatedRouteSnapshot),

state: [RouterStateSnapshot](https://angular.io/api/router/RouterStateSnapshot)): true|[UrlTree](https://angular.io/api/router/UrlTree) {

const url: string = state.url;

return this.checkLogin(url);

}

checkLogin(url: string): true|[UrlTree](https://angular.io/api/router/UrlTree) {

if (this.authService.isLoggedIn) { return true; }

// Store the attempted URL for redirecting

this.authService.redirectUrl = url;

// Redirect to the login page

return this.router.parseUrl('/login');

}

}

Notice that you inject the AuthService and the [Router](https://angular.io/api/router/Router) in the constructor. You haven't provided the AuthService yet but it's good to know that you can inject helpful services into routing guards.

This guard returns a synchronous boolean result. If the user is logged in, it returns true and the navigation continues.

The [ActivatedRouteSnapshot](https://angular.io/api/router/ActivatedRouteSnapshot) contains the future route that will be activated and the [RouterStateSnapshot](https://angular.io/api/router/RouterStateSnapshot) contains the future [RouterState](https://angular.io/api/router/RouterState) of the application, should you pass through the guard check.

If the user is not logged in, you store the attempted URL the user came from using the [RouterStateSnapshot.url](https://angular.io/api/router/RouterStateSnapshot#url) and tell the router to redirect to a login page—a page you haven't created yet. Returning a [UrlTree](https://angular.io/api/router/UrlTree) tells the [Router](https://angular.io/api/router/Router) to cancel the current navigation and schedule a new one to redirect the user.

#### Add the LoginComponent

You need a LoginComponent for the user to log in to the application. After logging in, you'll redirect to the stored URL if available, or use the default URL. There is nothing new about this component or the way you use it in the router configuration.

content\_copyng generate component auth/login

Register a /login route in the auth/auth-routing.module.ts. In app.module.ts, import and add the AuthModule to the AppModule imports.

src/app/app.module.ts

src/app/auth/login/login.component.html

src/app/auth/login/login.component.ts

src/app/auth/auth.module.ts

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

import { [BrowserModule](https://angular.io/api/platform-browser/BrowserModule) } from '@angular/platform-browser';

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

import { [BrowserAnimationsModule](https://angular.io/api/platform-browser/animations/BrowserAnimationsModule) } from '@angular/platform-browser/animations';

import { AppComponent } from './app.component';

import { PageNotFoundComponent } from './page-not-found/page-not-found.component';

import { ComposeMessageComponent } from './compose-message/compose-message.component';

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

import { HeroesModule } from './heroes/heroes.module';

import { AuthModule } from './auth/auth.module';

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

imports: [

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

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

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

HeroesModule,

AuthModule,

AppRoutingModule,

],

declarations: [

AppComponent,

ComposeMessageComponent,

PageNotFoundComponent

],

bootstrap: [ AppComponent ]

})

export class AppModule {

}

### [**CanActivateChild**](https://angular.io/api/router/CanActivateChild)**: guarding child routes**

You can also protect child routes with the [CanActivateChild](https://angular.io/api/router/CanActivateChild) guard. The [CanActivateChild](https://angular.io/api/router/CanActivateChild) guard is similar to the [CanActivate](https://angular.io/api/router/CanActivate) guard. The key difference is that it runs before any child route is activated.

You protected the admin feature module from unauthorized access. You should also protect child routes within the feature module.

Extend the AuthGuard to protect when navigating between the admin routes. Open auth.guard.ts and add the [CanActivateChild](https://angular.io/api/router/CanActivateChild) interface to the imported tokens from the router package.

Next, implement the canActivateChild() method which takes the same arguments as the canActivate() method: an [ActivatedRouteSnapshot](https://angular.io/api/router/ActivatedRouteSnapshot) and [RouterStateSnapshot](https://angular.io/api/router/RouterStateSnapshot). The canActivateChild() method can return an Observable<boolean|[UrlTree](https://angular.io/api/router/UrlTree)> or Promise<boolean|[UrlTree](https://angular.io/api/router/UrlTree)> for async checks and a boolean or [UrlTree](https://angular.io/api/router/UrlTree) for sync checks. This one returns either true to allow the user to access the admin feature module or [UrlTree](https://angular.io/api/router/UrlTree) to redirect the user to the login page instead:

src/app/auth/auth.guard.ts (excerpt)

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

import {

[CanActivate](https://angular.io/api/router/CanActivate), [Router](https://angular.io/api/router/Router),

[ActivatedRouteSnapshot](https://angular.io/api/router/ActivatedRouteSnapshot),

[RouterStateSnapshot](https://angular.io/api/router/RouterStateSnapshot),

[CanActivateChild](https://angular.io/api/router/CanActivateChild),

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

} from '@angular/router';

import { AuthService } from './auth.service';

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

providedIn: 'root',

})

export class AuthGuard implements [CanActivate](https://angular.io/api/router/CanActivate), [CanActivateChild](https://angular.io/api/router/CanActivateChild) {

constructor(private authService: AuthService, private router: [Router](https://angular.io/api/router/Router)) {}

canActivate(

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

state: [RouterStateSnapshot](https://angular.io/api/router/RouterStateSnapshot)): true|[UrlTree](https://angular.io/api/router/UrlTree) {

const url: string = state.url;

return this.checkLogin(url);

}

canActivateChild(

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

state: [RouterStateSnapshot](https://angular.io/api/router/RouterStateSnapshot)): true|[UrlTree](https://angular.io/api/router/UrlTree) {

return this.canActivate(route, state);

}

/\* . . . \*/

}

Add the same AuthGuard to the component-less admin route to protect all other child routes at one time instead of adding the AuthGuard to each route individually.

src/app/admin/admin-routing.module.ts (excerpt)

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

{

path: 'admin',

component: AdminComponent,

canActivate: [AuthGuard],

children: [

{

path: '',

canActivateChild: [AuthGuard],

children: [

{ path: 'crises', component: ManageCrisesComponent },

{ path: 'heroes', component: ManageHeroesComponent },

{ path: '', component: AdminDashboardComponent }

]

}

]

}

];

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

imports: [

RouterModule.forChild(adminRoutes)

],

exports: [

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

]

})

export class AdminRoutingModule {}

### [**CanDeactivate**](https://angular.io/api/router/CanDeactivate)**: handling unsaved changes**

Back in the "Heroes" workflow, the application accepts every change to a hero immediately without validation.

In the real world, you might have to accumulate the users changes, validate across fields, validate on the server, or hold changes in a pending state until the user confirms them as a group or cancels and reverts all changes.

When the user navigates away, you can let the user decide what to do with unsaved changes. If the user cancels, you'll stay put and allow more changes. If the user approves, the application can save.

You still might delay navigation until the save succeeds. If you let the user move to the next screen immediately and saving were to fail (perhaps the data is ruled invalid), you would lose the context of the error.

You need to stop the navigation while you wait, asynchronously, for the server to return with its answer.

The [CanDeactivate](https://angular.io/api/router/CanDeactivate) guard helps you decide what to do with unsaved changes and how to proceed.

#### Cancel and save

Users update crisis information in the CrisisDetailComponent. Unlike the HeroDetailComponent, the user changes do not update the crisis entity immediately. Instead, the application updates the entity when the user presses the Save button and discards the changes when the user presses the Cancel button.

Both buttons navigate back to the crisis list after save or cancel.

src/app/crisis-center/crisis-detail/crisis-detail.component.ts (cancel and save methods)

content\_copycancel() {

this.gotoCrises();

}

save() {

this.crisis.name = this.editName;

this.gotoCrises();

}

In this scenario, the user could click the heroes link, cancel, push the browser back button, or navigate away without saving.

This example application asks the user to be explicit with a confirmation dialog box that waits asynchronously for the user's response.

You could wait for the user's answer with synchronous, blocking code, however, the application is more responsive—and can do other work—by waiting for the user's answer asynchronously.

Generate a Dialog service to handle user confirmation.

content\_copyng generate service dialog

Add a confirm() method to the DialogService to prompt the user to confirm their intent. The window.confirm is a blocking action that displays a modal dialog and waits for user interaction.

src/app/dialog.service.ts

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

import { Observable, of } from 'rxjs';

/\*\*

\* Async modal dialog service

\* DialogService makes this app easier to test by faking this service.

\* TODO: better modal implementation that doesn't use window.confirm

\*/

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

providedIn: 'root',

})

export class DialogService {

/\*\*

\* Ask user to confirm an action. `message` explains the action and choices.

\* Returns observable resolving to `true`=confirm or `false`=cancel

\*/

confirm(message?: string): Observable<boolean> {

const confirmation = window.confirm(message || 'Is it OK?');

return of(confirmation);

}

}

It returns an Observable that resolves when the user eventually decides what to do: either to discard changes and navigate away (true) or to preserve the pending changes and stay in the crisis editor (false).

Generate a guard that checks for the presence of a canDeactivate() method in a component—any component.

content\_copyng generate guard can-deactivate

Paste the following code into your guard.

src/app/can-deactivate.guard.ts

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

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

import { Observable } from 'rxjs';

export interface CanComponentDeactivate {

canDeactivate: () => Observable<boolean> | Promise<boolean> | boolean;

}

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

providedIn: 'root',

})

export class CanDeactivateGuard implements [CanDeactivate](https://angular.io/api/router/CanDeactivate)<CanComponentDeactivate> {

canDeactivate(component: CanComponentDeactivate) {

return component.canDeactivate ? component.canDeactivate() : true;

}

}

While the guard doesn't have to know which component has a deactivate method, it can detect that the CrisisDetailComponent component has the canDeactivate() method and call it. The guard not knowing the details of any component's deactivation method makes the guard reusable.

Alternatively, you could make a component-specific [CanDeactivate](https://angular.io/api/router/CanDeactivate) guard for the CrisisDetailComponent. The canDeactivate() method provides you with the current instance of the component, the current [ActivatedRoute](https://angular.io/api/router/ActivatedRoute), and [RouterStateSnapshot](https://angular.io/api/router/RouterStateSnapshot) in case you needed to access some external information. This would be useful if you only wanted to use this guard for this component and needed to get the component's properties or confirm whether the router should allow navigation away from it.

src/app/can-deactivate.guard.ts (component-specific)

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

import { Observable } from 'rxjs';

import { [CanDeactivate](https://angular.io/api/router/CanDeactivate),

[ActivatedRouteSnapshot](https://angular.io/api/router/ActivatedRouteSnapshot),

[RouterStateSnapshot](https://angular.io/api/router/RouterStateSnapshot) } from '@angular/router';

import { CrisisDetailComponent } from './crisis-center/crisis-detail/crisis-detail.component';

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

providedIn: 'root',

})

export class CanDeactivateGuard implements [CanDeactivate](https://angular.io/api/router/CanDeactivate)<CrisisDetailComponent> {

canDeactivate(

component: CrisisDetailComponent,

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

state: [RouterStateSnapshot](https://angular.io/api/router/RouterStateSnapshot)

): Observable<boolean> | boolean {

// Get the Crisis Center ID

console.log(route.paramMap.get('id'));

// Get the current URL

console.log(state.url);

// Allow synchronous navigation (`true`) if no crisis or the crisis is unchanged

if (!component.crisis || component.crisis.name === component.editName) {

return true;

}

// Otherwise ask the user with the dialog service and return its

// observable which resolves to true or false when the user decides

return component.dialogService.confirm('Discard changes?');

}

}

Looking back at the CrisisDetailComponent, it implements the confirmation workflow for unsaved changes.

src/app/crisis-center/crisis-detail/crisis-detail.component.ts (excerpt)

content\_copycanDeactivate(): Observable<boolean> | boolean {

// Allow synchronous navigation (`true`) if no crisis or the crisis is unchanged

if (!this.crisis || this.crisis.name === this.editName) {

return true;

}

// Otherwise ask the user with the dialog service and return its

// observable which resolves to true or false when the user decides

return this.dialogService.confirm('Discard changes?');

}

Notice that the canDeactivate() method can return synchronously; it returns true immediately if there is no crisis or there are no pending changes. But it can also return a Promise or an Observable and the router will wait for that to resolve to truthy (navigate) or falsy (stay on the current route).

Add the Guard to the crisis detail route in crisis-center-routing.module.ts using the canDeactivate array property.

src/app/crisis-center/crisis-center-routing.module.ts (can deactivate guard)

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

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

import { CrisisCenterHomeComponent } from './crisis-center-home/crisis-center-home.component';

import { CrisisListComponent } from './crisis-list/crisis-list.component';

import { CrisisCenterComponent } from './crisis-center/crisis-center.component';

import { CrisisDetailComponent } from './crisis-detail/crisis-detail.component';

import { CanDeactivateGuard } from '../can-deactivate.guard';

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

{

path: 'crisis-center',

component: CrisisCenterComponent,

children: [

{

path: '',

component: CrisisListComponent,

children: [

{

path: ':id',

component: CrisisDetailComponent,

canDeactivate: [CanDeactivateGuard]

},

{

path: '',

component: CrisisCenterHomeComponent

}

]

}

]

}

];

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

imports: [

RouterModule.forChild(crisisCenterRoutes)

],

exports: [

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

]

})

export class CrisisCenterRoutingModule { }

Now you have given the user a safeguard against unsaved changes.

### **Resolve: pre-fetching component data**

In the Hero Detail and Crisis Detail, the application waited until the route was activated to fetch the respective hero or crisis.

If you were using a real world API, there might be some delay before the data to display is returned from the server. You don't want to display a blank component while waiting for the data.

To improve this behavior, you can pre-fetch data from the server using a resolver so it's ready the moment the route is activated. This also allows you to handle errors before routing to the component. There's no point in navigating to a crisis detail for an id that doesn't have a record. It'd be better to send the user back to the Crisis List that shows only valid crisis centers.

In summary, you want to delay rendering the routed component until all necessary data has been fetched.

#### Fetch data before navigating

At the moment, the CrisisDetailComponent retrieves the selected crisis. If the crisis is not found, the router navigates back to the crisis list view.

The experience might be better if all of this were handled first, before the route is activated. A CrisisDetailResolver service could retrieve a Crisis or navigate away, if the Crisis did not exist, before activating the route and creating the CrisisDetailComponent.

Generate a CrisisDetailResolver service file within the Crisis Center feature area.

content\_copyng generate service crisis-center/crisis-detail-resolver

src/app/crisis-center/crisis-detail-resolver.service.ts (generated)

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

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

providedIn: 'root',

})

export class CrisisDetailResolverService {

constructor() { }

}

Move the relevant parts of the crisis retrieval logic in CrisisDetailComponent.ngOnInit() into the CrisisDetailResolverService. Import the Crisis model, CrisisService, and the [Router](https://angular.io/api/router/Router) so you can navigate elsewhere if you can't fetch the crisis.

Be explicit and implement the [Resolve](https://angular.io/api/router/Resolve) interface with a type of Crisis.

Inject the CrisisService and [Router](https://angular.io/api/router/Router) and implement the resolve() method. That method could return a Promise, an Observable, or a synchronous return value.

The CrisisService.getCrisis() method returns an observable in order to prevent the route from loading until the data is fetched. The [Router](https://angular.io/api/router/Router) guards require an observable to complete, which means it has emitted all of its values. You use the take operator with an argument of 1 to ensure that the Observable completes after retrieving the first value from the Observable returned by the getCrisis() method.

If it doesn't return a valid Crisis, then return an empty Observable, cancel the previous in-progress navigation to the CrisisDetailComponent, and navigate the user back to the CrisisListComponent. The updated resolver service looks like this:

src/app/crisis-center/crisis-detail-resolver.service.ts

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

import {

[Router](https://angular.io/api/router/Router), [Resolve](https://angular.io/api/router/Resolve),

[RouterStateSnapshot](https://angular.io/api/router/RouterStateSnapshot),

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

} from '@angular/router';

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

import { mergeMap, take } from 'rxjs/operators';

import { CrisisService } from './crisis.service';

import { Crisis } from './crisis';

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

providedIn: 'root',

})

export class CrisisDetailResolverService implements [Resolve](https://angular.io/api/router/Resolve)<Crisis> {

constructor(private cs: CrisisService, private router: [Router](https://angular.io/api/router/Router)) {}

resolve(route: [ActivatedRouteSnapshot](https://angular.io/api/router/ActivatedRouteSnapshot), state: [RouterStateSnapshot](https://angular.io/api/router/RouterStateSnapshot)): Observable<Crisis> | Observable<never> {

const id = route.paramMap.get('id')!;

return this.cs.getCrisis(id).pipe(

take(1),

mergeMap(crisis => {

if (crisis) {

return of(crisis);

} else { // id not found

this.router.navigate(['/crisis-center']);

return EMPTY;

}

})

);

}

}

Import this resolver in the crisis-center-routing.module.ts and add a resolve object to the CrisisDetailComponent route configuration.

src/app/crisis-center/crisis-center-routing.module.ts (resolver)

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

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

import { CrisisCenterHomeComponent } from './crisis-center-home/crisis-center-home.component';

import { CrisisListComponent } from './crisis-list/crisis-list.component';

import { CrisisCenterComponent } from './crisis-center/crisis-center.component';

import { CrisisDetailComponent } from './crisis-detail/crisis-detail.component';

import { CanDeactivateGuard } from '../can-deactivate.guard';

import { CrisisDetailResolverService } from './crisis-detail-resolver.service';

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

{

path: 'crisis-center',

component: CrisisCenterComponent,

children: [

{

path: '',

component: CrisisListComponent,

children: [

{

path: ':id',

component: CrisisDetailComponent,

canDeactivate: [CanDeactivateGuard],

resolve: {

crisis: CrisisDetailResolverService

}

},

{

path: '',

component: CrisisCenterHomeComponent

}

]

}

]

}

];

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

imports: [

RouterModule.forChild(crisisCenterRoutes)

],

exports: [

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

]

})

export class CrisisCenterRoutingModule { }

The CrisisDetailComponent should no longer fetch the crisis. When you re-configured the route, you changed where the crisis is. Update the CrisisDetailComponent to get the crisis from the ActivatedRoute.data.crisis property instead;

src/app/crisis-center/crisis-detail/crisis-detail.component.ts (ngOnInit v2)

content\_copyngOnInit() {

this.route.data

.subscribe(data => {

const crisis: Crisis = data.crisis;

this.editName = crisis.name;

this.crisis = crisis;

});

}

Note the following three important points:

1. The router's [Resolve](https://angular.io/api/router/Resolve) interface is optional. The CrisisDetailResolverService doesn't inherit from a base class. The router looks for that method and calls it if found.
2. The router calls the resolver in any case where the user could navigate away so you don't have to code for each use case.
3. Returning an empty Observable in at least one resolver will cancel navigation.

The relevant Crisis Center code for this milestone follows.

app.component.html

crisis-center-home.component.html

crisis-center.component.html

crisis-center-routing.module.ts

crisis-list.component.html

crisis-list.component.ts

crisis-detail.component.html

crisis-detail.component.ts

crisis-detail-resolver.service.ts

crisis.service.ts

dialog.service.ts

content\_copy<div class="wrapper">

<h1 class="title">Angular [Router](https://angular.io/api/router/Router)</h1>

<nav>

<a [routerLink](https://angular.io/api/router/RouterLink)="/crisis-center" [routerLinkActive](https://angular.io/api/router/RouterLinkActive)="active">Crisis Center</a>

<a [routerLink](https://angular.io/api/router/RouterLink)="/superheroes" [routerLinkActive](https://angular.io/api/router/RouterLinkActive)="active">Heroes</a>

<a [routerLink](https://angular.io/api/router/RouterLink)="/admin" [routerLinkActive](https://angular.io/api/router/RouterLinkActive)="active">Admin</a>

<a [routerLink](https://angular.io/api/router/RouterLink)="/login" [routerLinkActive](https://angular.io/api/router/RouterLinkActive)="active">Login</a>

<a [[routerLink](https://angular.io/api/router/RouterLink)]="[{ outlets: { popup: ['compose'] } }]">Contact</a>

</nav>

<div [@routeAnimation]="getAnimationData(routerOutlet)">

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

</div>

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

</div>

Guards

auth.guard.ts

can-deactivate.guard.ts

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

import {

[CanActivate](https://angular.io/api/router/CanActivate), [Router](https://angular.io/api/router/Router),

[ActivatedRouteSnapshot](https://angular.io/api/router/ActivatedRouteSnapshot),

[RouterStateSnapshot](https://angular.io/api/router/RouterStateSnapshot),

[CanActivateChild](https://angular.io/api/router/CanActivateChild),

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

} from '@angular/router';

import { AuthService } from './auth.service';

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

providedIn: 'root',

})

export class AuthGuard implements [CanActivate](https://angular.io/api/router/CanActivate), [CanActivateChild](https://angular.io/api/router/CanActivateChild) {

constructor(private authService: AuthService, private router: [Router](https://angular.io/api/router/Router)) {}

canActivate(

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

state: [RouterStateSnapshot](https://angular.io/api/router/RouterStateSnapshot)): true|[UrlTree](https://angular.io/api/router/UrlTree) {

const url: string = state.url;

return this.checkLogin(url);

}

canActivateChild(

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

state: [RouterStateSnapshot](https://angular.io/api/router/RouterStateSnapshot)): true|[UrlTree](https://angular.io/api/router/UrlTree) {

return this.canActivate(route, state);

}

checkLogin(url: string): true|[UrlTree](https://angular.io/api/router/UrlTree) {

if (this.authService.isLoggedIn) { return true; }

// Store the attempted URL for redirecting

this.authService.redirectUrl = url;

// Redirect to the login page

return this.router.parseUrl('/login');

}

}

### **Query parameters and fragments**

In the [route parameters](https://angular.io/guide/router-tutorial-toh#optional-route-parameters) section, you only dealt with parameters specific to the route. However, you can use query parameters to get optional parameters available to all routes.

[Fragments](https://en.wikipedia.org/wiki/Fragment_identifier) refer to certain elements on the page identified with an id attribute.

Update the AuthGuard to provide a session\_id query that will remain after navigating to another route.

Add an anchor element so you can jump to a certain point on the page.

Add the [NavigationExtras](https://angular.io/api/router/NavigationExtras) object to the router.navigate() method that navigates you to the /login route.

src/app/auth/auth.guard.ts (v3)

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

import {

[CanActivate](https://angular.io/api/router/CanActivate), [Router](https://angular.io/api/router/Router),

[ActivatedRouteSnapshot](https://angular.io/api/router/ActivatedRouteSnapshot),

[RouterStateSnapshot](https://angular.io/api/router/RouterStateSnapshot),

[CanActivateChild](https://angular.io/api/router/CanActivateChild),

[NavigationExtras](https://angular.io/api/router/NavigationExtras),

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

} from '@angular/router';

import { AuthService } from './auth.service';

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

providedIn: 'root',

})

export class AuthGuard implements [CanActivate](https://angular.io/api/router/CanActivate), [CanActivateChild](https://angular.io/api/router/CanActivateChild) {

constructor(private authService: AuthService, private router: [Router](https://angular.io/api/router/Router)) {}

canActivate(route: [ActivatedRouteSnapshot](https://angular.io/api/router/ActivatedRouteSnapshot), state: [RouterStateSnapshot](https://angular.io/api/router/RouterStateSnapshot)): true|[UrlTree](https://angular.io/api/router/UrlTree) {

const url: string = state.url;

return this.checkLogin(url);

}

canActivateChild(route: [ActivatedRouteSnapshot](https://angular.io/api/router/ActivatedRouteSnapshot), state: [RouterStateSnapshot](https://angular.io/api/router/RouterStateSnapshot)): true|[UrlTree](https://angular.io/api/router/UrlTree) {

return this.canActivate(route, state);

}

checkLogin(url: string): true|[UrlTree](https://angular.io/api/router/UrlTree) {

if (this.authService.isLoggedIn) { return true; }

// Store the attempted URL for redirecting

this.authService.redirectUrl = url;

// Create a dummy session id

const sessionId = 123456789;

// Set our navigation extras object

// that contains our [global](https://angular.io/api/core/global) [query](https://angular.io/api/animations/query) params and fragment

const navigationExtras: [NavigationExtras](https://angular.io/api/router/NavigationExtras) = {

queryParams: { session\_id: sessionId },

fragment: 'anchor'

};

// Redirect to the login page with extras

return this.router.createUrlTree(['/login'], navigationExtras);

}

}

You can also preserve query parameters and fragments across navigations without having to provide them again when navigating. In the LoginComponent, you'll add an object as the second argument in the router.navigateUrl() function and provide the queryParamsHandling and preserveFragment to pass along the current query parameters and fragment to the next route.

src/app/auth/login/login.component.ts (preserve)

content\_copy// Set our navigation extras object

// that passes on our [global](https://angular.io/api/core/global) [query](https://angular.io/api/animations/query) params and fragment

const navigationExtras: [NavigationExtras](https://angular.io/api/router/NavigationExtras) = {

queryParamsHandling: 'preserve',

preserveFragment: true

};

// Redirect the user

this.router.navigate([redirectUrl], navigationExtras);

The queryParamsHandling feature also provides a merge option, which preserves and combines the current query parameters with any provided query parameters when navigating.

To navigate to the Admin Dashboard route after logging in, update admin-dashboard.component.ts to handle the query parameters and fragment.

src/app/admin/admin-dashboard/admin-dashboard.component.ts (v2)

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

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

import { Observable } from 'rxjs';

import { map } from 'rxjs/operators';

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

selector: 'app-admin-dashboard',

templateUrl: './admin-dashboard.component.html',

styleUrls: ['./admin-dashboard.component.css']

})

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

sessionId!: Observable<string>;

token!: Observable<string>;

constructor(private route: [ActivatedRoute](https://angular.io/api/router/ActivatedRoute)) {}

ngOnInit() {

// Capture the session ID if available

this.sessionId = this.route

.queryParamMap

.pipe(map(params => params.get('session\_id') || 'None'));

// Capture the fragment if available

this.token = this.route

.fragment

.pipe(map(fragment => fragment || 'None'));

}

}

Query parameters and fragments are also available through the [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) service. Like route parameters, the query parameters and fragments are provided as an Observable. The updated Crisis Admin component feeds the Observable directly into the template using the [AsyncPipe](https://angular.io/api/common/AsyncPipe).

Now, you can click on the Admin button, which takes you to the Login page with the provided queryParamMap and fragment. After you click the login button, notice that you have been redirected to the Admin Dashboard page with the query parameters and fragment still intact in the address bar.

You can use these persistent bits of information for things that need to be provided across pages like authentication tokens or session ids.

The [query](https://angular.io/api/animations/query) params and fragment can also be preserved using a [RouterLink](https://angular.io/api/router/RouterLink) with the queryParamsHandling and preserveFragment bindings respectively.

## **Milestone 6: Asynchronous routing**

As you've worked through the milestones, the application has naturally gotten larger. At some point you'll reach a point where the application takes a long time to load.

To remedy this issue, use asynchronous routing, which loads feature modules lazily, on request. Lazy loading has multiple benefits.

* You can load feature areas only when requested by the user.
* You can speed up load time for users that only visit certain areas of the application.
* You can continue expanding lazy loaded feature areas without increasing the size of the initial load bundle.

You're already part of the way there. By organizing the application into modules—AppModule, HeroesModule, AdminModule and CrisisCenterModule—you have natural candidates for lazy loading.

Some modules, like AppModule, must be loaded from the start. But others can and should be lazy loaded. The AdminModule, for example, is needed by a few authorized users, so you should only load it when requested by the right people.

### **Lazy Loading route configuration**

Change the admin path in the admin-routing.module.ts from 'admin' to an empty string, '', the empty path.

Use empty path routes to group routes together without adding any additional path segments to the URL. Users will still visit /admin and the AdminComponent still serves as the Routing Component containing child routes.

Open the AppRoutingModule and add a new admin route to its appRoutes array.

Give it a loadChildren property instead of a children property. The loadChildren property takes a function that returns a promise using the browser's built-in syntax for lazy loading code using dynamic imports import('...'). The path is the location of the AdminModule (relative to the application root). After the code is requested and loaded, the Promise resolves an object that contains the [NgModule](https://angular.io/api/core/NgModule), in this case the AdminModule.

app-routing.module.ts (load children)

content\_copy{

path: 'admin',

loadChildren: () => import('./admin/admin.module').then(m => m.AdminModule),

},

Note: When using absolute paths, the [NgModule](https://angular.io/api/core/NgModule) file location must begin with src/app in order to resolve correctly. For custom [path mapping with absolute paths](https://www.typescriptlang.org/docs/handbook/module-resolution.html#path-mapping), you must configure the baseUrl and paths properties in the project tsconfig.json.

When the router navigates to this route, it uses the loadChildren string to dynamically load the AdminModule. Then it adds the AdminModule routes to its current route configuration. Finally, it loads the requested route to the destination admin component.

The lazy loading and re-configuration happen just once, when the route is first requested; the module and routes are available immediately for subsequent requests.

Angular provides a built-in module loader that supports SystemJS to load modules asynchronously. If you were using another bundling tool, such as Webpack, you would use the Webpack mechanism for asynchronously loading modules.

Take the final step and detach the admin feature set from the main application. The root AppModule must neither load nor reference the AdminModule or its files.

In app.module.ts, remove the AdminModule import statement from the top of the file and remove the AdminModule from the NgModule's imports array.

### [**CanLoad**](https://angular.io/api/router/CanLoad)**: guarding unauthorized loading of feature modules**

You're already protecting the AdminModule with a [CanActivate](https://angular.io/api/router/CanActivate) guard that prevents unauthorized users from accessing the admin feature area. It redirects to the login page if the user is not authorized.

But the router is still loading the AdminModule even if the user can't visit any of its components. Ideally, you'd only load the AdminModule if the user is logged in.

Add a [CanLoad](https://angular.io/api/router/CanLoad) guard that only loads the AdminModule once the user is logged in and attempts to access the admin feature area.

The existing AuthGuard already has the essential logic in its checkLogin() method to support the [CanLoad](https://angular.io/api/router/CanLoad) guard.

Open auth.guard.ts. Import the [CanLoad](https://angular.io/api/router/CanLoad) interface from @angular/router. Add it to the AuthGuard class's implements list. Then implement canLoad() as follows:

src/app/auth/auth.guard.ts (CanLoad guard)

content\_copycanLoad(route: [Route](https://angular.io/api/router/Route)): boolean {

const url = `/${route.path}`;

return this.checkLogin(url);

}

The router sets the canLoad() method's route parameter to the intended destination URL. The checkLogin() method redirects to that URL once the user has logged in.

Now import the AuthGuard into the AppRoutingModule and add the AuthGuard to the canLoad array property for the admin route. The completed admin route looks like this:

app-routing.module.ts (lazy admin route)

content\_copy{

path: 'admin',

loadChildren: () => import('./admin/admin.module').then(m => m.AdminModule),

canLoad: [AuthGuard]

},

### **Preloading: background loading of feature areas**

In addition to loading modules on-demand, you can load modules asynchronously with preloading.

The AppModule is eagerly loaded when the application starts, meaning that it loads right away. Now the AdminModule loads only when the user clicks on a link, which is called lazy loading.

Preloading allows you to load modules in the background so that the data is ready to render when the user activates a particular route. Consider the Crisis Center. It isn't the first view that a user sees. By default, the Heroes are the first view. For the smallest initial payload and fastest launch time, you should eagerly load the AppModule and the HeroesModule.

You could lazy load the Crisis Center. But you're almost certain that the user will visit the Crisis Center within minutes of launching the app. Ideally, the application would launch with just the AppModule and the HeroesModule loaded and then, almost immediately, load the CrisisCenterModule in the background. By the time the user navigates to the Crisis Center, its module will have been loaded and ready.

#### How preloading works

After each successful navigation, the router looks in its configuration for an unloaded module that it can preload. Whether it preloads a module, and which modules it preloads, depends upon the preload strategy.

The [Router](https://angular.io/api/router/Router) offers two preloading strategies:

* No preloading, which is the default. Lazy loaded feature areas are still loaded on-demand.
* Preloading of all lazy loaded feature areas.

The router either never preloads, or preloads every lazy loaded module. The [Router](https://angular.io/api/router/Router) also supports [custom preloading strategies](https://angular.io/guide/router-tutorial-toh#custom-preloading) for fine control over which modules to preload and when.

This section guides you through updating the CrisisCenterModule to load lazily by default and use the [PreloadAllModules](https://angular.io/api/router/PreloadAllModules) strategy to load all lazy loaded modules.

#### Lazy load the crisis center

Update the route configuration to lazy load the CrisisCenterModule. Take the same steps you used to configure AdminModule for lazy loading.

1. Change the crisis-center path in the CrisisCenterRoutingModule to an empty string.
2. Add a crisis-center route to the AppRoutingModule.
3. Set the loadChildren string to load the CrisisCenterModule.
4. Remove all mention of the CrisisCenterModule from app.module.ts.

Here are the updated modules before enabling preload:

app.module.ts

app-routing.module.ts

crisis-center-routing.module.ts

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

import { [BrowserModule](https://angular.io/api/platform-browser/BrowserModule) } from '@angular/platform-browser';

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

import { [BrowserAnimationsModule](https://angular.io/api/platform-browser/animations/BrowserAnimationsModule) } from '@angular/platform-browser/animations';

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

import { AppComponent } from './app.component';

import { PageNotFoundComponent } from './page-not-found/page-not-found.component';

import { ComposeMessageComponent } from './compose-message/compose-message.component';

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

import { HeroesModule } from './heroes/heroes.module';

import { AuthModule } from './auth/auth.module';

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

imports: [

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

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

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

HeroesModule,

AuthModule,

AppRoutingModule,

],

declarations: [

AppComponent,

ComposeMessageComponent,

PageNotFoundComponent

],

bootstrap: [ AppComponent ]

})

export class AppModule {

}

You could try this now and confirm that the CrisisCenterModule loads after you click the "Crisis Center" button.

To enable preloading of all lazy loaded modules, import the [PreloadAllModules](https://angular.io/api/router/PreloadAllModules) token from the Angular router package.

The second argument in the [RouterModule.forRoot()](https://angular.io/api/router/RouterModule#forRoot) method takes an object for additional configuration options. The preloadingStrategy is one of those options. Add the [PreloadAllModules](https://angular.io/api/router/PreloadAllModules) token to the forRoot() call:

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

content\_copyRouterModule.forRoot(

appRoutes,

{

enableTracing: true, // <-- debugging purposes only

preloadingStrategy: [PreloadAllModules](https://angular.io/api/router/PreloadAllModules)

}

)

This configures the [Router](https://angular.io/api/router/Router) preloader to immediately load all lazy loaded routes (routes with a loadChildren property).

When you visit http://localhost:4200, the /heroes route loads immediately upon launch and the router starts loading the CrisisCenterModule right after the HeroesModule loads.

Currently, the AdminModule does not preload because [CanLoad](https://angular.io/api/router/CanLoad) is blocking it.

#### [CanLoad](https://angular.io/api/router/CanLoad) blocks preload

The [PreloadAllModules](https://angular.io/api/router/PreloadAllModules) strategy does not load feature areas protected by a [CanLoad](https://angular.io/guide/router-tutorial-toh#can-load-guard) guard.

You added a [CanLoad](https://angular.io/api/router/CanLoad) guard to the route in the AdminModule a few steps back to block loading of that module until the user is authorized. That [CanLoad](https://angular.io/api/router/CanLoad) guard takes precedence over the preload strategy.

If you want to preload a module as well as guard against unauthorized access, remove the canLoad() guard method and rely on the [canActivate()](https://angular.io/guide/router-tutorial-toh#can-activate-guard) guard alone.

### **Custom Preloading Strategy**

Preloading every lazy loaded module works well in many situations. However, in consideration of things such as low bandwidth and user metrics, you can use a custom preloading strategy for specific feature modules.

This section guides you through adding a custom strategy that only preloads routes whose data.preload flag is set to true. Recall that you can add anything to the data property of a route.

Set the data.preload flag in the crisis-center route in the AppRoutingModule.

src/app/app-routing.module.ts (route data preload)

content\_copy{

path: 'crisis-center',

loadChildren: () => import('./crisis-center/crisis-center.module').then(m => m.CrisisCenterModule),

data: { preload: true }

},

Generate a new SelectivePreloadingStrategy service.

content\_copyng generate service selective-preloading-strategy

Replace the contents of selective-preloading-strategy.service.ts with the following:

src/app/selective-preloading-strategy.service.ts

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

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

import { Observable, of } from 'rxjs';

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

providedIn: 'root',

})

export class SelectivePreloadingStrategyService implements [PreloadingStrategy](https://angular.io/api/router/PreloadingStrategy) {

preloadedModules: string[] = [];

preload(route: [Route](https://angular.io/api/router/Route), load: () => Observable<any>): Observable<any> {

if (route.data && route.data.preload && route.path != null) {

// add the route path to the preloaded module array

this.preloadedModules.push(route.path);

// log the route path to the console

console.log('Preloaded: ' + route.path);

return load();

} else {

return of(null);

}

}

}

SelectivePreloadingStrategyService implements the [PreloadingStrategy](https://angular.io/api/router/PreloadingStrategy), which has one method, preload().

The router calls the preload() method with two arguments:

1. The route to consider.
2. A loader function that can load the routed module asynchronously.

An implementation of preload must return an Observable. If the route does preload, it returns the observable returned by calling the loader function. If the route does not preload, it returns an Observable of null.

In this sample, the preload() method loads the route if the route's data.preload flag is truthy.

As a side-effect, SelectivePreloadingStrategyService logs the path of a selected route in its public preloadedModules array.

Shortly, you'll extend the AdminDashboardComponent to inject this service and display its preloadedModules array.

But first, make a few changes to the AppRoutingModule.

1. Import SelectivePreloadingStrategyService into AppRoutingModule.
2. Replace the [PreloadAllModules](https://angular.io/api/router/PreloadAllModules) strategy in the call to forRoot() with this SelectivePreloadingStrategyService.

Now edit the AdminDashboardComponent to display the log of preloaded routes.

1. Import the SelectivePreloadingStrategyService.
2. Inject it into the dashboard's constructor.
3. Update the template to display the strategy service's preloadedModules array.

Now the file is as follows:

src/app/admin/admin-dashboard/admin-dashboard.component.ts (preloaded modules)

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

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

import { Observable } from 'rxjs';

import { map } from 'rxjs/operators';

import { SelectivePreloadingStrategyService } from '../../selective-preloading-strategy.service';

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

selector: 'app-admin-dashboard',

templateUrl: './admin-dashboard.component.html',

styleUrls: ['./admin-dashboard.component.css']

})

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

sessionId!: Observable<string>;

token!: Observable<string>;

modules: string[] = [];

constructor(

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

preloadStrategy: SelectivePreloadingStrategyService

) {

this.modules = preloadStrategy.preloadedModules;

}

ngOnInit() {

// Capture the session ID if available

this.sessionId = this.route

.queryParamMap

.pipe(map(params => params.get('session\_id') || 'None'));

// Capture the fragment if available

this.token = this.route

.fragment

.pipe(map(fragment => fragment || 'None'));

}

}

Once the application loads the initial route, the CrisisCenterModule is preloaded. Verify this by logging in to the Admin feature area and noting that the crisis-center is listed in the Preloaded Modules. It also logs to the browser's console.

### **Migrating URLs with redirects**

You've setup the routes for navigating around your application and used navigation imperatively and declaratively. But like any application, requirements change over time. You've setup links and navigation to /heroes and /hero/:id from the HeroListComponent and HeroDetailComponent components. If there were a requirement that links to heroes become superheroes, you would still want the previous URLs to navigate correctly. You also don't want to update every link in your application, so redirects makes refactoring routes trivial.

#### Changing /heroes to /superheroes

This section guides you through migrating the Hero routes to new URLs. The [Router](https://angular.io/api/router/Router) checks for redirects in your configuration before navigating, so each redirect is triggered when needed. To support this change, add redirects from the old routes to the new routes in the heroes-routing.module.

src/app/heroes/heroes-routing.module.ts (heroes redirects)

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

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

import { HeroListComponent } from './hero-list/hero-list.component';

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

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

{ path: 'heroes', redirectTo: '/superheroes' },

{ path: 'hero/:id', redirectTo: '/superhero/:id' },

{ path: 'superheroes', component: HeroListComponent, data: { [animation](https://angular.io/api/animations/animation): 'heroes' } },

{ path: 'superhero/:id', component: HeroDetailComponent, data: { [animation](https://angular.io/api/animations/animation): 'hero' } }

];

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

imports: [

RouterModule.forChild(heroesRoutes)

],

exports: [

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

]

})

export class HeroesRoutingModule { }

Notice two different types of redirects. The first change is from /heroes to /superheroes without any parameters. The second change is from /hero/:id to /superhero/:id, which includes the :id route parameter. Router redirects also use powerful pattern-matching, so the [Router](https://angular.io/api/router/Router) inspects the URL and replaces route parameters in the path with their appropriate destination. Previously, you navigated to a URL such as /hero/15 with a route parameter id of 15.

The [Router](https://angular.io/api/router/Router) also supports [query parameters](https://angular.io/guide/router-tutorial-toh#query-parameters) and the [fragment](https://angular.io/guide/router-tutorial-toh#fragment) when using redirects.

* When using absolute redirects, the [Router](https://angular.io/api/router/Router) will use the query parameters and the fragment from the redirectTo in the route config.
* When using relative redirects, the [Router](https://angular.io/api/router/Router) use the query params and the fragment from the source URL.

Currently, the empty path route redirects to /heroes, which redirects to /superheroes. This won't work because the [Router](https://angular.io/api/router/Router) handles redirects once at each level of routing configuration. This prevents chaining of redirects, which can lead to endless redirect loops.

Instead, update the empty path route in app-routing.module.ts to redirect to /superheroes.

src/app/app-routing.module.ts (superheroes redirect)

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

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

import { ComposeMessageComponent } from './compose-message/compose-message.component';

import { PageNotFoundComponent } from './page-not-found/page-not-found.component';

import { AuthGuard } from './auth/auth.guard';

import { SelectivePreloadingStrategyService } from './selective-preloading-strategy.service';

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

{

path: 'compose',

component: ComposeMessageComponent,

outlet: 'popup'

},

{

path: 'admin',

loadChildren: () => import('./admin/admin.module').then(m => m.AdminModule),

canLoad: [AuthGuard]

},

{

path: 'crisis-center',

loadChildren: () => import('./crisis-center/crisis-center.module').then(m => m.CrisisCenterModule),

data: { preload: true }

},

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

{ path: '\*\*', component: PageNotFoundComponent }

];

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

imports: [

RouterModule.forRoot(

appRoutes,

{

enableTracing: false, // <-- debugging purposes only

preloadingStrategy: SelectivePreloadingStrategyService,

}

)

],

exports: [

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

]

})

export class AppRoutingModule { }

A [routerLink](https://angular.io/api/router/RouterLink) isn't tied to route configuration, so update the associated router links to remain active when the new route is active. Update the app.component.ts template for the /heroes [routerLink](https://angular.io/api/router/RouterLink).

src/app/app.component.html (superheroes active routerLink)

content\_copy<div class="wrapper">

<h1 class="title">Angular [Router](https://angular.io/api/router/Router)</h1>

<nav>

<a [routerLink](https://angular.io/api/router/RouterLink)="/crisis-center" [routerLinkActive](https://angular.io/api/router/RouterLinkActive)="active">Crisis Center</a>

<a [routerLink](https://angular.io/api/router/RouterLink)="/superheroes" [routerLinkActive](https://angular.io/api/router/RouterLinkActive)="active">Heroes</a>

<a [routerLink](https://angular.io/api/router/RouterLink)="/admin" [routerLinkActive](https://angular.io/api/router/RouterLinkActive)="active">Admin</a>

<a [routerLink](https://angular.io/api/router/RouterLink)="/login" [routerLinkActive](https://angular.io/api/router/RouterLinkActive)="active">Login</a>

<a [[routerLink](https://angular.io/api/router/RouterLink)]="[{ outlets: { popup: ['compose'] } }]">Contact</a>

</nav>

<div [@routeAnimation]="getAnimationData(routerOutlet)">

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

</div>

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

</div>

Update the goToHeroes() method in the hero-detail.component.ts to navigate back to /superheroes with the optional route parameters.

src/app/heroes/hero-detail/hero-detail.component.ts (goToHeroes)

content\_copygotoHeroes(hero: Hero) {

const heroId = hero ? hero.id : null;

// Pass along the hero id if available

// so that the HeroList component can select that hero.

// Include a junk 'foo' property for fun.

this.router.navigate(['/superheroes', { id: heroId, foo: 'foo' }]);

}

With the redirects setup, all previous routes now point to their new destinations and both URLs still function as intended.

### **Inspect the router's configuration**

To determine if your routes are actually evaluated [in the proper order](https://angular.io/guide/router-tutorial-toh#routing-module-order), you can inspect the router's configuration.

Do this by injecting the router and logging to the console its config property. For example, update the AppModule as follows and look in the browser console window to see the finished route configuration.

src/app/app.module.ts (inspect the router config)

content\_copyexport class AppModule {

// Diagnostic only: inspect router configuration

constructor(router: [Router](https://angular.io/api/router/Router)) {

// Use a custom replacer to display function names in the route configs

const replacer = (key, value) => (typeof value === 'function') ? value.name : value;

console.log('[Routes](https://angular.io/api/router/Routes): ', JSON.stringify(router.config, replacer, 2));

}

}

## **Final application**

For the completed router application, see the [live example](https://angular.io/generated/live-examples/router/stackblitz.html) / [download example](https://angular.io/generated/zips/router/router.zip) for the final source code.

# Router Reference

The following sections highlight some core router concepts.

### **Router imports**

The Angular Router is an optional service that presents a particular component view for a given URL. It is not part of the Angular core and thus is in its own library package, @angular/router.

Import what you need from it as you would from any other Angular package.

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

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

For more on browser URL styles, see [LocationStrategy and browser URL styles](https://angular.io/guide/router#browser-url-styles).

### **Configuration**

A routed Angular application has one singleton instance of the [Router](https://angular.io/api/router/Router) service. When the browser's URL changes, that router looks for a corresponding [Route](https://angular.io/api/router/Route) from which it can determine the component to display.

A router has no routes until you configure it. The following example creates five route definitions, configures the router via the [RouterModule.forRoot()](https://angular.io/api/router/RouterModule#forRoot) method, and adds the result to the AppModule's imports array.

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

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

{ path: 'crisis-center', component: CrisisListComponent },

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

{

path: 'heroes',

component: HeroListComponent,

data: { title: 'Heroes List' }

},

{ path: '',

redirectTo: '/heroes',

pathMatch: 'full'

},

{ path: '\*\*', component: PageNotFoundComponent }

];

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

imports: [

RouterModule.forRoot(

appRoutes,

{ enableTracing: true } // <-- debugging purposes only

)

// other imports here

],

...

})

export class AppModule { }

The appRoutes array of routes describes how to navigate. Pass it to the [RouterModule.forRoot()](https://angular.io/api/router/RouterModule#forRoot) method in the module imports to configure the router.

Each [Route](https://angular.io/api/router/Route) maps a URL path to a component. There are no leading slashes in the path. The router parses and builds the final URL for you, which allows you to use both relative and absolute paths when navigating between application views.

The :id in the second route is a token for a route parameter. In a URL such as /hero/42, "42" is the value of the id parameter. The corresponding HeroDetailComponent uses that value to find and present the hero whose id is 42.

The data property in the third route is a place to store arbitrary data associated with this specific route. The data property is accessible within each activated route. Use it to store items such as page titles, breadcrumb text, and other read-only, static data. You can use the [resolve guard](https://angular.io/guide/router-tutorial-toh#resolve-guard) to retrieve dynamic data.

The empty path in the fourth route represents the default path for the application—the place to go when the path in the URL is empty, as it typically is at the start. This default route redirects to the route for the /heroes URL and, therefore, displays the HeroesListComponent.

If you need to see what events are happening during the navigation lifecycle, there is the enableTracing option as part of the router's default configuration. This outputs each router event that took place during each navigation lifecycle to the browser console. Use enableTracing only for debugging purposes. You set the enableTracing: true option in the object passed as the second argument to the [RouterModule.forRoot()](https://angular.io/api/router/RouterModule#forRoot) method.

### **Router outlet**

The [RouterOutlet](https://angular.io/api/router/RouterOutlet) is a directive from the router library that is used like a component. It acts as a placeholder that marks the spot in the template where the router should display the components for that outlet.

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

<!-- Routed components go here -->

Given the configuration above, when the browser URL for this application becomes /heroes, the router matches that URL to the route path /heroes and displays the HeroListComponent as a sibling element to the [RouterOutlet](https://angular.io/api/router/RouterOutlet) that you've placed in the host component's template.

### **Router links**

To navigate as a result of some user action such as the click of an anchor tag, use [RouterLink](https://angular.io/api/router/RouterLink).

Consider the following template:

src/app/app.component.html

content\_copy<h1>Angular [Router](https://angular.io/api/router/Router)</h1>

<nav>

<a [routerLink](https://angular.io/api/router/RouterLink)="/crisis-center" [routerLinkActive](https://angular.io/api/router/RouterLinkActive)="active">Crisis Center</a>

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

</nav>

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

The [RouterLink](https://angular.io/api/router/RouterLink) directives on the anchor tags give the router control over those elements. The navigation paths are fixed, so you can assign a string to the [routerLink](https://angular.io/api/router/RouterLink) (a "one-time" binding).

Had the navigation path been more dynamic, you could have bound to a template expression that returned an array of route link parameters; that is, the [link parameters array](https://angular.io/guide/router#link-parameters-array). The router resolves that array into a complete URL.

### **Active router links**

The [RouterLinkActive](https://angular.io/api/router/RouterLinkActive) directive toggles CSS classes for active [RouterLink](https://angular.io/api/router/RouterLink) bindings based on the current [RouterState](https://angular.io/api/router/RouterState).

On each anchor tag, you see a [property binding](https://angular.io/guide/property-binding) to the [RouterLinkActive](https://angular.io/api/router/RouterLinkActive) directive that looks like [routerLinkActive](https://angular.io/api/router/RouterLinkActive)="...".

The template expression to the right of the equal sign, =, contains a space-delimited string of CSS classes that the Router adds when this link is active (and removes when the link is inactive). You set the [RouterLinkActive](https://angular.io/api/router/RouterLinkActive) directive to a string of classes such as [[routerLinkActive](https://angular.io/api/router/RouterLinkActive)]="'active fluffy'" or bind it to a component property that returns such a string.

Active route links cascade down through each level of the route tree, so parent and child router links can be active at the same time. To override this behavior, you can bind to the [routerLinkActiveOptions] input binding with the { exact: true } expression. By using { exact: true }, a given [RouterLink](https://angular.io/api/router/RouterLink) will only be active if its URL is an exact match to the current URL.

### **Router state**

After the end of each successful navigation lifecycle, the router builds a tree of [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) objects that make up the current state of the router. You can access the current [RouterState](https://angular.io/api/router/RouterState) from anywhere in the application using the [Router](https://angular.io/api/router/Router) service and the routerState property.

Each [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) in the [RouterState](https://angular.io/api/router/RouterState) provides methods to traverse up and down the route tree to get information from parent, child, and sibling routes.

### **Activated route**

The route path and parameters are available through an injected router service called the [ActivatedRoute](https://angular.io/api/router/ActivatedRoute). It has a great deal of useful information including:

|  |  |
| --- | --- |
| **Property** | **Description** |
| url | An Observable of the route path(s), represented as an array of strings for each part of the route path. |
| data | An Observable that contains the data object provided for the route. Also contains any resolved values from the [resolve guard](https://angular.io/guide/router-tutorial-toh#resolve-guard). |
| paramMap | An Observable that contains a [map](https://angular.io/api/router/ParamMap) of the required and [optional parameters](https://angular.io/guide/router-tutorial-toh#optional-route-parameters) specific to the route. The map supports retrieving single and multiple values from the same parameter. |
| queryParamMap | An Observable that contains a [map](https://angular.io/api/router/ParamMap) of the [query parameters](https://angular.io/guide/router-tutorial-toh#query-parameters) available to all routes. The map supports retrieving single and multiple values from the query parameter. |
| fragment | An Observable of the URL [fragment](https://angular.io/guide/router-tutorial-toh#fragment) available to all routes. |
| outlet | The name of the [RouterOutlet](https://angular.io/api/router/RouterOutlet) used to render the route. For an unnamed outlet, the outlet name is primary. |
| routeConfig | The route configuration used for the route that contains the origin path. |
| parent | The route's parent [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) when this route is a [child route](https://angular.io/guide/router-tutorial-toh#child-routing-component). |
| firstChild | Contains the first [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) in the list of this route's child routes. |
| children | Contains all the [child routes](https://angular.io/guide/router-tutorial-toh#child-routing-component) activated under the current route. |

Two older properties are still available; however, their replacements are preferable as they may be deprecated in a future Angular version.

* params: An Observable that contains the required and [optional parameters](https://angular.io/guide/router-tutorial-toh#optional-route-parameters) specific to the route. Use paramMap instead.
* queryParams: An Observable that contains the [query parameters](https://angular.io/guide/router-tutorial-toh#query-parameters) available to all routes. Use queryParamMap instead.

### **Router events**

During each navigation, the [Router](https://angular.io/api/router/Router) emits navigation events through the [Router.events](https://angular.io/api/router/Router#events) property. These events range from when the navigation starts and ends to many points in between. The full list of navigation events is displayed in the table below.

|  |  |
| --- | --- |
| **Router Event** | **Description** |
| [NavigationStart](https://angular.io/api/router/NavigationStart) | An [event](https://angular.io/api/router/NavigationStart) triggered when navigation starts. |
| [RouteConfigLoadStart](https://angular.io/api/router/RouteConfigLoadStart) | An [event](https://angular.io/api/router/RouteConfigLoadStart) triggered before the [Router](https://angular.io/api/router/Router) [lazy loads](https://angular.io/guide/router-tutorial-toh#asynchronous-routing) a route configuration. |
| [RouteConfigLoadEnd](https://angular.io/api/router/RouteConfigLoadEnd) | An [event](https://angular.io/api/router/RouteConfigLoadEnd) triggered after a route has been lazy loaded. |
| [RoutesRecognized](https://angular.io/api/router/RoutesRecognized) | An [event](https://angular.io/api/router/RoutesRecognized) triggered when the Router parses the URL and the routes are recognized. |
| [GuardsCheckStart](https://angular.io/api/router/GuardsCheckStart) | An [event](https://angular.io/api/router/GuardsCheckStart) triggered when the Router begins the Guards phase of routing. |
| [ChildActivationStart](https://angular.io/api/router/ChildActivationStart) | An [event](https://angular.io/api/router/ChildActivationStart) triggered when the Router begins activating a route's children. |
| [ActivationStart](https://angular.io/api/router/ActivationStart) | An [event](https://angular.io/api/router/ActivationStart) triggered when the Router begins activating a route. |
| [GuardsCheckEnd](https://angular.io/api/router/GuardsCheckEnd) | An [event](https://angular.io/api/router/GuardsCheckEnd) triggered when the Router finishes the Guards phase of routing successfully. |
| [ResolveStart](https://angular.io/api/router/ResolveStart) | An [event](https://angular.io/api/router/ResolveStart) triggered when the Router begins the Resolve phase of routing. |
| [ResolveEnd](https://angular.io/api/router/ResolveEnd) | An [event](https://angular.io/api/router/ResolveEnd) triggered when the Router finishes the Resolve phase of routing successfuly. |
| [ChildActivationEnd](https://angular.io/api/router/ChildActivationEnd) | An [event](https://angular.io/api/router/ChildActivationEnd) triggered when the Router finishes activating a route's children. |
| [ActivationEnd](https://angular.io/api/router/ActivationEnd) | An [event](https://angular.io/api/router/ActivationEnd) triggered when the Router finishes activating a route. |
| [NavigationEnd](https://angular.io/api/router/NavigationEnd) | An [event](https://angular.io/api/router/NavigationEnd) triggered when navigation ends successfully. |
| [NavigationCancel](https://angular.io/api/router/NavigationCancel) | An [event](https://angular.io/api/router/NavigationCancel) triggered when navigation is canceled. This can happen when a [Route Guard](https://angular.io/guide/router-tutorial-toh#guards) returns false during navigation, or redirects by returning a [UrlTree](https://angular.io/api/router/UrlTree). |
| [NavigationError](https://angular.io/api/router/NavigationError) | An [event](https://angular.io/api/router/NavigationError) triggered when navigation fails due to an unexpected error. |
| [Scroll](https://angular.io/api/router/Scroll) | An [event](https://angular.io/api/router/Scroll) that represents a scrolling event. |

When you enable the enableTracing option, Angular logs these events to the console. For an example of filtering router navigation events, see the [router section](https://angular.io/guide/observables-in-angular#router) of the [Observables in Angular](https://angular.io/guide/observables-in-angular) guide.

### **Router terminology**

Here are the key [Router](https://angular.io/api/router/Router) terms and their meanings:

|  |  |
| --- | --- |
| **Router Part** | **Meaning** |
| [Router](https://angular.io/api/router/Router) | Displays the application component for the active URL. Manages navigation from one component to the next. |
| [RouterModule](https://angular.io/api/router/RouterModule) | A separate NgModule that provides the necessary service providers and directives for navigating through application views. |
| [Routes](https://angular.io/api/router/Routes) | Defines an array of Routes, each mapping a URL path to a component. |
| [Route](https://angular.io/api/router/Route) | Defines how the router should navigate to a component based on a URL pattern. Most routes consist of a path and a component type. |
| [RouterOutlet](https://angular.io/api/router/RouterOutlet) | The directive (<[router-outlet](https://angular.io/api/router/RouterOutlet)>) that marks where the router displays a view. |
| [RouterLink](https://angular.io/api/router/RouterLink) | The directive for binding a clickable HTML element to a route. Clicking an element with a [routerLink](https://angular.io/api/router/RouterLink) directive that is bound to a *string* or a *link parameters array* triggers a navigation. |
| [RouterLinkActive](https://angular.io/api/router/RouterLinkActive) | The directive for adding/removing classes from an HTML element when an associated [routerLink](https://angular.io/api/router/RouterLink) contained on or inside the element becomes active/inactive. |
| [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) | A service that is provided to each route component that contains route specific information such as route parameters, static data, resolve data, global query params, and the global fragment. |
| [RouterState](https://angular.io/api/router/RouterState) | The current state of the router including a tree of the currently activated routes together with convenience methods for traversing the route tree. |
| ***Link parameters array*** | An array that the router interprets as a routing instruction. You can bind that array to a [RouterLink](https://angular.io/api/router/RouterLink) or pass the array as an argument to the Router.navigate method. |
| ***Routing component*** | An Angular component with a [RouterOutlet](https://angular.io/api/router/RouterOutlet) that displays views based on router navigations. |

# Reactive forms

Reactive forms provide a model-driven approach to handling form inputs whose values change over time. This guide shows you how to create and update a basic form control, progress to using multiple controls in a group, validate form values, and create dynamic forms where you can add or remove controls at run time.

Try this [Reactive Forms live-example](https://angular.io/generated/live-examples/reactive-forms/stackblitz.html) / [download example](https://angular.io/generated/zips/reactive-forms/reactive-forms.zip).

**Prerequisites**

Before going further into reactive forms, you should have a basic understanding of the following:

* [TypeScript](https://www.typescriptlang.org/) programming.
* Angular application-design fundamentals, as described in [Angular Concepts](https://angular.io/guide/architecture).
* The form-design concepts that are presented in [Introduction to Forms](https://angular.io/guide/forms-overview).

## **Overview of reactive forms**

Reactive forms use an explicit and immutable approach to managing the state of a form at a given point in time. Each change to the form state returns a new state, which maintains the integrity of the model between changes. Reactive forms are built around [observable](https://angular.io/guide/glossary#observable) streams, where form inputs and values are provided as streams of input values, which can be accessed synchronously.

Reactive forms also provide a straightforward path to testing because you are assured that your data is consistent and predictable when requested. Any consumers of the streams have access to manipulate that data safely.

Reactive forms differ from [template-driven forms](https://angular.io/guide/forms) in distinct ways. Reactive forms provide synchronous access to the data model, immutability with observable operators, and change tracking through observable streams.

Template-driven forms allow direct access to modify data in your template, but are less explicit than reactive forms because they rely on directives embedded in the template, along with mutable data to track changes asynchronously. See the [Forms Overview](https://angular.io/guide/forms-overview) for detailed comparisons between the two paradigms.

## **Adding a basic form control**

There are three steps to using form controls.

1. Register the reactive forms module in your application. This module declares the reactive-form directives that you need to use reactive forms.
2. Generate a new [FormControl](https://angular.io/api/forms/FormControl) instance and save it in the component.
3. Register the [FormControl](https://angular.io/api/forms/FormControl) in the template.

You can then display the form by adding the component to the template.

The following examples show how to add a single form control. In the example, the user enters their name into an input field, captures that input value, and displays the current value of the form control element.

**Register the reactive forms module**

To use reactive form controls, import [ReactiveFormsModule](https://angular.io/api/forms/ReactiveFormsModule) from the @angular/forms package and add it to your NgModule's imports array.

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

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

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

imports: [

// other imports ...

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

],

})

export class AppModule { }

**Generate a new**[FormControl](https://angular.io/api/forms/FormControl)

Use the [CLI command](https://angular.io/cli) ng generate to generate a component in your project to host the control.

content\_copyng generate component NameEditor

To register a single form control, import the [FormControl](https://angular.io/api/forms/FormControl) class and create a new instance of [FormControl](https://angular.io/api/forms/FormControl) to save as a class property.

src/app/name-editor/name-editor.component.ts

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

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

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

selector: 'app-name-editor',

templateUrl: './name-editor.component.html',

styleUrls: ['./name-editor.component.css']

})

export class NameEditorComponent {

name = new [FormControl](https://angular.io/api/forms/FormControl)('');

}

Use the constructor of [FormControl](https://angular.io/api/forms/FormControl) to set its initial value, which in this case is an empty string. By creating these controls in your component class, you get immediate access to listen for, update, and validate the state of the form input.

**Register the control in the template**

After you create the control in the component class, you must associate it with a form control element in the template. Update the template with the form control using the formControl binding provided by [FormControlDirective](https://angular.io/api/forms/FormControlDirective), which is also included in the [ReactiveFormsModule](https://angular.io/api/forms/ReactiveFormsModule).

src/app/name-editor/name-editor.component.html

content\_copy<label for="name">Name: </label>

<input id="name" type="text" [formControl]="name">

* For a summary of the classes and directives provided by [ReactiveFormsModule](https://angular.io/api/forms/ReactiveFormsModule), see the [Reactive forms API](https://angular.io/guide/reactive-forms#reactive-forms-api) section below.
* For complete syntax details of these classes and directives, see the API reference documentation for the [Forms package](https://angular.io/api/forms).

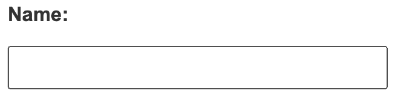
Using the template binding syntax, the form control is now registered to the name input element in the template. The form control and DOM element communicate with each other: the view reflects changes in the model, and the model reflects changes in the view.

**Display the component**

The form control assigned to name is displayed when the component is added to a template.

src/app/app.component.html (name editor)

content\_copy<app-name-editor></app-name-editor>



### **Displaying a form control value**

You can display the value in the following ways.

* Through the valueChanges observable where you can listen for changes in the form's value in the template using [AsyncPipe](https://angular.io/api/common/AsyncPipe) or in the component class using the subscribe() method.
* With the value property, which gives you a snapshot of the current value.

The following example shows you how to display the current value using interpolation in the template.

src/app/name-editor/name-editor.component.html (control value)

content\_copy<p>Value: {{ name.value }}</p>

The displayed value changes as you update the form control element.

Reactive forms provide access to information about a given control through properties and methods provided with each instance. These properties and methods of the underlying [AbstractControl](https://angular.io/api/forms/AbstractControl) class are used to control form state and determine when to display messages when handling [input validation](https://angular.io/guide/reactive-forms#basic-form-validation).

Read about other [FormControl](https://angular.io/api/forms/FormControl) properties and methods in the [API Reference](https://angular.io/api/forms/FormControl).

### **Replacing a form control value**

Reactive forms have methods to change a control's value programmatically, which gives you the flexibility to update the value without user interaction. A form control instance provides a setValue() method that updates the value of the form control and validates the structure of the value provided against the control's structure. For example, when retrieving form data from a backend API or service, use the setValue() method to update the control to its new value, replacing the old value entirely.

The following example adds a method to the component class to update the value of the control to Nancy using the setValue() method.

src/app/name-editor/name-editor.component.ts (update value)

content\_copyupdateName() {

this.name.setValue('Nancy');

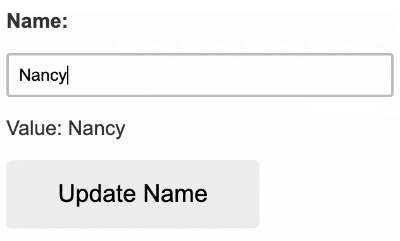
}

Update the template with a button to simulate a name update. When you click the **Update Name** button, the value entered in the form control element is reflected as its current value.

src/app/name-editor/name-editor.component.html (update value)

content\_copy<button (click)="updateName()">Update Name</button>

The form model is the source of truth for the control, so when you click the button, the value of the input is changed within the component class, overriding its current value.



**Note:** In this example, you're using a single control. When using the setValue() method with a [form group](https://angular.io/guide/reactive-forms#grouping-form-controls) or [form array](https://angular.io/guide/reactive-forms#creating-dynamic-forms) instance, the value needs to match the structure of the group or array.

## **Grouping form controls**

Forms typically contain several related controls. Reactive forms provide two ways of grouping multiple related controls into a single input form.

* A form group defines a form with a fixed set of controls that you can manage together. Form group basics are discussed in this section. You can also [nest form groups](https://angular.io/guide/reactive-forms#nested-groups) to create more complex forms.
* A form array defines a dynamic form, where you can add and remove controls at run time. You can also nest form arrays to create more complex forms. For more about this option, see [Creating dynamic forms](https://angular.io/guide/reactive-forms#dynamic-forms) below.

Just as a form control instance gives you control over a single input field, a form group instance tracks the form state of a group of form control instances (for example, a form). Each control in a form group instance is tracked by name when creating the form group. The following example shows how to manage multiple form control instances in a single group.

Generate a ProfileEditor component and import the [FormGroup](https://angular.io/api/forms/FormGroup) and [FormControl](https://angular.io/api/forms/FormControl) classes from the @angular/forms package.

content\_copyng generate component ProfileEditor

src/app/profile-editor/profile-editor.component.ts (imports)

content\_copyimport { [FormGroup](https://angular.io/api/forms/FormGroup), [FormControl](https://angular.io/api/forms/FormControl) } from '@angular/forms';

To add a form group to this component, take the following steps.

1. Create a [FormGroup](https://angular.io/api/forms/FormGroup) instance.
2. Associate the [FormGroup](https://angular.io/api/forms/FormGroup) model and view.
3. Save the form data.

**Create a FormGroup instance**

Create a property in the component class named profileForm and set the property to a new form group instance. To initialize the form group, provide the constructor with an object of named keys mapped to their control.

For the profile form, add two form control instances with the names firstName and lastName.

src/app/profile-editor/profile-editor.component.ts (form group)

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

import { [FormGroup](https://angular.io/api/forms/FormGroup), [FormControl](https://angular.io/api/forms/FormControl) } from '@angular/forms';

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

selector: 'app-profile-editor',

templateUrl: './profile-editor.component.html',

styleUrls: ['./profile-editor.component.css']

})

export class ProfileEditorComponent {

profileForm = new [FormGroup](https://angular.io/api/forms/FormGroup)({

firstName: new [FormControl](https://angular.io/api/forms/FormControl)(''),

lastName: new [FormControl](https://angular.io/api/forms/FormControl)(''),

});

}

The individual form controls are now collected within a group. A [FormGroup](https://angular.io/api/forms/FormGroup) instance provides its model value as an object reduced from the values of each control in the group. A form group instance has the same properties (such as value and untouched) and methods (such as setValue()) as a form control instance.

**Associate the FormGroup model and view**

A form group tracks the status and changes for each of its controls, so if one of the controls changes, the parent control also emits a new status or value change. The model for the group is maintained from its members. After you define the model, you must update the template to reflect the model in the view.

src/app/profile-editor/profile-editor.component.html (template form group)

content\_copy<form [formGroup]="profileForm">

<label for="first-name">First Name: </label>

<input id="first-name" type="text" [formControlName](https://angular.io/api/forms/FormControlName)="firstName">

<label for="last-name">Last Name: </label>

<input id="last-name" type="text" [formControlName](https://angular.io/api/forms/FormControlName)="lastName">

</form>

Note that just as a form group contains a group of controls, the profileForm [FormGroup](https://angular.io/api/forms/FormGroup) is bound to the form element with the [FormGroup](https://angular.io/api/forms/FormGroup) directive, creating a communication layer between the model and the form containing the inputs. The [formControlName](https://angular.io/api/forms/FormControlName) input provided by the [FormControlName](https://angular.io/api/forms/FormControlName) directive binds each individual input to the form control defined in [FormGroup](https://angular.io/api/forms/FormGroup). The form controls communicate with their respective elements. They also communicate changes to the form group instance, which provides the source of truth for the model value.

**Save form data**

The ProfileEditor component accepts input from the user, but in a real scenario you want to capture the form value and make available for further processing outside the component. The [FormGroup](https://angular.io/api/forms/FormGroup) directive listens for the submit event emitted by the form element and emits an ngSubmit event that you can bind to a callback function.

Add an ngSubmit event listener to the form tag with the onSubmit() callback method.

src/app/profile-editor/profile-editor.component.html (submit event)

content\_copy<form [formGroup]="profileForm" (ngSubmit)="onSubmit()">

The onSubmit() method in the ProfileEditor component captures the current value of profileForm. Use [EventEmitter](https://angular.io/api/core/EventEmitter) to keep the form encapsulated and to provide the form value outside the component. The following example uses console.warn to log a message to the browser console.

src/app/profile-editor/profile-editor.component.ts (submit method)

content\_copyonSubmit() {

// TODO: Use [EventEmitter](https://angular.io/api/core/EventEmitter) with form value

console.warn(this.profileForm.value);

}

The submit event is emitted by the form tag using the native DOM event. You trigger the event by clicking a button with submit type. This allows the user to press the **Enter** key to submit the completed form.

Use a button element to add a button to the bottom of the form to trigger the form submission.

src/app/profile-editor/profile-editor.component.html (submit button)

content\_copy<p>Complete the form to enable button.</p>

<button type="submit" [disabled]="!profileForm.valid">Submit</button>

**Note:** The button in the snippet above also has a disabled binding attached to it to disable the button when profileForm is invalid. You aren't performing any validation yet, so the button is always enabled. Basic form validation is covered in the [Validating form input](https://angular.io/guide/reactive-forms#basic-form-validation) section.

**Display the component**

To display the ProfileEditor component that contains the form, add it to a component template.

src/app/app.component.html (profile editor)

content\_copy<app-profile-editor></app-profile-editor>

ProfileEditor allows you to manage the form control instances for the firstName and lastName controls within the form group instance.



### **Creating nested form groups**

Form groups can accept both individual form control instances and other form group instances as children. This makes composing complex form models easier to maintain and logically group together.

When building complex forms, managing the different areas of information is easier in smaller sections. Using a nested form group instance allows you to break large forms groups into smaller, more manageable ones.

To make more complex forms, use the following steps.

1. Create a nested group.
2. Group the nested form in the template.

Some types of information naturally fall into the same group. A name and address are typical examples of such nested groups, and are used in the following examples.

**Create a nested group**

To create a nested group in profileForm, add a nested address element to the form group instance.

src/app/profile-editor/profile-editor.component.ts (nested form group)

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

import { [FormGroup](https://angular.io/api/forms/FormGroup), [FormControl](https://angular.io/api/forms/FormControl) } from '@angular/forms';

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

selector: 'app-profile-editor',

templateUrl: './profile-editor.component.html',

styleUrls: ['./profile-editor.component.css']

})

export class ProfileEditorComponent {

profileForm = new [FormGroup](https://angular.io/api/forms/FormGroup)({

firstName: new [FormControl](https://angular.io/api/forms/FormControl)(''),

lastName: new [FormControl](https://angular.io/api/forms/FormControl)(''),

address: new [FormGroup](https://angular.io/api/forms/FormGroup)({

street: new [FormControl](https://angular.io/api/forms/FormControl)(''),

city: new [FormControl](https://angular.io/api/forms/FormControl)(''),

state: new [FormControl](https://angular.io/api/forms/FormControl)(''),

zip: new [FormControl](https://angular.io/api/forms/FormControl)('')

})

});

}

In this example, address group combines the current firstName and lastName controls with the new street, city, state, and zip controls. Even though the address element in the form group is a child of the overall profileForm element in the form group, the same rules apply with value and status changes. Changes in status and value from the nested form group propagate to the parent form group, maintaining consistency with the overall model.

**Group the nested form in the template**

After you update the model in the component class, update the template to connect the form group instance and its input elements.

Add the address form group containing the street, city, state, and zip fields to the ProfileEditor template.

src/app/profile-editor/profile-editor.component.html (template nested form group)

content\_copy<div [formGroupName](https://angular.io/api/forms/FormGroupName)="address">

<h2>Address</h2>

<label for="street">Street: </label>

<input id="street" type="text" [formControlName](https://angular.io/api/forms/FormControlName)="street">

<label for="city">City: </label>

<input id="city" type="text" [formControlName](https://angular.io/api/forms/FormControlName)="city">

<label for="state">State: </label>

<input id="state" type="text" [formControlName](https://angular.io/api/forms/FormControlName)="state">

<label for="zip">Zip Code: </label>

<input id="zip" type="text" [formControlName](https://angular.io/api/forms/FormControlName)="zip">

</div>

The ProfileEditor form is displayed as one group, but the model is broken down further to represent the logical grouping areas.



**Tip** Display the value for the form group instance in the component template using the value property and [JsonPipe](https://angular.io/api/common/JsonPipe).

### **Updating parts of the data model**

When updating the value for a form group instance that contains multiple controls, you may only want to update parts of the model. This section covers how to update specific parts of a form control data model.

There are two ways to update the model value:

* Use the setValue() method to set a new value for an individual control. The setValue() method strictly adheres to the structure of the form group and replaces the entire value for the control.
* Use the patchValue() method to replace any properties defined in the object that have changed in the form model.

The strict checks of the setValue() method help catch nesting errors in complex forms, while patchValue() fails silently on those errors.

In ProfileEditorComponent, use the updateProfile method with the example below to update the first name and street address for the user.

src/app/profile-editor/profile-editor.component.ts (patch value)

content\_copyupdateProfile() {

this.profileForm.patchValue({

firstName: 'Nancy',

address: {

street: '123 Drew Street'

}

});

}

Simulate an update by adding a button to the template to update the user profile on demand.

src/app/profile-editor/profile-editor.component.html (update value)

content\_copy<button (click)="updateProfile()">Update Profile</button>

When a user clicks the button, the profileForm model is updated with new values for firstName and street. Notice that street is provided in an object inside the address property. This is necessary because the patchValue() method applies the update against the model structure. PatchValue() only updates properties that the form model defines.

## **Using the FormBuilder service to generate controls**

Creating form control instances manually can become repetitive when dealing with multiple forms. The [FormBuilder](https://angular.io/api/forms/FormBuilder) service provides convenient methods for generating controls.

Use the following steps to take advantage of this service.

1. Import the [FormBuilder](https://angular.io/api/forms/FormBuilder) class.
2. Inject the [FormBuilder](https://angular.io/api/forms/FormBuilder) service.
3. Generate the form contents.

The following examples show how to refactor the ProfileEditor component to use the form builder service to create form control and form group instances.

**Import the FormBuilder class**

Import the [FormBuilder](https://angular.io/api/forms/FormBuilder) class from the @angular/forms package.

src/app/profile-editor/profile-editor.component.ts (import)

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

**Inject the FormBuilder service**

The [FormBuilder](https://angular.io/api/forms/FormBuilder) service is an injectable provider that is provided with the reactive forms module. Inject this dependency by adding it to the component constructor.

src/app/profile-editor/profile-editor.component.ts (constructor)

content\_copyconstructor(private fb: [FormBuilder](https://angular.io/api/forms/FormBuilder)) { }

**Generate form controls**

The [FormBuilder](https://angular.io/api/forms/FormBuilder) service has three methods: control(), group(), and array(). These are factory methods for generating instances in your component classes including form controls, form groups, and form arrays.

Use the group method to create the profileForm controls.

src/app/profile-editor/profile-editor.component.ts (form builder)

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

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

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

selector: 'app-profile-editor',

templateUrl: './profile-editor.component.html',

styleUrls: ['./profile-editor.component.css']

})

export class ProfileEditorComponent {

profileForm = this.fb.group({

firstName: [''],

lastName: [''],

address: this.fb.group({

street: [''],

city: [''],

state: [''],

zip: ['']

}),

});

constructor(private fb: [FormBuilder](https://angular.io/api/forms/FormBuilder)) { }

}

In the example above, you use the group() method with the same object to define the properties in the model. The value for each control name is an array containing the initial value as the first item in the array.

**Tip** You can define the control with just the initial value, but if your controls need sync or async validation, add sync and async validators as the second and third items in the array.

Compare using the form builder to creating the instances manually.

src/app/profile-editor/profile-editor.component.ts (instances)

src/app/profile-editor/profile-editor.component.ts (form builder)

content\_copyprofileForm = new [FormGroup](https://angular.io/api/forms/FormGroup)({

firstName: new [FormControl](https://angular.io/api/forms/FormControl)(''),

lastName: new [FormControl](https://angular.io/api/forms/FormControl)(''),

address: new [FormGroup](https://angular.io/api/forms/FormGroup)({

street: new [FormControl](https://angular.io/api/forms/FormControl)(''),

city: new [FormControl](https://angular.io/api/forms/FormControl)(''),

state: new [FormControl](https://angular.io/api/forms/FormControl)(''),

zip: new [FormControl](https://angular.io/api/forms/FormControl)('')

})

});

## **Validating form input**

Form validation is used to ensure that user input is complete and correct. This section covers adding a single validator to a form control and displaying the overall form status. Form validation is covered more extensively in the [Form Validation](https://angular.io/guide/form-validation) guide.

Use the following steps to add form validation.

1. Import a validator function in your form component.
2. Add the validator to the field in the form.
3. Add logic to handle the validation status.

The most common validation is making a field required. The following example shows how to add a required validation to the firstName control and display the result of validation.

**Import a validator function**

Reactive forms include a set of validator functions for common use cases. These functions receive a control to validate against and return an error object or a null value based on the validation check.

Import the [Validators](https://angular.io/api/forms/Validators) class from the @angular/forms package.

src/app/profile-editor/profile-editor.component.ts (import)

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

**Make a field required**

In the ProfileEditor component, add the Validators.required static method as the second item in the array for the firstName control.

src/app/profile-editor/profile-editor.component.ts (required validator)

content\_copyprofileForm = this.fb.group({

firstName: ['', Validators.required],

lastName: [''],

address: this.fb.group({

street: [''],

city: [''],

state: [''],

zip: ['']

}),

});

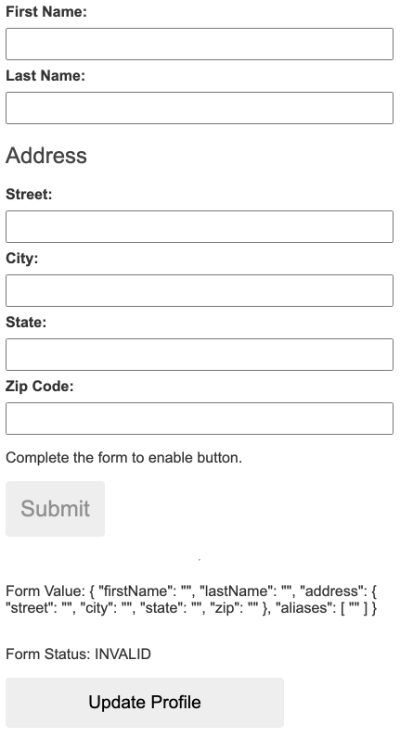
**Display form status**

When you add a required field to the form control, its initial status is invalid. This invalid status propagates to the parent form group element, making its status invalid. Access the current status of the form group instance through its status property.

Display the current status of profileForm using interpolation.

src/app/profile-editor/profile-editor.component.html (display status)

content\_copy<p>[Form](https://angular.io/api/forms/Form) Status: {{ profileForm.status }}</p>



The **Submit** button is disabled because profileForm is invalid due to the required firstName form control. After you fill out the firstName input, the form becomes valid and the **Submit** button is enabled.

For more on form validation, visit the [Form Validation](https://angular.io/guide/form-validation) guide.

## **Creating dynamic forms**

[FormArray](https://angular.io/api/forms/FormArray) is an alternative to [FormGroup](https://angular.io/api/forms/FormGroup) for managing any number of unnamed controls. As with form group instances, you can dynamically insert and remove controls from form array instances, and the form array instance value and validation status is calculated from its child controls. However, you don't need to define a key for each control by name, so this is a great option if you don't know the number of child values in advance.

To define a dynamic form, take the following steps.

1. Import the [FormArray](https://angular.io/api/forms/FormArray) class.
2. Define a [FormArray](https://angular.io/api/forms/FormArray) control.
3. Access the [FormArray](https://angular.io/api/forms/FormArray) control with a getter method.
4. Display the form array in a template.

The following example shows you how to manage an array of aliases in ProfileEditor.

**Import the FormArray class**

Import the [FormArray](https://angular.io/api/forms/FormArray) class from @angular/forms to use for type information. The [FormBuilder](https://angular.io/api/forms/FormBuilder) service is ready to create a [FormArray](https://angular.io/api/forms/FormArray) instance.

src/app/profile-editor/profile-editor.component.ts (import)

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

**Define a FormArray control**

You can initialize a form array with any number of controls, from zero to many, by defining them in an array. Add an aliases property to the form group instance for profileForm to define the form array.

Use the [FormBuilder.array()](https://angular.io/api/forms/FormBuilder#array) method to define the array, and the [FormBuilder.control()](https://angular.io/api/forms/FormBuilder#control) method to populate the array with an initial control.

src/app/profile-editor/profile-editor.component.ts (aliases form array)

content\_copyprofileForm = this.fb.group({

firstName: ['', Validators.required],

lastName: [''],

address: this.fb.group({

street: [''],

city: [''],

state: [''],

zip: ['']

}),

aliases: this.fb.array([

this.fb.control('')

])

});

The aliases control in the form group instance is now populated with a single control until more controls are added dynamically.

**Access the FormArray control**

A getter provides easy access to the aliases in the form array instance compared to repeating the profileForm.get() method to get each instance. The form array instance represents an undefined number of controls in an array. It's convenient to access a control through a getter, and this approach is easy to repeat for additional controls.

Use the getter syntax to create an aliases class property to retrieve the alias's form array control from the parent form group.

src/app/profile-editor/profile-editor.component.ts (aliases getter)

content\_copyget aliases() {

return this.profileForm.get('aliases') as [FormArray](https://angular.io/api/forms/FormArray);

}

**Note:** Because the returned control is of the type [AbstractControl](https://angular.io/api/forms/AbstractControl), you need to provide an explicit type to access the method syntax for the form array instance.

Define a method to dynamically insert an alias control into the alias's form array. The [FormArray.push()](https://angular.io/api/forms/FormArray#push) method inserts the control as a new item in the array.

src/app/profile-editor/profile-editor.component.ts (add alias)

content\_copyaddAlias() {

this.aliases.push(this.fb.control(''));

}

In the template, each control is displayed as a separate input field.

**Display the form array in the template**

To attach the aliases from your form model, you must add it to the template. Similar to the [formGroupName](https://angular.io/api/forms/FormGroupName) input provided by FormGroupNameDirective, [formArrayName](https://angular.io/api/forms/FormArrayName) binds communication from the form array instance to the template with FormArrayNameDirective.

Add the template HTML below after the <div> closing the [formGroupName](https://angular.io/api/forms/FormGroupName) element.

src/app/profile-editor/profile-editor.component.html (aliases form array template)

content\_copy<div [formArrayName](https://angular.io/api/forms/FormArrayName)="aliases">

<h2>Aliases</h2>

<button (click)="addAlias()">+ Add another alias</button>

<div \*[ngFor](https://angular.io/api/common/NgForOf)="let alias of aliases.controls; let i=index">

<!-- The repeated alias template -->

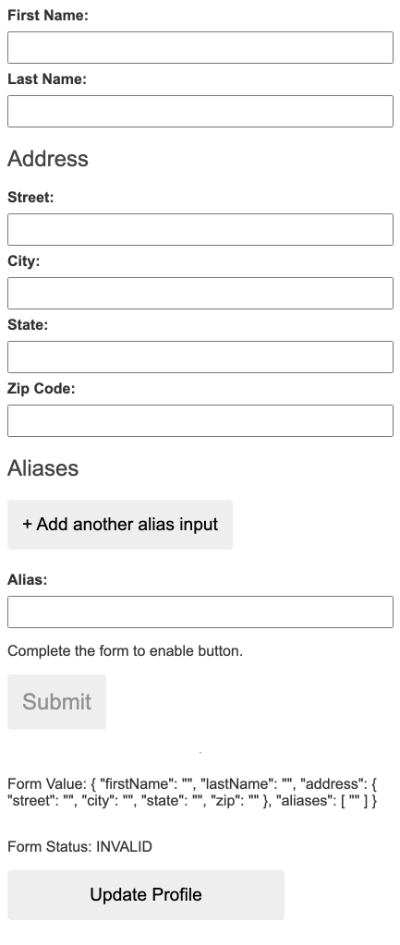
<label for="alias-{{ i }}">Alias:</label>

<input id="alias-{{ i }}" type="text" [[formControlName](https://angular.io/api/forms/FormControlName)]="i">

</div>

</div>

The \*[ngFor](https://angular.io/api/common/NgForOf) directive iterates over each form control instance provided by the aliases form array instance. Because form array elements are unnamed, you assign the index to the i variable and pass it to each control to bind it to the [formControlName](https://angular.io/api/forms/FormControlName) input.



Each time a new alias instance is added, the new form array instance is provided its control based on the index. This allows you to track each individual control when calculating the status and value of the root control.

**Add an alias**

Initially, the form contains one Alias field. To add another field, click the **Add Alias** button. You can also validate the array of aliases reported by the form model displayed by [Form](https://angular.io/api/forms/Form) Value at the bottom of the template.

**Note:** Instead of a form control instance for each alias, you can compose another form group instance with additional fields. The process of defining a control for each item is the same.

## **Reactive forms API summary**

The following table lists the base classes and services used to create and manage reactive form controls. For complete syntax details, see the API reference documentation for the [Forms package](https://angular.io/api/forms).

#### Classes

|  |  |
| --- | --- |
| **Class** | **Description** |
| [AbstractControl](https://angular.io/api/forms/AbstractControl) | The abstract base class for the concrete form control classes [FormControl](https://angular.io/api/forms/FormControl), [FormGroup](https://angular.io/api/forms/FormGroup), and [FormArray](https://angular.io/api/forms/FormArray). It provides their common behaviors and properties. |
| [FormControl](https://angular.io/api/forms/FormControl) | Manages the value and validity status of an individual form control. It corresponds to an HTML form control such as <input> or <select>. |
| [FormGroup](https://angular.io/api/forms/FormGroup) | Manages the value and validity state of a group of [AbstractControl](https://angular.io/api/forms/AbstractControl) instances. The group's properties include its child controls. The top-level form in your component is [FormGroup](https://angular.io/api/forms/FormGroup). |
| [FormArray](https://angular.io/api/forms/FormArray) | Manages the value and validity state of a numerically indexed array of [AbstractControl](https://angular.io/api/forms/AbstractControl) instances. |
| [FormBuilder](https://angular.io/api/forms/FormBuilder) | An injectable service that provides factory methods for creating control instances. |

#### Directives

|  |  |
| --- | --- |
| **Directive** | **Description** |
| [FormControlDirective](https://angular.io/api/forms/FormControlDirective) | Syncs a standalone [FormControl](https://angular.io/api/forms/FormControl) instance to a form control element. |
| [FormControlName](https://angular.io/api/forms/FormControlName) | Syncs [FormControl](https://angular.io/api/forms/FormControl) in an existing [FormGroup](https://angular.io/api/forms/FormGroup) instance to a form control element by name. |
| [FormGroupDirective](https://angular.io/api/forms/FormGroupDirective) | Syncs an existing [FormGroup](https://angular.io/api/forms/FormGroup) instance to a DOM element. |
| [FormGroupName](https://angular.io/api/forms/FormGroupName) | Syncs a nested [FormGroup](https://angular.io/api/forms/FormGroup) instance to a DOM element. |
| [FormArrayName](https://angular.io/api/forms/FormArrayName) | Syncs a nested [FormArray](https://angular.io/api/forms/FormArray) instance to a DOM element. |

# Validating form input

You can improve overall data quality by validating user input for accuracy and completeness. This page shows how to validate user input from the UI and display useful validation messages, in both reactive and template-driven forms.

**Prerequisites**

Before reading about form validation, you should have a basic understanding of the following.

* [TypeScript](https://www.typescriptlang.org/) and HTML5 programming.
* Fundamental concepts of [Angular application design](https://angular.io/guide/architecture).
* The [two types of forms that Angular supports](https://angular.io/guide/forms-overview).
* Basics of either [Template-driven Forms](https://angular.io/guide/forms) or [Reactive Forms](https://angular.io/guide/reactive-forms).

Get the complete example code for the reactive and template-driven forms used here to illustrate form validation. Run the [live example](https://angular.io/generated/live-examples/form-validation/stackblitz.html) / [download example](https://angular.io/generated/zips/form-validation/form-validation.zip).

## **Validating input in template-driven forms**

To add validation to a template-driven form, you add the same validation attributes as you would with [native HTML form validation](https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/HTML5/Constraint_validation). Angular uses directives to match these attributes with validator functions in the framework.

Every time the value of a form control changes, Angular runs validation and generates either a list of validation errors that results in an INVALID status, or null, which results in a VALID status.

You can then inspect the control's state by exporting [ngModel](https://angular.io/api/forms/NgModel) to a local template variable. The following example exports [NgModel](https://angular.io/api/forms/NgModel) into a variable called name:

template/hero-form-template.component.html (name)

content\_copy<input type="text" id="name" name="name" class="form-control"

required [minlength](https://angular.io/api/forms/MinLengthValidator)="4" appForbiddenName="bob"

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

<div \*[ngIf](https://angular.io/api/common/NgIf)="name.invalid && (name.dirty || name.touched)"

class="alert">

<div \*[ngIf](https://angular.io/api/common/NgIf)="name.errors?.required">

Name is required.

</div>

<div \*[ngIf](https://angular.io/api/common/NgIf)="name.errors?.minlength">

Name must be at least 4 characters long.

</div>

<div \*[ngIf](https://angular.io/api/common/NgIf)="name.errors?.forbiddenName">

Name cannot be Bob.

</div>

</div>

Notice the following features illustrated by the example.

* The <input> element carries the HTML validation attributes: required and [minlength](https://angular.io/api/forms/MinLengthValidator). It also carries a custom validator directive, forbiddenName. For more information, see the [Custom validators](https://angular.io/guide/form-validation#custom-validators) section.
* #name="[ngModel](https://angular.io/api/forms/NgModel)" exports [NgModel](https://angular.io/api/forms/NgModel) into a local variable called name. [NgModel](https://angular.io/api/forms/NgModel) mirrors many of the properties of its underlying [FormControl](https://angular.io/api/forms/FormControl) instance, so you can use this in the template to check for control states such as valid and dirty. For a full list of control properties, see the [AbstractControl](https://angular.io/api/forms/AbstractControl) API reference.
  + The \*[ngIf](https://angular.io/api/common/NgIf) on the <div> element reveals a set of nested message divs but only if the name is invalid and the control is either dirty or touched.
  + Each nested <div> can present a custom message for one of the possible validation errors. There are messages for required, [minlength](https://angular.io/api/forms/MinLengthValidator), and forbiddenName.

To prevent the validator from displaying errors before the user has a chance to edit the form, you should check for either the dirty or touched states in a control.

* When the user changes the value in the watched field, the control is marked as "dirty".
* When the user blurs the form control element, the control is marked as "touched".

## **Validating input in reactive forms**

In a reactive form, the source of truth is the component class. Instead of adding validators through attributes in the template, you add validator functions directly to the form control model in the component class. Angular then calls these functions whenever the value of the control changes.

### **Validator functions**

Validator functions can be either synchronous or asynchronous.

* **Sync validators**: Synchronous functions that take a control instance and immediately return either a set of validation errors or null. You can pass these in as the second argument when you instantiate a [FormControl](https://angular.io/api/forms/FormControl).
* **Async validators**: Asynchronous functions that take a control instance and return a Promise or Observable that later emits a set of validation errors or null. You can pass these in as the third argument when you instantiate a [FormControl](https://angular.io/api/forms/FormControl).

For performance reasons, Angular only runs async validators if all sync validators pass. Each must complete before errors are set.

### **Built-in validator functions**

You can choose to [write your own validator functions](https://angular.io/guide/form-validation#custom-validators), or you can use some of Angular's built-in validators.

The same built-in validators that are available as attributes in template-driven forms, such as required and [minlength](https://angular.io/api/forms/MinLengthValidator), are all available to use as functions from the [Validators](https://angular.io/api/forms/Validators) class. For a full list of built-in validators, see the [Validators](https://angular.io/api/forms/Validators) API reference.

To update the hero form to be a reactive form, you can use some of the same built-in validators—this time, in function form, as in the following example.

reactive/hero-form-reactive.component.ts (validator functions)

content\_copyngOnInit(): void {

this.heroForm = new [FormGroup](https://angular.io/api/forms/FormGroup)({

name: new [FormControl](https://angular.io/api/forms/FormControl)(this.hero.name, [

Validators.required,

Validators.minLength(4),

forbiddenNameValidator(/bob/i) // <-- Here's how you pass in the custom validator.

]),

alterEgo: new [FormControl](https://angular.io/api/forms/FormControl)(this.hero.alterEgo),

power: new [FormControl](https://angular.io/api/forms/FormControl)(this.hero.power, Validators.required)

});

}

get name() { return this.heroForm.get('name'); }

get power() { return this.heroForm.get('power'); }

In this example, the name control sets up two built-in validators—Validators.required and Validators.minLength(4)—and one custom validator, forbiddenNameValidator. (For more details see [custom validators](https://angular.io/guide/form-validation#custom-validators) below.)

All of these validators are synchronous, so they are passed as the second argument. Notice that you can support multiple validators by passing the functions in as an array.

This example also adds a few getter methods. In a reactive form, you can always access any form control through the get method on its parent group, but sometimes it's useful to define getters as shorthand for the template.

If you look at the template for the name input again, it is fairly similar to the template-driven example.

reactive/hero-form-reactive.component.html (name with error msg)

content\_copy<input type="text" id="name" class="form-control"

[formControlName](https://angular.io/api/forms/FormControlName)="name" required>

<div \*[ngIf](https://angular.io/api/common/NgIf)="name.invalid && (name.dirty || name.touched)"

class="alert alert-danger">

<div \*[ngIf](https://angular.io/api/common/NgIf)="name.errors?.required">

Name is required.

</div>

<div \*[ngIf](https://angular.io/api/common/NgIf)="name.errors?.minlength">

Name must be at least 4 characters long.

</div>

<div \*[ngIf](https://angular.io/api/common/NgIf)="name.errors?.forbiddenName">

Name cannot be Bob.

</div>

</div>

This form differs from the template-driven version in that it no longer exports any directives. Instead, it uses the name getter defined in the component class.

Notice that the required attribute is still present in the template. Although it's not necessary for validation, it should be retained to for accessibility purposes.

## **Defining custom validators**

The built-in validators don't always match the exact use case of your application, so you sometimes need to create a custom validator.

Consider the forbiddenNameValidator function from previous [reactive-form examples](https://angular.io/guide/form-validation#reactive-component-class). Here's what the definition of that function looks like.

shared/forbidden-name.directive.ts (forbiddenNameValidator)

content\_copy/\*\* A hero's name can't match the given regular expression \*/

export function forbiddenNameValidator(nameRe: RegExp): [ValidatorFn](https://angular.io/api/forms/ValidatorFn) {

return (control: [AbstractControl](https://angular.io/api/forms/AbstractControl)): [ValidationErrors](https://angular.io/api/forms/ValidationErrors) | null => {

const forbidden = nameRe.test(control.value);

return forbidden ? {forbiddenName: {value: control.value}} : null;

};

}

The function is a factory that takes a regular expression to detect a specific forbidden name and returns a validator function.

In this sample, the forbidden name is "bob", so the validator will reject any hero name containing "bob". Elsewhere it could reject "alice" or any name that the configuring regular expression matches.

The forbiddenNameValidator factory returns the configured validator function. That function takes an Angular control object and returns either null if the control value is valid or a validation error object. The validation error object typically has a property whose name is the validation key, 'forbiddenName', and whose value is an arbitrary dictionary of values that you could insert into an error message, {name}.

Custom async validators are similar to sync validators, but they must instead return a Promise or observable that later emits null or a validation error object. In the case of an observable, the observable must complete, at which point the form uses the last value emitted for validation.

### **Adding custom validators to reactive forms**

In reactive forms, add a custom validator by passing the function directly to the [FormControl](https://angular.io/api/forms/FormControl).

reactive/hero-form-reactive.component.ts (validator functions)

content\_copythis.heroForm = new [FormGroup](https://angular.io/api/forms/FormGroup)({

name: new [FormControl](https://angular.io/api/forms/FormControl)(this.hero.name, [

Validators.required,

Validators.minLength(4),

forbiddenNameValidator(/bob/i) // <-- Here's how you pass in the custom validator.

]),

alterEgo: new [FormControl](https://angular.io/api/forms/FormControl)(this.hero.alterEgo),

power: new [FormControl](https://angular.io/api/forms/FormControl)(this.hero.power, Validators.required)

});

### **Adding custom validators to template-driven forms**

In template-driven forms, add a directive to the template, where the directive wraps the validator function. For example, the corresponding ForbiddenValidatorDirective serves as a wrapper around the forbiddenNameValidator.

Angular recognizes the directive's role in the validation process because the directive registers itself with the [NG\_VALIDATORS](https://angular.io/api/forms/NG_VALIDATORS) provider, as shown in the following example. [NG\_VALIDATORS](https://angular.io/api/forms/NG_VALIDATORS) is a predefined provider with an extensible collection of validators.

shared/forbidden-name.directive.ts (providers)

content\_copyproviders: [{provide: [NG\_VALIDATORS](https://angular.io/api/forms/NG_VALIDATORS), useExisting: ForbiddenValidatorDirective, multi: true}]

The directive class then implements the [Validator](https://angular.io/api/forms/Validator) interface, so that it can easily integrate with Angular forms. Here is the rest of the directive to help you get an idea of how it all comes together.

shared/forbidden-name.directive.ts (directive)

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

selector: '[appForbiddenName]',

providers: [{provide: [NG\_VALIDATORS](https://angular.io/api/forms/NG_VALIDATORS), useExisting: ForbiddenValidatorDirective, multi: true}]

})

export class ForbiddenValidatorDirective implements [Validator](https://angular.io/api/forms/Validator) {

@[Input](https://angular.io/api/core/Input)('appForbiddenName') forbiddenName = '';

validate(control: [AbstractControl](https://angular.io/api/forms/AbstractControl)): [ValidationErrors](https://angular.io/api/forms/ValidationErrors) | null {

return this.forbiddenName ? forbiddenNameValidator(new RegExp(this.forbiddenName, 'i'))(control)

: null;

}

}

Once the ForbiddenValidatorDirective is ready, you can add its selector, appForbiddenName, to any input element to activate it. For example:

template/hero-form-template.component.html (forbidden-name-input)

content\_copy<input type="text" id="name" name="name" class="form-control"

required [minlength](https://angular.io/api/forms/MinLengthValidator)="4" appForbiddenName="bob"

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

Notice that the custom validation directive is instantiated with useExisting rather than useClass. The registered validator must be this instance of the ForbiddenValidatorDirective—the instance in the form with its forbiddenName property bound to “bob".

If you were to replace useExisting with useClass, then you’d be registering a new class instance, one that doesn’t have a forbiddenName.

## **Control status CSS classes**

Angular automatically mirrors many control properties onto the form control element as CSS classes. You can use these classes to style form control elements according to the state of the form. The following classes are currently supported.

* .ng-valid
* .ng-invalid
* .ng-pending
* .ng-pristine
* .ng-dirty
* .ng-untouched
* .ng-touched
* .ng-submitted (enclosing form element only)

In the following example, the hero form uses the .ng-valid and .ng-invalid classes to set the color of each form control's border.

forms.css (status classes)

content\_copy.ng-valid[required], .ng-valid.required {

border-left: 5px solid #42A948; /\* green \*/

}

.ng-invalid:not(form) {

border-left: 5px solid #a94442; /\* red \*/

}

.alert div {

background-color: #fed3d3;

color: #820000;

padding: 1rem;

margin-bottom: 1rem;

}

.form-group {

margin-bottom: 1rem;

}

label {

display: block;

margin-bottom: .5rem;

}

select {

width: 100%;

padding: .5rem;

}

## **Cross-field validation**

A cross-field validator is a [custom validator](https://angular.io/guide/form-validation#custom-validators) that compares the values of different fields in a form and accepts or rejects them in combination. For example, you might have a form that offers mutually incompatible options, so that if the user can choose A or B, but not both. Some field values might also depend on others; a user might be allowed to choose B only if A is also chosen.

The following cross validation examples show how to do the following:

* Validate reactive or template-based form input based on the values of two sibling controls,
* Show a descriptive error message after the user interacted with the form and the validation failed.

The examples use cross-validation to ensure that heroes do not reveal their true identities by filling out the Hero Form. The validators do this by checking that the hero names and alter egos do not match.

### **Adding cross-validation to reactive forms**

The form has the following structure:

content\_copyconst heroForm = new [FormGroup](https://angular.io/api/forms/FormGroup)({

'name': new [FormControl](https://angular.io/api/forms/FormControl)(),

'alterEgo': new [FormControl](https://angular.io/api/forms/FormControl)(),

'power': new [FormControl](https://angular.io/api/forms/FormControl)()

});

Notice that the name and alterEgo are sibling controls. To evaluate both controls in a single custom validator, you must perform the validation in a common ancestor control: the [FormGroup](https://angular.io/api/forms/FormGroup). You query the [FormGroup](https://angular.io/api/forms/FormGroup) for its child controls so that you can compare their values.

To add a validator to the [FormGroup](https://angular.io/api/forms/FormGroup), pass the new validator in as the second argument on creation.

content\_copyconst heroForm = new [FormGroup](https://angular.io/api/forms/FormGroup)({

'name': new [FormControl](https://angular.io/api/forms/FormControl)(),

'alterEgo': new [FormControl](https://angular.io/api/forms/FormControl)(),

'power': new [FormControl](https://angular.io/api/forms/FormControl)()

}, { validators: identityRevealedValidator });

The validator code is as follows.

shared/identity-revealed.directive.ts

content\_copy/\*\* A hero's name can't match the hero's alter ego \*/

export const identityRevealedValidator: [ValidatorFn](https://angular.io/api/forms/ValidatorFn) = (control: [AbstractControl](https://angular.io/api/forms/AbstractControl)): [ValidationErrors](https://angular.io/api/forms/ValidationErrors) | null => {

const name = control.get('name');

const alterEgo = control.get('alterEgo');

return name && alterEgo && name.value === alterEgo.value ? { identityRevealed: true } : null;

};

The identity validator implements the [ValidatorFn](https://angular.io/api/forms/ValidatorFn) interface. It takes an Angular control object as an argument and returns either null if the form is valid, or [ValidationErrors](https://angular.io/api/forms/ValidationErrors) otherwise.

The validator retrieves the child controls by calling the [FormGroup](https://angular.io/api/forms/FormGroup)'s [get](https://angular.io/api/forms/AbstractControl#get) method, then compares the values of the name and alterEgo controls.

If the values do not match, the hero's identity remains secret, both are valid, and the validator returns null. If they do match, the hero's identity is revealed and the validator must mark the form as invalid by returning an error object.

To provide better user experience, the template shows an appropriate error message when the form is invalid.

reactive/hero-form-template.component.html

content\_copy<div \*[ngIf](https://angular.io/api/common/NgIf)="heroForm.errors?.identityRevealed && (heroForm.touched || heroForm.dirty)" class="cross-validation-error-message alert alert-danger">

Name cannot match alter ego.

</div>

This \*[ngIf](https://angular.io/api/common/NgIf) displays the error if the [FormGroup](https://angular.io/api/forms/FormGroup) has the cross validation error returned by the identityRevealed validator, but only if the user has finished [interacting with the form](https://angular.io/guide/form-validation#dirty-or-touched).

### **Adding cross-validation to template-driven forms**

For a template-driven form, you must create a directive to wrap the validator function. You provide that directive as the validator using the [NG\_VALIDATORS token](https://angular.io/guide/form-validation#adding-to-template-driven-forms), as shown in the following example.

shared/identity-revealed.directive.ts

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

selector: '[appIdentityRevealed]',

providers: [{ provide: [NG\_VALIDATORS](https://angular.io/api/forms/NG_VALIDATORS), useExisting: IdentityRevealedValidatorDirective, multi: true }]

})

export class IdentityRevealedValidatorDirective implements [Validator](https://angular.io/api/forms/Validator) {

validate(control: [AbstractControl](https://angular.io/api/forms/AbstractControl)): [ValidationErrors](https://angular.io/api/forms/ValidationErrors) | null {

return identityRevealedValidator(control);

}

}

You must add the new directive to the HTML template. Because the validator must be registered at the highest level in the form, the following template puts the directive on the form tag.

template/hero-form-template.component.html

content\_copy<form #heroForm="[ngForm](https://angular.io/api/forms/NgForm)" appIdentityRevealed>

To provide better user experience, we show an appropriate error message when the form is invalid.

template/hero-form-template.component.html

content\_copy<div \*[ngIf](https://angular.io/api/common/NgIf)="heroForm.errors?.identityRevealed && (heroForm.touched || heroForm.dirty)" class="cross-validation-error-message alert">

Name cannot match alter ego.

</div>

This is the same in both template-driven and reactive forms.

## **Creating asynchronous validators**

Asynchronous validators implement the [AsyncValidatorFn](https://angular.io/api/forms/AsyncValidatorFn) and [AsyncValidator](https://angular.io/api/forms/AsyncValidator) interfaces. These are very similar to their synchronous counterparts, with the following differences.

* The validate() functions must return a Promise or an observable,
* The observable returned must be finite, meaning it must complete at some point. To convert an infinite observable into a finite one, pipe the observable through a filtering operator such as first, last, take, or takeUntil.

Asynchronous validation happens after the synchronous validation, and is performed only if the synchronous validation is successful. This check allows forms to avoid potentially expensive async validation processes (such as an HTTP request) if the more basic validation methods have already found invalid input.

After asynchronous validation begins, the form control enters a pending state. You can inspect the control's pending property and use it to give visual feedback about the ongoing validation operation.

A common UI pattern is to show a spinner while the async validation is being performed. The following example shows how to achieve this in a template-driven form.

content\_copy<input [([ngModel](https://angular.io/api/forms/NgModel))]="name" #model="[ngModel](https://angular.io/api/forms/NgModel)" appSomeAsyncValidator>

<app-spinner \*[ngIf](https://angular.io/api/common/NgIf)="model.pending"></app-spinner>

### **Implementing a custom async validator**

In the following example, an async validator ensures that heroes pick an alter ego that is not already taken. New heroes are constantly enlisting and old heroes are leaving the service, so the list of available alter egos cannot be retrieved ahead of time. To validate the potential alter ego entry, the validator must initiate an asynchronous operation to consult a central database of all currently enlisted heroes.

The following code create the validator class, UniqueAlterEgoValidator, which implements the [AsyncValidator](https://angular.io/api/forms/AsyncValidator) interface.

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

export class UniqueAlterEgoValidator implements [AsyncValidator](https://angular.io/api/forms/AsyncValidator) {

constructor(private heroesService: HeroesService) {}

validate(

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

): Promise<[ValidationErrors](https://angular.io/api/forms/ValidationErrors) | null> | Observable<[ValidationErrors](https://angular.io/api/forms/ValidationErrors) | null> {

return this.heroesService.isAlterEgoTaken(ctrl.value).pipe(

map(isTaken => (isTaken ? { uniqueAlterEgo: true } : null)),

catchError(() => of(null))

);

}

}

The constructor injects the HeroesService, which defines the following interface.

content\_copyinterface HeroesService {

isAlterEgoTaken: (alterEgo: string) => Observable<boolean>;

}

In a real world application, the HeroesService would be responsible for making an HTTP request to the hero database to check if the alter ego is available. From the validator's point of view, the actual implementation of the service is not important, so the example can just code against the HeroesService interface.

As the validation begins, the UniqueAlterEgoValidator delegates to the HeroesService isAlterEgoTaken() method with the current control value. At this point the control is marked as pending and remains in this state until the observable chain returned from the validate() method completes.

The isAlterEgoTaken() method dispatches an HTTP request that checks if the alter ego is available, and returns Observable<boolean> as the result. The validate() method pipes the response through the map operator and transforms it into a validation result.

The method then, like any validator, returns null if the form is valid, and [ValidationErrors](https://angular.io/api/forms/ValidationErrors) if it is not. This validator handles any potential errors with the catchError operator. In this case, the validator treats the isAlterEgoTaken() error as a successful validation, because failure to make a validation request does not necessarily mean that the alter ego is invalid. You could handle the error differently and return the ValidationError object instead.

After some time passes, the observable chain completes and the asynchronous validation is done. The pending flag is set to false, and the form validity is updated.

### **Optimizing performance of async validators**

By default, all validators run after every form value change. With synchronous validators, this does not normally have a noticeable impact on application performance. Async validators, however, commonly perform some kind of HTTP request to validate the control. Dispatching an HTTP request after every keystroke could put a strain on the backend API, and should be avoided if possible.

You can delay updating the form validity by changing the updateOn property from change (default) to submit or blur.

With template-driven forms, set the property in the template.

content\_copy<input [([ngModel](https://angular.io/api/forms/NgModel))]="name" [ngModelOptions]="{updateOn: 'blur'}">

With reactive forms, set the property in the [FormControl](https://angular.io/api/forms/FormControl) instance.

content\_copynew [FormControl](https://angular.io/api/forms/FormControl)('', {updateOn: 'blur'});

## **Interaction with native HTML form validation**

By default, Angular disables [native HTML form validation](https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Constraint_validation) by adding the novalidate attribute on the enclosing <form> and uses directives to match these attributes with validator functions in the framework. If you want to use native validation **in combination** with Angular-based validation, you can re-enable it with the ngNativeValidate directive. See the [API docs](https://angular.io/api/forms/NgForm#native-dom-validation-ui) for details.

# Building dynamic forms

Many forms, such as questionaires, can be very similar to one another in format and intent. To make it faster and easier to generate different versions of such a form, you can create a dynamic form template based on metadata that describes the business object model. You can then use the template to generate new forms automatically, according to changes in the data model.

The technique is particularly useful when you have a type of form whose content must change frequently to meet rapidly changing business and regulatory requirements. A typical use case is a questionaire. You might need to get input from users in different contexts. The format and style of the forms a user sees should remain constant, while the actual questions you need to ask vary with the context.

In this tutorial you will build a dynamic form that presents a basic questionaire. You will build an online application for heroes seeking employment. The agency is constantly tinkering with the application process, but by using the dynamic form you can create the new forms on the fly without changing the application code.

The tutorial walks you through the following steps.

1. Enable reactive forms for a project.
2. Establish a data model to represent form controls.
3. Populate the model with sample data.
4. Develop a component to create form controls dynamically.

The form you create uses input validation and styling to improve the user experience. It has a Submit button that is only enabled when all user input is valid, and flags invalid input with color coding and error messages.

The basic version can evolve to support a richer variety of questions, more graceful rendering, and superior user experience.

See the [live example](https://angular.io/generated/live-examples/dynamic-form/stackblitz.html) / [download example](https://angular.io/generated/zips/dynamic-form/dynamic-form.zip).

## **Prerequisites**

Before doing this tutorial, you should have a basic understanding to the following.

* [TypeScript](https://www.typescriptlang.org/) and HTML5 programming.
* Fundamental concepts of [Angular app design](https://angular.io/guide/architecture).
* Basic knowledge of [reactive forms](https://angular.io/guide/reactive-forms).

## **Enable reactive forms for your project**

Dynamic forms are based on reactive forms. To give the application access reactive forms directives, the [root module](https://angular.io/guide/bootstrapping) imports [ReactiveFormsModule](https://angular.io/api/forms/ReactiveFormsModule) from the @angular/forms library.

The following code from the example shows the setup in the root module.

app.module.ts

main.ts

content\_copyimport { [BrowserModule](https://angular.io/api/platform-browser/BrowserModule) } from '@angular/platform-browser';

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

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

import { AppComponent } from './app.component';

import { DynamicFormComponent } from './dynamic-form.component';

import { DynamicFormQuestionComponent } from './dynamic-form-question.component';

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

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

declarations: [ AppComponent, DynamicFormComponent, DynamicFormQuestionComponent ],

bootstrap: [ AppComponent ]

})

export class AppModule {

constructor() {

}

}

## **Create a form object model**

A dynamic form requires an object model that can describe all scenarios needed by the form functionality. The example hero-application form is a set of questions—that is, each control in the form must ask a question and accept an answer.

The data model for this type of form must represent a question. The example includes the DynamicFormQuestionComponent, which defines a question as the fundamental object in the model.

The following QuestionBase is a base class for a set of controls that can represent the question and its answer in the form.

src/app/question-base.ts

content\_copyexport class QuestionBase<T> {

value: T|undefined;

key: string;

label: string;

required: boolean;

order: number;

controlType: string;

type: string;

options: {key: string, value: string}[];

constructor(options: {

value?: T;

key?: string;

label?: string;

required?: boolean;

order?: number;

controlType?: string;

type?: string;

options?: {key: string, value: string}[];

} = {}) {

this.value = options.value;

this.key = options.key || '';

this.label = options.label || '';

this.required = !!options.required;

this.order = options.order === undefined ? 1 : options.order;

this.controlType = options.controlType || '';

this.type = options.type || '';

this.options = options.options || [];

}

}

### **Define control classes**

From this base, the example derives two new classes, TextboxQuestion and DropdownQuestion, that represent different control types. When you create the form template in the next step, you will instantiate these specific question types in order to render the appropriate controls dynamically.

* The TextboxQuestion control type presents a question and allows users to enter input.

src/app/question-textbox.ts

content\_copyimport { QuestionBase } from './question-base';

export class TextboxQuestion extends QuestionBase<string> {

controlType = 'textbox';

}

The TextboxQuestion control type will be represented in a form template using an <input> element. The type attribute of the element will be defined based on the type field specified in the options argument (for example text, [email](https://angular.io/api/forms/EmailValidator), url).

* The DropdownQuestion control presents a list of choices in a select box.

src/app/question-dropdown.ts

content\_copyimport { QuestionBase } from './question-base';

export class DropdownQuestion extends QuestionBase<string> {

controlType = 'dropdown';

}

### **Compose form groups**

A dynamic form uses a service to create grouped sets of input controls, based on the form model. The following QuestionControlService collects a set of [FormGroup](https://angular.io/api/forms/FormGroup) instances that consume the metadata from the question model. You can specify default values and validation rules.

src/app/question-control.service.ts

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

import { [FormControl](https://angular.io/api/forms/FormControl), [FormGroup](https://angular.io/api/forms/FormGroup), [Validators](https://angular.io/api/forms/Validators) } from '@angular/forms';

import { QuestionBase } from './question-base';

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

export class QuestionControlService {

constructor() { }

toFormGroup(questions: QuestionBase<string>[] ) {

const group: any = {};

questions.forEach(question => {

group[question.key] = question.required ? new [FormControl](https://angular.io/api/forms/FormControl)(question.value || '', Validators.required)

: new [FormControl](https://angular.io/api/forms/FormControl)(question.value || '');

});

return new [FormGroup](https://angular.io/api/forms/FormGroup)(group);

}

}

## **Compose dynamic form contents**

The dynamic form itself will be represented by a container component, which you will add in a later step. Each question is represented in the form component's template by an <app-question> tag, which matches an instance of DynamicFormQuestionComponent.

The DynamicFormQuestionComponent is responsible for rendering the details of an individual question based on values in the data-bound question object. The form relies on a [[formGroup] directive](https://angular.io/api/forms/FormGroupDirective) to connect the template HTML to the underlying control objects. The DynamicFormQuestionComponent creates form groups and populates them with controls defined in the question model, specifying display and validation rules.

dynamic-form-question.component.html

dynamic-form-question.component.ts

content\_copy<div [formGroup]="form">

<label [attr.for]="question.key">{{question.label}}</label>

<div [[ngSwitch](https://angular.io/api/common/NgSwitch)]="question.controlType">

<input \*[ngSwitchCase](https://angular.io/api/common/NgSwitchCase)="'textbox'" [[formControlName](https://angular.io/api/forms/FormControlName)]="question.key"

[id]="question.key" [type]="question.type">

<select [id]="question.key" \*[ngSwitchCase](https://angular.io/api/common/NgSwitchCase)="'dropdown'" [[formControlName](https://angular.io/api/forms/FormControlName)]="question.key">

<option \*[ngFor](https://angular.io/api/common/NgForOf)="let opt of question.options" [value]="opt.key">{{opt.value}}</option>

</select>

</div>

<div class="errorMessage" \*[ngIf](https://angular.io/api/common/NgIf)="!isValid">{{question.label}} is required</div>

</div>

The goal of the DynamicFormQuestionComponent is to present question types defined in your model. You only have two types of questions at this point but you can imagine many more. The [ngSwitch](https://angular.io/api/common/NgSwitch) statement in the template determines which type of question to display. The switch uses directives with the [formControlName](https://angular.io/api/forms/FormControlName) and [formGroup](https://angular.io/api/forms/FormGroupDirective) selectors. Both directives are defined in [ReactiveFormsModule](https://angular.io/api/forms/ReactiveFormsModule).

### **Supply data**

Another service is needed to supply a specific set of questions from which to build an individual form. For this exercise you will create the QuestionService to supply this array of questions from the hard-coded sample data. In a real-world app, the service might fetch data from a backend system. The key point, however, is that you control the hero job-application questions entirely through the objects returned from QuestionService. To maintain the questionnaire as requirements change, you only need to add, update, and remove objects from the questions array.

The QuestionService supplies a set of questions in the form of an array bound to @[Input](https://angular.io/api/core/Input)() questions.

src/app/question.service.ts

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

import { DropdownQuestion } from './question-dropdown';

import { QuestionBase } from './question-base';

import { TextboxQuestion } from './question-textbox';

import { of } from 'rxjs';

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

export class QuestionService {

// TODO: get from a remote source of question metadata

getQuestions() {

const questions: QuestionBase<string>[] = [

new DropdownQuestion({

key: 'brave',

label: 'Bravery Rating',

options: [

{key: 'solid', value: 'Solid'},

{key: 'great', value: 'Great'},

{key: 'good', value: 'Good'},

{key: 'unproven', value: 'Unproven'}

],

order: 3

}),

new TextboxQuestion({

key: 'firstName',

label: 'First name',

value: 'Bombasto',

required: true,

order: 1

}),

new TextboxQuestion({

key: 'emailAddress',

label: 'Email',

type: '[email](https://angular.io/api/forms/EmailValidator)',

order: 2

})

];

return of(questions.sort((a, b) => a.order - b.order));

}

}

## **Create a dynamic form template**

The DynamicFormComponent component is the entry point and the main container for the form, which is represented using the <app-dynamic-form> in a template.

The DynamicFormComponent component presents a list of questions by binding each one to an <app-question> element that matches the DynamicFormQuestionComponent.

dynamic-form.component.html

dynamic-form.component.ts

content\_copy<div>

<form (ngSubmit)="onSubmit()" [formGroup]="form">

<div \*[ngFor](https://angular.io/api/common/NgForOf)="let question of questions" class="form-row">

<app-question [question]="question" [form]="form"></app-question>

</div>

<div class="form-row">

<button type="submit" [disabled]="!form.valid">Save</button>

</div>

</form>

<div \*[ngIf](https://angular.io/api/common/NgIf)="payLoad" class="form-row">

<strong>Saved the following values</strong><br>{{payLoad}}

</div>

</div>

### **Display the form**

To display an instance of the dynamic form, the AppComponent shell template passes the questions array returned by the QuestionService to the form container component, <app-dynamic-form>.

app.component.ts

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

import { QuestionService } from './question.service';

import { QuestionBase } from './question-base';

import { Observable } from 'rxjs';

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

selector: 'app-root',

template: `

<div>

<h2>Job Application for Heroes</h2>

<app-dynamic-form [questions]="questions$ | async"></app-dynamic-form>

</div>

`,

providers: [QuestionService]

})

export class AppComponent {

questions$: Observable<QuestionBase<any>[]>;

constructor(service: QuestionService) {

this.questions$ = service.getQuestions();

}

}

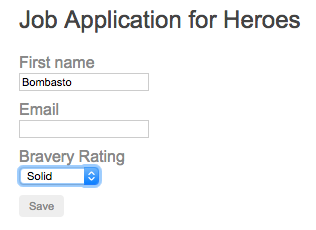
The example provides a model for a job application for heroes, but there are no references to any specific hero question other than the objects returned by QuestionService. This separation of model and data allows you to repurpose the components for any type of survey as long as it's compatible with the question object model.

### **Ensuring valid data**

The form template uses dynamic data binding of metadata to render the form without making any hardcoded assumptions about specific questions. It adds both control metadata and validation criteria dynamically.

To ensure valid input, the Save button is disabled until the form is in a valid state. When the form is valid, you can click Save and the application renders the current form values as JSON.

The following figure shows the final form.



## **Next steps**

* **Different types of forms and control collection**

This tutorial shows how to build a a questionaire, which is just one kind of dynamic form. The example uses [FormGroup](https://angular.io/api/forms/FormGroup) to collect a set of controls. For an example of a different type of dynamic form, see the section [Creating dynamic forms](https://angular.io/guide/reactive-forms#creating-dynamic-forms) in the Reactive Forms guide. That example also shows how to use [FormArray](https://angular.io/api/forms/FormArray) instead of [FormGroup](https://angular.io/api/forms/FormGroup) to collect a set of controls.

* **Validating user input**

The section [Validating form input](https://angular.io/guide/reactive-forms#validating-form-input) introduces the basics of how input validation works in reactive forms.

The [Form validation guide](https://angular.io/guide/form-validation) covers the topic in more depth.

# Communicating with backend services using HTTP

Most front-end applications need to communicate with a server over the HTTP protocol, in order to download or upload data and access other back-end services. Angular provides a client HTTP API for Angular applications, the [HttpClient](https://angular.io/api/common/http/HttpClient) service class in @angular/common/[http](https://angular.io/api/common/http).

The HTTP client service offers the following major features.

* The ability to request [typed response objects](https://angular.io/guide/http#typed-response).
* Streamlined [error handling](https://angular.io/guide/http#error-handling).
* [Testability](https://angular.io/guide/http#testing-requests) features.
* Request and response [interception](https://angular.io/guide/http#intercepting-requests-and-responses).

##### Prerequisites

Before working with the [HttpClientModule](https://angular.io/api/common/http/HttpClientModule), you should have a basic understanding of the following:

* TypeScript programming
* Usage of the HTTP protocol
* Angular app-design fundamentals, as described in [Angular Concepts](https://angular.io/guide/architecture)
* Observable techniques and operators. See the [Observables](https://angular.io/guide/observables) guide.

## **Setup for server communication**

Before you can use [HttpClient](https://angular.io/api/common/http/HttpClient), you need to import the Angular [HttpClientModule](https://angular.io/api/common/http/HttpClientModule). Most apps do so in the root AppModule.

app/app.module.ts (excerpt)

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

import { [BrowserModule](https://angular.io/api/platform-browser/BrowserModule) } from '@angular/platform-browser';

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

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

imports: [

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

// import [HttpClientModule](https://angular.io/api/common/http/HttpClientModule) after BrowserModule.

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

],

declarations: [

AppComponent,

],

bootstrap: [ AppComponent ]

})

export class AppModule {}

You can then inject the [HttpClient](https://angular.io/api/common/http/HttpClient) service as a dependency of an application class, as shown in the following ConfigService example.

app/config/config.service.ts (excerpt)

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

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

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

export class ConfigService {

constructor(private [http](https://angular.io/api/common/http): [HttpClient](https://angular.io/api/common/http/HttpClient)) { }

}

The [HttpClient](https://angular.io/api/common/http/HttpClient) service makes use of [observables](https://angular.io/guide/glossary#observable) for all transactions. You must import the RxJS observable and operator symbols that appear in the example snippets. These ConfigService imports are typical.

app/config/config.service.ts (RxJS imports)

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

import { catchError, retry } from 'rxjs/operators';

You can run the [live example](https://angular.io/generated/live-examples/http/stackblitz.html) / [download example](https://angular.io/generated/zips/http/http.zip) that accompanies this guide.

The sample app does not require a data server. It relies on the [Angularin-memory-web-api](https://github.com/angular/angular/tree/master/packages/misc/angular-in-memory-web-api), which replaces the HttpClient module's [HttpBackend](https://angular.io/api/common/http/HttpBackend). The replacement service simulates the behavior of a REST-like backend.

Look at the AppModule imports to see how it is configured.

## **Requesting data from a server**

Use the [HttpClient.get()](https://angular.io/api/common/http/HttpClient#get) method to fetch data from a server. The asynchronous method sends an HTTP request, and returns an Observable that emits the requested data when the response is received. The return type varies based on the observe and responseType values that you pass to the call.

The get() method takes two arguments; the endpoint URL from which to fetch, and an options object that you can use to configure the request.

content\_copyoptions: {

headers?: [HttpHeaders](https://angular.io/api/common/http/HttpHeaders) | {[header: string]: string | string[]},

observe?: 'body' | 'events' | 'response',

params?: [HttpParams](https://angular.io/api/common/http/HttpParams)|{[param: string]: string | [number](https://angular.io/api/common/DecimalPipe) | boolean | ReadonlyArray<string | [number](https://angular.io/api/common/DecimalPipe) | boolean>},

reportProgress?: boolean,

responseType?: 'arraybuffer'|'blob'|'json'|'text',

withCredentials?: boolean,

}

Important options include the observe and responseType properties.

* The observe option specifies how much of the response to return.
* The responseType option specifies the format in which to return data.

You can use the options object to configure various other aspects of an outgoing request. In [Adding headers](https://angular.io/guide/http#adding-headers), for example, the service set the default headers using the headers option property.

Use the params property to configure a request with [HTTP URL parameters](https://angular.io/guide/http#url-params), and the reportProgress option to [listen for progress events](https://angular.io/guide/http#report-progress) when transferring large amounts of data.

Applications often request JSON data from a server. In the ConfigService example, the app needs a configuration file on the server, config.json, that specifies resource URLs.

assets/config.json

content\_copy{

"heroesUrl": "api/heroes",

"textfile": "assets/textfile.txt",

"date": "2020-01-29"

}

To fetch this kind of data, the get() call needs the following options: {observe: 'body', responseType: 'json'}. These are the default values for those options, so the following examples do not pass the options object. Later sections show some of the additional option possibilities.

The example conforms to the best practices for creating scalable solutions by defining a re-usable [injectable service](https://angular.io/guide/glossary#service) to perform the data-handling functionality. In addition to fetching data, the service can post-process the data, add error handling, and add retry logic.

The ConfigService fetches this file using the [HttpClient.get()](https://angular.io/api/common/http/HttpClient#get) method.

app/config/config.service.ts (getConfig v.1)

content\_copyconfigUrl = 'assets/config.json';

getConfig() {

return this.http.get<Config>(this.configUrl);

}

The ConfigComponent injects the ConfigService and calls the getConfig service method.

Because the service method returns an Observable of configuration data, the component subscribes to the method's return value. The subscription callback performs minimal post-processing. It copies the data fields into the component's config object, which is data-bound in the component template for display.

app/config/config.component.ts (showConfig v.1)

content\_copyshowConfig() {

this.configService.getConfig()

.subscribe((data: Config) => this.config = {

heroesUrl: data.heroesUrl,

textfile: data.textfile,

date: data.date,

});

}

### **Requesting a typed response**

You can structure your [HttpClient](https://angular.io/api/common/http/HttpClient) request to declare the type of the response object, to make consuming the output easier and more obvious. Specifying the response type acts as a type assertion at compile time.

Specifying the response type is a declaration to TypeScript that it should treat your response as being of the given type. This is a build-time check and doesn't guarantee that the server will actually respond with an object of this type. It is up to the server to ensure that the type specified by the server API is returned.

To specify the response object type, first define an interface with the required properties. Use an interface rather than a class, because the response is a plain object that cannot be automatically converted to an instance of a class.

content\_copyexport interface Config {

heroesUrl: string;

textfile: string;

date: any;

}

Next, specify that interface as the [HttpClient.get()](https://angular.io/api/common/http/HttpClient#get) call's type parameter in the service.

app/config/config.service.ts (getConfig v.2)

content\_copygetConfig() {

// now returns an Observable of Config

return this.http.get<Config>(this.configUrl);

}

When you pass an interface as a type parameter to the [HttpClient.get()](https://angular.io/api/common/http/HttpClient#get) method, you can use the [RxJS map operator](https://angular.io/guide/rx-library#operators) to transform the response data as needed by the UI. You can then pass the transformed data to the [async pipe](https://angular.io/api/common/AsyncPipe).

The callback in the updated component method receives a typed data object, which is easier and safer to consume:

app/config/config.component.ts (showConfig v.2)

content\_copyconfig: Config | undefined;

showConfig() {

this.configService.getConfig()

// clone the data object, using its known Config shape

.subscribe((data: Config) => this.config = { ...data });

}

To access properties that are defined in an interface, you must explicitly convert the plain object you get from the JSON to the required response type. For example, the following subscribe callback receives data as an Object, and then type-casts it in order to access the properties.

content\_copy.subscribe(data => this.config = {

heroesUrl: (data as any).heroesUrl,

textfile: (data as any).textfile,

});

\*OBSERVE\* AND \*RESPONSE\* TYPES

The types of the observe and response options are string unions, rather than plain strings.

content\_copyoptions: {

...

observe?: 'body' | 'events' | 'response',

...

responseType?: 'arraybuffer'|'blob'|'json'|'text',

...

}

This can cause confusion. For example:

content\_copy// this works

client.get('/foo', {responseType: 'text'})

// but this does NOT work

const options = {

responseType: 'text',

};

client.get('/foo', options)

In the second case, TypeScript infers the type of options to be {responseType: string}. The type is too wide to pass to HttpClient.get which is expecting the type of responseType to be one of the specific strings. [HttpClient](https://angular.io/api/common/http/HttpClient) is typed explicitly this way so that the compiler can report the correct return type based on the options you provided.

Use as const to let TypeScript know that you really do mean to use a constant string type:

content\_copyconst options = {

responseType: 'text' as const,

};

client.get('/foo', options);

### **Reading the full response**

In the previous example, the call to [HttpClient.get()](https://angular.io/api/common/http/HttpClient#get) did not specify any options. By default, it returned the JSON data contained in the response body.

You might need more information about the transaction than is contained in the response body. Sometimes servers return special headers or status codes to indicate certain conditions that are important to the application workflow.

Tell [HttpClient](https://angular.io/api/common/http/HttpClient) that you want the full response with the observe option of the get() method:

content\_copygetConfigResponse(): Observable<[HttpResponse](https://angular.io/api/common/http/HttpResponse)<Config>> {

return this.http.get<Config>(

this.configUrl, { observe: 'response' });

}

Now [HttpClient.get()](https://angular.io/api/common/http/HttpClient#get) returns an Observable of type [HttpResponse](https://angular.io/api/common/http/HttpResponse) rather than just the JSON data contained in the body.

The component's showConfigResponse() method displays the response headers as well as the configuration:

app/config/config.component.ts (showConfigResponse)

content\_copyshowConfigResponse() {

this.configService.getConfigResponse()

// resp is of type `[HttpResponse](https://angular.io/api/common/http/HttpResponse)<Config>`

.subscribe(resp => {

// display its headers

const keys = resp.headers.keys();

this.headers = keys.map(key =>

`${key}: ${resp.headers.get(key)}`);

// access the body directly, which is typed as `Config`.

this.config = { ...resp.body! };

});

}

As you can see, the response object has a body property of the correct type.

### **Making a JSONP request**

Apps can use the [HttpClient](https://angular.io/api/common/http/HttpClient) to make [JSONP](https://en.wikipedia.org/wiki/JSONP) requests across domains when a server doesn't support [CORS protocol](https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS).

Angular JSONP requests return an Observable. Follow the pattern for subscribing to observables and use the RxJS map operator to transform the response before using the [async pipe](https://angular.io/api/common/AsyncPipe) to manage the results.

In Angular, use JSONP by including [HttpClientJsonpModule](https://angular.io/api/common/http/HttpClientJsonpModule) in the [NgModule](https://angular.io/api/core/NgModule) imports. In the following example, the searchHeroes() method uses a JSONP request to query for heroes whose names contain the search term.

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

searchHeroes(term: string): Observable {

term = term.trim();

const heroesURL = `${this.heroesURL}?${term}`;

return this.http.jsonp(heroesUrl, 'callback').pipe(

catchError(this.handleError('searchHeroes', [])) // then handle the error

);

}

This request passes the heroesURL as the first parameter and the callback function name as the second parameter. The response is wrapped in the callback function, which takes the observables returned by the JSONP method and pipes them through to the error handler.

### **Requesting non-JSON data**

Not all APIs return JSON data. In this next example, a DownloaderService method reads a text file from the server and logs the file contents, before returning those contents to the caller as an Observable<string>.

app/downloader/downloader.service.ts (getTextFile)

content\_copygetTextFile(filename: string) {

// The Observable returned by get() is of type Observable<string>

// because a text response was specified.

// There's no need to pass a <string> type parameter to get().

return this.http.get(filename, {responseType: 'text'})

.pipe(

tap( // Log the result or error

data => this.log(filename, data),

error => this.logError(filename, error)

)

);

}

[HttpClient.get()](https://angular.io/api/common/http/HttpClient#get) returns a string rather than the default JSON because of the responseType option.

The RxJS tap operator (as in "wiretap") lets the code inspect both success and error values passing through the observable without disturbing them.

A download() method in the DownloaderComponent initiates the request by subscribing to the service method.

app/downloader/downloader.component.ts (download)

content\_copydownload() {

this.downloaderService.getTextFile('assets/textfile.txt')

.subscribe(results => this.contents = results);

}

## **Handling request errors**

If the request fails on the server, [HttpClient](https://angular.io/api/common/http/HttpClient) returns an error object instead of a successful response.

The same service that performs your server transactions should also perform error inspection, interpretation, and resolution.

When an error occurs, you can obtain details of what failed in order to inform your user. In some cases, you might also automatically [retry the request](https://angular.io/guide/http#retry).

### **Getting error details**

An app should give the user useful feedback when data access fails. A raw error object is not particularly useful as feedback. In addition to detecting that an error has occurred, you need to get error details and use those details to compose a user-friendly response.

Two types of errors can occur.

* The server backend might reject the request, returning an HTTP response with a status code such as 404 or 500. These are error responses.
* Something could go wrong on the client-side such as a network error that prevents the request from completing successfully or an exception thrown in an RxJS operator. These errors have status set to 0 and the error property contains a ProgressEvent object, whose type might provide further information.

[HttpClient](https://angular.io/api/common/http/HttpClient) captures both kinds of errors in its [HttpErrorResponse](https://angular.io/api/common/http/HttpErrorResponse). You can inspect that response to identify the error's cause.

The following example defines an error handler in the previously defined [ConfigService](https://angular.io/guide/http#config-service).

app/config/config.service.ts (handleError)

content\_copyprivate handleError(error: [HttpErrorResponse](https://angular.io/api/common/http/HttpErrorResponse)) {

if (error.status === 0) {

// A client-side or network error occurred. Handle it accordingly.

console.error('An error occurred:', error.error);

} else {

// The backend returned an unsuccessful response code.

// The response body may contain clues as to what went wrong.

console.error(

`Backend returned code ${error.status}, ` +

`body was: ${error.error}`);

}

// Return an observable with a user-facing error message.

return throwError(

'Something bad happened; please try again later.');

}

The handler returns an RxJS ErrorObservable with a user-friendly error message. The following code updates the getConfig() method, using a [pipe](https://angular.io/guide/pipes) to send all observables returned by the [HttpClient.get()](https://angular.io/api/common/http/HttpClient#get) call to the error handler.

app/config/config.service.ts (getConfig v.3 with error handler)

content\_copygetConfig() {

return this.http.get<Config>(this.configUrl)

.pipe(

catchError(this.handleError)

);

}

### **Retrying a failed request**

Sometimes the error is transient and goes away automatically if you try again. For example, network interruptions are common in mobile scenarios, and trying again can produce a successful result.

The [RxJS library](https://angular.io/guide/rx-library) offers several retry operators. For example, the retry() operator automatically re-subscribes to a failed Observable a specified number of times. Re-subscribing to the result of an [HttpClient](https://angular.io/api/common/http/HttpClient) method call has the effect of reissuing the HTTP request.

The following example shows how you can pipe a failed request to the retry() operator before passing it to the error handler.

app/config/config.service.ts (getConfig with retry)

content\_copygetConfig() {

return this.http.get<Config>(this.configUrl)

.pipe(

retry(3), // retry a failed request up to 3 times

catchError(this.handleError) // then handle the error

);

}

## **Sending data to a server**

In addition to fetching data from a server, [HttpClient](https://angular.io/api/common/http/HttpClient) supports other HTTP methods such as PUT, POST, and DELETE, which you can use to modify the remote data.

The sample app for this guide includes an abridged version of the "Tour of Heroes" example that fetches heroes and enables users to add, delete, and update them. The following sections show examples of the data-update methods from the sample's HeroesService.

### **Making a POST request**

Apps often send data to a server with a POST request when submitting a form. In the following example, the HeroesService makes an HTTP POST request when adding a hero to the database.

app/heroes/heroes.service.ts (addHero)

content\_copy/\*\* POST: add a new hero to the database \*/

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

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

.pipe(

catchError(this.handleError('addHero', hero))

);

}

The [HttpClient.post()](https://angular.io/api/common/http/HttpClient#post) method is similar to get() in that it has a type parameter, which you can use to specify that you expect the server to return data of a given type. The method takes a resource URL and two additional parameters:

* body - The data to POST in the body of the request.
* options - An object containing method options which, in this case, [specify required headers](https://angular.io/guide/http#adding-headers).

The example catches errors as [described above](https://angular.io/guide/http#error-details).

The HeroesComponent initiates the actual POST operation by subscribing to the Observable returned by this service method.

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

content\_copythis.heroesService

.addHero(newHero)

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

When the server responds successfully with the newly added hero, the component adds that hero to the displayed heroes list.

### **Making a DELETE request**

This application deletes a hero with the HttpClient.delete method by passing the hero's id in the request URL.

app/heroes/heroes.service.ts (deleteHero)

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

deleteHero(id: number): Observable<unknown> {

const url = `${this.heroesUrl}/${id}`; // DELETE api/heroes/42

return this.http.delete(url, httpOptions)

.pipe(

catchError(this.handleError('deleteHero'))

);

}

The HeroesComponent initiates the actual DELETE operation by subscribing to the Observable returned by this service method.

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

content\_copythis.heroesService

.deleteHero(hero.id)

.subscribe();

The component isn't expecting a result from the delete operation, so it subscribes without a callback. Even though you are not using the result, you still have to subscribe. Calling the subscribe() method executes the observable, which is what initiates the DELETE request.

You must call subscribe() or nothing happens. Just calling HeroesService.deleteHero() does not initiate the DELETE request.

content\_copy// oops ... subscribe() is missing so nothing happens

this.heroesService.deleteHero(hero.id);

**Always**subscribe**!**

An [HttpClient](https://angular.io/api/common/http/HttpClient) method does not begin its HTTP request until you call subscribe() on the observable returned by that method. This is true for all [HttpClient](https://angular.io/api/common/http/HttpClient) methods.

The [AsyncPipe](https://angular.io/api/common/AsyncPipe) subscribes (and unsubscribes) for you automatically.

All observables returned from [HttpClient](https://angular.io/api/common/http/HttpClient) methods are cold by design. Execution of the HTTP request is deferred, allowing you to extend the observable with additional operations such as tap and catchError before anything actually happens.

Calling subscribe(...) triggers execution of the observable and causes [HttpClient](https://angular.io/api/common/http/HttpClient) to compose and send the HTTP request to the server.

You can think of these observables as blueprints for actual HTTP requests.

In fact, each subscribe() initiates a separate, independent execution of the observable. Subscribing twice results in two HTTP requests.

content\_copyconst req = http.get<Heroes>('/api/heroes');

// 0 requests made - .subscribe() not called.

req.subscribe();

// 1 request made.

req.subscribe();

// 2 requests made.

### **Making a PUT request**

An app can send PUT requests using the HTTP client service. The following HeroesService example, like the POST example, replaces a resource with updated data.

app/heroes/heroes.service.ts (updateHero)

content\_copy/\*\* PUT: update the hero on the server. Returns the updated hero upon success. \*/

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

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

.pipe(

catchError(this.handleError('updateHero', hero))

);

}

As for any of the HTTP methods that return an observable, the caller, HeroesComponent.update() [must subscribe()](https://angular.io/guide/http#always-subscribe) to the observable returned from the [HttpClient.put()](https://angular.io/api/common/http/HttpClient#put) in order to initiate the request.

### **Adding and updating headers**

Many servers require extra headers for save operations. For example, a server might require an authorization token, or "Content-Type" header to explicitly declare the MIME type of the request body.

##### Adding headers

The HeroesService defines such headers in an httpOptions object that are passed to every [HttpClient](https://angular.io/api/common/http/HttpClient) save method.

app/heroes/heroes.service.ts (httpOptions)

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

const httpOptions = {

headers: new [HttpHeaders](https://angular.io/api/common/http/HttpHeaders)({

'Content-Type': 'application/json',

Authorization: 'my-auth-token'

})

};

##### Updating headers

You can't directly modify the existing headers within the previous options object because instances of the [HttpHeaders](https://angular.io/api/common/http/HttpHeaders) class are immutable. Use the set() method instead, to return a clone of the current instance with the new changes applied.

The following example shows how, when an old token has expired, you can update the authorization header before making the next request.

content\_copyhttpOptions.headers =

httpOptions.headers.set('Authorization', 'my-new-auth-token');

## **Configuring HTTP URL parameters**

Use the [HttpParams](https://angular.io/api/common/http/HttpParams) class with the params request option to add URL query strings in your [HttpRequest](https://angular.io/api/common/http/HttpRequest).

The following example, the searchHeroes() method queries for heroes whose names contain the search term.

Start by importing [HttpParams](https://angular.io/api/common/http/HttpParams) class.

import {[HttpParams](https://angular.io/api/common/http/HttpParams)} from "@angular/common/[http](https://angular.io/api/common/http)";

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

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

term = term.trim();

// Add safe, URL encoded search parameter if there is a search term

const options = term ?

{ params: new [HttpParams](https://angular.io/api/common/http/HttpParams)().set('name', term) } : {};

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

.pipe(

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

);

}

If there is a search term, the code constructs an options object with an HTML URL-encoded search parameter. If the term is "cat", for example, the GET request URL would be api/heroes?name=cat.

The [HttpParams](https://angular.io/api/common/http/HttpParams) object is immutable. If you need to update the options, save the returned value of the .set() method.

You can also create HTTP parameters directly from a query string by using the fromString variable:

const params = new [HttpParams](https://angular.io/api/common/http/HttpParams)({fromString: 'name=foo'});

## **Intercepting requests and responses**

With interception, you declare interceptors that inspect and transform HTTP requests from your application to a server. The same interceptors can also inspect and transform a server's responses on their way back to the application. Multiple interceptors form a forward-and-backward chain of request/response handlers.

Interceptors can perform a variety of implicit tasks, from authentication to logging, in a routine, standard way, for every HTTP request/response.

Without interception, developers would have to implement these tasks explicitly for each [HttpClient](https://angular.io/api/common/http/HttpClient) method call.

### **Write an interceptor**

To implement an interceptor, declare a class that implements the intercept() method of the [HttpInterceptor](https://angular.io/api/common/http/HttpInterceptor) interface.

Here is a do-nothing noop interceptor that passes the request through without touching it:

app/http-interceptors/noop-interceptor.ts

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

import {

[HttpEvent](https://angular.io/api/common/http/HttpEvent), [HttpInterceptor](https://angular.io/api/common/http/HttpInterceptor), [HttpHandler](https://angular.io/api/common/http/HttpHandler), [HttpRequest](https://angular.io/api/common/http/HttpRequest)

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

import { Observable } from 'rxjs';

/\*\* Pass untouched request through to the next request handler. \*/

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

export class NoopInterceptor implements [HttpInterceptor](https://angular.io/api/common/http/HttpInterceptor) {

intercept(req: [HttpRequest](https://angular.io/api/common/http/HttpRequest)<any>, next: [HttpHandler](https://angular.io/api/common/http/HttpHandler)):

Observable<[HttpEvent](https://angular.io/api/common/http/HttpEvent)<any>> {

return next.handle(req);

}

}

The intercept method transforms a request into an Observable that eventually returns the HTTP response. In this sense, each interceptor is fully capable of handling the request entirely by itself.

Most interceptors inspect the request on the way in and forward the (perhaps altered) request to the handle() method of the next object which implements the [HttpHandler](https://angular.io/api/common/http/HttpHandler) interface.

content\_copyexport abstract class [HttpHandler](https://angular.io/api/common/http/HttpHandler) {

abstract handle(req: [HttpRequest](https://angular.io/api/common/http/HttpRequest)<any>): Observable<[HttpEvent](https://angular.io/api/common/http/HttpEvent)<any>>;

}

Like intercept(), the handle() method transforms an HTTP request into an Observable of [HttpEvents](https://angular.io/guide/http#interceptor-events) which ultimately include the server's response. The intercept() method could inspect that observable and alter it before returning it to the caller.

This no-op interceptor calls next.handle() with the original request and returns the observable without doing a thing.

### **The next object**

The next object represents the next interceptor in the chain of interceptors. The final next in the chain is the [HttpClient](https://angular.io/api/common/http/HttpClient) backend handler that sends the request to the server and receives the server's response.

Most interceptors call next.handle() so that the request flows through to the next interceptor and, eventually, the backend handler. An interceptor could skip calling next.handle(), short-circuit the chain, and [return its own Observable](https://angular.io/guide/http#caching) with an artificial server response.

This is a common middleware pattern found in frameworks such as Express.js.

### **Provide the interceptor**

The NoopInterceptor is a service managed by Angular's [dependency injection (DI)](https://angular.io/guide/dependency-injection) system. Like other services, you must provide the interceptor class before the app can use it.

Because interceptors are (optional) dependencies of the [HttpClient](https://angular.io/api/common/http/HttpClient) service, you must provide them in the same injector (or a parent of the injector) that provides [HttpClient](https://angular.io/api/common/http/HttpClient). Interceptors provided after DI creates the [HttpClient](https://angular.io/api/common/http/HttpClient) are ignored.

This app provides [HttpClient](https://angular.io/api/common/http/HttpClient) in the app's root injector, as a side-effect of importing the [HttpClientModule](https://angular.io/api/common/http/HttpClientModule) in AppModule. You should provide interceptors in AppModule as well.

After importing the [HTTP\_INTERCEPTORS](https://angular.io/api/common/http/HTTP_INTERCEPTORS) injection token from @angular/common/[http](https://angular.io/api/common/http), write the NoopInterceptor provider like this:

content\_copy{ provide: [HTTP\_INTERCEPTORS](https://angular.io/api/common/http/HTTP_INTERCEPTORS), useClass: NoopInterceptor, multi: true },

Note the multi: true option. This required setting tells Angular that [HTTP\_INTERCEPTORS](https://angular.io/api/common/http/HTTP_INTERCEPTORS) is a token for a multiprovider that injects an array of values, rather than a single value.

You could add this provider directly to the providers array of the AppModule. However, it's rather verbose and there's a good chance that you'll create more interceptors and provide them in the same way. You must also pay [close attention to the order](https://angular.io/guide/http#interceptor-order) in which you provide these interceptors.

Consider creating a "barrel" file that gathers all the interceptor providers into an httpInterceptorProviders array, starting with this first one, the NoopInterceptor.

app/http-interceptors/index.ts

content\_copy/\* "Barrel" of Http Interceptors \*/

import { [HTTP\_INTERCEPTORS](https://angular.io/api/common/http/HTTP_INTERCEPTORS) } from '@angular/common/[http](https://angular.io/api/common/http)';

import { NoopInterceptor } from './noop-interceptor';

/\*\* Http interceptor providers in outside-in order \*/

export const httpInterceptorProviders = [

{ provide: [HTTP\_INTERCEPTORS](https://angular.io/api/common/http/HTTP_INTERCEPTORS), useClass: NoopInterceptor, multi: true },

];

Then import and add it to the AppModule providers array like this:

app/app.module.ts (interceptor providers)

content\_copyproviders: [

httpInterceptorProviders

],

As you create new interceptors, add them to the httpInterceptorProviders array and you won't have to revisit the AppModule.

There are many more interceptors in the complete sample code.

### **Interceptor order**

Angular applies interceptors in the order that you provide them. For example, consider a situation in which you want to handle the authentication of your HTTP requests and log them before sending them to a server. To accomplish this task, you could provide an AuthInterceptor service and then a LoggingInterceptor service. Outgoing requests would flow from the AuthInterceptor to the LoggingInterceptor. Responses from these requests would flow in the other direction, from LoggingInterceptor back to AuthInterceptor. The following is a visual representation of the process:

The last interceptor in the process is always the [HttpBackend](https://angular.io/api/common/http/HttpBackend) that handles communication with the server.

You cannot change the order or remove interceptors later. If you need to enable and disable an interceptor dynamically, you'll have to build that capability into the interceptor itself.

### **Handling interceptor events**

Most [HttpClient](https://angular.io/api/common/http/HttpClient) methods return observables of [HttpResponse](https://angular.io/api/common/http/HttpResponse)<any>. The [HttpResponse](https://angular.io/api/common/http/HttpResponse) class itself is actually an event, whose type is [HttpEventType.Response](https://angular.io/api/common/http/HttpEventType#Response). A single HTTP request can, however, generate multiple events of other types, including upload and download progress events. The methods HttpInterceptor.intercept() and HttpHandler.handle() return observables of [HttpEvent](https://angular.io/api/common/http/HttpEvent)<any>.

Many interceptors are only concerned with the outgoing request and return the event stream from next.handle() without modifying it. Some interceptors, however, need to examine and modify the response from next.handle(); these operations can see all of these events in the stream.

Although interceptors are capable of modifying requests and responses, the [HttpRequest](https://angular.io/api/common/http/HttpRequest) and [HttpResponse](https://angular.io/api/common/http/HttpResponse) instance properties are readonly, rendering them largely immutable. They are immutable for a good reason: an app might retry a request several times before it succeeds, which means that the interceptor chain can re-process the same request multiple times. If an interceptor could modify the original request object, the re-tried operation would start from the modified request rather than the original. Immutability ensures that interceptors see the same request for each try.

Your interceptor should return every event without modification unless it has a compelling reason to do otherwise.

TypeScript prevents you from setting [HttpRequest](https://angular.io/api/common/http/HttpRequest) read-only properties.

content\_copy// Typescript disallows the following assignment because req.url is readonly

req.url = req.url.replace('[http](https://angular.io/api/common/http)://', 'https://');

If you must alter a request, clone it first and modify the clone before passing it to next.handle(). You can clone and modify the request in a single step, as shown in the following example.

app/http-interceptors/ensure-https-interceptor.ts (excerpt)

content\_copy// clone request and replace '[http](https://angular.io/api/common/http)://' with 'https://' at the same time

const secureReq = req.clone({

url: req.url.replace('[http](https://angular.io/api/common/http)://', 'https://')

});

// send the cloned, "secure" request to the next handler.

return next.handle(secureReq);

The clone() method's hash argument allows you to mutate specific properties of the request while copying the others.

#### Modifying a request body

The readonly assignment guard can't prevent deep updates and, in particular, it can't prevent you from modifying a property of a request body object.

content\_copyreq.body.name = req.body.name.trim(); // bad idea!

If you must modify the request body, follow these steps.

1. Copy the body and make your change in the copy.
2. Clone the request object, using its clone() method.
3. Replace the clone's body with the modified copy.

app/http-interceptors/trim-name-interceptor.ts (excerpt)

content\_copy// copy the body and trim whitespace from the name property

const newBody = { ...body, name: body.name.trim() };

// clone request and set its body

const newReq = req.clone({ body: newBody });

// send the cloned request to the next handler.

return next.handle(newReq);

#### Clearing the request body in a clone

Sometimes you need to clear the request body rather than replace it. To do this, set the cloned request body to null.

**Tip**: If you set the cloned request body to undefined, Angular assumes you intend to leave the body as is.

content\_copynewReq = req.clone({ ... }); // body not mentioned => preserve original body

newReq = req.clone({ body: undefined }); // preserve original body

newReq = req.clone({ body: null }); // clear the body

## **Http interceptor use-cases**

Below are a number of common uses for interceptors.

### **Setting default headers**

Apps often use an interceptor to set default headers on outgoing requests.

The sample app has an AuthService that produces an authorization token. Here is its AuthInterceptor that injects that service to get the token and adds an authorization header with that token to every outgoing request:

app/http-interceptors/auth-interceptor.ts

content\_copyimport { AuthService } from '../auth.service';

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

export class AuthInterceptor implements [HttpInterceptor](https://angular.io/api/common/http/HttpInterceptor) {

constructor(private auth: AuthService) {}

intercept(req: [HttpRequest](https://angular.io/api/common/http/HttpRequest)<any>, next: [HttpHandler](https://angular.io/api/common/http/HttpHandler)) {

// Get the auth token from the service.

const authToken = this.auth.getAuthorizationToken();

// Clone the request and replace the original headers with

// cloned headers, updated with the authorization.

const authReq = req.clone({

headers: req.headers.set('Authorization', authToken)

});

// send cloned request with header to the next handler.

return next.handle(authReq);

}

}

The practice of cloning a request to set new headers is so common that there's a setHeaders shortcut for it:

content\_copy// Clone the request and set the new header in one step.

const authReq = req.clone({ setHeaders: { Authorization: authToken } });

An interceptor that alters headers can be used for a number of different operations, including:

* Authentication/authorization
* Caching behavior; for example, If-Modified-Since
* XSRF protection

### **Logging request and response pairs**

Because interceptors can process the request and response together, they can perform tasks such as timing and logging an entire HTTP operation.

Consider the following LoggingInterceptor, which captures the time of the request, the time of the response, and logs the outcome with the elapsed time with the injected MessageService.

app/http-interceptors/logging-interceptor.ts)

content\_copyimport { finalize, tap } from 'rxjs/operators';

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

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

export class LoggingInterceptor implements [HttpInterceptor](https://angular.io/api/common/http/HttpInterceptor) {

constructor(private messenger: MessageService) {}

intercept(req: [HttpRequest](https://angular.io/api/common/http/HttpRequest)<any>, next: [HttpHandler](https://angular.io/api/common/http/HttpHandler)) {

const started = Date.now();

let ok: string;

// extend server response observable with logging

return next.handle(req)

.pipe(

tap(

// Succeeds when there is a response; ignore other events

event => ok = event instanceof [HttpResponse](https://angular.io/api/common/http/HttpResponse) ? 'succeeded' : '',

// Operation failed; error is an [HttpErrorResponse](https://angular.io/api/common/http/HttpErrorResponse)

error => ok = 'failed'

),

// Log when response observable either completes or errors

finalize(() => {

const elapsed = Date.now() - started;

const msg = `${req.method} "${req.urlWithParams}"

${ok} in ${elapsed} ms.`;

this.messenger.add(msg);

})

);

}

}

The RxJS tap operator captures whether the request succeeded or failed. The RxJS finalize operator is called when the response observable either errors or completes (which it must), and reports the outcome to the MessageService.

Neither tap nor finalize touch the values of the observable stream returned to the caller.

### **Custom JSON parsing**

Interceptors can be used to replace the built-in JSON parsing with a custom implementation.

The CustomJsonInterceptor in the following example demonstrates how to achieve this. If the intercepted request expects a 'json' response, the responseType is changed to 'text' to disable the built-in JSON parsing. Then the response is parsed via the injected JsonParser.

app/http-interceptors/custom-json-interceptor.ts

content\_copy// The JsonParser class acts as a base class for custom parsers and as the DI token.

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

export abstract class JsonParser {

abstract parse(text: string): any;

}

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

export class CustomJsonInterceptor implements [HttpInterceptor](https://angular.io/api/common/http/HttpInterceptor) {

constructor(private jsonParser: JsonParser) {}

intercept(httpRequest: [HttpRequest](https://angular.io/api/common/http/HttpRequest)<any>, next: [HttpHandler](https://angular.io/api/common/http/HttpHandler)) {

if (httpRequest.responseType === 'json') {

// If the expected response type is JSON then handle it here.

return this.handleJsonResponse(httpRequest, next);

} else {

return next.handle(httpRequest);

}

}

private handleJsonResponse(httpRequest: [HttpRequest](https://angular.io/api/common/http/HttpRequest)<any>, next: [HttpHandler](https://angular.io/api/common/http/HttpHandler)) {

// Override the responseType to disable the default JSON parsing.

httpRequest = httpRequest.clone({responseType: 'text'});

// Handle the response using the custom parser.

return next.handle(httpRequest).pipe(map(event => this.parseJsonResponse(event)));

}

private parseJsonResponse(event: [HttpEvent](https://angular.io/api/common/http/HttpEvent)<any>) {

if (event instanceof [HttpResponse](https://angular.io/api/common/http/HttpResponse) && typeof event.body === 'string') {

return event.clone({body: this.jsonParser.parse(event.body)});

} else {

return event;

}

}

}

You can then implement your own custom JsonParser. Here is a custom JsonParser that has a special date reviver.

app/http-interceptors/custom-json-interceptor.ts

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

export class CustomJsonParser implements JsonParser {

parse(text: string): any {

return JSON.parse(text, dateReviver);

}

}

function dateReviver(key: string, value: any) {

/\* . . . \*/

}

You provide the CustomParser along with the CustomJsonInterceptor.

app/http-interceptors/index.ts

content\_copy{ provide: [HTTP\_INTERCEPTORS](https://angular.io/api/common/http/HTTP_INTERCEPTORS), useClass: CustomJsonInterceptor, multi: true },

{ provide: JsonParser, useClass: CustomJsonParser },

### **Caching requests**

Interceptors can handle requests by themselves, without forwarding to next.handle().

For example, you might decide to cache certain requests and responses to improve performance. You can delegate caching to an interceptor without disturbing your existing data services.

The CachingInterceptor in the following example demonstrates this approach.

app/http-interceptors/caching-interceptor.ts)

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

export class CachingInterceptor implements [HttpInterceptor](https://angular.io/api/common/http/HttpInterceptor) {

constructor(private cache: RequestCache) {}

intercept(req: [HttpRequest](https://angular.io/api/common/http/HttpRequest)<any>, next: [HttpHandler](https://angular.io/api/common/http/HttpHandler)) {

// continue if not cacheable.

if (!isCacheable(req)) { return next.handle(req); }

const cachedResponse = this.cache.get(req);

return cachedResponse ?

of(cachedResponse) : sendRequest(req, next, this.cache);

}

}

* The isCacheable() function determines if the request is cacheable. In this sample, only GET requests to the npm package search API are cacheable.
* If the request is not cacheable, the interceptor forwards the request to the next handler in the chain.
* If a cacheable request is found in the cache, the interceptor returns an of() observable with the cached response, by-passing the next handler (and all other interceptors downstream).
* If a cacheable request is not in cache, the code calls sendRequest(). This function creates a [request clone](https://angular.io/guide/http#immutability) without headers, because the npm API forbids them. The function then forwards the clone of the request to next.handle() which ultimately calls the server and returns the server's response.

content\_copy/\*\*

\* Get server response observable by sending request to `next()`.

\* Will add the response to the cache on the way out.

\*/

function sendRequest(

req: [HttpRequest](https://angular.io/api/common/http/HttpRequest)<any>,

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

cache: RequestCache): Observable<[HttpEvent](https://angular.io/api/common/http/HttpEvent)<any>> {

// No headers allowed in npm search request

const noHeaderReq = req.clone({ headers: new [HttpHeaders](https://angular.io/api/common/http/HttpHeaders)() });

return next.handle(noHeaderReq).pipe(

tap(event => {

// There may be other events besides the response.

if (event instanceof [HttpResponse](https://angular.io/api/common/http/HttpResponse)) {

cache.put(req, event); // Update the cache.

}

})

);

}

Note how sendRequest() intercepts the response on its way back to the application. This method pipes the response through the tap() operator, whose callback adds the response to the cache.

The original response continues untouched back up through the chain of interceptors to the application caller.

Data services, such as PackageSearchService, are unaware that some of their [HttpClient](https://angular.io/api/common/http/HttpClient) requests actually return cached responses.

### **Using interceptors to request multiple values**

The [HttpClient.get()](https://angular.io/api/common/http/HttpClient#get) method normally returns an observable that emits a single value, either the data or an error. An interceptor can change this to an observable that emits [multiple values](https://angular.io/guide/observables).

The following revised version of the CachingInterceptor optionally returns an observable that immediately emits the cached response, sends the request on to the npm web API, and emits again later with the updated search results.

content\_copy// cache-then-refresh

if (req.headers.get('x-refresh')) {

const results$ = sendRequest(req, next, this.cache);

return cachedResponse ?

results$.pipe( startWith(cachedResponse) ) :

results$;

}

// cache-or-fetch

return cachedResponse ?

of(cachedResponse) : sendRequest(req, next, this.cache);

The cache-then-refresh option is triggered by the presence of a custom x-refresh header.

A checkbox on the PackageSearchComponent toggles a withRefresh flag, which is one of the arguments to PackageSearchService.search(). That search() method creates the custom x-refresh header and adds it to the request before calling [HttpClient.get()](https://angular.io/api/common/http/HttpClient#get).

The revised CachingInterceptor sets up a server request whether there's a cached value or not, using the same sendRequest() method described [above](https://angular.io/guide/http#send-request). The results$ observable makes the request when subscribed.

* If there's no cached value, the interceptor returns results$.
* If there is a cached value, the code pipes the cached response onto results$, producing a recomposed observable that emits twice, the cached response first (and immediately), followed later by the response from the server. Subscribers see a sequence of two responses.

## **Tracking and showing request progress**

Sometimes applications transfer large amounts of data and those transfers can take a long time. File uploads are a typical example. You can give the users a better experience by providing feedback on the progress of such transfers.

To make a request with progress events enabled, you can create an instance of [HttpRequest](https://angular.io/api/common/http/HttpRequest) with the reportProgress option set true to enable tracking of progress events.

app/uploader/uploader.service.ts (upload request)

content\_copyconst req = new [HttpRequest](https://angular.io/api/common/http/HttpRequest)('POST', '/upload/file', file, {

reportProgress: true

});

**Tip**: Every progress event triggers change detection, so only turn them on if you need to report progress in the UI.

When using [HttpClient.request()](https://angular.io/api/common/http/HttpClient#request) with an HTTP method, configure the method with [observe: 'events'](https://angular.io/api/common/http/HttpClient#request) to see all events, including the progress of transfers.

Next, pass this request object to the [HttpClient.request()](https://angular.io/api/common/http/HttpClient#request) method, which returns an Observable of HttpEvents (the same events processed by [interceptors](https://angular.io/guide/http#interceptor-events)).

app/uploader/uploader.service.ts (upload body)

content\_copy// The `HttpClient.request` API produces a raw event stream

// which includes start (sent), progress, and response events.

return this.http.request(req).pipe(

map(event => this.getEventMessage(event, file)),

tap(message => this.showProgress(message)),

last(), // return last (completed) message to caller

catchError(this.handleError(file))

);

The getEventMessage method interprets each type of [HttpEvent](https://angular.io/api/common/http/HttpEvent) in the event stream.

app/uploader/uploader.service.ts (getEventMessage)

content\_copy/\*\* Return distinct message for sent, upload progress, & response events \*/

private getEventMessage(event: [HttpEvent](https://angular.io/api/common/http/HttpEvent)<any>, file: File) {

switch (event.type) {

case [HttpEventType.Sent](https://angular.io/api/common/http/HttpEventType#Sent):

return `Uploading file "${file.name}" of size ${file.size}.`;

case [HttpEventType.UploadProgress](https://angular.io/api/common/http/HttpEventType#UploadProgress):

// Compute and show the % done:

const percentDone = Math.round(100 \* event.loaded / (event.total ?? 0));

return `File "${file.name}" is ${percentDone}% uploaded.`;

case [HttpEventType.Response](https://angular.io/api/common/http/HttpEventType#Response):

return `File "${file.name}" was completely uploaded!`;

default:

return `File "${file.name}" surprising upload event: ${event.type}.`;

}

}

The sample app for this guide doesn't have a server that accepts uploaded files. The UploadInterceptor in app/http-interceptors/upload-interceptor.ts intercepts and short-circuits upload requests by returning an observable of simulated events.

## **Optimizing server interaction with debouncing**

If you need to make an HTTP request in response to user input, it's not efficient to send a request for every keystroke. It's better to wait until the user stops typing and then send a request. This technique is known as debouncing.

Consider the following template, which lets a user enter a search term to find an npm package by name. When the user enters a name in a search-box, the PackageSearchComponent sends a search request for a package with that name to the npm web API.

app/package-search/package-search.component.html (search)

content\_copy<input type="text" (keyup)="search(getValue($event))" id="name" placeholder="Search"/>

<ul>

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

<b>{{package.name}} v.{{package.version}}</b> -

<i>{{package.description}}</i>

</li>

</ul>

Here, the keyup event binding sends every keystroke to the component's search() method.

The type of $event.target is only EventTarget in the template. In the getValue() method, the target is cast to an HTMLInputElement to allow type-safe access to its value property.

content\_copygetValue(event: [Event](https://angular.io/api/router/Event)): string {

return (event.target as HTMLInputElement).value;

}

The following snippet implements debouncing for this input using RxJS operators.

app/package-search/package-search.component.ts (excerpt)

content\_copywithRefresh = false;

packages$!: Observable<NpmPackageInfo[]>;

private searchText$ = new Subject<string>();

search(packageName: string) {

this.searchText$.next(packageName);

}

ngOnInit() {

this.packages$ = this.searchText$.pipe(

debounceTime(500),

distinctUntilChanged(),

switchMap(packageName =>

this.searchService.search(packageName, this.withRefresh))

);

}

constructor(private searchService: PackageSearchService) { }

The searchText$ is the sequence of search-box values coming from the user. It's defined as an RxJS Subject, which means it is a multicasting Observable that can also emit values for itself by calling next(value), as happens in the search() method.

Rather than forward every searchText value directly to the injected PackageSearchService, the code in ngOnInit() pipes search values through three operators, so that a search value reaches the service only if it's a new value and the user has stopped typing.

* debounceTime(500)⁠—Wait for the user to stop typing (1/2 second in this case).
* distinctUntilChanged()⁠—Wait until the search text changes.
* switchMap()⁠—Send the search request to the service.

The code sets packages$ to this re-composed Observable of search results. The template subscribes to packages$ with the [AsyncPipe](https://angular.io/api/common/AsyncPipe) and displays search results as they arrive.

See [Using interceptors to request multiple values](https://angular.io/guide/http#cache-refresh) for more about the withRefresh option.

### **Using the switchMap() operator**

The switchMap() operator takes a function argument that returns an Observable. In the example, PackageSearchService.search returns an Observable, as other data service methods do. If a previous search request is still in-flight (as when the network connection is poor), the operator cancels that request and sends a new one.

Note that switchMap() returns service responses in their original request order, even if the server returns them out of order.

If you think you'll reuse this debouncing logic, consider moving it to a utility function or into the PackageSearchService itself.

## **Security: XSRF protection**

[Cross-Site Request Forgery (XSRF or CSRF)](https://en.wikipedia.org/wiki/Cross-site_request_forgery) is an attack technique by which the attacker can trick an authenticated user into unknowingly executing actions on your website. [HttpClient](https://angular.io/api/common/http/HttpClient) supports a [common mechanism](https://en.wikipedia.org/wiki/Cross-site_request_forgery#Cookie-to-header_token) used to prevent XSRF attacks. When performing HTTP requests, an interceptor reads a token from a cookie, by default XSRF-TOKEN, and sets it as an HTTP header, X-XSRF-TOKEN. Since only code that runs on your domain could read the cookie, the backend can be certain that the HTTP request came from your client application and not an attacker.

By default, an interceptor sends this header on all mutating requests (such as POST) to relative URLs, but not on GET/HEAD requests or on requests with an absolute URL.

To take advantage of this, your server needs to set a token in a JavaScript readable session cookie called XSRF-TOKEN on either the page load or the first GET request. On subsequent requests the server can verify that the cookie matches the X-XSRF-TOKEN HTTP header, and therefore be sure that only code running on your domain could have sent the request. The token must be unique for each user and must be verifiable by the server; this prevents the client from making up its own tokens. Set the token to a digest of your site's authentication cookie with a salt for added security.

In order to prevent collisions in environments where multiple Angular apps share the same domain or subdomain, give each application a unique cookie name.

[*HttpClient*](https://angular.io/api/common/http/HttpClient) supports only the client half of the XSRF protection scheme. Your backend service must be configured to set the cookie for your page, and to verify that the header is present on all eligible requests. Failing to do so renders Angular's default protection ineffective.

### **Configuring custom cookie/header names**

If your backend service uses different names for the XSRF token cookie or header, use [HttpClientXsrfModule.withOptions()](https://angular.io/api/common/http/HttpClientXsrfModule#withOptions) to override the defaults.

content\_copyimports: [

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

HttpClientXsrfModule.withOptions({

cookieName: 'My-Xsrf-Cookie',

headerName: 'My-Xsrf-Header',

}),

],

## **Testing HTTP requests**

As for any external dependency, you must mock the HTTP backend so your tests can simulate interaction with a remote server. The @angular/common/[http](https://angular.io/api/common/http)/testing library makes it straightforward to set up such mocking.

Angular's HTTP testing library is designed for a pattern of testing in which the app executes code and makes requests first. The test then expects that certain requests have or have not been made, performs assertions against those requests, and finally provides responses by "flushing" each expected request.

At the end, tests can verify that the app has made no unexpected requests.

You can run [these sample tests](https://angular.io/generated/live-examples/http/specs.stackblitz.html) / [download example](https://angular.io/generated/zips/http/specs.http.zip) in a live coding environment.

The tests described in this guide are in src/testing/http-client.spec.ts. There are also tests of an application data service that call [HttpClient](https://angular.io/api/common/http/HttpClient) in src/app/heroes/heroes.service.spec.ts.

### **Setup for testing**

To begin testing calls to [HttpClient](https://angular.io/api/common/http/HttpClient), import the [HttpClientTestingModule](https://angular.io/api/common/http/testing/HttpClientTestingModule) and the mocking controller, [HttpTestingController](https://angular.io/api/common/http/testing/HttpTestingController), along with the other symbols your tests require.

app/testing/http-client.spec.ts (imports)

content\_copy// Http testing module and mocking controller

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

// Other imports

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

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

Then add the [HttpClientTestingModule](https://angular.io/api/common/http/testing/HttpClientTestingModule) to the [TestBed](https://angular.io/api/core/testing/TestBed) and continue with the setup of the service-under-test.

app/testing/http-client.spec.ts(setup)

content\_copydescribe('[HttpClient](https://angular.io/api/common/http/HttpClient) testing', () => {

let httpClient: [HttpClient](https://angular.io/api/common/http/HttpClient);

let httpTestingController: [HttpTestingController](https://angular.io/api/common/http/testing/HttpTestingController);

beforeEach(() => {

TestBed.configureTestingModule({

imports: [ [HttpClientTestingModule](https://angular.io/api/common/http/testing/HttpClientTestingModule) ]

});

// [Inject](https://angular.io/api/core/Inject) the [http](https://angular.io/api/common/http) service and test controller for each test

httpClient = TestBed.inject([HttpClient](https://angular.io/api/common/http/HttpClient));

httpTestingController = TestBed.inject([HttpTestingController](https://angular.io/api/common/http/testing/HttpTestingController));

});

/// Tests begin ///

});

Now requests made in the course of your tests hit the testing backend instead of the normal backend.

This setup also calls TestBed.inject() to inject the [HttpClient](https://angular.io/api/common/http/HttpClient) service and the mocking controller so they can be referenced during the tests.

### **Expecting and answering requests**

Now you can write a test that expects a GET Request to occur and provides a mock response.

app/testing/http-client.spec.ts (HttpClient.get)

content\_copyit('can test HttpClient.get', () => {

const testData: [Data](https://angular.io/api/router/Data) = {name: 'Test [Data](https://angular.io/api/router/Data)'};

// Make an HTTP GET request

httpClient.get<[Data](https://angular.io/api/router/Data)>(testUrl)

.subscribe(data =>

// When observable resolves, result should match test data

expect(data).toEqual(testData)

);

// The following `expectOne()` will match the request's URL.

// If no requests or [multiple](https://angular.io/api/forms/SelectMultipleControlValueAccessor) requests matched that URL

// `expectOne()` would throw.

const req = httpTestingController.expectOne('/data');

// Assert that the request is a GET.

expect(req.request.method).toEqual('GET');

// Respond with mock data, causing Observable to resolve.

// Subscribe callback asserts that correct data was returned.

req.flush(testData);

// Finally, assert that there are no outstanding requests.

httpTestingController.verify();

});

The last step, verifying that no requests remain outstanding, is common enough for you to move it into an afterEach() step:

content\_copyafterEach(() => {

// After every test, assert that there are no more pending requests.

httpTestingController.verify();

});

#### Custom request expectations

If matching by URL isn't sufficient, it's possible to implement your own matching function. For example, you could look for an outgoing request that has an authorization header:

content\_copy// Expect one request with an authorization header

const req = httpTestingController.expectOne(

request => request.headers.has('Authorization')

);

As with the previous expectOne(), the test fails if 0 or 2+ requests satisfy this predicate.

#### Handling more than one request

If you need to respond to duplicate requests in your test, use the match() API instead of expectOne(). It takes the same arguments but returns an array of matching requests. Once returned, these requests are removed from future matching and you are responsible for flushing and verifying them.

content\_copy// get all pending requests that match the given URL

const requests = httpTestingController.match(testUrl);

expect(requests.length).toEqual(3);

// Respond to each request with different results

requests[0].flush([]);

requests[1].flush([testData[0]]);

requests[2].flush(testData);

### **Testing for errors**

You should test the app's defenses against HTTP requests that fail.

Call request.flush() with an error message, as seen in the following example.

content\_copyit('can test for 404 error', () => {

const emsg = 'deliberate 404 error';

httpClient.get<[Data](https://angular.io/api/router/Data)[]>(testUrl).subscribe(

data => fail('should have failed with the 404 error'),

(error: [HttpErrorResponse](https://angular.io/api/common/http/HttpErrorResponse)) => {

expect(error.status).toEqual(404, 'status');

expect(error.error).toEqual(emsg, 'message');

}

);

const req = httpTestingController.expectOne(testUrl);

// Respond with mock error

req.flush(emsg, { status: 404, statusText: 'Not Found' });

});

Alternatively, you can call request.error() with an ErrorEvent.

content\_copyit('can test for network error', () => {

const emsg = 'simulated network error';

httpClient.get<[Data](https://angular.io/api/router/Data)[]>(testUrl).subscribe(

data => fail('should have failed with the network error'),

(error: [HttpErrorResponse](https://angular.io/api/common/http/HttpErrorResponse)) => {

expect(error.error.message).toEqual(emsg, 'message');

}

);

const req = httpTestingController.expectOne(testUrl);

// Create mock ErrorEvent, raised when something goes wrong at the network level.

// Connection timeout, DNS error, offline, etc

const mockError = new ErrorEvent('Network error', {

message: emsg,

});

// Respond with mock error

req.error(mockError);

});

## **Passing metadata to interceptors**

Many interceptors require or benefit from configuration. Consider an interceptor that retries failed requests. By default, the interceptor might retry a request three times, but you might want to override this retry count for particularly error-prone or sensitive requests.

[HttpClient](https://angular.io/api/common/http/HttpClient) requests contain a context that can carry metadata about the request. This context is available for interceptors to read or modify, though it is not transmitted to the backend server when the request is sent. This allows applications or other interceptors to tag requests with configuration parameters, such as how many times to retry a request.

### **Creating a context token**

Angular stores and retrieves a value in the context using an [HttpContextToken](https://angular.io/api/common/http/HttpContextToken). You can create a context token using the new operator, as in the following example:

creating a context token

content\_copyexport const RETRY\_COUNT = new [HttpContextToken](https://angular.io/api/common/http/HttpContextToken)(() => 3);

The lambda function () => 3 passed during the creation of the [HttpContextToken](https://angular.io/api/common/http/HttpContextToken) serves two purposes:

1. It allows TypeScript to infer the type of this token: [HttpContextToken](https://angular.io/api/common/http/HttpContextToken)<number>. The request context is type-safe—reading a token from a request's context returns a value of the appropriate type.
2. It sets the default value for the token. This is the value that the request context returns if no other value has been set for this token. Using a default value avoids the need to check if a particular value is set.

### **Setting context values when making a request**

When making a request, you can provide an [HttpContext](https://angular.io/api/common/http/HttpContext) instance, in which you have already set the context values.

setting context values

content\_copythis.httpClient

.get('/data/feed', {

context: new [HttpContext](https://angular.io/api/common/http/HttpContext)().set(RETRY\_COUNT, 5),

})

.subscribe(results => {/\* ... \*/});

### **Reading context values in an interceptor**

Within an interceptor, you can read the value of a token in a given request's context with [HttpContext.get()](https://angular.io/api/common/http/HttpContext#get). If you have not explicitly set a value for the token, Angular returns the default value specified in the token.

reading context values in an interceptor

content\_copyimport {retry} from 'rxjs';

export class RetryInterceptor implements [HttpInterceptor](https://angular.io/api/common/http/HttpInterceptor) {

intercept(req: [HttpRequest](https://angular.io/api/common/http/HttpRequest)<any>, next: [HttpHandler](https://angular.io/api/common/http/HttpHandler)): Observable<[HttpEvent](https://angular.io/api/common/http/HttpEvent)<any>> {

const retryCount = req.context.get(RETRY\_COUNT);

return next.handle(req).pipe(

// Retry the request a configurable number of times.

retry(retryCount),

);

}

}

### **Contexts are mutable**

Unlike most other aspects of [HttpRequest](https://angular.io/api/common/http/HttpRequest) instances, the request context is mutable and persists across other immutable transformations of the request. This allows interceptors to coordinate operations through the context. For instance, the RetryInterceptor example could use a second context token to track how many errors occur during the execution of a given request:

coordinating operations through the context

content\_copyimport {retry, tap} from 'rxjs/operators';

export const RETRY\_COUNT = new [HttpContextToken](https://angular.io/api/common/http/HttpContextToken)(() => 3);

export const ERROR\_COUNT = new [HttpContextToken](https://angular.io/api/common/http/HttpContextToken)(() => 0);

export class RetryInterceptor implements [HttpInterceptor](https://angular.io/api/common/http/HttpInterceptor) {

intercept(req: [HttpRequest](https://angular.io/api/common/http/HttpRequest)<any>, next: [HttpHandler](https://angular.io/api/common/http/HttpHandler)): Observable<[HttpEvent](https://angular.io/api/common/http/HttpEvent)<any>> {

const retryCount = req.context.get(RETRY\_COUNT);

return next.handle(req).pipe(

tap(null,

() => {

// An error has occurred, so increment this request's ERROR\_COUNT.

req.context.set(ERROR\_COUNT, req.context.get(ERROR\_COUNT) + 1);

}),

// Retry the request a configurable number of times.

retry(retryCount),

);

}

}

# Find out how much code you're testing

The CLI can run unit tests and create code coverage reports. Code coverage reports show you any parts of your code base that may not be properly tested by your unit tests.

For a hands-on experience you can [run tests and explore the test code](https://angular.io/generated/live-examples/testing/specs.stackblitz.html) in your browser as your read this guide.

If you'd like to experiment with the application that this guide describes, you can [run it in your browser](https://angular.io/generated/live-examples/testing/stackblitz.html) or [download and run it locally](https://angular.io/generated/zips/testing/testing.zip).

To generate a coverage report run the following command in the root of your project.

content\_copyng test --no-watch --code-coverage

When the tests are complete, the command creates a new /coverage folder in the project. Open the index.html file to see a report with your source code and code coverage values.

If you want to create code-coverage reports every time you test, you can set the following option in the CLI configuration file, angular.json:

content\_copy"test": {

"options": {

"codeCoverage": true

}

}

## **Code coverage enforcement**

The code coverage percentages let you estimate how much of your code is tested. If your team decides on a set minimum amount to be unit tested, you can enforce this minimum with the Angular CLI.

For example, suppose you want the code base to have a minimum of 80% code coverage. To enable this, open the [Karma](https://karma-runner.github.io/) test platform configuration file, karma.conf.js, and add the check property in the coverageReporter: key.

content\_copycoverageReporter: {

dir: require('path').join(\_\_dirname, './coverage/<project-name>'),

subdir: '.',

reporters: [

{ type: 'html' },

{ type: 'text-summary' }

],

check: {

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

statements: 80,

branches: 80,

functions: 80,

lines: 80

}

}

}

The check property causes the tool to enforce a minimum of 80% code coverage when the unit tests are run in the project.

You can find more information about the different coverage configuration options [here](https://github.com/karma-runner/karma-coverage/blob/master/docs/configuration.md).

# Testing services

To check that your services are working as you intend, you can write tests specifically for them.

For a hands-on experience, you can [run tests and explore the test code](https://angular.io/generated/live-examples/testing/specs.stackblitz.html) in your browser as you read this guide.

If you'd like to experiment with the application that this guide describes, you can [run it in your browser](https://angular.io/generated/live-examples/testing/stackblitz.html) or [download and run it locally](https://angular.io/generated/zips/testing/testing.zip).

Services are often the easiest files to unit test. Here are some synchronous and asynchronous unit tests of the ValueService written without assistance from Angular testing utilities.

app/demo/demo.spec.ts

content\_copy// Straight Jasmine testing without Angular's testing support

describe('ValueService', () => {

let service: ValueService;

beforeEach(() => { service = new ValueService(); });

it('#getValue should return real value', () => {

expect(service.getValue()).toBe('real value');

});

it('#getObservableValue should return value from observable',

(done: DoneFn) => {

service.getObservableValue().subscribe(value => {

expect(value).toBe('observable value');

done();

});

});

it('#getPromiseValue should return value from a promise',

(done: DoneFn) => {

service.getPromiseValue().then(value => {

expect(value).toBe('promise value');

done();

});

});

});

## **Services with dependencies**

Services often depend on other services that Angular injects into the constructor. In many cases, it's easy to create and inject these dependencies by hand while calling the service's constructor.

The MasterService is a simple example:

app/demo/demo.ts

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

export class MasterService {

constructor(private valueService: ValueService) { }

getValue() { return this.valueService.getValue(); }

}

MasterService delegates its only method, getValue, to the injected ValueService.

Here are several ways to test it.

app/demo/demo.spec.ts

content\_copydescribe('MasterService without Angular testing support', () => {

let masterService: MasterService;

it('#getValue should return real value from the real service', () => {

masterService = new MasterService(new ValueService());

expect(masterService.getValue()).toBe('real value');

});

it('#getValue should return faked value from a fakeService', () => {

masterService = new MasterService(new FakeValueService());

expect(masterService.getValue()).toBe('faked service value');

});

it('#getValue should return faked value from a fake object', () => {

const fake = { getValue: () => 'fake value' };

masterService = new MasterService(fake as ValueService);

expect(masterService.getValue()).toBe('fake value');

});

it('#getValue should return stubbed value from a spy', () => {

// create `getValue` spy on an object representing the ValueService

const valueServiceSpy =

jasmine.createSpyObj('ValueService', ['getValue']);

// set the value to return when the `getValue` spy is called.

const stubValue = 'stub value';

valueServiceSpy.getValue.and.returnValue(stubValue);

masterService = new MasterService(valueServiceSpy);

expect(masterService.getValue())

.toBe(stubValue, 'service returned stub value');

expect(valueServiceSpy.getValue.calls.count())

.toBe(1, 'spy method was called once');

expect(valueServiceSpy.getValue.calls.mostRecent().returnValue)

.toBe(stubValue);

});

});

The first test creates a ValueService with new and passes it to the MasterService constructor.

However, injecting the real service rarely works well as most dependent services are difficult to create and control.

Instead you can mock the dependency, use a dummy value, or create a [spy](https://jasmine.github.io/tutorials/your_first_suite#section-Spies) on the pertinent service method.

Prefer spies as they are usually the easiest way to mock services.

These standard testing techniques are great for unit testing services in isolation.

However, you almost always inject services into application classes using Angular dependency injection and you should have tests that reflect that usage pattern. Angular testing utilities make it easy to investigate how injected services behave.

## **Testing services with the TestBed**

Your application relies on Angular [dependency injection (DI)](https://angular.io/guide/dependency-injection) to create services. When a service has a dependent service, DI finds or creates that dependent service. And if that dependent service has its own dependencies, DI finds-or-creates them as well.

As service consumer, you don't worry about any of this. You don't worry about the order of constructor arguments or how they're created.

As a service tester, you must at least think about the first level of service dependencies but you can let Angular DI do the service creation and deal with constructor argument order when you use the [TestBed](https://angular.io/api/core/testing/TestBed) testing utility to provide and create services.

## **Angular TestBed**

The [TestBed](https://angular.io/api/core/testing/TestBed) is the most important of the Angular testing utilities. The [TestBed](https://angular.io/api/core/testing/TestBed) creates a dynamically-constructed Angular test module that emulates an Angular [@NgModule](https://angular.io/guide/ngmodules).

The TestBed.configureTestingModule() method takes a metadata object that can have most of the properties of an [@NgModule](https://angular.io/guide/ngmodules).

To test a service, you set the providers metadata property with an array of the services that you'll test or mock.

app/demo/demo.testbed.spec.ts (provide ValueService in beforeEach)

content\_copylet service: ValueService;

beforeEach(() => {

TestBed.configureTestingModule({ providers: [ValueService] });

});

Then inject it inside a test by calling TestBed.inject() with the service class as the argument.

**Note:** TestBed.get() was deprecated as of Angular version 9. To help minimize breaking changes, Angular introduces a new function called TestBed.inject(), which you should use instead. For information on the removal of TestBed.get(), see its entry in the [Deprecations index](https://angular.io/guide/deprecations#index).

content\_copyit('should use ValueService', () => {

service = TestBed.inject(ValueService);

expect(service.getValue()).toBe('real value');

});

Or inside the beforeEach() if you prefer to inject the service as part of your setup.

content\_copybeforeEach(() => {

TestBed.configureTestingModule({ providers: [ValueService] });

service = TestBed.inject(ValueService);

});

When testing a service with a dependency, provide the mock in the providers array.

In the following example, the mock is a spy object.

content\_copylet masterService: MasterService;

let valueServiceSpy: jasmine.SpyObj<ValueService>;

beforeEach(() => {

const spy = jasmine.createSpyObj('ValueService', ['getValue']);

TestBed.configureTestingModule({

// Provide both the service-to-test and its (spy) dependency

providers: [

MasterService,

{ provide: ValueService, useValue: spy }

]

});

// [Inject](https://angular.io/api/core/Inject) both the service-to-test and its (spy) dependency

masterService = TestBed.inject(MasterService);

valueServiceSpy = TestBed.inject(ValueService) as jasmine.SpyObj<ValueService>;

});

The test consumes that spy in the same way it did earlier.

content\_copyit('#getValue should return stubbed value from a spy', () => {

const stubValue = 'stub value';

valueServiceSpy.getValue.and.returnValue(stubValue);

expect(masterService.getValue())

.toBe(stubValue, 'service returned stub value');

expect(valueServiceSpy.getValue.calls.count())

.toBe(1, 'spy method was called once');

expect(valueServiceSpy.getValue.calls.mostRecent().returnValue)

.toBe(stubValue);

});

## **Testing without beforeEach()**

Most test suites in this guide call beforeEach() to set the preconditions for each it() test and rely on the [TestBed](https://angular.io/api/core/testing/TestBed) to create classes and inject services.

There's another school of testing that never calls beforeEach() and prefers to create classes explicitly rather than use the [TestBed](https://angular.io/api/core/testing/TestBed).

Here's how you might rewrite one of the MasterService tests in that style.

Begin by putting re-usable, preparatory code in a setup function instead of beforeEach().

app/demo/demo.spec.ts (setup)

content\_copyfunction setup() {

const valueServiceSpy =

jasmine.createSpyObj('ValueService', ['getValue']);

const stubValue = 'stub value';

const masterService = new MasterService(valueServiceSpy);

valueServiceSpy.getValue.and.returnValue(stubValue);

return { masterService, stubValue, valueServiceSpy };

}

The setup() function returns an object literal with the variables, such as masterService, that a test might reference. You don't define semi-global variables (for example, let masterService: MasterService) in the body of the describe().

Then each test invokes setup() in its first line, before continuing with steps that manipulate the test subject and assert expectations.

content\_copyit('#getValue should return stubbed value from a spy', () => {

const { masterService, stubValue, valueServiceSpy } = setup();

expect(masterService.getValue())

.toBe(stubValue, 'service returned stub value');

expect(valueServiceSpy.getValue.calls.count())

.toBe(1, 'spy method was called once');

expect(valueServiceSpy.getValue.calls.mostRecent().returnValue)

.toBe(stubValue);

});

Notice how the test uses [destructuring assignment](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment) to extract the setup variables that it needs.

content\_copyconst { masterService, stubValue, valueServiceSpy } = setup();

Many developers feel this approach is cleaner and more explicit than the traditional beforeEach() style.

Although this testing guide follows the traditional style and the default [CLI schematics](https://github.com/angular/angular-cli) generate test files with beforeEach() and [TestBed](https://angular.io/api/core/testing/TestBed), feel free to adopt this alternative approach in your own projects.

## **Testing HTTP services**

Data services that make HTTP calls to remote servers typically inject and delegate to the Angular [HttpClient](https://angular.io/guide/http) service for XHR calls.

You can test a data service with an injected [HttpClient](https://angular.io/api/common/http/HttpClient) spy as you would test any service with a dependency.app/model/hero.service.spec.ts (tests with spies)

content\_copylet httpClientSpy: { get: jasmine.Spy };

let heroService: HeroService;

beforeEach(() => {

// TODO: spy on other methods too

httpClientSpy = jasmine.createSpyObj('[HttpClient](https://angular.io/api/common/http/HttpClient)', ['get']);

heroService = new HeroService(httpClientSpy as any);

});

it('should return expected heroes ([HttpClient](https://angular.io/api/common/http/HttpClient) called once)', (done: DoneFn) => {

const expectedHeroes: Hero[] =

[{ id: 1, name: 'A' }, { id: 2, name: 'B' }];

httpClientSpy.get.and.returnValue(asyncData(expectedHeroes));

heroService.getHeroes().subscribe(

heroes => {

expect(heroes).toEqual(expectedHeroes, 'expected heroes');

done();

},

done.fail

);

expect(httpClientSpy.get.calls.count()).toBe(1, 'one call');

});

it('should return an error when the server returns a 404', (done: DoneFn) => {

const errorResponse = new [HttpErrorResponse](https://angular.io/api/common/http/HttpErrorResponse)({

error: 'test 404 error',

status: 404, statusText: 'Not Found'

});

httpClientSpy.get.and.returnValue(asyncError(errorResponse));

heroService.getHeroes().subscribe(

heroes => done.fail('expected an error, not heroes'),

error => {

expect(error.message).toContain('test 404 error');

done();

}

);

});

The HeroService methods return Observables. You must subscribe to an observable to (a) cause it to execute and (b) assert that the method succeeds or fails.

The subscribe() method takes a success (next) and fail (error) callback. Make sure you provide both callbacks so that you capture errors. Neglecting to do so produces an asynchronous uncaught observable error that the test runner will likely attribute to a completely different test.

## **HttpClientTestingModule**

Extended interactions between a data service and the [HttpClient](https://angular.io/api/common/http/HttpClient) can be complex and difficult to mock with spies.

The [HttpClientTestingModule](https://angular.io/api/common/http/testing/HttpClientTestingModule) can make these testing scenarios more manageable.

While the code sample accompanying this guide demonstrates [HttpClientTestingModule](https://angular.io/api/common/http/testing/HttpClientTestingModule), this page defers to the [Http guide](https://angular.io/guide/http#testing-http-requests), which covers testing with the [HttpClientTestingModule](https://angular.io/api/common/http/testing/HttpClientTestingModule) in detail.

# Basics of testing components

A component, unlike all other parts of an Angular application, combines an HTML template and a TypeScript class. The component truly is the template and the class working together. To adequately test a component, you should test that they work together as intended.

Such tests require creating the component's host element in the browser DOM, as Angular does, and investigating the component class's interaction with the DOM as described by its template.

The Angular [TestBed](https://angular.io/api/core/testing/TestBed) facilitates this kind of testing as you'll see in the sections below. But in many cases, testing the component class alone, without DOM involvement, can validate much of the component's behavior in an easier, more obvious way.

For a hands-on experience you can [run tests and explore the test code](https://angular.io/generated/live-examples/testing/specs.stackblitz.html) in your browser as your read this guide.

If you'd like to experiment with the application that this guide describes, you can [run it in your browser](https://angular.io/generated/live-examples/testing/stackblitz.html) or [download and run it locally](https://angular.io/generated/zips/testing/testing.zip).

## **Component class testing**

Test a component class on its own as you would test a service class.

Component class testing should be kept very clean and simple. It should test only a single unit. At first glance, you should be able to understand what the test is testing.

Consider this LightswitchComponent which toggles a light on and off (represented by an on-screen message) when the user clicks the button.

app/demo/demo.ts (LightswitchComp)

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

selector: 'lightswitch-comp',

template: `

<button (click)="clicked()">Click me!</button>

<span>{{message}}</span>`

})

export class LightswitchComponent {

isOn = false;

clicked() { this.isOn = !this.isOn; }

get message() { return `The light is ${this.isOn ? 'On' : 'Off'}`; }

}

You might decide only to test that the clicked() method toggles the light's on/off state and sets the message appropriately.

This component class has no dependencies. To test these types of classes, follow the same steps as you would for a service that has no dependencies:

1. Create a component using the new keyword.
2. Poke at its API.
3. Assert expectations on its public state.

app/demo/demo.spec.ts (Lightswitch tests)

content\_copydescribe('LightswitchComp', () => {

it('#clicked() should toggle #isOn', () => {

const comp = new LightswitchComponent();

expect(comp.isOn).toBe(false, 'off at first');

comp.clicked();

expect(comp.isOn).toBe(true, 'on after click');

comp.clicked();

expect(comp.isOn).toBe(false, 'off after second click');

});

it('#clicked() should set #message to "is on"', () => {

const comp = new LightswitchComponent();

expect(comp.message).toMatch(/is off/i, 'off at first');

comp.clicked();

expect(comp.message).toMatch(/is on/i, 'on after clicked');

});

});

Here is the DashboardHeroComponent from the Tour of Heroes tutorial.

app/dashboard/dashboard-hero.component.ts (component)

content\_copyexport class DashboardHeroComponent {

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

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

click() { this.selected.emit(this.hero); }

}

It appears within the template of a parent component, which binds a hero to the @[Input](https://angular.io/api/core/Input) property and listens for an event raised through the selected @[Output](https://angular.io/api/core/Output) property.

You can test that the class code works without creating the DashboardHeroComponent or its parent component.

app/dashboard/dashboard-hero.component.spec.ts (class tests)

content\_copyit('raises the selected event when clicked', () => {

const comp = new DashboardHeroComponent();

const hero: Hero = {id: 42, name: 'Test'};

comp.hero = hero;

comp.selected.pipe(first()).subscribe((selectedHero: Hero) => expect(selectedHero).toBe(hero));

comp.click();

});

When a component has dependencies, you may want to use the [TestBed](https://angular.io/api/core/testing/TestBed) to both create the component and its dependencies.

The following WelcomeComponent depends on the UserService to know the name of the user to greet.

app/welcome/welcome.component.ts

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

welcome = '';

constructor(private userService: UserService) { }

ngOnInit(): void {

this.welcome = this.userService.isLoggedIn ?

'Welcome, ' + this.userService.user.name : 'Please log in.';

}

}

You might start by creating a mock of the UserService that meets the minimum needs of this component.

app/welcome/welcome.component.spec.ts (MockUserService)

content\_copyclass MockUserService {

isLoggedIn = true;

user = { name: 'Test User'};

}

Then provide and inject both the **component** and the service in the [TestBed](https://angular.io/api/core/testing/TestBed) configuration.

app/welcome/welcome.component.spec.ts (class-only setup)

content\_copybeforeEach(() => {

TestBed.configureTestingModule({

// provide the component-under-test and dependent service

providers: [

WelcomeComponent,

{ provide: UserService, useClass: MockUserService }

]

});

// inject both the component and the dependent service.

comp = TestBed.inject(WelcomeComponent);

userService = TestBed.inject(UserService);

});

Then exercise the component class, remembering to call the [lifecycle hook methods](https://angular.io/guide/lifecycle-hooks) as Angular does when running the application.

app/welcome/welcome.component.spec.ts (class-only tests)

content\_copyit('should not have welcome message after construction', () => {

expect(comp.welcome).toBe('');

});

it('should welcome logged in user after Angular calls ngOnInit', () => {

comp.ngOnInit();

expect(comp.welcome).toContain(userService.user.name);

});

it('should ask user to log in if not logged in after ngOnInit', () => {

userService.isLoggedIn = false;

comp.ngOnInit();

expect(comp.welcome).not.toContain(userService.user.name);

expect(comp.welcome).toContain('log in');

});

## **Component DOM testing**

Testing the component class is as easy as [testing a service](https://angular.io/guide/testing-services).

But a component is more than just its class. A component interacts with the DOM and with other components. The class-only tests can tell you about class behavior. They cannot tell you if the component is going to render properly, respond to user input and gestures, or integrate with its parent and child components.

None of the class-only tests above can answer key questions about how the components actually behave on screen.

* Is Lightswitch.clicked() bound to anything such that the user can invoke it?
* Is the Lightswitch.message displayed?
* Can the user actually select the hero displayed by DashboardHeroComponent?
* Is the hero name displayed as expected (i.e, in uppercase)?
* Is the welcome message displayed by the template of WelcomeComponent?

These may not be troubling questions for the simple components illustrated above. But many components have complex interactions with the DOM elements described in their templates, causing HTML to appear and disappear as the component state changes.

To answer these kinds of questions, you have to create the DOM elements associated with the components, you must examine the DOM to confirm that component state displays properly at the appropriate times, and you must simulate user interaction with the screen to determine whether those interactions cause the component to behave as expected.

To write these kinds of test, you'll use additional features of the [TestBed](https://angular.io/api/core/testing/TestBed) as well as other testing helpers.

### **CLI-generated tests**

The CLI creates an initial test file for you by default when you ask it to generate a new component.

For example, the following CLI command generates a BannerComponent in the app/banner folder (with inline template and styles):

content\_copyng generate component banner --inline-template --inline-style --module app

It also generates an initial test file for the component, banner-external.component.spec.ts, that looks like this:

app/banner/banner-external.component.spec.ts (initial)

content\_copyimport { [ComponentFixture](https://angular.io/api/core/testing/ComponentFixture), [TestBed](https://angular.io/api/core/testing/TestBed), [waitForAsync](https://angular.io/api/core/testing/waitForAsync) } from '@angular/core/testing';

import { BannerComponent } from './banner.component';

describe('BannerComponent', () => {

let component: BannerComponent;

let fixture: [ComponentFixture](https://angular.io/api/core/testing/ComponentFixture)<BannerComponent>;

beforeEach([waitForAsync](https://angular.io/api/core/testing/waitForAsync)(() => {

TestBed.configureTestingModule({declarations: [BannerComponent]}).compileComponents();

}));

beforeEach(() => {

fixture = TestBed.createComponent(BannerComponent);

component = fixture.componentInstance;

fixture.detectChanges();

});

it('should create', () => {

expect(component).toBeDefined();

});

});

Because compileComponents is asynchronous, it uses the [waitForAsync](https://angular.io/api/core/testing/waitForAsync) utility function imported from @angular/core/testing.

Refer to the [waitForAsync](https://angular.io/guide/testing-components-scenarios#waitForAsync) section for more details.

### **Reduce the setup**

Only the last three lines of this file actually test the component and all they do is assert that Angular can create the component.

The rest of the file is boilerplate setup code anticipating more advanced tests that might become necessary if the component evolves into something substantial.

You'll learn about these advanced test features below. For now, you can radically reduce this test file to a more manageable size:

app/banner/banner-initial.component.spec.ts (minimal)

content\_copydescribe('BannerComponent (minimal)', () => {

it('should create', () => {

TestBed.configureTestingModule({declarations: [BannerComponent]});

const fixture = TestBed.createComponent(BannerComponent);

const component = fixture.componentInstance;

expect(component).toBeDefined();

});

});

In this example, the metadata object passed to TestBed.configureTestingModule simply declares BannerComponent, the component to test.

content\_copyTestBed.configureTestingModule({declarations: [BannerComponent]});

There's no need to declare or import anything else. The default test module is pre-configured with something like the [BrowserModule](https://angular.io/api/platform-browser/BrowserModule) from @angular/platform-browser.

Later you'll call TestBed.configureTestingModule() with imports, providers, and more declarations to suit your testing needs. Optional override methods can further fine-tune aspects of the configuration.

### **createComponent()**

After configuring [TestBed](https://angular.io/api/core/testing/TestBed), you call its createComponent() method.

content\_copyconst fixture = TestBed.createComponent(BannerComponent);

TestBed.createComponent() creates an instance of the BannerComponent, adds a corresponding element to the test-runner DOM, and returns a [ComponentFixture](https://angular.io/guide/testing-components-basics#component-fixture).

Do not re-configure [TestBed](https://angular.io/api/core/testing/TestBed) after calling createComponent.

The createComponent method freezes the current [TestBed](https://angular.io/api/core/testing/TestBed) definition, closing it to further configuration.

You cannot call any more [TestBed](https://angular.io/api/core/testing/TestBed) configuration methods, not configureTestingModule(), nor get(), nor any of the override... methods. If you try, [TestBed](https://angular.io/api/core/testing/TestBed) throws an error.

### **ComponentFixture**

The [ComponentFixture](https://angular.io/api/core/testing/ComponentFixture) is a test harness for interacting with the created component and its corresponding element.

Access the component instance through the fixture and confirm it exists with a Jasmine expectation:

content\_copyconst component = fixture.componentInstance;

expect(component).toBeDefined();

### **beforeEach()**

You will add more tests as this component evolves. Rather than duplicate the [TestBed](https://angular.io/api/core/testing/TestBed) configuration for each test, you refactor to pull the setup into a Jasmine beforeEach() and some supporting variables:

content\_copydescribe('BannerComponent (with beforeEach)', () => {

let component: BannerComponent;

let fixture: [ComponentFixture](https://angular.io/api/core/testing/ComponentFixture)<BannerComponent>;

beforeEach(() => {

TestBed.configureTestingModule({declarations: [BannerComponent]});

fixture = TestBed.createComponent(BannerComponent);

component = fixture.componentInstance;

});

it('should create', () => {

expect(component).toBeDefined();

});

});

Now add a test that gets the component's element from fixture.nativeElement and looks for the expected text.

content\_copyit('should contain "banner works!"', () => {

const bannerElement: HTMLElement = fixture.nativeElement;

expect(bannerElement.textContent).toContain('banner works!');

});

### **nativeElement**

The value of [ComponentFixture.nativeElement](https://angular.io/api/core/testing/ComponentFixture#nativeElement) has the any type. Later you'll encounter the [DebugElement.nativeElement](https://angular.io/api/core/DebugElement#nativeElement) and it too has the any type.

Angular can't know at compile time what kind of HTML element the nativeElement is or if it even is an HTML element. The application might be running on a non-browser platform, such as the server or a [Web Worker](https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API), where the element may have a diminished API or not exist at all.

The tests in this guide are designed to run in a browser so a nativeElement value will always be an HTMLElement or one of its derived classes.

Knowing that it is an HTMLElement of some sort, you can use the standard HTML querySelector to dive deeper into the element tree.

Here's another test that calls HTMLElement.querySelector to get the paragraph element and look for the banner text:

content\_copyit('should have <p> with "banner works!"', () => {

const bannerElement: HTMLElement = fixture.nativeElement;

const p = bannerElement.querySelector('p')!;

expect(p.textContent).toEqual('banner works!');

});

### **DebugElement**

The Angular fixture provides the component's element directly through the fixture.nativeElement.

content\_copyconst bannerElement: HTMLElement = fixture.nativeElement;

This is actually a convenience method, implemented as fixture.debugElement.nativeElement.

content\_copyconst bannerDe: [DebugElement](https://angular.io/api/core/DebugElement) = fixture.debugElement;

const bannerEl: HTMLElement = bannerDe.nativeElement;

There's a good reason for this circuitous path to the element.

The properties of the nativeElement depend upon the runtime environment. You could be running these tests on a non-browser platform that doesn't have a DOM or whose DOM-emulation doesn't support the full HTMLElement API.

Angular relies on the [DebugElement](https://angular.io/api/core/DebugElement) abstraction to work safely across all supported platforms. Instead of creating an HTML element tree, Angular creates a [DebugElement](https://angular.io/api/core/DebugElement) tree that wraps the native elements for the runtime platform. The nativeElement property unwraps the [DebugElement](https://angular.io/api/core/DebugElement) and returns the platform-specific element object.

Because the sample tests for this guide are designed to run only in a browser, a nativeElement in these tests is always an HTMLElement whose familiar methods and properties you can explore within a test.

Here's the previous test, re-implemented with fixture.debugElement.nativeElement:

content\_copyit('should find the <p> with fixture.debugElement.nativeElement)', () => {

const bannerDe: [DebugElement](https://angular.io/api/core/DebugElement) = fixture.debugElement;

const bannerEl: HTMLElement = bannerDe.nativeElement;

const p = bannerEl.querySelector('p')!;

expect(p.textContent).toEqual('banner works!');

});

The [DebugElement](https://angular.io/api/core/DebugElement) has other methods and properties that are useful in tests, as you'll see elsewhere in this guide.

You import the [DebugElement](https://angular.io/api/core/DebugElement) symbol from the Angular core library.

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

### **By.css()**

Although the tests in this guide all run in the browser, some applications might run on a different platform at least some of the time.

For example, the component might render first on the server as part of a strategy to make the application launch faster on poorly connected devices. The server-side renderer might not support the full HTML element API. If it doesn't support querySelector, the previous test could fail.

The [DebugElement](https://angular.io/api/core/DebugElement) offers query methods that work for all supported platforms. These query methods take a predicate function that returns true when a node in the [DebugElement](https://angular.io/api/core/DebugElement) tree matches the selection criteria.

You create a predicate with the help of a [By](https://angular.io/api/platform-browser/By) class imported from a library for the runtime platform. Here's the [By](https://angular.io/api/platform-browser/By) import for the browser platform:

content\_copyimport { [By](https://angular.io/api/platform-browser/By) } from '@angular/platform-browser';

The following example re-implements the previous test with DebugElement.query() and the browser's By.css method.

content\_copyit('should find the <p> with fixture.debugElement.query(By.css)', () => {

const bannerDe: [DebugElement](https://angular.io/api/core/DebugElement) = fixture.debugElement;

const paragraphDe = bannerDe.query(By.css('p'));

const p: HTMLElement = paragraphDe.nativeElement;

expect(p.textContent).toEqual('banner works!');

});

Some noteworthy observations:

* The [By.css()](https://angular.io/api/platform-browser/By#css) static method selects [DebugElement](https://angular.io/api/core/DebugElement) nodes with a [standard CSS selector](https://developer.mozilla.org/en-US/docs/Web/Guide/CSS/Getting_started/Selectors).
* The query returns a [DebugElement](https://angular.io/api/core/DebugElement) for the paragraph.
* You must unwrap that result to get the paragraph element.

When you're filtering by CSS selector and only testing properties of a browser's native element, the By.css approach may be overkill.

It's often easier and more clear to filter with a standard HTMLElement method such as querySelector() or querySelectorAll().

# Component testing scenarios

This guide explores common component testing use cases.

For a hands-on experience you can [run tests and explore the test code](https://angular.io/generated/live-examples/testing/specs.stackblitz.html) in your browser as your read this guide.

If you'd like to experiment with the application that this guide describes, you can [run it in your browser](https://angular.io/generated/live-examples/testing/stackblitz.html) or [download and run it locally](https://angular.io/generated/zips/testing/testing.zip).

## **Component binding**

In the example app, the BannerComponent presents static title text in the HTML template.

After a few changes, the BannerComponent presents a dynamic title by binding to the component's title property like this.

app/banner/banner.component.ts

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

selector: 'app-banner',

template: '<h1>{{title}}</h1>',

styles: ['h1 { color: green; font-size: 350%}']

})

export class BannerComponent {

title = 'Test Tour of Heroes';

}

As minimal as this is, you decide to add a test to confirm that component actually displays the right content where you think it should.

#### Query for the <h1>

You'll write a sequence of tests that inspect the value of the <h1> element that wraps the title property interpolation binding.

You update the beforeEach to find that element with a standard HTML querySelector and assign it to the h1 variable.

app/banner/banner.component.spec.ts (setup)

content\_copylet component: BannerComponent;

let fixture: [ComponentFixture](https://angular.io/api/core/testing/ComponentFixture)<BannerComponent>;

let h1: HTMLElement;

beforeEach(() => {

TestBed.configureTestingModule({

declarations: [ BannerComponent ],

});

fixture = TestBed.createComponent(BannerComponent);

component = fixture.componentInstance; // BannerComponent test instance

h1 = fixture.nativeElement.querySelector('h1');

});

#### createComponent() does not bind data

For your first test you'd like to see that the screen displays the default title. Your instinct is to write a test that immediately inspects the <h1> like this:

content\_copyit('should display original title', () => {

expect(h1.textContent).toContain(component.title);

});

That test fails with the message:

content\_copyexpected '' to contain 'Test Tour of Heroes'.

Binding happens when Angular performs **change detection**.

In production, change detection kicks in automatically when Angular creates a component or the user enters a keystroke or an asynchronous activity (for example, AJAX) completes.

The TestBed.createComponent does not trigger change detection; a fact confirmed in the revised test:

content\_copyit('no title in the DOM after createComponent()', () => {

expect(h1.textContent).toEqual('');

});

#### detectChanges()

You must tell the [TestBed](https://angular.io/api/core/testing/TestBed) to perform data binding by calling fixture.detectChanges(). Only then does the <h1> have the expected title.

content\_copyit('should display original title after detectChanges()', () => {

fixture.detectChanges();

expect(h1.textContent).toContain(component.title);

});

Delayed change detection is intentional and useful. It gives the tester an opportunity to inspect and change the state of the component before Angular initiates data binding and calls [*lifecycle hooks*](https://angular.io/guide/lifecycle-hooks).

Here's another test that changes the component's title property before calling fixture.detectChanges().

content\_copyit('should display a different test title', () => {

component.title = 'Test [Title](https://angular.io/api/platform-browser/Title)';

fixture.detectChanges();

expect(h1.textContent).toContain('Test [Title](https://angular.io/api/platform-browser/Title)');

});

#### Automatic change detection

The BannerComponent tests frequently call detectChanges. Some testers prefer that the Angular test environment run change detection automatically.

That's possible by configuring the [TestBed](https://angular.io/api/core/testing/TestBed) with the [ComponentFixtureAutoDetect](https://angular.io/api/core/testing/ComponentFixtureAutoDetect) provider. First import it from the testing utility library:

app/banner/banner.component.detect-changes.spec.ts (import)

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

Then add it to the providers array of the testing module configuration:

app/banner/banner.component.detect-changes.spec.ts (AutoDetect)

content\_copyTestBed.configureTestingModule({

declarations: [ BannerComponent ],

providers: [

{ provide: [ComponentFixtureAutoDetect](https://angular.io/api/core/testing/ComponentFixtureAutoDetect), useValue: true }

]

});

Here are three tests that illustrate how automatic change detection works.

app/banner/banner.component.detect-changes.spec.ts (AutoDetect Tests)

content\_copyit('should display original title', () => {

// Hooray! No `fixture.detectChanges()` needed

expect(h1.textContent).toContain(comp.title);

});

it('should still see original title after comp.title change', () => {

const oldTitle = comp.title;

comp.title = 'Test [Title](https://angular.io/api/platform-browser/Title)';

// Displayed title is old because Angular didn't hear the change :(

expect(h1.textContent).toContain(oldTitle);

});

it('should display updated title after detectChanges', () => {

comp.title = 'Test [Title](https://angular.io/api/platform-browser/Title)';

fixture.detectChanges(); // detect changes explicitly

expect(h1.textContent).toContain(comp.title);

});

The first test shows the benefit of automatic change detection.

The second and third test reveal an important limitation. The Angular testing environment does not know that the test changed the component's title. The [ComponentFixtureAutoDetect](https://angular.io/api/core/testing/ComponentFixtureAutoDetect) service responds to asynchronous activities such as promise resolution, timers, and DOM events. But a direct, synchronous update of the component property is invisible. The test must call fixture.detectChanges() manually to trigger another cycle of change detection.

Rather than wonder when the test fixture will or won't perform change detection, the samples in this guide always call detectChanges() explicitly. There is no harm in calling detectChanges() more often than is strictly necessary.

#### Change an input value with dispatchEvent()

To simulate user input, you can find the input element and set its value property.

You will call fixture.detectChanges() to trigger Angular's change detection. But there is an essential, intermediate step.

Angular doesn't know that you set the input element's value property. It won't read that property until you raise the element's input event by calling dispatchEvent(). Then you call detectChanges().

The following example demonstrates the proper sequence.

app/hero/hero-detail.component.spec.ts (pipe test)

content\_copyit('should convert hero name to [Title](https://angular.io/api/platform-browser/Title) Case', () => {

// get the name's input and display elements from the DOM

const hostElement: HTMLElement = fixture.nativeElement;

const nameInput: HTMLInputElement = hostElement.querySelector('input')!;

const nameDisplay: HTMLElement = hostElement.querySelector('span')!;

// simulate user entering a new name into the input box

nameInput.value = 'quick BROWN fOx';

// Dispatch a DOM event so that Angular learns of input value change.

// In older browsers, such as IE, you might need a CustomEvent instead. See

// https://developer.mozilla.org/en-US/docs/Web/API/CustomEvent/CustomEvent#Polyfill

nameInput.dispatchEvent(new [Event](https://angular.io/api/router/Event)('input'));

// Tell Angular to update the display binding through the title pipe

fixture.detectChanges();

expect(nameDisplay.textContent).toBe('Quick Brown Fox');

});

## **Component with external files**

The BannerComponent above is defined with an inline template and inline css, specified in the @[Component.template](https://angular.io/api/core/Component#template) and @[Component.styles](https://angular.io/api/core/Component#styles) properties respectively.

Many components specify external templates and external css with the @[Component.templateUrl](https://angular.io/api/core/Component#templateUrl) and @[Component.styleUrls](https://angular.io/api/core/Component#styleUrls) properties respectively, as the following variant of BannerComponent does.

app/banner/banner-external.component.ts (metadata)

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

selector: 'app-banner',

templateUrl: './banner-external.component.html',

styleUrls: ['./banner-external.component.css']

})

This syntax tells the Angular compiler to read the external files during component compilation.

That's not a problem when you run the CLI ng test command because it compiles the application before running the tests.

However, if you run the tests in a **non-CLI environment**, tests of this component may fail. For example, if you run the BannerComponent tests in a web coding environment such as [plunker](https://plnkr.co/), you'll see a message like this one:

Error: This test module uses the component BannerComponent

which is using a "templateUrl" or "styleUrls", but they were never compiled.

Please call "TestBed.compileComponents" before your test.

You get this test failure message when the runtime environment compiles the source code during the tests themselves.

To correct the problem, call compileComponents() as explained [below](https://angular.io/guide/testing-components-scenarios#compile-components).

## **Component with a dependency**

Components often have service dependencies.

The WelcomeComponent displays a welcome message to the logged in user. It knows who the user is based on a property of the injected UserService:

app/welcome/welcome.component.ts

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

import { UserService } from '../model/user.service';

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

selector: 'app-welcome',

template: '<h3 class="welcome"><i>{{welcome}}</i></h3>'

})

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

welcome = '';

constructor(private userService: UserService) { }

ngOnInit(): void {

this.welcome = this.userService.isLoggedIn ?

'Welcome, ' + this.userService.user.name : 'Please log in.';

}

}

The WelcomeComponent has decision logic that interacts with the service, logic that makes this component worth testing. Here's the testing module configuration for the spec file:

app/welcome/welcome.component.spec.ts

content\_copyTestBed.configureTestingModule({

declarations: [ WelcomeComponent ],

// providers: [ UserService ], // NO! Don't provide the real service!

// Provide a test-double instead

providers: [ { provide: UserService, useValue: userServiceStub } ],

});

This time, in addition to declaring the component-under-test, the configuration adds a UserService provider to the providers list. But not the real UserService.

#### Provide service test doubles

A component-under-test doesn't have to be injected with real services. In fact, it is usually better if they are test doubles (stubs, fakes, spies, or mocks). The purpose of the spec is to test the component, not the service, and real services can be trouble.

Injecting the real UserService could be a nightmare. The real service might ask the user for login credentials and attempt to reach an authentication server. These behaviors can be hard to intercept. It is far easier and safer to create and register a test double in place of the real UserService.

This particular test suite supplies a minimal mock of the UserService that satisfies the needs of the WelcomeComponent and its tests:

app/welcome/welcome.component.spec.ts

content\_copylet userServiceStub: Partial<UserService>;

userServiceStub = {

isLoggedIn: true,

user: { name: 'Test User' },

};

#### Get injected services

The tests need access to the (stub) UserService injected into the WelcomeComponent.

Angular has a hierarchical injection system. There can be injectors at multiple levels, from the root injector created by the [TestBed](https://angular.io/api/core/testing/TestBed) down through the component tree.

The safest way to get the injected service, the way that always works, is to **get it from the injector of the**component-under-test. The component injector is a property of the fixture's [DebugElement](https://angular.io/api/core/DebugElement).

WelcomeComponent's injector

content\_copy// UserService actually injected into the component

userService = fixture.debugElement.injector.get(UserService);

#### TestBed.inject()

You may also be able to get the service from the root injector using TestBed.inject(). This is easier to remember and less verbose. But it only works when Angular injects the component with the service instance in the test's root injector.

In this test suite, the only provider of UserService is the root testing module, so it is safe to call TestBed.inject() as follows:

TestBed injector

content\_copy// UserService from the root injector

userService = TestBed.inject(UserService);

For a use case in which TestBed.inject() does not work, see the [Override component providers](https://angular.io/guide/testing-components-scenarios#component-override) section that explains when and why you must get the service from the component's injector instead.

#### Final setup and tests

Here's the complete beforeEach(), using TestBed.inject():

app/welcome/welcome.component.spec.ts

content\_copylet userServiceStub: Partial<UserService>;

beforeEach(() => {

// stub UserService for test purposes

userServiceStub = {

isLoggedIn: true,

user: { name: 'Test User' },

};

TestBed.configureTestingModule({

declarations: [ WelcomeComponent ],

providers: [ { provide: UserService, useValue: userServiceStub } ],

});

fixture = TestBed.createComponent(WelcomeComponent);

comp = fixture.componentInstance;

// UserService from the root injector

userService = TestBed.inject(UserService);

// get the "welcome" element by CSS selector (e.g., by class name)

el = fixture.nativeElement.querySelector('.welcome');

});

And here are some tests:

app/welcome/welcome.component.spec.ts

content\_copyit('should welcome the user', () => {

fixture.detectChanges();

const content = el.textContent;

expect(content).toContain('Welcome', '"Welcome ..."');

expect(content).toContain('Test User', 'expected name');

});

it('should welcome "Bubba"', () => {

userService.user.name = 'Bubba'; // welcome message hasn't been shown yet

fixture.detectChanges();

expect(el.textContent).toContain('Bubba');

});

it('should request login if not logged in', () => {

userService.isLoggedIn = false; // welcome message hasn't been shown yet

fixture.detectChanges();

const content = el.textContent;

expect(content).not.toContain('Welcome', 'not welcomed');

expect(content).toMatch(/log in/i, '"log in"');

});

The first is a sanity test; it confirms that the stubbed UserService is called and working.

The second parameter to the Jasmine matcher (for example, 'expected name') is an optional failure label. If the expectation fails, Jasmine appends this label to the expectation failure message. In a spec with multiple expectations, it can help clarify what went wrong and which expectation failed.

The remaining tests confirm the logic of the component when the service returns different values. The second test validates the effect of changing the user name. The third test checks that the component displays the proper message when there is no logged-in user.

## **Component with async service**

In this sample, the AboutComponent template hosts a TwainComponent. The TwainComponent displays Mark Twain quotes.

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

content\_copytemplate: `

<p class="twain"><i>{{quote | [async](https://angular.io/api/common/AsyncPipe)}}</i></p>

<button (click)="getQuote()">Next quote</button>

<p class="error" \*[ngIf](https://angular.io/api/common/NgIf)="errorMessage">{{ errorMessage }}</p>`,

Note that the value of the component's quote property passes through an [AsyncPipe](https://angular.io/api/common/AsyncPipe). That means the property returns either a Promise or an Observable.

In this example, the TwainComponent.getQuote() method tells you that the quote property returns an Observable.

app/twain/twain.component.ts (getQuote)

content\_copygetQuote() {

this.errorMessage = '';

this.quote = this.twainService.getQuote().pipe(

startWith('...'),

catchError( (err: any) => {

// Wait a turn because errorMessage already set once this turn

setTimeout(() => this.errorMessage = err.message || err.toString());

return of('...'); // reset message to placeholder

})

);

The TwainComponent gets quotes from an injected TwainService. The component starts the returned Observable with a placeholder value ('...'), before the service can return its first quote.

The catchError intercepts service errors, prepares an error message, and returns the placeholder value on the success channel. It must wait a tick to set the errorMessage in order to avoid updating that message twice in the same change detection cycle.

These are all features you'll want to test.

#### Testing with a spy

When testing a component, only the service's public API should matter. In general, tests themselves should not make calls to remote servers. They should emulate such calls. The setup in this app/twain/twain.component.spec.ts shows one way to do that:

app/twain/twain.component.spec.ts (setup)

content\_copybeforeEach(() => {

testQuote = 'Test Quote';

// Create a fake TwainService object with a `getQuote()` spy

const twainService = jasmine.createSpyObj('TwainService', ['getQuote']);

// Make the spy return a synchronous Observable with the test data

getQuoteSpy = twainService.getQuote.and.returnValue(of(testQuote));

TestBed.configureTestingModule({

declarations: [TwainComponent],

providers: [{provide: TwainService, useValue: twainService}]

});

fixture = TestBed.createComponent(TwainComponent);

component = fixture.componentInstance;

quoteEl = fixture.nativeElement.querySelector('.twain');

});

Focus on the spy.

content\_copy// Create a fake TwainService object with a `getQuote()` spy

const twainService = jasmine.createSpyObj('TwainService', ['getQuote']);

// Make the spy return a synchronous Observable with the test data

getQuoteSpy = twainService.getQuote.and.returnValue(of(testQuote));

The spy is designed such that any call to getQuote receives an observable with a test quote. Unlike the real getQuote() method, this spy bypasses the server and returns a synchronous observable whose value is available immediately.

You can write many useful tests with this spy, even though its Observable is synchronous.

#### Synchronous tests

A key advantage of a synchronous Observable is that you can often turn asynchronous processes into synchronous tests.

content\_copyit('should show quote after component initialized', () => {

fixture.detectChanges(); // onInit()

// sync spy result shows testQuote immediately after init

expect(quoteEl.textContent).toBe(testQuote);

expect(getQuoteSpy.calls.any()).toBe(true, 'getQuote called');

});

Because the spy result returns synchronously, the getQuote() method updates the message on screen immediately after the first change detection cycle during which Angular calls ngOnInit.

You're not so lucky when testing the error path. Although the service spy will return an error synchronously, the component method calls setTimeout(). The test must wait at least one full turn of the JavaScript engine before the value becomes available. The test must become asynchronous.

#### Async test with fakeAsync()

To use [fakeAsync](https://angular.io/api/core/testing/fakeAsync)() functionality, you must import zone.js/testing in your test setup file. If you created your project with the Angular CLI, zone-testing is already imported in src/test.ts.

The following test confirms the expected behavior when the service returns an ErrorObservable.

content\_copyit('should display error when TwainService fails', [fakeAsync](https://angular.io/api/core/testing/fakeAsync)(() => {

// tell spy to return an error observable

getQuoteSpy.and.returnValue(throwError('TwainService test failure'));

fixture.detectChanges(); // onInit()

// sync spy errors immediately after init

[tick](https://angular.io/api/core/testing/tick)(); // [flush](https://angular.io/api/core/testing/flush) the component's setTimeout()

fixture.detectChanges(); // update errorMessage within setTimeout()

expect(errorMessage()).toMatch(/test failure/, 'should display error');

expect(quoteEl.textContent).toBe('...', 'should show placeholder');

}));

Note that the it() function receives an argument of the following form.

content\_copy[fakeAsync](https://angular.io/api/core/testing/fakeAsync)(() => { /\* test body \*/ })

The [fakeAsync](https://angular.io/api/core/testing/fakeAsync)() function enables a linear coding style by running the test body in a special [fakeAsync](https://angular.io/api/core/testing/fakeAsync) test zone. The test body appears to be synchronous. There is no nested syntax (like a Promise.then()) to disrupt the flow of control.

Limitation: The [fakeAsync](https://angular.io/api/core/testing/fakeAsync)() function won't work if the test body makes an XMLHttpRequest (XHR) call. XHR calls within a test are rare, but if you need to call XHR, see [waitForAsync()](https://angular.io/guide/testing-components-scenarios#waitForAsync), below.

#### The tick() function

You do have to call [tick()](https://angular.io/api/core/testing/tick) to advance the (virtual) clock.

Calling [tick()](https://angular.io/api/core/testing/tick) simulates the passage of time until all pending asynchronous activities finish. In this case, it waits for the error handler's setTimeout().

The [tick()](https://angular.io/api/core/testing/tick) function accepts milliseconds and tickOptions as parameters, the millisecond (defaults to 0 if not provided) parameter represents how much the virtual clock advances. For example, if you have a setTimeout(fn, 100) in a [fakeAsync](https://angular.io/api/core/testing/fakeAsync)() test, you need to use tick(100) to trigger the fn callback. The tickOptions is an optional parameter with a property called processNewMacroTasksSynchronously (defaults to true) that represents whether to invoke new generated macro tasks when ticking.

content\_copyit('should run timeout callback with delay after call [tick](https://angular.io/api/core/testing/tick) with millis', [fakeAsync](https://angular.io/api/core/testing/fakeAsync)(() => {

let called = false;

setTimeout(() => {

called = true;

}, 100);

[tick](https://angular.io/api/core/testing/tick)(100);

expect(called).toBe(true);

}));

The [tick()](https://angular.io/api/core/testing/tick) function is one of the Angular testing utilities that you import with [TestBed](https://angular.io/api/core/testing/TestBed). It's a companion to [fakeAsync](https://angular.io/api/core/testing/fakeAsync)() and you can only call it within a [fakeAsync](https://angular.io/api/core/testing/fakeAsync)() body.

#### tickOptions

content\_copyit('should run new macro task callback with delay after call [tick](https://angular.io/api/core/testing/tick) with millis',

[fakeAsync](https://angular.io/api/core/testing/fakeAsync)(() => {

function nestedTimer(cb: () => any): void {

setTimeout(() => setTimeout(() => cb()));

}

const callback = jasmine.createSpy('callback');

nestedTimer(callback);

expect(callback).not.toHaveBeenCalled();

[tick](https://angular.io/api/core/testing/tick)(0);

// the nested timeout will also be triggered

expect(callback).toHaveBeenCalled();

}));

In this example, we have a new macro task (nested setTimeout), by default, when we [tick](https://angular.io/api/core/testing/tick), the setTimeout outside and nested will both be triggered.

content\_copyit('should not run new macro task callback with delay after call [tick](https://angular.io/api/core/testing/tick) with millis',

[fakeAsync](https://angular.io/api/core/testing/fakeAsync)(() => {

function nestedTimer(cb: () => any): void {

setTimeout(() => setTimeout(() => cb()));

}

const callback = jasmine.createSpy('callback');

nestedTimer(callback);

expect(callback).not.toHaveBeenCalled();

[tick](https://angular.io/api/core/testing/tick)(0, {processNewMacroTasksSynchronously: false});

// the nested timeout will not be triggered

expect(callback).not.toHaveBeenCalled();

[tick](https://angular.io/api/core/testing/tick)(0);

expect(callback).toHaveBeenCalled();

}));

And in some case, we don't want to trigger the new macro task when ticking, we can use [tick](https://angular.io/api/core/testing/tick)(milliseconds, {processNewMacroTasksSynchronously: false}) to not invoke new macro task.

#### Comparing dates inside fakeAsync()

[fakeAsync](https://angular.io/api/core/testing/fakeAsync)() simulates passage of time, which allows you to calculate the difference between dates inside [fakeAsync](https://angular.io/api/core/testing/fakeAsync)().

content\_copyit('should get Date diff correctly in [fakeAsync](https://angular.io/api/core/testing/fakeAsync)', [fakeAsync](https://angular.io/api/core/testing/fakeAsync)(() => {

const start = Date.now();

[tick](https://angular.io/api/core/testing/tick)(100);

const end = Date.now();

expect(end - start).toBe(100);

}));

#### jasmine.clock with fakeAsync()

Jasmine also provides a clock feature to mock dates. Angular automatically runs tests that are run after jasmine.clock().install() is called inside a [fakeAsync](https://angular.io/api/core/testing/fakeAsync)() method until jasmine.clock().uninstall() is called. [fakeAsync](https://angular.io/api/core/testing/fakeAsync)() is not needed and throws an error if nested.

By default, this feature is disabled. To enable it, set a global flag before importing zone-testing.

If you use the Angular CLI, configure this flag in src/test.ts.

content\_copy(window as any)['\_\_zone\_symbol\_\_fakeAsyncPatchLock'] = true;

import 'zone.js/testing';

content\_copydescribe('use jasmine.clock()', () => {

// need to config \_\_zone\_symbol\_\_fakeAsyncPatchLock flag

// before loading zone.js/testing

beforeEach(() => {

jasmine.clock().install();

});

afterEach(() => {

jasmine.clock().uninstall();

});

it('should auto enter [fakeAsync](https://angular.io/api/core/testing/fakeAsync)', () => {

// is in [fakeAsync](https://angular.io/api/core/testing/fakeAsync) now, don't need to call [fakeAsync](https://angular.io/api/core/testing/fakeAsync)(testFn)

let called = false;

setTimeout(() => {

called = true;

}, 100);

jasmine.clock().tick(100);

expect(called).toBe(true);

});

});

#### Using the RxJS scheduler inside fakeAsync()

You can also use RxJS scheduler in [fakeAsync](https://angular.io/api/core/testing/fakeAsync)() just like using setTimeout() or setInterval(), but you need to import zone.js/plugins/zone-patch-rxjs-fake-async to patch RxJS scheduler.

content\_copyit('should get Date diff correctly in [fakeAsync](https://angular.io/api/core/testing/fakeAsync) with rxjs scheduler', [fakeAsync](https://angular.io/api/core/testing/fakeAsync)(() => {

// need to add `import 'zone.js/plugins/zone-patch-rxjs-fake-async'

// to patch rxjs scheduler

let result = '';

of('hello').pipe(delay(1000)).subscribe(v => {

result = v;

});

expect(result).toBe('');

[tick](https://angular.io/api/core/testing/tick)(1000);

expect(result).toBe('hello');

const start = new Date().getTime();

let dateDiff = 0;

interval(1000).pipe(take(2)).subscribe(() => dateDiff = (new Date().getTime() - start));

[tick](https://angular.io/api/core/testing/tick)(1000);

expect(dateDiff).toBe(1000);

[tick](https://angular.io/api/core/testing/tick)(1000);

expect(dateDiff).toBe(2000);

}));

#### Support more macroTasks

By default, [fakeAsync](https://angular.io/api/core/testing/fakeAsync)() supports the following macro tasks.

* setTimeout
* setInterval
* requestAnimationFrame
* webkitRequestAnimationFrame
* mozRequestAnimationFrame

If you run other macro tasks such as HTMLCanvasElement.toBlob(), an "Unknown macroTask scheduled in fake async test" error will be thrown.

src/app/shared/canvas.component.spec.ts (failing)

src/app/shared/canvas.component.ts

content\_copyimport { [fakeAsync](https://angular.io/api/core/testing/fakeAsync), [TestBed](https://angular.io/api/core/testing/TestBed), [tick](https://angular.io/api/core/testing/tick) } from '@angular/core/testing';

import { CanvasComponent } from './canvas.component';

describe('CanvasComponent', () => {

beforeEach(async () => {

await [TestBed](https://angular.io/api/core/testing/TestBed)

.configureTestingModule({

declarations: [CanvasComponent],

})

.compileComponents();

});

it('should be able to generate blob data from canvas', [fakeAsync](https://angular.io/api/core/testing/fakeAsync)(() => {

const fixture = TestBed.createComponent(CanvasComponent);

const canvasComp = fixture.componentInstance;

fixture.detectChanges();

expect(canvasComp.blobSize).toBe(0);

[tick](https://angular.io/api/core/testing/tick)();

expect(canvasComp.blobSize).toBeGreaterThan(0);

}));

});

If you want to support such a case, you need to define the macro task you want to support in beforeEach(). For example:

src/app/shared/canvas.component.spec.ts (excerpt)

content\_copybeforeEach(() => {

(window as any).\_\_zone\_symbol\_\_FakeAsyncTestMacroTask = [

{

source: 'HTMLCanvasElement.toBlob',

callbackArgs: [{size: 200}],

},

];

});

Note that in order to make the <canvas> element Zone.js-aware in your app, you need to import the zone-patch-canvas patch (either in polyfills.ts or in the specific file that uses <canvas>):

src/polyfills.ts or src/app/shared/canvas.component.ts

content\_copy// Import patch to make async `HTMLCanvasElement` methods (such as `.toBlob()`) Zone.js-aware.

// Either import in `polyfills.ts` (if used in more than one places in the app) or in the component

// file using `HTMLCanvasElement` (if it is only used in a single file).

import 'zone.js/plugins/zone-patch-canvas';

#### Async observables

You might be satisfied with the test coverage of these tests.

However, you might be troubled by the fact that the real service doesn't quite behave this way. The real service sends requests to a remote server. A server takes time to respond and the response certainly won't be available immediately as in the previous two tests.

Your tests will reflect the real world more faithfully if you return an asynchronous observable from the getQuote() spy like this.

content\_copy// Simulate delayed observable values with the `asyncData()` helper

getQuoteSpy.and.returnValue(asyncData(testQuote));

#### Async observable helpers

The async observable was produced by an asyncData helper. The asyncData helper is a utility function that you'll have to write yourself, or you can copy this one from the sample code.

testing/async-observable-helpers.ts

content\_copy/\*\*

\* Create async observable that emits-once and completes

\* after a JS engine turn

\*/

export function asyncData<T>(data: T) {

return defer(() => Promise.resolve(data));

}

This helper's observable emits the data value in the next turn of the JavaScript engine.

The [RxJSdefer()operator](http://reactivex.io/documentation/operators/defer.html) returns an observable. It takes a factory function that returns either a promise or an observable. When something subscribes to defer's observable, it adds the subscriber to a new observable created with that factory.

The defer() operator transforms the Promise.resolve() into a new observable that, like [HttpClient](https://angular.io/api/common/http/HttpClient), emits once and completes. Subscribers are unsubscribed after they receive the data value.

There's a similar helper for producing an async error.

content\_copy/\*\*

\* Create async observable error that errors

\* after a JS engine turn

\*/

export function asyncError<T>(errorObject: any) {

return defer(() => Promise.reject(errorObject));

}

#### More async tests

Now that the getQuote() spy is returning async observables, most of your tests will have to be async as well.

Here's a [fakeAsync](https://angular.io/api/core/testing/fakeAsync)() test that demonstrates the data flow you'd expect in the real world.

content\_copyit('should show quote after getQuote ([fakeAsync](https://angular.io/api/core/testing/fakeAsync))', [fakeAsync](https://angular.io/api/core/testing/fakeAsync)(() => {

fixture.detectChanges(); // ngOnInit()

expect(quoteEl.textContent).toBe('...', 'should show placeholder');

[tick](https://angular.io/api/core/testing/tick)(); // [flush](https://angular.io/api/core/testing/flush) the observable to get the quote

fixture.detectChanges(); // update view

expect(quoteEl.textContent).toBe(testQuote, 'should show quote');

expect(errorMessage()).toBeNull('should not show error');

}));

Notice that the quote element displays the placeholder value ('...') after ngOnInit(). The first quote hasn't arrived yet.

To flush the first quote from the observable, you call [tick()](https://angular.io/api/core/testing/tick). Then call detectChanges() to tell Angular to update the screen.

Then you can assert that the quote element displays the expected text.

#### Async test with waitForAsync()

To use [waitForAsync](https://angular.io/api/core/testing/waitForAsync)() functionality, you must import zone.js/testing in your test setup file. If you created your project with the Angular CLI, zone-testing is already imported in src/test.ts.

Here's the previous [fakeAsync](https://angular.io/api/core/testing/fakeAsync)() test, re-written with the [waitForAsync](https://angular.io/api/core/testing/waitForAsync)() utility.

content\_copyit('should show quote after getQuote ([waitForAsync](https://angular.io/api/core/testing/waitForAsync))', [waitForAsync](https://angular.io/api/core/testing/waitForAsync)(() => {

fixture.detectChanges(); // ngOnInit()

expect(quoteEl.textContent).toBe('...', 'should show placeholder');

fixture.whenStable().then(() => { // wait for async getQuote

fixture.detectChanges(); // update view with quote

expect(quoteEl.textContent).toBe(testQuote);

expect(errorMessage()).toBeNull('should not show error');

});

}));

The [waitForAsync](https://angular.io/api/core/testing/waitForAsync)() utility hides some asynchronous boilerplate by arranging for the tester's code to run in a special async test zone. You don't need to pass Jasmine's done() into the test and call done() because it is undefined in promise or observable callbacks.

But the test's asynchronous nature is revealed by the call to fixture.whenStable(), which breaks the linear flow of control.

When using an intervalTimer() such as setInterval() in [waitForAsync](https://angular.io/api/core/testing/waitForAsync)(), remember to cancel the timer with clearInterval() after the test, otherwise the [waitForAsync](https://angular.io/api/core/testing/waitForAsync)() never ends.

#### whenStable

The test must wait for the getQuote() observable to emit the next quote. Instead of calling [tick()](https://angular.io/api/core/testing/tick), it calls fixture.whenStable().

The fixture.whenStable() returns a promise that resolves when the JavaScript engine's task queue becomes empty. In this example, the task queue becomes empty when the observable emits the first quote.

The test resumes within the promise callback, which calls detectChanges() to update the quote element with the expected text.

#### Jasmine done()

While the [waitForAsync](https://angular.io/api/core/testing/waitForAsync)() and [fakeAsync](https://angular.io/api/core/testing/fakeAsync)() functions greatly simplify Angular asynchronous testing, you can still fall back to the traditional technique and pass it a function that takes a [donecallback](https://jasmine.github.io/2.0/introduction.html#section-Asynchronous_Support).

You can't call done() in [waitForAsync](https://angular.io/api/core/testing/waitForAsync)() or [fakeAsync](https://angular.io/api/core/testing/fakeAsync)() functions, because the done parameter is undefined.

Now you are responsible for chaining promises, handling errors, and calling done() at the appropriate moments.

Writing test functions with done(), is more cumbersome than [waitForAsync](https://angular.io/api/core/testing/waitForAsync)()and [fakeAsync](https://angular.io/api/core/testing/fakeAsync)(), but it is occasionally necessary when code involves the intervalTimer() like setInterval.

Here are two more versions of the previous test, written with done(). The first one subscribes to the Observable exposed to the template by the component's quote property.

content\_copyit('should show last quote (quote done)', (done: DoneFn) => {

fixture.detectChanges();

component.quote.pipe(last()).subscribe(() => {

fixture.detectChanges(); // update view with quote

expect(quoteEl.textContent).toBe(testQuote);

expect(errorMessage()).toBeNull('should not show error');

done();

});

});

The RxJS last() operator emits the observable's last value before completing, which will be the test quote. The subscribe callback calls detectChanges() to update the quote element with the test quote, in the same manner as the earlier tests.

In some tests, you're more interested in how an injected service method was called and what values it returned, than what appears on screen.

A service spy, such as the qetQuote() spy of the fake TwainService, can give you that information and make assertions about the state of the view.

content\_copyit('should show quote after getQuote (spy done)', (done: DoneFn) => {

fixture.detectChanges();

// the spy's most recent call returns the observable with the test quote

getQuoteSpy.calls.mostRecent().returnValue.subscribe(() => {

fixture.detectChanges(); // update view with quote

expect(quoteEl.textContent).toBe(testQuote);

expect(errorMessage()).toBeNull('should not show error');

done();

});

});

## **Component marble tests**

The previous TwainComponent tests simulated an asynchronous observable response from the TwainService with the asyncData and asyncError utilities.

These are short, simple functions that you can write yourself. Unfortunately, they're too simple for many common scenarios. An observable often emits multiple times, perhaps after a significant delay. A component may coordinate multiple observables with overlapping sequences of values and errors.

**RxJS marble testing** is a great way to test observable scenarios, both simple and complex. You've likely seen the [marble diagrams](https://rxmarbles.com/) that illustrate how observables work. Marble testing uses a similar marble language to specify the observable streams and expectations in your tests.

The following examples revisit two of the TwainComponent tests with marble testing.

Start by installing the jasmine-marbles npm package. Then import the symbols you need.

app/twain/twain.component.marbles.spec.ts (import marbles)

content\_copyimport { cold, getTestScheduler } from 'jasmine-marbles';

Here's the complete test for getting a quote:

content\_copyit('should show quote after getQuote (marbles)', () => {

// observable test quote value and complete(), after delay

const q$ = cold('---x|', { x: testQuote });

getQuoteSpy.and.returnValue( q$ );

fixture.detectChanges(); // ngOnInit()

expect(quoteEl.textContent).toBe('...', 'should show placeholder');

getTestScheduler().flush(); // [flush](https://angular.io/api/core/testing/flush) the observables

fixture.detectChanges(); // update view

expect(quoteEl.textContent).toBe(testQuote, 'should show quote');

expect(errorMessage()).toBeNull('should not show error');

});

Notice that the Jasmine test is synchronous. There's no [fakeAsync](https://angular.io/api/core/testing/fakeAsync)(). Marble testing uses a test scheduler to simulate the passage of time in a synchronous test.

The beauty of marble testing is in the visual definition of the observable streams. This test defines a [cold observable](https://angular.io/guide/testing-components-scenarios#cold-observable) that waits three [frames](https://angular.io/guide/testing-components-scenarios#marble-frame) (---), emits a value (x), and completes (|). In the second argument you map the value marker (x) to the emitted value (testQuote).

content\_copyconst q$ = cold('---x|', { x: testQuote });

The marble library constructs the corresponding observable, which the test sets as the getQuote spy's return value.

When you're ready to activate the marble observables, you tell the TestScheduler to flush its queue of prepared tasks like this.

content\_copygetTestScheduler().flush(); // [flush](https://angular.io/api/core/testing/flush) the observables

This step serves a purpose analogous to [tick()](https://angular.io/api/core/testing/tick) and whenStable() in the earlier [fakeAsync](https://angular.io/api/core/testing/fakeAsync)() and [waitForAsync](https://angular.io/api/core/testing/waitForAsync)() examples. The balance of the test is the same as those examples.

#### Marble error testing

Here's the marble testing version of the getQuote() error test.

content\_copyit('should display error when TwainService fails', [fakeAsync](https://angular.io/api/core/testing/fakeAsync)(() => {

// observable error after delay

const q$ = cold('---#|', null, new Error('TwainService test failure'));

getQuoteSpy.and.returnValue( q$ );

fixture.detectChanges(); // ngOnInit()

expect(quoteEl.textContent).toBe('...', 'should show placeholder');

getTestScheduler().flush(); // [flush](https://angular.io/api/core/testing/flush) the observables

[tick](https://angular.io/api/core/testing/tick)(); // component shows error after a setTimeout()

fixture.detectChanges(); // update error message

expect(errorMessage()).toMatch(/test failure/, 'should display error');

expect(quoteEl.textContent).toBe('...', 'should show placeholder');

}));

It's still an async test, calling [fakeAsync](https://angular.io/api/core/testing/fakeAsync)() and [tick()](https://angular.io/api/core/testing/tick), because the component itself calls setTimeout() when processing errors.

Look at the marble observable definition.

content\_copyconst q$ = cold('---#|', null, new Error('TwainService test failure'));

This is a cold observable that waits three frames and then emits an error, The hash (#) indicates the timing of the error that is specified in the third argument. The second argument is null because the observable never emits a value.

#### Learn about marble testing

A marble frame is a virtual unit of testing time. Each symbol (-, x, |, #) marks the passing of one frame.

A cold observable doesn't produce values until you subscribe to it. Most of your application observables are cold. All [HttpClient](https://angular.io/guide/http) methods return cold observables.

A hot observable is already producing values before you subscribe to it. The [Router.events](https://angular.io/api/router/Router#events) observable, which reports router activity, is a hot observable.

RxJS marble testing is a rich subject, beyond the scope of this guide. Learn about it on the web, starting with the [official documentation](https://rxjs.dev/guide/testing/marble-testing).

## **Component with inputs and outputs**

A component with inputs and outputs typically appears inside the view template of a host component. The host uses a property binding to set the input property and an event binding to listen to events raised by the output property.

The testing goal is to verify that such bindings work as expected. The tests should set input values and listen for output events.

The DashboardHeroComponent is a tiny example of a component in this role. It displays an individual hero provided by the DashboardComponent. Clicking that hero tells the DashboardComponent that the user has selected the hero.

The DashboardHeroComponent is embedded in the DashboardComponent template like this:

app/dashboard/dashboard.component.html (excerpt)

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

[hero]=hero (selected)="gotoDetail($event)" >

</dashboard-hero>

The DashboardHeroComponent appears in an \*[ngFor](https://angular.io/api/common/NgForOf) repeater, which sets each component's hero input property to the looping value and listens for the component's selected event.

Here's the component's full definition:

app/dashboard/dashboard-hero.component.ts (component)

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

selector: 'dashboard-hero',

template: `

<div (click)="click()" class="hero">

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

</div>`,

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

})

export class DashboardHeroComponent {

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

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

click() { this.selected.emit(this.hero); }

}

While testing a component this simple has little intrinsic value, it's worth knowing how. You can use one of these approaches:

* Test it as used by DashboardComponent.
* Test it as a stand-alone component.
* Test it as used by a substitute for DashboardComponent.

A quick look at the DashboardComponent constructor discourages the first approach:

app/dashboard/dashboard.component.ts (constructor)

content\_copyconstructor(

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

private heroService: HeroService) {

}

The DashboardComponent depends on the Angular router and the HeroService. You'd probably have to replace them both with test doubles, which is a lot of work. The router seems particularly challenging.

The [discussion below](https://angular.io/guide/testing-components-scenarios#routing-component) covers testing components that require the router.

The immediate goal is to test the DashboardHeroComponent, not the DashboardComponent, so, try the second and third options.

#### Test DashboardHeroComponent stand-alone

Here's the meat of the spec file setup.

app/dashboard/dashboard-hero.component.spec.ts (setup)

content\_copy[TestBed](https://angular.io/api/core/testing/TestBed)

.configureTestingModule({declarations: [DashboardHeroComponent]})

fixture = TestBed.createComponent(DashboardHeroComponent);

comp = fixture.componentInstance;

// find the hero's [DebugElement](https://angular.io/api/core/DebugElement) and element

heroDe = fixture.debugElement.query(By.css('.hero'));

heroEl = heroDe.nativeElement;

// mock the hero supplied by the parent component

expectedHero = {id: 42, name: 'Test Name'};

// simulate the parent setting the input property with that hero

comp.hero = expectedHero;

// [trigger](https://angular.io/api/animations/trigger) initial data binding

fixture.detectChanges();

Note how the setup code assigns a test hero (expectedHero) to the component's hero property, emulating the way the DashboardComponent would set it using the property binding in its repeater.

The following test verifies that the hero name is propagated to the template using a binding.

content\_copyit('should display hero name in uppercase', () => {

const expectedPipedName = expectedHero.name.toUpperCase();

expect(heroEl.textContent).toContain(expectedPipedName);

});

Because the [template](https://angular.io/guide/testing-components-scenarios#dashboard-hero-component) passes the hero name through the Angular [UpperCasePipe](https://angular.io/api/common/UpperCasePipe), the test must match the element value with the upper-cased name.

This small test demonstrates how Angular tests can verify a component's visual representation—something not possible with [component class tests](https://angular.io/guide/testing-components-basics#component-class-testing)—at low cost and without resorting to much slower and more complicated end-to-end tests.

#### Clicking

Clicking the hero should raise a selected event that the host component (DashboardComponent presumably) can hear:

content\_copyit('should raise selected event when clicked (triggerEventHandler)', () => {

let selectedHero: Hero | undefined;

comp.selected.pipe(first()).subscribe((hero: Hero) => selectedHero = hero);

heroDe.triggerEventHandler('click', null);

expect(selectedHero).toBe(expectedHero);

});

The component's selected property returns an [EventEmitter](https://angular.io/api/core/EventEmitter), which looks like an RxJS synchronous Observable to consumers. The test subscribes to it explicitly just as the host component does implicitly.

If the component behaves as expected, clicking the hero's element should tell the component's selected property to emit the hero object.

The test detects that event through its subscription to selected.

#### triggerEventHandler

The heroDe in the previous test is a [DebugElement](https://angular.io/api/core/DebugElement) that represents the hero <div>.

It has Angular properties and methods that abstract interaction with the native element. This test calls the DebugElement.triggerEventHandler with the "click" event name. The "click" event binding responds by calling DashboardHeroComponent.click().

The Angular DebugElement.triggerEventHandler can raise any data-bound event by its event name. The second parameter is the event object passed to the handler.

The test triggered a "click" event with a null event object.

content\_copyheroDe.triggerEventHandler('click', null);

The test assumes (correctly in this case) that the runtime event handler—the component's click() method—doesn't care about the event object.

Other handlers are less forgiving. For example, the [RouterLink](https://angular.io/api/router/RouterLink) directive expects an object with a button property that identifies which mouse button (if any) was pressed during the click. The [RouterLink](https://angular.io/api/router/RouterLink) directive throws an error if the event object is missing.

#### Click the element

The following test alternative calls the native element's own click() method, which is perfectly fine for this component.

content\_copyit('should raise selected event when clicked (element.click)', () => {

let selectedHero: Hero | undefined;

comp.selected.pipe(first()).subscribe((hero: Hero) => selectedHero = hero);

heroEl.click();

expect(selectedHero).toBe(expectedHero);

});

#### click() helper

Clicking a button, an anchor, or an arbitrary HTML element is a common test task.

Make that consistent and easy by encapsulating the click-triggering process in a helper such as the click() function below:

testing/index.ts (click helper)

content\_copy/\*\* Button events to pass to `DebugElement.triggerEventHandler` for [RouterLink](https://angular.io/api/router/RouterLink) event handler \*/

export const ButtonClickEvents = {

left: { button: 0 },

right: { button: 2 }

};

/\*\* Simulate element click. Defaults to mouse left-button click event. \*/

export function click(el: [DebugElement](https://angular.io/api/core/DebugElement) | HTMLElement, eventObj: any = ButtonClickEvents.left): void {

if (el instanceof HTMLElement) {

el.click();

} else {

el.triggerEventHandler('click', eventObj);

}

}

The first parameter is the element-to-click. If you want, you can pass a custom event object as the second parameter. The default is a (partial) [left-button mouse event object](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/button) accepted by many handlers including the [RouterLink](https://angular.io/api/router/RouterLink) directive.

The click() helper function is **not** one of the Angular testing utilities. It's a function defined in this guide's sample code. All of the sample tests use it. If you like it, add it to your own collection of helpers.

Here's the previous test, rewritten using the click helper.

app/dashboard/dashboard-hero.component.spec.ts (test with click helper)

content\_copyit('should raise selected event when clicked (click helper with [DebugElement](https://angular.io/api/core/DebugElement))', () => {

let selectedHero: Hero | undefined;

comp.selected.pipe(first()).subscribe((hero: Hero) => selectedHero = hero);

click(heroDe); // click helper with [DebugElement](https://angular.io/api/core/DebugElement)

expect(selectedHero).toBe(expectedHero);

});

## **Component inside a test host**

The previous tests played the role of the host DashboardComponent themselves. But does the DashboardHeroComponent work correctly when properly data-bound to a host component?

You could test with the actual DashboardComponent. But doing so could require a lot of setup, especially when its template features an \*[ngFor](https://angular.io/api/common/NgForOf) repeater, other components, layout HTML, additional bindings, a constructor that injects multiple services, and it starts interacting with those services right away.

Imagine the effort to disable these distractions, just to prove a point that can be made satisfactorily with a test host like this one:

app/dashboard/dashboard-hero.component.spec.ts (test host)

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

template: `

<dashboard-hero

[hero]="hero" (selected)="onSelected($event)">

</dashboard-hero>`

})

class TestHostComponent {

hero: Hero = {id: 42, name: 'Test Name'};

selectedHero: Hero | undefined;

onSelected(hero: Hero) {

this.selectedHero = hero;

}

}

This test host binds to DashboardHeroComponent as the DashboardComponent would but without the noise of the [Router](https://angular.io/api/router/Router), the HeroService, or the \*[ngFor](https://angular.io/api/common/NgForOf) repeater.

The test host sets the component's hero input property with its test hero. It binds the component's selected event with its onSelected handler, which records the emitted hero in its selectedHero property.

Later, the tests will be able to easily check selectedHero to verify that the DashboardHeroComponent.selected event emitted the expected hero.

The setup for the test-host tests is similar to the setup for the stand-alone tests:

app/dashboard/dashboard-hero.component.spec.ts (test host setup)

content\_copy[TestBed](https://angular.io/api/core/testing/TestBed)

.configureTestingModule({declarations: [DashboardHeroComponent, TestHostComponent]})

// create TestHostComponent instead of DashboardHeroComponent

fixture = TestBed.createComponent(TestHostComponent);

testHost = fixture.componentInstance;

heroEl = fixture.nativeElement.querySelector('.hero');

fixture.detectChanges(); // [trigger](https://angular.io/api/animations/trigger) initial data binding

This testing module configuration shows three important differences:

1. It declares both the DashboardHeroComponent and the TestHostComponent.
2. It creates the TestHostComponent instead of the DashboardHeroComponent.
3. The TestHostComponent sets the DashboardHeroComponent.hero with a binding.

The createComponent returns a fixture that holds an instance of TestHostComponent instead of an instance of DashboardHeroComponent.

Creating the TestHostComponent has the side-effect of creating a DashboardHeroComponent because the latter appears within the template of the former. The query for the hero element (heroEl) still finds it in the test DOM, albeit at greater depth in the element tree than before.

The tests themselves are almost identical to the stand-alone version:

app/dashboard/dashboard-hero.component.spec.ts (test-host)

content\_copyit('should display hero name', () => {

const expectedPipedName = testHost.hero.name.toUpperCase();

expect(heroEl.textContent).toContain(expectedPipedName);

});

it('should raise selected event when clicked', () => {

click(heroEl);

// selected hero should be the same data bound hero

expect(testHost.selectedHero).toBe(testHost.hero);

});

Only the selected event test differs. It confirms that the selected DashboardHeroComponent hero really does find its way up through the event binding to the host component.

## **Routing component**

A routing component is a component that tells the [Router](https://angular.io/api/router/Router) to navigate to another component. The DashboardComponent is a routing component because the user can navigate to the HeroDetailComponent by clicking on one of the hero buttons on the dashboard.

Routing is pretty complicated. Testing the DashboardComponent seemed daunting in part because it involves the [Router](https://angular.io/api/router/Router), which it injects together with the HeroService.

app/dashboard/dashboard.component.ts (constructor)

content\_copyconstructor(

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

private heroService: HeroService) {

}

Mocking the HeroService with a spy is a [familiar story](https://angular.io/guide/testing-components-scenarios#component-with-async-service). But the [Router](https://angular.io/api/router/Router) has a complicated API and is entwined with other services and application preconditions. Might it be difficult to mock?

Fortunately, not in this case because the DashboardComponent isn't doing much with the [Router](https://angular.io/api/router/Router)

app/dashboard/dashboard.component.ts (goToDetail)

content\_copygotoDetail(hero: Hero) {

const url = `/heroes/${hero.id}`;

this.router.navigateByUrl(url);

}

This is often the case with routing components. As a rule you test the component, not the router, and care only if the component navigates with the right address under the given conditions.

Providing a router spy for this component test suite happens to be as easy as providing a HeroService spy.

app/dashboard/dashboard.component.spec.ts (spies)

content\_copyconst routerSpy = jasmine.createSpyObj('[Router](https://angular.io/api/router/Router)', ['navigateByUrl']);

const heroServiceSpy = jasmine.createSpyObj('HeroService', ['getHeroes']);

[TestBed](https://angular.io/api/core/testing/TestBed)

.configureTestingModule({

providers: [

{provide: HeroService, useValue: heroServiceSpy}, {provide: [Router](https://angular.io/api/router/Router), useValue: routerSpy}

]

})

The following test clicks the displayed hero and confirms that Router.navigateByUrl is called with the expected url.

app/dashboard/dashboard.component.spec.ts (navigate test)

content\_copyit('should tell ROUTER to navigate when hero clicked', () => {

heroClick(); // [trigger](https://angular.io/api/animations/trigger) click on first inner <div class="hero">

// args passed to router.navigateByUrl() spy

const spy = router.navigateByUrl as jasmine.Spy;

const navArgs = spy.calls.first().args[0];

// expecting to navigate to id of the component's first hero

const id = comp.heroes[0].id;

expect(navArgs).toBe('/heroes/' + id, 'should nav to HeroDetail for first hero');

});

## **Routed components**

A routed component is the destination of a [Router](https://angular.io/api/router/Router) navigation. It can be trickier to test, especially when the route to the component includes parameters. The HeroDetailComponent is a routed component that is the destination of such a route.

When a user clicks a Dashboard hero, the DashboardComponent tells the [Router](https://angular.io/api/router/Router) to navigate to heroes/:id. The :id is a route parameter whose value is the id of the hero to edit.

The [Router](https://angular.io/api/router/Router) matches that URL to a route to the HeroDetailComponent. It creates an [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) object with the routing information and injects it into a new instance of the HeroDetailComponent.

Here's the HeroDetailComponent constructor:

app/hero/hero-detail.component.ts (constructor)

content\_copyconstructor(

private heroDetailService: HeroDetailService,

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

private router: [Router](https://angular.io/api/router/Router)) {

}

The HeroDetail component needs the id parameter so it can fetch the corresponding hero using the HeroDetailService. The component has to get the id from the [ActivatedRoute.paramMap](https://angular.io/api/router/ActivatedRoute#paramMap) property which is an Observable.

It can't just reference the id property of the [ActivatedRoute.paramMap](https://angular.io/api/router/ActivatedRoute#paramMap). The component has to subscribe to the [ActivatedRoute.paramMap](https://angular.io/api/router/ActivatedRoute#paramMap) observable and be prepared for the id to change during its lifetime.

app/hero/hero-detail.component.ts (ngOnInit)

content\_copyngOnInit(): void {

// get hero when `id` param changes

this.route.paramMap.subscribe(pmap => this.getHero(pmap.get('id')));

}

The [ActivatedRoute in action](https://angular.io/guide/router-tutorial-toh#activated-route-in-action) section of the [Router tutorial: tour of heroes](https://angular.io/guide/router-tutorial-toh) guide covers [ActivatedRoute.paramMap](https://angular.io/api/router/ActivatedRoute#paramMap) in more detail.

Tests can explore how the HeroDetailComponent responds to different id parameter values by manipulating the [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) injected into the component's constructor.

You know how to spy on the [Router](https://angular.io/api/router/Router) and a data service.

You'll take a different approach with [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) because

* paramMap returns an Observable that can emit more than one value during a test.
* You need the router helper function, [convertToParamMap](https://angular.io/api/router/convertToParamMap)(), to create a [ParamMap](https://angular.io/api/router/ParamMap).
* Other routed component tests need a test double for [ActivatedRoute](https://angular.io/api/router/ActivatedRoute).

These differences argue for a re-usable stub class.

#### ActivatedRouteStub

The following ActivatedRouteStub class serves as a test double for [ActivatedRoute](https://angular.io/api/router/ActivatedRoute).

testing/activated-route-stub.ts (ActivatedRouteStub)

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

import { ReplaySubject } from 'rxjs';

/\*\*

\* An ActivateRoute test double with a `paramMap` observable.

\* Use the `setParamMap()` method to add the next `paramMap` value.

\*/

export class ActivatedRouteStub {

// Use a ReplaySubject to share previous values with subscribers

// and pump new values into the `paramMap` observable

private subject = new ReplaySubject<[ParamMap](https://angular.io/api/router/ParamMap)>();

constructor(initialParams?: [Params](https://angular.io/api/router/Params)) {

this.setParamMap(initialParams);

}

/\*\* The mock paramMap observable \*/

readonly paramMap = this.subject.asObservable();

/\*\* Set the paramMap observable's next value \*/

setParamMap(params: [Params](https://angular.io/api/router/Params) = {}) {

this.subject.next([convertToParamMap](https://angular.io/api/router/convertToParamMap)(params));

}

}

Consider placing such helpers in a testing folder sibling to the app folder. This sample puts ActivatedRouteStub in testing/activated-route-stub.ts.

Consider writing a more capable version of this stub class with the [marble testing library](https://angular.io/guide/testing-components-scenarios#marble-testing).

#### Testing with ActivatedRouteStub

Here's a test demonstrating the component's behavior when the observed id refers to an existing hero:

app/hero/hero-detail.component.spec.ts (existing id)

content\_copydescribe('when navigate to existing hero', () => {

let expectedHero: Hero;

beforeEach(async () => {

expectedHero = firstHero;

activatedRoute.setParamMap({id: expectedHero.id});

await createComponent();

});

it('should display that hero\'s name', () => {

expect(page.nameDisplay.textContent).toBe(expectedHero.name);

});

});

The createComponent() method and page object are discussed [below](https://angular.io/guide/testing-components-scenarios#page-object). Rely on your intuition for now.

When the id cannot be found, the component should re-route to the HeroListComponent.

The test suite setup provided the same router spy [described above](https://angular.io/guide/testing-components-scenarios#routing-component) which spies on the router without actually navigating.

This test expects the component to try to navigate to the HeroListComponent.

app/hero/hero-detail.component.spec.ts (bad id)

content\_copydescribe('when navigate to non-existent hero id', () => {

beforeEach(async () => {

activatedRoute.setParamMap({id: 99999});

await createComponent();

});

it('should try to navigate back to hero list', () => {

expect(page.gotoListSpy.calls.any()).toBe(true, 'comp.gotoList called');

expect(page.navigateSpy.calls.any()).toBe(true, 'router.navigate called');

});

});

While this application doesn't have a route to the HeroDetailComponent that omits the id parameter, it might add such a route someday. The component should do something reasonable when there is no id.

In this implementation, the component should create and display a new hero. New heroes have id=0 and a blank name. This test confirms that the component behaves as expected:

app/hero/hero-detail.component.spec.ts (no id)

content\_copydescribe('when navigate with no hero id', () => {

beforeEach(async () => {

await createComponent();

});

it('should have hero.id === 0', () => {

expect(component.hero.id).toBe(0);

});

it('should display empty hero name', () => {

expect(page.nameDisplay.textContent).toBe('');

});

});

## **Nested component tests**

Component templates often have nested components, whose templates may contain more components.

The component tree can be very deep and, most of the time, the nested components play no role in testing the component at the top of the tree.

The AppComponent, for example, displays a navigation bar with anchors and their [RouterLink](https://angular.io/api/router/RouterLink) directives.

app/app.component.html

content\_copy<app-banner></app-banner>

<app-welcome></app-welcome>

<nav>

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

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

<a [routerLink](https://angular.io/api/router/RouterLink)="/about">About</a>

</nav>

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

While the AppComponent class is empty, you may want to write unit tests to confirm that the links are wired properly to the [RouterLink](https://angular.io/api/router/RouterLink) directives, perhaps for the reasons [explained below](https://angular.io/guide/testing-components-scenarios#why-stubbed-routerlink-tests).

To validate the links, you don't need the [Router](https://angular.io/api/router/Router) to navigate and you don't need the <[router-outlet](https://angular.io/api/router/RouterOutlet)> to mark where the [Router](https://angular.io/api/router/Router) inserts routed components.

The BannerComponent and WelcomeComponent (indicated by <app-banner> and <app-welcome>) are also irrelevant.

Yet any test that creates the AppComponent in the DOM will also create instances of these three components and, if you let that happen, you'll have to configure the [TestBed](https://angular.io/api/core/testing/TestBed) to create them.

If you neglect to declare them, the Angular compiler won't recognize the <app-banner>, <app-welcome>, and <[router-outlet](https://angular.io/api/router/RouterOutlet)> tags in the AppComponent template and will throw an error.

If you declare the real components, you'll also have to declare their nested components and provide for all services injected in any component in the tree.

That's too much effort just to answer a few simple questions about links.

This section describes two techniques for minimizing the setup. Use them, alone or in combination, to stay focused on testing the primary component.

##### Stubbing unneeded components

In the first technique, you create and declare stub versions of the components and directive that play little or no role in the tests.

app/app.component.spec.ts (stub declaration)

content\_copy@[Component](https://angular.io/api/core/Component)({selector: 'app-banner', template: ''})

class BannerStubComponent {

}

@[Component](https://angular.io/api/core/Component)({selector: '[router-outlet](https://angular.io/api/router/RouterOutlet)', template: ''})

class RouterOutletStubComponent {

}

@[Component](https://angular.io/api/core/Component)({selector: 'app-welcome', template: ''})

class WelcomeStubComponent {

}

The stub selectors match the selectors for the corresponding real components. But their templates and classes are empty.

Then declare them in the [TestBed](https://angular.io/api/core/testing/TestBed) configuration next to the components, directives, and pipes that need to be real.

app/app.component.spec.ts (TestBed stubs)

content\_copy[TestBed](https://angular.io/api/core/testing/TestBed)

.configureTestingModule({

declarations: [

AppComponent, RouterLinkDirectiveStub, BannerStubComponent, RouterOutletStubComponent,

WelcomeStubComponent

]

})

The AppComponent is the test subject, so of course you declare the real version.

The RouterLinkDirectiveStub, [described later](https://angular.io/guide/testing-components-scenarios#routerlink), is a test version of the real [RouterLink](https://angular.io/api/router/RouterLink) that helps with the link tests.

The rest are stubs.

#### NO\_ERRORS\_SCHEMA

In the second approach, add [NO\_ERRORS\_SCHEMA](https://angular.io/api/core/NO_ERRORS_SCHEMA) to the TestBed.schemas metadata.

app/app.component.spec.ts (NO\_ERRORS\_SCHEMA)

content\_copy[TestBed](https://angular.io/api/core/testing/TestBed)

.configureTestingModule({

declarations: [

AppComponent,

RouterLinkDirectiveStub

],

schemas: [[NO\_ERRORS\_SCHEMA](https://angular.io/api/core/NO_ERRORS_SCHEMA)]

})

The [NO\_ERRORS\_SCHEMA](https://angular.io/api/core/NO_ERRORS_SCHEMA) tells the Angular compiler to ignore unrecognized elements and attributes.

The compiler will recognize the <app-root> element and the [routerLink](https://angular.io/api/router/RouterLink) attribute because you declared a corresponding AppComponent and RouterLinkDirectiveStub in the [TestBed](https://angular.io/api/core/testing/TestBed) configuration.

But the compiler won't throw an error when it encounters <app-banner>, <app-welcome>, or <[router-outlet](https://angular.io/api/router/RouterOutlet)>. It simply renders them as empty tags and the browser ignores them.

You no longer need the stub components.

#### Use both techniques together

These are techniques for Shallow Component Testing , so-named because they reduce the visual surface of the component to just those elements in the component's template that matter for tests.

The [NO\_ERRORS\_SCHEMA](https://angular.io/api/core/NO_ERRORS_SCHEMA) approach is the easier of the two but don't overuse it.

The [NO\_ERRORS\_SCHEMA](https://angular.io/api/core/NO_ERRORS_SCHEMA) also prevents the compiler from telling you about the missing components and attributes that you omitted inadvertently or misspelled. You could waste hours chasing phantom bugs that the compiler would have caught in an instant.

The stub component approach has another advantage. While the stubs in this example were empty, you could give them stripped-down templates and classes if your tests need to interact with them in some way.

In practice you will combine the two techniques in the same setup, as seen in this example.

app/app.component.spec.ts (mixed setup)

content\_copy[TestBed](https://angular.io/api/core/testing/TestBed)

.configureTestingModule({

declarations: [

AppComponent,

BannerStubComponent,

RouterLinkDirectiveStub

],

schemas: [[NO\_ERRORS\_SCHEMA](https://angular.io/api/core/NO_ERRORS_SCHEMA)]

})

The Angular compiler creates the BannerComponentStub for the <app-banner> element and applies the RouterLinkStubDirective to the anchors with the [routerLink](https://angular.io/api/router/RouterLink) attribute, but it ignores the <app-welcome> and <[router-outlet](https://angular.io/api/router/RouterOutlet)> tags.

## **Components with RouterLink**

The real RouterLinkDirective is quite complicated and entangled with other components and directives of the [RouterModule](https://angular.io/api/router/RouterModule). It requires challenging setup to mock and use in tests.

The RouterLinkDirectiveStub in this sample code replaces the real directive with an alternative version designed to validate the kind of anchor tag wiring seen in the AppComponent template.

testing/router-link-directive-stub.ts (RouterLinkDirectiveStub)

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

selector: '[[routerLink](https://angular.io/api/router/RouterLink)]'

})

export class RouterLinkDirectiveStub {

@[Input](https://angular.io/api/core/Input)('[routerLink](https://angular.io/api/router/RouterLink)') linkParams: any;

navigatedTo: any = null;

@[HostListener](https://angular.io/api/core/HostListener)('click')

onClick() {

this.navigatedTo = this.linkParams;

}

}

The URL bound to the [[routerLink](https://angular.io/api/router/RouterLink)] attribute flows in to the directive's linkParams property.

The [HostListener](https://angular.io/api/core/HostListener) wires the click event of the host element (the <a> anchor elements in AppComponent) to the stub directive's onClick method.

Clicking the anchor should trigger the onClick() method, which sets the stub's telltale navigatedTo property. Tests inspect navigatedTo to confirm that clicking the anchor sets the expected route definition.

Whether the router is configured properly to navigate with that route definition is a question for a separate set of tests.

#### By.directive and injected directives

A little more setup triggers the initial data binding and gets references to the navigation links:

app/app.component.spec.ts (test setup)

content\_copybeforeEach(() => {

fixture.detectChanges(); // [trigger](https://angular.io/api/animations/trigger) initial data binding

// find DebugElements with an attached RouterLinkStubDirective

linkDes = fixture.debugElement.queryAll(By.directive(RouterLinkDirectiveStub));

// get attached link directive instances

// using each [DebugElement](https://angular.io/api/core/DebugElement)'s injector

routerLinks = linkDes.map(de => de.injector.get(RouterLinkDirectiveStub));

});

Three points of special interest:

1. You can locate the anchor elements with an attached directive using By.directive.
2. The query returns [DebugElement](https://angular.io/api/core/DebugElement) wrappers around the matching elements.
3. Each [DebugElement](https://angular.io/api/core/DebugElement) exposes a dependency injector with the specific instance of the directive attached to that element.

The AppComponent links to validate are as follows:

app/app.component.html (navigation links)

content\_copy<nav>

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

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

<a [routerLink](https://angular.io/api/router/RouterLink)="/about">About</a>

</nav>

Here are some tests that confirm those links are wired to the [routerLink](https://angular.io/api/router/RouterLink) directives as expected:

app/app.component.spec.ts (selected tests)

content\_copyit('can get RouterLinks from template', () => {

expect(routerLinks.length).toBe(3, 'should have 3 routerLinks');

expect(routerLinks[0].linkParams).toBe('/dashboard');

expect(routerLinks[1].linkParams).toBe('/heroes');

expect(routerLinks[2].linkParams).toBe('/about');

});

it('can click Heroes link in template', () => {

const heroesLinkDe = linkDes[1]; // heroes link [DebugElement](https://angular.io/api/core/DebugElement)

const heroesLink = routerLinks[1]; // heroes link directive

expect(heroesLink.navigatedTo).toBeNull('should not have navigated yet');

heroesLinkDe.triggerEventHandler('click', null);

fixture.detectChanges();

expect(heroesLink.navigatedTo).toBe('/heroes');

});

The "click" test in this example is misleading. It tests the RouterLinkDirectiveStub rather than the component. This is a common failing of directive stubs.

It has a legitimate purpose in this guide. It demonstrates how to find a [RouterLink](https://angular.io/api/router/RouterLink) element, click it, and inspect a result, without engaging the full router machinery. This is a skill you may need to test a more sophisticated component, one that changes the display, re-calculates parameters, or re-arranges navigation options when the user clicks the link.

#### What good are these tests?

Stubbed [RouterLink](https://angular.io/api/router/RouterLink) tests can confirm that a component with links and an outlet is setup properly, that the component has the links it should have, and that they are all pointing in the expected direction. These tests do not concern whether the application will succeed in navigating to the target component when the user clicks a link.

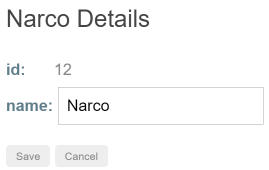
Stubbing the RouterLink and RouterOutlet is the best option for such limited testing goals. Relying on the real router would make them brittle. They could fail for reasons unrelated to the component. For example, a navigation guard could prevent an unauthorized user from visiting the HeroListComponent. That's not the fault of the AppComponent and no change to that component could cure the failed test.

A different battery of tests can explore whether the application navigates as expected in the presence of conditions that influence guards such as whether the user is authenticated and authorized.

A future guide update will explain how to write such tests with the [RouterTestingModule](https://angular.io/api/router/testing/RouterTestingModule).

## **Use a page object**

The HeroDetailComponent is a simple view with a title, two hero fields, and two buttons.



But there's plenty of template complexity even in this simple form.

app/hero/hero-detail.component.html

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

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

<div>

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

<div>

<label for="name">name: </label>

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

</div>

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

<button (click)="cancel()">Cancel</button>

</div>

Tests that exercise the component need ...

* to wait until a hero arrives before elements appear in the DOM.
* a reference to the title text.
* a reference to the name input box to inspect and set it.
* references to the two buttons so they can click them.
* spies for some of the component and router methods.

Even a small form such as this one can produce a mess of tortured conditional setup and CSS element selection.

Tame the complexity with a Page class that handles access to component properties and encapsulates the logic that sets them.

Here is such a Page class for the hero-detail.component.spec.ts

app/hero/hero-detail.component.spec.ts (Page)

content\_copyclass Page {

// getter properties wait to [query](https://angular.io/api/animations/query) the DOM until called.

get buttons() {

return this.queryAll<HTMLButtonElement>('button');

}

get saveBtn() {

return this.buttons[0];

}

get cancelBtn() {

return this.buttons[1];

}

get nameDisplay() {

return this.query<HTMLElement>('span');

}

get nameInput() {

return this.query<HTMLInputElement>('input');

}

gotoListSpy: jasmine.Spy;

navigateSpy: jasmine.Spy;

constructor(someFixture: [ComponentFixture](https://angular.io/api/core/testing/ComponentFixture)<HeroDetailComponent>) {

// get the navigate spy from the injected router spy object

const routerSpy = someFixture.debugElement.injector.get([Router](https://angular.io/api/router/Router)) as any;

this.navigateSpy = routerSpy.navigate;

// spy on component's `gotoList()` method

const someComponent = someFixture.componentInstance;

this.gotoListSpy = spyOn(someComponent, 'gotoList').and.callThrough();

}

//// [query](https://angular.io/api/animations/query) helpers ////

private [query](https://angular.io/api/animations/query)<T>(selector: string): T {

return fixture.nativeElement.querySelector(selector);

}

private queryAll<T>(selector: string): T[] {

return fixture.nativeElement.querySelectorAll(selector);

}

}

Now the important hooks for component manipulation and inspection are neatly organized and accessible from an instance of Page.

A createComponent method creates a page object and fills in the blanks once the hero arrives.

app/hero/hero-detail.component.spec.ts (createComponent)

content\_copy/\*\* Create the HeroDetailComponent, initialize it, set test variables \*/

function createComponent() {

fixture = TestBed.createComponent(HeroDetailComponent);

component = fixture.componentInstance;

page = new Page(fixture);

// 1st change detection triggers ngOnInit which gets a hero

fixture.detectChanges();

return fixture.whenStable().then(() => {

// 2nd change detection displays the async-fetched hero

fixture.detectChanges();

});

}

The [HeroDetailComponent tests](https://angular.io/guide/testing-components-scenarios#tests-w-test-double) in an earlier section demonstrate how createComponent and page keep the tests short and on message. There are no distractions: no waiting for promises to resolve and no searching the DOM for element values to compare.

Here are a few more HeroDetailComponent tests to reinforce the point.

app/hero/hero-detail.component.spec.ts (selected tests)

content\_copyit('should display that hero\'s name', () => {

expect(page.nameDisplay.textContent).toBe(expectedHero.name);

});

it('should navigate when click cancel', () => {

click(page.cancelBtn);

expect(page.navigateSpy.calls.any()).toBe(true, 'router.navigate called');

});

it('should save when click save but not navigate immediately', () => {

// Get service injected into component and spy on its`saveHero` method.

// It delegates to fake `HeroService.updateHero` which delivers a safe test result.

const hds = fixture.debugElement.injector.get(HeroDetailService);

const saveSpy = spyOn(hds, 'saveHero').and.callThrough();

click(page.saveBtn);

expect(saveSpy.calls.any()).toBe(true, 'HeroDetailService.save called');

expect(page.navigateSpy.calls.any()).toBe(false, 'router.navigate not called');

});

it('should navigate when click save and save resolves', [fakeAsync](https://angular.io/api/core/testing/fakeAsync)(() => {

click(page.saveBtn);

[tick](https://angular.io/api/core/testing/tick)(); // wait for async save to complete

expect(page.navigateSpy.calls.any()).toBe(true, 'router.navigate called');

}));

it('should convert hero name to [Title](https://angular.io/api/platform-browser/Title) Case', () => {

// get the name's input and display elements from the DOM

const hostElement: HTMLElement = fixture.nativeElement;

const nameInput: HTMLInputElement = hostElement.querySelector('input')!;

const nameDisplay: HTMLElement = hostElement.querySelector('span')!;

// simulate user entering a new name into the input box

nameInput.value = 'quick BROWN fOx';

// Dispatch a DOM event so that Angular learns of input value change.

// In older browsers, such as IE, you might need a CustomEvent instead. See

// https://developer.mozilla.org/en-US/docs/Web/API/CustomEvent/CustomEvent#Polyfill

nameInput.dispatchEvent(new [Event](https://angular.io/api/router/Event)('input'));

// Tell Angular to update the display binding through the title pipe

fixture.detectChanges();

expect(nameDisplay.textContent).toBe('Quick Brown Fox');

});

## **Calling compileComponents()**

You can ignore this section if you only run tests with the CLI ng test command because the CLI compiles the application before running the tests.

If you run tests in a **non-CLI environment**, the tests may fail with a message like this one:

Error: This test module uses the component BannerComponent

which is using a "templateUrl" or "styleUrls", but they were never compiled.

Please call "TestBed.compileComponents" before your test.

The root of the problem is at least one of the components involved in the test specifies an external template or CSS file as the following version of the BannerComponent does.

app/banner/banner-external.component.ts (external template & css)

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

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

selector: 'app-banner',

templateUrl: './banner-external.component.html',

styleUrls: ['./banner-external.component.css']

})

export class BannerComponent {

title = 'Test Tour of Heroes';

}

The test fails when the [TestBed](https://angular.io/api/core/testing/TestBed) tries to create the component.

app/banner/banner-external.component.spec.ts (setup that fails)

content\_copybeforeEach(async () => {

await TestBed.configureTestingModule({

declarations: [ BannerComponent ],

}); // missing call to compileComponents()

fixture = TestBed.createComponent(BannerComponent);

});

Recall that the application hasn't been compiled. So when you call createComponent(), the [TestBed](https://angular.io/api/core/testing/TestBed) compiles implicitly.

That's not a problem when the source code is in memory. But the BannerComponent requires external files that the compiler must read from the file system, an inherently asynchronous operation.

If the [TestBed](https://angular.io/api/core/testing/TestBed) were allowed to continue, the tests would run and fail mysteriously before the compiler could finished.

The preemptive error message tells you to compile explicitly with compileComponents().

#### compileComponents() is async

You must call compileComponents() within an asynchronous test function.

If you neglect to make the test function async (for example, forget to use [waitForAsync](https://angular.io/api/core/testing/waitForAsync)() as described below), you'll see this error message

Error: ViewDestroyedError: Attempt to use a destroyed view

A typical approach is to divide the setup logic into two separate beforeEach() functions:

1. An async beforeEach() that compiles the components
2. A synchronous beforeEach() that performs the remaining setup.

#### The async beforeEach

Write the first async beforeEach like this.

app/banner/banner-external.component.spec.ts (async beforeEach)

content\_copybeforeEach(async () => {

await TestBed.configureTestingModule({

declarations: [ BannerComponent ],

}).compileComponents(); // compile template and css

});

The TestBed.configureTestingModule() method returns the [TestBed](https://angular.io/api/core/testing/TestBed) class so you can chain calls to other [TestBed](https://angular.io/api/core/testing/TestBed) static methods such as compileComponents().

In this example, the BannerComponent is the only component to compile. Other examples configure the testing module with multiple components and may import application modules that hold yet more components. Any of them could require external files.

The TestBed.compileComponents method asynchronously compiles all components configured in the testing module.

Do not re-configure the [TestBed](https://angular.io/api/core/testing/TestBed) after calling compileComponents().

Calling compileComponents() closes the current [TestBed](https://angular.io/api/core/testing/TestBed) instance to further configuration. You cannot call any more [TestBed](https://angular.io/api/core/testing/TestBed) configuration methods, not configureTestingModule() nor any of the override... methods. The [TestBed](https://angular.io/api/core/testing/TestBed) throws an error if you try.

Make compileComponents() the last step before calling TestBed.createComponent().

#### The synchronous beforeEach

The second, synchronous beforeEach() contains the remaining setup steps, which include creating the component and querying for elements to inspect.

app/banner/banner-external.component.spec.ts (synchronous beforeEach)

content\_copybeforeEach(() => {

fixture = TestBed.createComponent(BannerComponent);

component = fixture.componentInstance; // BannerComponent test instance

h1 = fixture.nativeElement.querySelector('h1');

});

You can count on the test runner to wait for the first asynchronous beforeEach to finish before calling the second.

#### Consolidated setup

You can consolidate the two beforeEach() functions into a single, async beforeEach().

The compileComponents() method returns a promise so you can perform the synchronous setup tasks after compilation by moving the synchronous code into a then(...) callback.

app/banner/banner-external.component.spec.ts (one beforeEach)

content\_copybeforeEach(async () => {

await TestBed.configureTestingModule({

declarations: [ BannerComponent ],

}).compileComponents();

fixture = TestBed.createComponent(BannerComponent);

component = fixture.componentInstance;

h1 = fixture.nativeElement.querySelector('h1');

});

#### compileComponents() is harmless

There's no harm in calling compileComponents() when it's not required.

The component test file generated by the CLI calls compileComponents() even though it is never required when running ng test.

The tests in this guide only call compileComponents when necessary.

## **Setup with module imports**

Earlier component tests configured the testing module with a few declarations like this:

app/dashboard/dashboard-hero.component.spec.ts (configure TestBed)

content\_copy[TestBed](https://angular.io/api/core/testing/TestBed)

.configureTestingModule({declarations: [DashboardHeroComponent]})

The DashboardComponent is simple. It needs no help. But more complex components often depend on other components, directives, pipes, and providers and these must be added to the testing module too.

Fortunately, the TestBed.configureTestingModule parameter parallels the metadata passed to the @[NgModule](https://angular.io/api/core/NgModule) decorator which means you can also specify providers and imports.

The HeroDetailComponent requires a lot of help despite its small size and simple construction. In addition to the support it receives from the default testing module [CommonModule](https://angular.io/api/common/CommonModule), it needs:

* [NgModel](https://angular.io/api/forms/NgModel) and friends in the [FormsModule](https://angular.io/api/forms/FormsModule) to enable two-way data binding.
* The [TitleCasePipe](https://angular.io/api/common/TitleCasePipe) from the shared folder.
* Router services (which these tests are stubbing).
* Hero data access services (also stubbed).

One approach is to configure the testing module from the individual pieces as in this example:

app/hero/hero-detail.component.spec.ts (FormsModule setup)

content\_copybeforeEach(async () => {

const routerSpy = createRouterSpy();

await [TestBed](https://angular.io/api/core/testing/TestBed)

.configureTestingModule({

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

declarations: [HeroDetailComponent, [TitleCasePipe](https://angular.io/api/common/TitleCasePipe)],

providers: [

{provide: [ActivatedRoute](https://angular.io/api/router/ActivatedRoute), useValue: activatedRoute},

{provide: HeroService, useClass: TestHeroService},

{provide: [Router](https://angular.io/api/router/Router), useValue: routerSpy},

]

})

.compileComponents();

});

Notice that the beforeEach() is asynchronous and calls TestBed.compileComponents because the HeroDetailComponent has an external template and css file.

As explained in [Calling compileComponents()](https://angular.io/guide/testing-components-scenarios#compile-components) above, these tests could be run in a non-CLI environment where Angular would have to compile them in the browser.

#### Import a shared module

Because many application components need the [FormsModule](https://angular.io/api/forms/FormsModule) and the [TitleCasePipe](https://angular.io/api/common/TitleCasePipe), the developer created a SharedModule to combine these and other frequently requested parts.

The test configuration can use the SharedModule too as seen in this alternative setup:

app/hero/hero-detail.component.spec.ts (SharedModule setup)

content\_copybeforeEach(async () => {

const routerSpy = createRouterSpy();

await [TestBed](https://angular.io/api/core/testing/TestBed)

.configureTestingModule({

imports: [SharedModule],

declarations: [HeroDetailComponent],

providers: [

{provide: [ActivatedRoute](https://angular.io/api/router/ActivatedRoute), useValue: activatedRoute},

{provide: HeroService, useClass: TestHeroService},

{provide: [Router](https://angular.io/api/router/Router), useValue: routerSpy},

]

})

.compileComponents();

});

It's a bit tighter and smaller, with fewer import statements (not shown).

#### Import a feature module

The HeroDetailComponent is part of the HeroModule [Feature Module](https://angular.io/guide/feature-modules) that aggregates more of the interdependent pieces including the SharedModule. Try a test configuration that imports the HeroModule like this one:

app/hero/hero-detail.component.spec.ts (HeroModule setup)

content\_copybeforeEach(async () => {

const routerSpy = createRouterSpy();

await [TestBed](https://angular.io/api/core/testing/TestBed)

.configureTestingModule({

imports: [HeroModule],

providers: [

{provide: [ActivatedRoute](https://angular.io/api/router/ActivatedRoute), useValue: activatedRoute},

{provide: HeroService, useClass: TestHeroService},

{provide: [Router](https://angular.io/api/router/Router), useValue: routerSpy},

]

})

.compileComponents();

});

That's really crisp. Only the test doubles in the providers remain. Even the HeroDetailComponent declaration is gone.

In fact, if you try to declare it, Angular will throw an error because HeroDetailComponent is declared in both the HeroModule and the DynamicTestModule created by the [TestBed](https://angular.io/api/core/testing/TestBed).

Importing the component's feature module can be the easiest way to configure tests when there are many mutual dependencies within the module and the module is small, as feature modules tend to be.

## **Override component providers**

The HeroDetailComponent provides its own HeroDetailService.

app/hero/hero-detail.component.ts (prototype)

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

selector: 'app-hero-detail',

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

styleUrls: ['./hero-detail.component.css' ],

providers: [ HeroDetailService ]

})

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

constructor(

private heroDetailService: HeroDetailService,

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

private router: [Router](https://angular.io/api/router/Router)) {

}

}

It's not possible to stub the component's HeroDetailService in the providers of the TestBed.configureTestingModule. Those are providers for the testing module, not the component. They prepare the dependency injector at the fixture level.

Angular creates the component with its own injector, which is a child of the fixture injector. It registers the component's providers (the HeroDetailService in this case) with the child injector.

A test cannot get to child injector services from the fixture injector. And TestBed.configureTestingModule can't configure them either.

Angular has been creating new instances of the real HeroDetailService all along!

These tests could fail or timeout if the HeroDetailService made its own XHR calls to a remote server. There might not be a remote server to call.

Fortunately, the HeroDetailService delegates responsibility for remote data access to an injected HeroService.

app/hero/hero-detail.service.ts (prototype)

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

export class HeroDetailService {

constructor(private heroService: HeroService) { }

/\* . . . \*/

}

The [previous test configuration](https://angular.io/guide/testing-components-scenarios#feature-module-import) replaces the real HeroService with a TestHeroService that intercepts server requests and fakes their responses.

What if you aren't so lucky. What if faking the HeroService is hard? What if HeroDetailService makes its own server requests?

The TestBed.overrideComponent method can replace the component's providers with easy-to-manage test doubles as seen in the following setup variation:

app/hero/hero-detail.component.spec.ts (Override setup)

content\_copybeforeEach(async () => {

const routerSpy = createRouterSpy();

await [TestBed](https://angular.io/api/core/testing/TestBed)

.configureTestingModule({

imports: [HeroModule],

providers: [

{provide: [ActivatedRoute](https://angular.io/api/router/ActivatedRoute), useValue: activatedRoute},

{provide: [Router](https://angular.io/api/router/Router), useValue: routerSpy},

]

})

// Override component's own provider

.overrideComponent(

HeroDetailComponent,

{set: {providers: [{provide: HeroDetailService, useClass: HeroDetailServiceSpy}]}})

.compileComponents();

});

Notice that TestBed.configureTestingModule no longer provides a (fake) HeroService because it's [not needed](https://angular.io/guide/testing-components-scenarios#spy-stub).

#### The overrideComponent method

Focus on the overrideComponent method.

app/hero/hero-detail.component.spec.ts (overrideComponent)

content\_copy.overrideComponent(

HeroDetailComponent,

{set: {providers: [{provide: HeroDetailService, useClass: HeroDetailServiceSpy}]}})

It takes two arguments: the component type to override (HeroDetailComponent) and an override metadata object. The [override metadata object](https://angular.io/guide/testing-utility-apis#metadata-override-object) is a generic defined as follows:

content\_copytype [MetadataOverride](https://angular.io/api/core/testing/MetadataOverride)<T> = {

add?: Partial<T>;

remove?: Partial<T>;

set?: Partial<T>;

};

A metadata override object can either add-and-remove elements in metadata properties or completely reset those properties. This example resets the component's providers metadata.

The type parameter, T, is the kind of metadata you'd pass to the @[Component](https://angular.io/api/core/Component) decorator:

content\_copyselector?: string;

template?: string;

templateUrl?: string;

providers?: any[];

...

#### Provide a spy stub (HeroDetailServiceSpy)

This example completely replaces the component's providers array with a new array containing a HeroDetailServiceSpy.

The HeroDetailServiceSpy is a stubbed version of the real HeroDetailService that fakes all necessary features of that service. It neither injects nor delegates to the lower level HeroService so there's no need to provide a test double for that.

The related HeroDetailComponent tests will assert that methods of the HeroDetailService were called by spying on the service methods. Accordingly, the stub implements its methods as spies:

app/hero/hero-detail.component.spec.ts (HeroDetailServiceSpy)

content\_copyclass HeroDetailServiceSpy {

testHero: Hero = {id: 42, name: 'Test Hero'};

/\* emit cloned test hero \*/

getHero = jasmine.createSpy('getHero').and.callFake(

() => asyncData(Object.assign({}, this.testHero)));

/\* emit clone of test hero, with changes merged in \*/

saveHero = jasmine.createSpy('saveHero')

.and.callFake((hero: Hero) => asyncData(Object.assign(this.testHero, hero)));

}

#### The override tests

Now the tests can control the component's hero directly by manipulating the spy-stub's testHero and confirm that service methods were called.

app/hero/hero-detail.component.spec.ts (override tests)

content\_copylet hdsSpy: HeroDetailServiceSpy;

beforeEach(async () => {

await createComponent();

// get the component's injected HeroDetailServiceSpy

hdsSpy = fixture.debugElement.injector.get(HeroDetailService) as any;

});

it('should have called `getHero`', () => {

expect(hdsSpy.getHero.calls.count()).toBe(1, 'getHero called once');

});

it('should display stub hero\'s name', () => {

expect(page.nameDisplay.textContent).toBe(hdsSpy.testHero.name);

});

it('should save stub hero change', [fakeAsync](https://angular.io/api/core/testing/fakeAsync)(() => {

const origName = hdsSpy.testHero.name;

const newName = 'New Name';

page.nameInput.value = newName;

// In older browsers, such as IE, you might need a CustomEvent instead. See

// https://developer.mozilla.org/en-US/docs/Web/API/CustomEvent/CustomEvent#Polyfill

page.nameInput.dispatchEvent(new [Event](https://angular.io/api/router/Event)('input')); // tell Angular

expect(component.hero.name).toBe(newName, 'component hero has new name');

expect(hdsSpy.testHero.name).toBe(origName, 'service hero unchanged before save');

click(page.saveBtn);

expect(hdsSpy.saveHero.calls.count()).toBe(1, 'saveHero called once');

[tick](https://angular.io/api/core/testing/tick)(); // wait for async save to complete

expect(hdsSpy.testHero.name).toBe(newName, 'service hero has new name after save');

expect(page.navigateSpy.calls.any()).toBe(true, 'router.navigate called');

}));

#### More overrides

The TestBed.overrideComponent method can be called multiple times for the same or different components. The [TestBed](https://angular.io/api/core/testing/TestBed) offers similar overrideDirective, overrideModule, and overridePipe methods for digging into and replacing parts of these other classes.

Explore the options and combinations on your own.

# Testing Attribute Directives

An attribute directive modifies the behavior of an element, component or another directive. Its name reflects the way the directive is applied: as an attribute on a host element.

For a hands-on experience you can [run tests and explore the test code](https://angular.io/generated/live-examples/testing/specs.stackblitz.html) in your browser as your read this guide.

If you'd like to experiment with the application that this guide describes, you can [run it in your browser](https://angular.io/generated/live-examples/testing/stackblitz.html) or [download and run it locally](https://angular.io/generated/zips/testing/testing.zip).

## **Testing the HighlightDirective**

The sample application's HighlightDirective sets the background color of an element based on either a data bound color or a default color (lightgray). It also sets a custom property of the element (customProperty) to true for no reason other than to show that it can.

app/shared/highlight.directive.ts

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

@[Directive](https://angular.io/api/core/Directive)({ selector: '[highlight]' })

/\*\*

\* Set backgroundColor for the attached element to highlight color

\* and set the element's customProperty to true

\*/

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

defaultColor = 'rgb(211, 211, 211)'; // lightgray

@[Input](https://angular.io/api/core/Input)('highlight') bgColor = '';

constructor(private el: [ElementRef](https://angular.io/api/core/ElementRef)) {

el.nativeElement.style.customProperty = true;

}

ngOnChanges() {

this.el.nativeElement.style.backgroundColor = this.bgColor || this.defaultColor;

}

}

It's used throughout the application, perhaps most simply in the AboutComponent:

app/about/about.component.ts

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

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

template: `

<h2 highlight="skyblue">About</h2>

<h3>Quote of the day:</h3>

<twain-quote></twain-quote>

`

})

export class AboutComponent { }

Testing the specific use of the HighlightDirective within the AboutComponent requires only the techniques explored in the ["Nested component tests"](https://angular.io/guide/testing-components-scenarios#nested-component-tests) section of [Component testing scenarios](https://angular.io/guide/testing-components-scenarios).

app/about/about.component.spec.ts

content\_copybeforeEach(() => {

fixture = TestBed.configureTestingModule({

declarations: [ AboutComponent, HighlightDirective ],

schemas: [ [CUSTOM\_ELEMENTS\_SCHEMA](https://angular.io/api/core/CUSTOM_ELEMENTS_SCHEMA) ]

})

.createComponent(AboutComponent);

fixture.detectChanges(); // initial binding

});

it('should have skyblue <h2>', () => {

const h2: HTMLElement = fixture.nativeElement.querySelector('h2');

const bgColor = h2.style.backgroundColor;

expect(bgColor).toBe('skyblue');

});

However, testing a single use case is unlikely to explore the full range of a directive's capabilities. Finding and testing all components that use the directive is tedious, brittle, and almost as unlikely to afford full coverage.

Class-only tests might be helpful, but attribute directives like this one tend to manipulate the DOM. Isolated unit tests don't touch the DOM and, therefore, do not inspire confidence in the directive's efficacy.

A better solution is to create an artificial test component that demonstrates all ways to apply the directive.

app/shared/highlight.directive.spec.ts (TestComponent)

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

template: `

<h2 highlight="yellow">Something Yellow</h2>

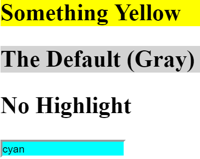
<h2 highlight>The Default (Gray)</h2>

<h2>No Highlight</h2>

<input #box [highlight]="box.value" value="cyan"/>`

})

class TestComponent { }



The <input> case binds the HighlightDirective to the name of a color value in the input box. The initial value is the word "cyan" which should be the background color of the input box.

Here are some tests of this component:

app/shared/highlight.directive.spec.ts (selected tests)

content\_copybeforeEach(() => {

fixture = TestBed.configureTestingModule({

declarations: [ HighlightDirective, TestComponent ]

})

.createComponent(TestComponent);

fixture.detectChanges(); // initial binding

// all elements with an attached HighlightDirective

des = fixture.debugElement.queryAll(By.directive(HighlightDirective));

// the h2 without the HighlightDirective

bareH2 = fixture.debugElement.query(By.css('h2:not([highlight])'));

});

// color tests

it('should have three highlighted elements', () => {

expect(des.length).toBe(3);

});

it('should color 1st <h2> background "yellow"', () => {

const bgColor = des[0].nativeElement.style.backgroundColor;

expect(bgColor).toBe('yellow');

});

it('should color 2nd <h2> background w/ default color', () => {

const dir = des[1].injector.get(HighlightDirective) as HighlightDirective;

const bgColor = des[1].nativeElement.style.backgroundColor;

expect(bgColor).toBe(dir.defaultColor);

});

it('should bind <input> background to value color', () => {

// easier to work with nativeElement

const input = des[2].nativeElement as HTMLInputElement;

expect(input.style.backgroundColor).toBe('cyan', 'initial backgroundColor');

input.value = 'green';

// Dispatch a DOM event so that Angular responds to the input value change.

// In older browsers, such as IE, you might need a CustomEvent instead. See

// https://developer.mozilla.org/en-US/docs/Web/API/CustomEvent/CustomEvent#Polyfill

input.dispatchEvent(new [Event](https://angular.io/api/router/Event)('input'));

fixture.detectChanges();

expect(input.style.backgroundColor).toBe('green', 'changed backgroundColor');

});

it('bare <h2> should not have a customProperty', () => {

expect(bareH2.properties.customProperty).toBeUndefined();

});

A few techniques are noteworthy:

* The By.directive predicate is a great way to get the elements that have this directive when their element types are unknown.
* The [:notpseudo-class](https://developer.mozilla.org/en-US/docs/Web/CSS/:not) in By.css('h2:not([highlight])') helps find <h2> elements that do not have the directive. By.css('\*:not([highlight])') finds any element that does not have the directive.
* [DebugElement.styles](https://angular.io/api/core/DebugElement#styles) affords access to element styles even in the absence of a real browser, thanks to the [DebugElement](https://angular.io/api/core/DebugElement) abstraction. But feel free to exploit the nativeElement when that seems easier or more clear than the abstraction.
* Angular adds a directive to the injector of the element to which it is applied. The test for the default color uses the injector of the second <h2> to get its HighlightDirective instance and its defaultColor.
* [DebugElement.properties](https://angular.io/api/core/DebugElement#properties) affords access to the artificial custom property that is set by the directive.

# Testing Pipes

You can test [pipes](https://angular.io/guide/pipes) without the Angular testing utilities.

For a hands-on experience you can [run tests and explore the test code](https://angular.io/generated/live-examples/testing/specs.stackblitz.html) in your browser as your read this guide.

If you'd like to experiment with the application that this guide describes, you can [run it in your browser](https://angular.io/generated/live-examples/testing/stackblitz.html) or [download and run it locally](https://angular.io/generated/zips/testing/testing.zip).

## **Testing the**[**TitleCasePipe**](https://angular.io/api/common/TitleCasePipe)

A pipe class has one method, transform, that manipulates the input value into a transformed output value. The transform implementation rarely interacts with the DOM. Most pipes have no dependence on Angular other than the @[Pipe](https://angular.io/api/core/Pipe) metadata and an interface.

Consider a [TitleCasePipe](https://angular.io/api/common/TitleCasePipe) that capitalizes the first letter of each word. Here's an implementation with a regular expression.

app/shared/title-case.pipe.ts

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

@[Pipe](https://angular.io/api/core/Pipe)({name: 'titlecase', pure: true})

/\*\* Transform to [Title](https://angular.io/api/platform-browser/Title) Case: uppercase the first letter of the words in a string. \*/

export class [TitleCasePipe](https://angular.io/api/common/TitleCasePipe) implements [PipeTransform](https://angular.io/api/core/PipeTransform) {

transform(input: string): string {

return input.length === 0 ? '' :

input.replace(/\w\S\*/g, (txt => txt[0].toUpperCase() + txt.substr(1).toLowerCase() ));

}

}

Anything that uses a regular expression is worth testing thoroughly. Use simple Jasmine to explore the expected cases and the edge cases.

app/shared/title-case.pipe.spec.ts

content\_copydescribe('[TitleCasePipe](https://angular.io/api/common/TitleCasePipe)', () => {

// This pipe is a pure, stateless function so no need for BeforeEach

const pipe = new [TitleCasePipe](https://angular.io/api/common/TitleCasePipe)();

it('transforms "abc" to "Abc"', () => {

expect(pipe.transform('abc')).toBe('Abc');

});

it('transforms "abc def" to "Abc Def"', () => {

expect(pipe.transform('abc def')).toBe('Abc Def');

});

// ... more tests ...

});

## **Writing DOM tests to support a pipe test**

These are tests of the pipe in isolation. They can't tell if the [TitleCasePipe](https://angular.io/api/common/TitleCasePipe) is working properly as applied in the application components.

Consider adding component tests such as this one:

app/hero/hero-detail.component.spec.ts (pipe test)

content\_copyit('should convert hero name to [Title](https://angular.io/api/platform-browser/Title) Case', () => {

// get the name's input and display elements from the DOM

const hostElement: HTMLElement = fixture.nativeElement;

const nameInput: HTMLInputElement = hostElement.querySelector('input')!;

const nameDisplay: HTMLElement = hostElement.querySelector('span')!;

// simulate user entering a new name into the input box

nameInput.value = 'quick BROWN fOx';

// Dispatch a DOM event so that Angular learns of input value change.

// In older browsers, such as IE, you might need a CustomEvent instead. See

// https://developer.mozilla.org/en-US/docs/Web/API/CustomEvent/CustomEvent#Polyfill

nameInput.dispatchEvent(new [Event](https://angular.io/api/router/Event)('input'));

// Tell Angular to update the display binding through the title pipe

fixture.detectChanges();

expect(nameDisplay.textContent).toBe('Quick Brown Fox');

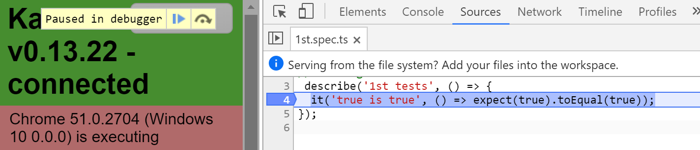
});

# Debugging tests

If your tests aren't working as you expect them to, you can inspect and debug them in the browser.

Debug specs in the browser in the same way that you debug an application.

1. Reveal the Karma browser window. See [Set up testing](https://angular.io/guide/testing#set-up-testing) if you need help with this step.
2. Click the **DEBUG** button; it opens a new browser tab and re-runs the tests.
3. Open the browser's “Developer Tools” (Ctrl-Shift-I on Windows; Command-Option-I in macOS).
4. Pick the "sources" section.
5. Open the 1st.spec.ts test file (Control/Command-P, then start typing the name of the file).
6. Set a breakpoint in the test.
7. Refresh the browser, and it stops at the breakpoint.



# Testing Utility APIs

This page describes the most useful Angular testing features.

The Angular testing utilities include the [TestBed](https://angular.io/api/core/testing/TestBed), the [ComponentFixture](https://angular.io/api/core/testing/ComponentFixture), and a handful of functions that control the test environment. The [TestBed](https://angular.io/guide/testing-utility-apis#testbed-api-summary) and [ComponentFixture](https://angular.io/guide/testing-utility-apis#component-fixture-api-summary) classes are covered separately.

Here's a summary of the stand-alone functions, in order of likely utility:

|  |  |
| --- | --- |
| **Function** | **Description** |
| [waitForAsync](https://angular.io/api/core/testing/waitForAsync) | Runs the body of a test (it) or setup (beforeEach) function within a special async test zone. See [waitForAsync](https://angular.io/guide/testing-components-scenarios#waitForAsync). |
| [fakeAsync](https://angular.io/api/core/testing/fakeAsync) | Runs the body of a test (it) within a special fakeAsync test zone, enabling a linear control flow coding style. See [fakeAsync](https://angular.io/guide/testing-components-scenarios#fake-async). |
| [tick](https://angular.io/api/core/testing/tick) | Simulates the passage of time and the completion of pending asynchronous activities by flushing both timer and micro-task queues within the fakeAsync test zone.  The curious, dedicated reader might enjoy this lengthy blog post, ["Tasks, microtasks, queues and schedules"](https://jakearchibald.com/2015/tasks-microtasks-queues-and-schedules/).  Accepts an optional argument that moves the virtual clock forward by the specified number of milliseconds, clearing asynchronous activities scheduled within that timeframe. See [tick](https://angular.io/guide/testing-components-scenarios#tick). |
| inject | Injects one or more services from the current [TestBed](https://angular.io/api/core/testing/TestBed) injector into a test function. It cannot inject a service provided by the component itself. See discussion of the [debugElement.injector](https://angular.io/guide/testing-components-scenarios#get-injected-services). |
| [discardPeriodicTasks](https://angular.io/api/core/testing/discardPeriodicTasks) | When a [fakeAsync](https://angular.io/api/core/testing/fakeAsync)() test ends with pending timer event tasks (queued setTimeOut and setInterval callbacks), the test fails with a clear error message.  In general, a test should end with no queued tasks. When pending timer tasks are expected, call [discardPeriodicTasks](https://angular.io/api/core/testing/discardPeriodicTasks) to flush the task queue and avoid the error. |
| [flushMicrotasks](https://angular.io/api/core/testing/flushMicrotasks) | When a [fakeAsync](https://angular.io/api/core/testing/fakeAsync)() test ends with pending micro-tasks such as unresolved promises, the test fails with a clear error message.  In general, a test should wait for micro-tasks to finish. When pending microtasks are expected, call [flushMicrotasks](https://angular.io/api/core/testing/flushMicrotasks) to flush the micro-task queue and avoid the error. |
| [ComponentFixtureAutoDetect](https://angular.io/api/core/testing/ComponentFixtureAutoDetect) | A provider token for a service that turns on [automatic change detection](https://angular.io/guide/testing-components-scenarios#automatic-change-detection). |
| [getTestBed](https://angular.io/api/core/testing/getTestBed) | Gets the current instance of the [TestBed](https://angular.io/api/core/testing/TestBed). Usually unnecessary because the static class methods of the [TestBed](https://angular.io/api/core/testing/TestBed) class are typically sufficient. The [TestBed](https://angular.io/api/core/testing/TestBed) instance exposes a few rarely used members that are not available as static methods. |

## **TestBed class summary**

The [TestBed](https://angular.io/api/core/testing/TestBed) class is one of the principal Angular testing utilities. Its API is quite large and can be overwhelming until you've explored it, a little at a time. Read the early part of this guide first to get the basics before trying to absorb the full API.

The module definition passed to configureTestingModule is a subset of the @[NgModule](https://angular.io/api/core/NgModule) metadata properties.

content\_copytype [TestModuleMetadata](https://angular.io/api/core/testing/TestModuleMetadata) = {

providers?: any[];

declarations?: any[];

imports?: any[];

schemas?: Array<[SchemaMetadata](https://angular.io/api/core/SchemaMetadata) | any[]>;

};

Each override method takes a [MetadataOverride](https://angular.io/api/core/testing/MetadataOverride)<T> where T is the kind of metadata appropriate to the method, that is, the parameter of an @[NgModule](https://angular.io/api/core/NgModule), @[Component](https://angular.io/api/core/Component), @[Directive](https://angular.io/api/core/Directive), or @[Pipe](https://angular.io/api/core/Pipe).

content\_copytype [MetadataOverride](https://angular.io/api/core/testing/MetadataOverride)<T> = {

add?: Partial<T>;

remove?: Partial<T>;

set?: Partial<T>;

};

The [TestBed](https://angular.io/api/core/testing/TestBed) API consists of static class methods that either update or reference a global instance of the [TestBed](https://angular.io/api/core/testing/TestBed).

Internally, all static methods cover methods of the current runtime [TestBed](https://angular.io/api/core/testing/TestBed) instance, which is also returned by the [getTestBed](https://angular.io/api/core/testing/getTestBed)() function.

Call [TestBed](https://angular.io/api/core/testing/TestBed) methods within a beforeEach() to ensure a fresh start before each individual test.

Here are the most important static methods, in order of likely utility.

|  |  |
| --- | --- |
| **Methods** | **Description** |
| configureTestingModule | The testing shims (karma-test-shim, browser-test-shim) establish the [initial test environment](https://angular.io/guide/testing) and a default testing module. The default testing module is configured with basic declaratives and some Angular service substitutes that every tester needs.  Call configureTestingModule to refine the testing module configuration for a particular set of tests by adding and removing imports, declarations (of components, directives, and pipes), and providers. |
| compileComponents | Compile the testing module asynchronously after you've finished configuring it. You **must** call this method if any of the testing module components have a templateUrl or styleUrls because fetching component template and style files is necessarily asynchronous. See [compileComponents](https://angular.io/guide/testing-components-scenarios#compile-components).  After calling compileComponents, the [TestBed](https://angular.io/api/core/testing/TestBed) configuration is frozen for the duration of the current spec. |
| createComponent | Create an instance of a component of type T based on the current [TestBed](https://angular.io/api/core/testing/TestBed) configuration. After calling compileComponent, the [TestBed](https://angular.io/api/core/testing/TestBed) configuration is frozen for the duration of the current spec. |
| overrideModule | Replace metadata for the given [NgModule](https://angular.io/api/core/NgModule). Recall that modules can import other modules. The overrideModule method can reach deeply into the current testing module to modify one of these inner modules. |
| overrideComponent | Replace metadata for the given component class, which could be nested deeply within an inner module. |
| overrideDirective | Replace metadata for the given directive class, which could be nested deeply within an inner module. |
| overridePipe | Replace metadata for the given pipe class, which could be nested deeply within an inner module. |
| inject | Retrieve a service from the current [TestBed](https://angular.io/api/core/testing/TestBed) injector.  The inject function is often adequate for this purpose. But inject throws an error if it can't provide the service.  What if the service is optional?  The TestBed.inject() method takes an optional second parameter, the object to return if Angular can't find the provider (null in this example):  app/demo/demo.testbed.spec.ts  content\_copyexpect(TestBed.inject(NotProvided, null)).toBeNull();  After calling TestBed.inject, the [TestBed](https://angular.io/api/core/testing/TestBed) configuration is frozen for the duration of the current spec. |
| initTestEnvironment | Initialize the testing environment for the entire test run.  The testing shims (karma-test-shim, browser-test-shim) call it for you so there is rarely a reason for you to call it yourself.  You may call this method exactly once. If you must change this default in the middle of your test run, call resetTestEnvironment first.  Specify the Angular compiler factory, a [PlatformRef](https://angular.io/api/core/PlatformRef), and a default Angular testing module. Alternatives for non-browser platforms are available in the general form @angular/platform-<platform\_name>/testing/<platform\_name>. |
| resetTestEnvironment | Reset the initial test environment, including the default testing module. |

A few of the [TestBed](https://angular.io/api/core/testing/TestBed) instance methods are not covered by static [TestBed](https://angular.io/api/core/testing/TestBed) class methods. These are rarely needed.

## **The ComponentFixture**

The TestBed.createComponent<T> creates an instance of the component T and returns a strongly typed [ComponentFixture](https://angular.io/api/core/testing/ComponentFixture) for that component.

The [ComponentFixture](https://angular.io/api/core/testing/ComponentFixture) properties and methods provide access to the component, its DOM representation, and aspects of its Angular environment.

### **ComponentFixture properties**

Here are the most important properties for testers, in order of likely utility.

|  |  |
| --- | --- |
| **Properties** | **Description** |
| componentInstance | The instance of the component class created by TestBed.createComponent. |
| debugElement | The [DebugElement](https://angular.io/api/core/DebugElement) associated with the root element of the component.  The debugElement provides insight into the component and its DOM element during test and debugging. It's a critical property for testers. The most interesting members are covered [below](https://angular.io/guide/testing-utility-apis#debug-element-details). |
| nativeElement | The native DOM element at the root of the component. |
| changeDetectorRef | The [ChangeDetectorRef](https://angular.io/api/core/ChangeDetectorRef) for the component.  The [ChangeDetectorRef](https://angular.io/api/core/ChangeDetectorRef) is most valuable when testing a component that has the [ChangeDetectionStrategy.OnPush](https://angular.io/api/core/ChangeDetectionStrategy#OnPush) method or the component's change detection is under your programmatic control. |

### **ComponentFixture methods**

The fixture methods cause Angular to perform certain tasks on the component tree. Call these method to trigger Angular behavior in response to simulated user action.

Here are the most useful methods for testers.

|  |  |
| --- | --- |
| **Methods** | **Description** |
| detectChanges | Trigger a change detection cycle for the component.  Call it to initialize the component (it calls ngOnInit) and after your test code, change the component's data bound property values. Angular can't see that you've changed personComponent.name and won't update the name binding until you call detectChanges.  Runs checkNoChanges afterwards to confirm that there are no circular updates unless called as detectChanges(false); |
| autoDetectChanges | Set this to true when you want the fixture to detect changes automatically.  When autodetect is true, the test fixture calls detectChanges immediately after creating the component. Then it listens for pertinent zone events and calls detectChanges accordingly. When your test code modifies component property values directly, you probably still have to call fixture.detectChanges to trigger data binding updates.  The default is false. Testers who prefer fine control over test behavior tend to keep it false. |
| checkNoChanges | Do a change detection run to make sure there are no pending changes. Throws an exceptions if there are. |
| isStable | If the fixture is currently stable, returns true. If there are async tasks that have not completed, returns false. |
| whenStable | Returns a promise that resolves when the fixture is stable.  To resume testing after completion of asynchronous activity or asynchronous change detection, hook that promise. See [whenStable](https://angular.io/guide/testing-components-scenarios#when-stable). |
| destroy | Trigger component destruction. |

#### DebugElement

The [DebugElement](https://angular.io/api/core/DebugElement) provides crucial insights into the component's DOM representation.

From the test root component's [DebugElement](https://angular.io/api/core/DebugElement) returned by fixture.debugElement, you can walk (and query) the fixture's entire element and component subtrees.

Here are the most useful [DebugElement](https://angular.io/api/core/DebugElement) members for testers, in approximate order of utility:

|  |  |
| --- | --- |
| **Member** | **Description** |
| nativeElement | The corresponding DOM element in the browser (null for WebWorkers). |
| [query](https://angular.io/api/animations/query) | Calling [query](https://angular.io/api/animations/query)(predicate: [Predicate](https://angular.io/api/core/Predicate)<[DebugElement](https://angular.io/api/core/DebugElement)>) returns the first [DebugElement](https://angular.io/api/core/DebugElement) that matches the [predicate](https://angular.io/guide/testing-utility-apis#query-predicate) at any depth in the subtree. |
| queryAll | Calling queryAll(predicate: [Predicate](https://angular.io/api/core/Predicate)<[DebugElement](https://angular.io/api/core/DebugElement)>) returns all DebugElements that matches the [predicate](https://angular.io/guide/testing-utility-apis#query-predicate) at any depth in subtree. |
| injector | The host dependency injector. For example, the root element's component instance injector. |
| componentInstance | The element's own component instance, if it has one. |
| context | An object that provides parent context for this element. Often an ancestor component instance that governs this element.  When an element is repeated within \*[ngFor](https://angular.io/api/common/NgForOf), the context is an [NgForOf](https://angular.io/api/common/NgForOf) whose $implicit property is the value of the row instance value. For example, the hero in \*[ngFor](https://angular.io/api/common/NgForOf)="let hero of heroes". |
| children | The immediate [DebugElement](https://angular.io/api/core/DebugElement) children. Walk the tree by descending through children.  [DebugElement](https://angular.io/api/core/DebugElement) also has childNodes, a list of [DebugNode](https://angular.io/api/core/DebugNode) objects. [DebugElement](https://angular.io/api/core/DebugElement) derives from [DebugNode](https://angular.io/api/core/DebugNode) objects and there are often more nodes than elements. Testers can usually ignore plain nodes. |
| parent | The [DebugElement](https://angular.io/api/core/DebugElement) parent. Null if this is the root element. |
| name | The element tag name, if it is an element. |
| triggerEventHandler | Triggers the event by its name if there is a corresponding listener in the element's listeners collection. The second parameter is the event object expected by the handler. See [triggerEventHandler](https://angular.io/guide/testing-components-scenarios#trigger-event-handler).  If the event lacks a listener or there's some other problem, consider calling nativeElement.dispatchEvent(eventObject). |
| listeners | The callbacks attached to the component's @[Output](https://angular.io/api/core/Output) properties and/or the element's event properties. |
| providerTokens | This component's injector lookup tokens. Includes the component itself plus the tokens that the component lists in its providers metadata. |
| source | Where to find this element in the source component template. |
| references | Dictionary of objects associated with template local variables (e.g. #foo), keyed by the local variable name. |

The DebugElement.query(predicate) and DebugElement.queryAll(predicate) methods take a predicate that filters the source element's subtree for matching [DebugElement](https://angular.io/api/core/DebugElement).

The predicate is any method that takes a [DebugElement](https://angular.io/api/core/DebugElement) and returns a truthy value. The following example finds all DebugElements with a reference to a template local variable named "content":

app/demo/demo.testbed.spec.ts

content\_copy// Filter for DebugElements with a #content reference

const contentRefs = el.queryAll( de => de.references.content);

The Angular [By](https://angular.io/api/platform-browser/By) class has three static methods for common predicates:

* By.all - return all elements.
* By.css(selector) - return elements with matching CSS selectors.
* By.directive(directive) - return elements that Angular matched to an instance of the directive class.

app/hero/hero-list.component.spec.ts

content\_copy// Can find [DebugElement](https://angular.io/api/core/DebugElement) either by css selector or by directive

const h2 = fixture.debugElement.query(By.css('h2'));

const directive = fixture.debugElement.query(By.directive(HighlightDirective));

# Localizing your app

Internationalization (i18n) is the process of designing and preparing your app to be usable in different locales around the world. Localization is the process of building versions of your app for different locales, including extracting text for translation into different languages, and formatting data for particular locales.

A locale identifies a region (such as a country) in which people speak a particular language or language variant. The locale determines the formatting and parsing of dates, times, numbers, and currencies as well as measurement units and the translated names for time zones, languages, and countries.

Create an adaptable user interface for all of your target locales that takes into consideration the differences in spacing for different languages. For details, see [How to approach internationalization](https://marketfinder.thinkwithgoogle.com/intl/en_us/guide/how-to-approach-i18n/#overview).

Use Angular to internationalize your app:

* Use built-in pipes to display dates, numbers, percentages, and currencies in a local format.
* Mark text in component templates for translation.
* Mark plural forms of expressions for translation.
* Mark alternate text for translation.

After preparing your app for an international audience, use the [Angular CLI](https://angular.io/cli) to localize your app by performing the following tasks:

* Use the CLI to extract marked text to a source language file.
* Make a copy of this file for each language, and send these translation files to a translator or service.
* Use the CLI to merge the finished translation files when building your app for one or more locales.

To explore the sample app with French translations used in this guide, see the [live example](https://angular.io/generated/live-examples/i18n/stackblitz.html) / [download example](https://angular.io/generated/zips/i18n/i18n.zip).

## **Prerequisites**

To prepare your app for translations, you should have a basic understanding of the following:

* [Templates](https://angular.io/guide/glossary#template)
* [Components](https://angular.io/guide/glossary#component)
* [Angular CLI](https://angular.io/guide/glossary#command-line-interface-cli) command-line tool for managing the Angular development cycle
* [Extensible Markup Language (XML)](https://www.w3.org/XML/) used for translation files

## **Steps to localize your app**

To localize your app, follow these general steps:

1. [Add the localize package](https://angular.io/guide/i18n#setting-up-cli).
2. [Refer to locales by ID](https://angular.io/guide/i18n#setting-up-locale).
3. [Format data based on locale](https://angular.io/guide/i18n#i18n-pipes).
4. [Prepare templates for translations](https://angular.io/guide/i18n#Template-translations).
5. [Work with translation files](https://angular.io/guide/i18n#ng-xi18n).
6. [Merge translations into the app](https://angular.io/guide/i18n#merge).
7. [Deploy multiple locales](https://angular.io/guide/i18n#deploy-locales).

While following these steps, you can [explore the translated example app](https://angular.io/guide/i18n#app-pre-translation).

The following are optional practices that may be required in special cases:

* [Set the source locale manually](https://angular.io/guide/i18n#set-source-manually) if you need to set the [LOCALE\_ID](https://angular.io/api/core/LOCALE_ID) token.
* [Import global variants of the locale data](https://angular.io/guide/i18n#import-locale) for extra locale data.
* [Manage marked text with custom IDs](https://angular.io/guide/i18n#custom-id) if you require more control over matching translations.

## **Add the localize package**

To take advantage of Angular's localization features, use the Angular CLI to add the @angular/localize package to your project:

content\_copyng add @angular/localize

This command updates your project's package.json and polyfills.ts files to import the @angular/localize package.

For more information about package.json and polyfill packages, see [Workspace npm dependencies](https://angular.io/guide/npm-packages).

If @angular/localize is not installed, the Angular CLI may generate an error when you try to build a localized version of your app.

## **Refer to locales by ID**

Refer to a locale using the Unicode locale identifier (ID), which specifies the language, country, and an optional code for further variants or subdivisions.

UNICODE LOCALE IDENTIFIERS

* For a list of language codes, see [ISO 639-2](https://www.loc.gov/standards/iso639-2/).
* IDs conform to the Unicode Common Locale Data Repository (CLDR). For more information about Unicode locale identifiers, see the [CLDR core specification](http://cldr.unicode.org/core-spec#Unicode_Language_and_Locale_Identifiers).
* CLDR and Angular base their identifiers on [BCP47 tags](https://tools.ietf.org/html/bcp47).

The ID consists of a language identifier, such as en for English or fr for French, followed by a dash (-) and a locale extension, such as US for the United States or CA for Canada. For example, en-US refers to English in the United States, and fr-CA refers to French in Canada. Angular uses this ID to find the correct corresponding locale data.

Many countries, such as France and Canada, use the same language (French, identified as fr) but differ in grammar, punctuation, and formats for currency, decimal numbers, and dates. Use a more specific locale ID, such as French for Canada (fr-CA), when localizing your app.

Angular by default uses en-US (English in the United States) as your app's source locale.

The [Angular repository](https://github.com/angular/angular/tree/master/packages/common/locales) includes common locales. You can change your app's source locale for the build by setting the source locale in the sourceLocale field of your app's [workspace configuration](https://angular.io/guide/workspace-config) file (angular.json). The build process (described in [Merge translations into the app](https://angular.io/guide/i18n#merge) in this guide) uses your app's angular.json file to automatically set the [LOCALE\_ID](https://angular.io/api/core/LOCALE_ID) token and load the locale data.

## **Format data based on locale**

Angular provides the following built-in data transformation [pipes](https://angular.io/guide/glossary#pipe) that use the [LOCALE\_ID](https://angular.io/api/core/LOCALE_ID) token to format data according to the locale's rules:

* [DatePipe](https://angular.io/api/common/DatePipe): Formats a date value.
* [CurrencyPipe](https://angular.io/api/common/CurrencyPipe): Transforms a number to a currency string.
* [DecimalPipe](https://angular.io/api/common/DecimalPipe): Transforms a number into a decimal number string.
* [PercentPipe](https://angular.io/api/common/PercentPipe): Transforms a number to a percentage string.

For example, {{today | [date](https://angular.io/api/common/DatePipe)}} uses [DatePipe](https://angular.io/api/common/DatePipe) to display the current date in the format for the locale in [LOCALE\_ID](https://angular.io/api/core/LOCALE_ID).

To override the value of [LOCALE\_ID](https://angular.io/api/core/LOCALE_ID), add the locale parameter. For example, to force the currency to use en-US no matter which language-locale you set for [LOCALE\_ID](https://angular.io/api/core/LOCALE_ID), use this form: {{amount | [currency](https://angular.io/api/common/CurrencyPipe) : 'en-US'}}.

## **Prepare templates for translations**

To translate your app's templates, you need to prepare the text for a translator or translation service by marking text, attributes, and other elements with the Angular i18n attribute. Follow these general steps:

1. [Mark text for translations](https://angular.io/guide/i18n#i18n-attribute).
2. [Add helpful descriptions and meanings](https://angular.io/guide/i18n#help-translator) to help the translator with additional information or context.
3. [Translate text not for display](https://angular.io/guide/i18n#no-element).
4. [Mark element attributes for translations](https://angular.io/guide/i18n#translate-attributes), such as an image's title attribute.
5. [Mark plurals and alternates for translation](https://angular.io/guide/i18n#plurals-alternates) in order to comply with the pluralization rules and grammatical constructions of different languages.

### **Mark text for translations**

Mark the static text messages in your component templates for translation using the i18n attribute. Place it on every element tag with fixed text to be translated.

For example, the following <h1> tag displays a simple English language greeting, "Hello i18n!"

src/app/app.component.html

content\_copy<h1>Hello i18n!</h1>

To mark the greeting for translation, add the i18n attribute to the <h1> tag.

src/app/app.component.html

content\_copy<h1 i18n>Hello i18n!</h1>

i18n is a custom attribute, recognized by Angular tools and compilers. After translation, the compiler removes it. It is not an Angular directive.

### **Add helpful descriptions and meanings**

To translate a text message accurately, the translator may need additional information or context. Add a description of the text message as the value of the i18n attribute, as shown in the following example:

src/app/app.component.html

content\_copy<h1 i18n="An introduction header for this sample">Hello i18n!</h1>

The translator may also need to know the meaning or intent of the text message within this particular app context, in order to translate it the same way as other text with the same meaning. Start the i18n attribute value with the meaning and separate it from the description with the | character: <meaning>|<description>.

For example, you can add the meaning that this <h1> tag is a site header that needs to be translated the same way not only when used as a header, but also when referred to from another section of text:

src/app/app.component.html

content\_copy<h1 i18n="site header|An introduction header for this sample">Hello i18n!</h1>

As a result, any text marked with site header as the meaning is translated exactly the same way.

HOW MEANINGS CONTROL TEXT EXTRACTION AND MERGING

The Angular extraction tool (described in [Work with translation files](https://angular.io/guide/i18n#ng-xi18n) in this guide) generates a translation unit entry for each i18n attribute in a template. It assigns each translation unit a unique ID based on the meaning and description.

The same text elements with different meanings are extracted with separate IDs. For example, if the word "right" appears with the meaning correct (as in "You are right") in one place, and with the meaning direction (as in "Turn right") in another place, the word is translated differently and merged back into the app as different translation entries.

If the same text elements have different descriptions but the same meaning, they are extracted only once, with only one ID. That one translation entry is merged back into the app wherever the same text elements appear.

### **Translate text not for display**

While you can translate non-displayed text using a <span> tag, you are creating a new DOM element. To avoid doing so, wrap the text in an [<ng-container>](https://angular.io/api/core/ng-container) element, which is transformed into a non-displayed HTML comment as shown in this example:

content\_copy<ng-container i18n>I don't output any element</ng-container>

### **Mark element attributes for translations**

HTML attributes such as title include text that should be translated along with the rest of the displayed text in the template. The following example shows an image with a title attribute:

src/app/app.component.html

content\_copy<img [src]="logo" title="Angular logo">

To mark an attribute for translation, add i18n-attribute in which attribute is the attribute to translate. The following example shows how to mark the title attribute on the img tag by adding i18n-title:

src/app/app.component.html

content\_copy<img [src]="logo" i18n-title title="Angular logo" />

You can use i18n-attribute with any attribute of any element. You also can assign a meaning, description, and custom ID with the i18n-attribute="<meaning>|<description>@@<id>" syntax.

### **Mark plurals and alternates for translation**

Different languages have different pluralization rules and grammatical constructions that can make translation difficult. To simplify translation, use International Components for Unicode (ICU) clauses with regular expressions, such as plural to mark the uses of plural numbers, and select to mark alternate text choices.

The ICU clauses adhere to the [ICU Message Format](http://userguide.icu-project.org/formatparse/messages) specified in the [CLDR pluralization rules](http://cldr.unicode.org/index/cldr-spec/plural-rules).

#### Mark plurals

Use the plural clause to mark expressions that may not be meaningful if translated word-for-word.

For example, if you want to display "updated x minutes ago" in English, you may want to display "just now", "one minute ago", or "x minutes ago" (with x as the actual number). Other languages might express this cardinality differently. The following example shows how to use a plural clause to express these three options:

src/app/app.component.html

content\_copy<span i18n>Updated {minutes, plural, =0 {just now} =1 {one minute ago} other {{{minutes}} minutes ago}}</span>

In the above example:

* The first parameter, minutes, is bound to the component property (minutes), which determines the number of minutes.
* The second parameter identifies this as a plural translation type.
* The third parameter defines a pattern of pluralization categories and their matching values:
  + For zero minutes, use =0 {just now}.
  + For one minute, use =1 {one minute}.
  + For any unmatched cardinality, use other {{{minutes}} minutes ago}. You can use HTML markup and [interpolations](https://angular.io/guide/glossary#interpolation) such as {{{minutes}} with the plural clause in expressions.
  + After the pluralization category, put the default text (English) within braces ({}).

Pluralization categories include (depending on the language):

* =0 (or any other number)
* zero
* one
* two
* few
* many
* other

LOCALES MAY NOT SUPPORT SOME PLURALIZATION CATEGORIES

Many locales don't support some of the pluralization categories. For example, the default locale (en-US) and other locales (such as es) have very simple plural() functions that don't support the few category. The following shows the [en-US](https://github.com/angular/angular/blob/ecffc3557fe1bff9718c01277498e877ca44588d/packages/core/src/i18n/locale_en.ts#L15-L18) plural() function:

content\_copyfunction plural(n: number): number {

let i = Math.floor(Math.abs(n)), v = n.toString().replace(/^[^.]\*\.?/, '').length;

if (i === 1 && v === 0) return 1;

return 5;

}

The function will only ever return 1 (one) or 5 (other). The few category will never match. If none of the pluralization categories match, Angular will try to match other. Use other as the standard fallback for a missing category.

For more information about pluralization categories, see [Choosing plural category names](http://cldr.unicode.org/index/cldr-spec/plural-rules#TOC-Choosing-Plural-Category-Names) in the CLDR - Unicode Common Locale Data Repository.

### **Mark alternates and nested expressions**

If you need to display alternate text depending on the value of a variable, you need to translate all of the alternates.

The select clause, similar to the plural clause, marks choices for alternate text based on your defined string values. For example, the following clause in the component template binds to the component's gender property, which outputs one of the following string values: "male", "female" or "other". The clause maps those values to the appropriate translations:

src/app/app.component.html

content\_copy<span i18n>The author is {gender, select, male {male} female {female} other {other}}</span>

You can also nest different clauses together, such as the plural and select clauses in the following example:

src/app/app.component.html

content\_copy<span i18n>Updated: {minutes, plural,

=0 {just now}

=1 {one minute ago}

other {{{minutes}} minutes ago by {gender, select, male {male} female {female} other {other}}}}

</span>

## **Work with translation files**

After preparing a template for translation, use the Angular CLI [extract-i18n](https://angular.io/cli/extract-i18n) command to extract the marked text in the template into a source language file. The marked text includes text marked with i18n and attributes marked with i18n-attribute as described in the previous section. Follow these steps:

1. [Extract the source language file](https://angular.io/guide/i18n#create-source). You can optionally change the location, format, and name.
2. [Create a translation file for each language](https://angular.io/guide/i18n#localization-folder) by copying the source language file.
3. [Translate each translation file](https://angular.io/guide/i18n#translate-text-nodes).
4. [Translate plurals and alternate expressions](https://angular.io/guide/i18n#translate-plural-select) separately.

### **Extract the source language file**

To extract the source language file, open a terminal window, change to the root directory of your app project, and run the following CLI command:

content\_copyng extract-i18n

The extract-i18n command creates a source language file named messages.xlf in your project's root directory using the [XML Localization Interchange File Format (XLIFF, version 1.2)](https://en.wikipedia.org/wiki/XLIFF).

Use the following [extract-i18n command options](https://angular.io/cli/extract-i18n) to change the source language file location, format, and file name:

* --output-path: Change the location.
* --format: Change the format.
* --outFile: Change the file name.

Note: The --i18n-locale option is deprecated. Angular 9 uses the source locale configured in your app's [workspace configuration](https://angular.io/guide/workspace-config) file (angular.json).

#### Change the source language file location

To create a file in the src/locale directory, specify the output path as an option, as shown in the following example:

content\_copyng extract-i18n --output-path src/locale

#### Change the source language file format

The extract-i18n command can read and write files in three translation formats:

* XLIFF 1.2 (default)
* XLIFF 2
* [XML Message Bundle (XMB)](http://cldr.unicode.org/development/development-process/design-proposals/xmb)
* JSON
* [ARB](https://github.com/google/app-resource-bundle/wiki/ApplicationResourceBundleSpecification)

Specify the translation format explicitly with the --format command option, as shown in the following examples:

content\_copyng extract-i18n --format=xlf

ng extract-i18n --format=xlf2

ng extract-i18n --format=xmb

ng extract-i18n --format=json

ng extract-i18n --format=arb

XLIFF files use the extension .xlf. The XMB format generates .xmb source language files but uses.xtb (XML Translation Bundle: XTB) translation files.

#### Change the source language file name

To change the name of the source language file generated by the extraction tool, use the --outFile command option:

content\_copyng extract-i18n --out-file source.xlf

### **Create a translation file for each language**

The ng extract-i18n command (with no options) generates a source language file named messages.xlf in the project src folder. Create translation files for each language by copying the source language file. To avoid confusion with multiple translations, you should organize the language translation files by locale in a dedicated locale folder under src/. Use a filename extension that matches the associated locale, such as messages.fr.xlf.

For example, to create a French translation file, follow these steps:

1. Make a copy of the messages.xlf source language file.
2. Put the copy in the src/locale folder.
3. Rename the copy to messages.fr.xlf for the French language (fr) translation. Send this translation file to the translator.

Repeat the above steps for each language you want to add to your app.

### **Translate each translation file**

Unless you are fluent in the language and have the time to edit translations, you would likely send each translation file to a translator, who would then use an XLIFF file editor to create and edit the translation.

To demonstrate this process, see the messages.fr.xlf file in the [live example](https://angular.io/generated/live-examples/i18n/stackblitz.html) / [download example](https://angular.io/generated/zips/i18n/i18n.zip), which includes a French translation you can edit without a special XLIFF editor or knowledge of French. Follow these steps:

1. Open messages.fr.xlf and find the first <trans-unit> element. This is a translation unit, also known as a text node, representing the translation of the <h1> greeting tag that was previously marked with the i18n attribute:

src/locale/messages.fr.xlf (<trans-unit>)

content\_copy<trans-unit id="introductionHeader" datatype="html">

<source>Hello i18n!</source>

<note priority="1" from="description">An introduction header for this sample</note>

<note priority="1" from="meaning">User welcome</note>

</trans-unit>

The id="introductionHeader" is a [custom ID](https://angular.io/guide/i18n#custom-id), but without the @@ prefix required in the source HTML.

1. Duplicate the <source>...</source> element in the text node, rename it to target, and then replace its content with the French text:

src/locale/messages.fr.xlf (<trans-unit>, after translation)

content\_copy<trans-unit id="introductionHeader" datatype="html">

<source>Hello i18n!</source>

<target>Bonjour i18n !</target>

<note priority="1" from="description">An introduction header for this sample</note>

<note priority="1" from="meaning">User welcome</note>

</trans-unit>

In a more complex translation, the information and context in the [description and meaning elements](https://angular.io/guide/i18n#help-translator) described previously would help you choose the right words for translation.

1. Translate the other text nodes the same way as shown in the following example:

src/locale/messages.fr.xlf (<trans-unit>)

content\_copy<trans-unit id="ba0cc104d3d69bf669f97b8d96a4c5d8d9559aa3" datatype="html">

<source>I don&apos;t output any element</source>

<target>Je n'affiche aucun élément</target>

</trans-unit>

<trans-unit id="701174153757adf13e7c24a248c8a873ac9f5193" datatype="html">

<source>Angular logo</source>

<target>Logo d'Angular</target>

</trans-unit>

Don't change the IDs for translation units. Each id is generated by Angular and depends on the content of the template text and its assigned meaning. If you change either the text or the meaning, then the id changes. For more about managing text updates and IDs, see the section on [custom IDs](https://angular.io/guide/i18n#custom-id).

### **Translate plurals and alternate expressions**

The [plural and select ICU expressions](https://angular.io/guide/i18n#plurals-alternates) are extracted as additional messages, so you must translate them separately.

#### Translate plurals

To translate a plural, translate its ICU format match values as shown in the following example:

* just now
* one minute ago
* <x id="INTERPOLATION" equiv-text="{{minutes}}"/> minutes ago

src/locale/messages.fr.xlf (<trans-unit>)

content\_copy<trans-unit id="5a134dee893586d02bffc9611056b9cadf9abfad" datatype="html">

<source>{VAR\_PLURAL, plural, =0 {just now} =1 {one minute ago} other {<x id="INTERPOLATION" equiv-text="{{minutes}}"/> minutes ago} }</source>

<target>{VAR\_PLURAL, plural, =0 {à l'instant} =1 {il y a une minute} other {il y a <x id="INTERPOLATION" equiv-text="{{minutes}}"/> minutes} }</target>

</trans-unit>

You can add or remove plural cases as needed for each language.

For language plural rules, see [CLDR plural rules](http://www.unicode.org/cldr/charts/latest/supplemental/language_plural_rules.html).

#### Translate alternate expressions

Angular also extracts alternate select ICU expressions as separate translation units. The following shows a select ICU expression in the component template:

src/app/app.component.html

content\_copy<span i18n>The author is {gender, select, male {male} female {female} other {other}}</span>

In this example, Angular extracts the expression into two translation units. The first contains the text outside of the select clause, and uses a placeholder for select (<x id="ICU">):

src/locale/messages.fr.xlf (<trans-unit>)

content\_copy<trans-unit id="f99f34ac9bd4606345071bd813858dec29f3b7d1" datatype="html">

<source>The author is <x id="ICU" equiv-text="{gender, select, male {...} female {...} other {...}}"/></source>

<target>L'auteur est <x id="ICU" equiv-text="{gender, select, male {...} female {...} other {...}}"/></target>

</trans-unit>

When translating the text, you can move the placeholder if necessary, but don't remove it. If you remove the placeholder, the ICU expression will not appear in your translated app.

The second translation unit contains the select clause:

src/locale/messages.fr.xlf (<trans-unit>)

content\_copy<trans-unit id="eff74b75ab7364b6fa888f1cbfae901aaaf02295" datatype="html">

<source>{VAR\_SELECT, select, male {male} female {female} other {other} }</source>

<target>{VAR\_SELECT, select, male {un homme} female {une femme} other {autre} }</target>

</trans-unit>

The following example shows both translation units after translating:

src/locale/messages.fr.xlf (<trans-unit>)

content\_copy<trans-unit id="f99f34ac9bd4606345071bd813858dec29f3b7d1" datatype="html">

<source>The author is <x id="ICU" equiv-text="{gender, select, male {...} female {...} other {...}}"/></source>

<target>L'auteur est <x id="ICU" equiv-text="{gender, select, male {...} female {...} other {...}}"/></target>

</trans-unit>

<trans-unit id="eff74b75ab7364b6fa888f1cbfae901aaaf02295" datatype="html">

<source>{VAR\_SELECT, select, male {male} female {female} other {other} }</source>

<target>{VAR\_SELECT, select, male {un homme} female {une femme} other {autre} }</target>

</trans-unit>

#### Translate nested expressions

Angular treats a nested expression in the same manner as an alternate expression, extracting it into two translation units. The first contains the text outside of the nested expression:

src/locale/messages.fr.xlf (<trans-unit>)

content\_copy<trans-unit id="972cb0cf3e442f7b1c00d7dab168ac08d6bdf20c" datatype="html">

<source>Updated: <x id="ICU" equiv-text="{minutes, plural, =0 {...} =1 {...} other {...}}"/></source>

<target>Mis à jour: <x id="ICU" equiv-text="{minutes, plural, =0 {...} =1 {...} other {...}}"/></target>

</trans-unit>

The second translation unit contains the complete nested expression:

src/locale/messages.fr.xlf (<trans-unit>)

content\_copy<trans-unit id="7151c2e67748b726f0864fc443861d45df21d706" datatype="html">

<source>{VAR\_PLURAL, plural, =0 {just now} =1 {one minute ago} other {<x id="INTERPOLATION" equiv-text="{{minutes}}"/> minutes ago by {VAR\_SELECT, select, male {male} female {female} other {other} }} }</source>

<target>{VAR\_PLURAL, plural, =0 {à l'instant} =1 {il y a une minute} other {il y a <x id="INTERPOLATION" equiv-text="{{minutes}}"/> minutes par {VAR\_SELECT, select, male {un homme} female {une femme} other {autre} }} }</target>

</trans-unit>

The following example shows both translation units after translating:

src/locale/messages.fr.xlf (<trans-unit>)

content\_copy<trans-unit id="972cb0cf3e442f7b1c00d7dab168ac08d6bdf20c" datatype="html">

<source>Updated: <x id="ICU" equiv-text="{minutes, plural, =0 {...} =1 {...} other {...}}"/></source>

<target>Mis à jour: <x id="ICU" equiv-text="{minutes, plural, =0 {...} =1 {...} other {...}}"/></target>

</trans-unit>

<trans-unit id="7151c2e67748b726f0864fc443861d45df21d706" datatype="html">

<source>{VAR\_PLURAL, plural, =0 {just now} =1 {one minute ago} other {<x id="INTERPOLATION" equiv-text="{{minutes}}"/> minutes ago by {VAR\_SELECT, select, male {male} female {female} other {other} }} }</source>

<target>{VAR\_PLURAL, plural, =0 {à l'instant} =1 {il y a une minute} other {il y a <x id="INTERPOLATION" equiv-text="{{minutes}}"/> minutes par {VAR\_SELECT, select, male {un homme} female {une femme} other {autre} }} }</target>

</trans-unit>

## **Merge translations into the app**

To merge the completed translations into the app, use the [Angular CLI](https://angular.io/guide/glossary#command-line-interface-cli) to build a copy of the app's distributable files for each locale. The build process replaces the original text with translated text, and sets the [LOCALE\_ID](https://angular.io/api/core/LOCALE_ID) token for each distributable copy of the app. It also loads and registers the locale data.

After merging, you can serve each distributable copy of the app using server-side language detection or different subdirectories, as described in the next section about [deploying multiple locales](https://angular.io/guide/i18n#deploy-locales).

The build process uses [ahead-of-time (AOT) compilation](https://angular.io/guide/glossary#ahead-of-time-aot-compilation) to produce a small, fast, ready-to-run app. With Ivy in Angular version 9, AOT is used by default for both development and production builds, and AOT is required to localize component templates.

For a detailed explanation of the build process, see [Building and serving Angular apps](https://angular.io/guide/build). This build process works for translation files in the .xlf format or in another format that Angular understands, such as .xtb.

Ivy does not support merging i18n translations when using JIT mode. If you [disable Ivy](https://angular.io/guide/ivy#opting-out-of-ivy-in-version-9) and are using JIT mode, see [merging with the JIT compiler](https://v8.angular.io/guide/i18n#merge-with-the-jit-compiler).

To build a separate distributable copy of the app for each locale, [define the locales in the build configuration](https://angular.io/guide/i18n#localize-config) in your project's workspace configuration file [angular.json](https://angular.io/guide/workspace-config). This method shortens the build process by removing the requirement to perform a full app build for each locale.

You can then [generate app versions for each locale](https://angular.io/guide/i18n#localize-generate) using the "localize" option in angular.json. You can also [build from the command line](https://angular.io/guide/i18n#localize-build-command) using the Angular CLI [build](https://angular.io/cli/build) command with the --localize option.

You can optionally [apply specific build options for just one locale](https://angular.io/guide/i18n#localize-build-one-locale) for a custom locale configuration.

### **Define locales in the build configuration**

Use the i18n project option in your app's build configuration file ([angular.json](https://angular.io/guide/workspace-config)) to define locales for a project. The following sub-options identify the source language and tell the compiler where to find supported translations for the project:

* sourceLocale: The locale you use within the app source code (en-US by default)
* locales: A map of locale identifiers to translation files

For example, the following excerpt of an angular.json file sets the source locale to en-US and provides the path to the fr (French) locale translation file:

angular.json

content\_copy...

"projects": {

"angular.io-example": {

...

"i18n": {

"sourceLocale": "en-US",

"locales": {

"fr": "src/locale/messages.fr.xlf"

}

},

"architect": {

...

}

}

### **Generate app versions for each locale**

To use your locale definition in the build configuration, use the "localize" option in angular.json to tell the CLI which locales to generate for the build configuration:

* Set "localize" to true for all the locales previously defined in the build configuration.
* Set "localize" to an array of a subset of the previously-defined locale identifiers to build only those locale versions.
* Set "localize" to false to disable localization and not generate any locale-specific versions.

Note: [Ahead-of-time (AOT) compilation](https://angular.io/guide/glossary#ahead-of-time-aot-compilation) is required to localize component templates. If you changed this setting, set "aot" to true in order to use AOT.

The following example shows the "localize" option set to true in angular.json so that all locales defined in the build configuration are built:

angular.json

content\_copy"build": {

"builder": "@angular-devkit/build-angular:browser",

"options": {

"localize": true,

"aot": true,

...

Due to the deployment complexities of i18n and the need to minimize rebuild time, the development server only supports localizing a single locale at a time. Setting the "localize" option to true will cause an error when using ng serve if more than one locale is defined. Setting the option to a specific locale, such as "localize": ["fr"], can work if you want to develop against a specific locale (such as fr).

The CLI loads and registers the locale data, places each generated version in a locale-specific directory to keep it separate from other locale versions, and puts the directories within the configured outputPath for the project. For each application variant the lang attribute of the html element is set to the locale. The CLI also adjusts the HTML base HREF for each version of the app by adding the locale to the configured baseHref.

You can set the "localize" property as a shared configuration that all the configurations effectively inherit (or can override).

### **Build from the command line**

You can also use the --localize option with the [ng build](https://angular.io/cli/build) command and your existing production configuration. The CLI builds all locales defined in the build configuration, which is similar to setting the "localize" option to true as described in the previous section.

content\_copyng build --localize

### **Apply specific build options for just one locale**

To apply specific build options to only one locale, you can create a custom locale-specific configuration by specifying a single locale as shown in the following example:

angular.json

content\_copy"build": {

...

"configurations": {

...

"fr": {

"localize": ["fr"],

"main": "src/main.fr.ts",

...

}

}

},

"serve": {

...

"configurations": {

...

"fr": {

"browserTarget": "\*project-name\*:build:fr"

}

}

}

You can then pass this configuration to the ng serve or ng build commands. The following shows how to serve the French language file created in the example for this guide:

content\_copyng serve --configuration=fr

You can use the CLI development server (ng serve), but only with a single locale.

For production builds, you can use configuration composition to execute both configurations:

content\_copyng build --configuration=production,fr

angular.json

content\_copy...

"architect": {

"build": {

"builder": "@angular-devkit/build-angular:browser",

"options": { ... },

"configurations": {

"fr": {

"localize": ["fr"],

}

}

},

...

"serve": {

"builder": "@angular-devkit/build-angular:dev-server",

"options": {

"browserTarget": "my-project:build"

},

"configurations": {

"production": {

"browserTarget": "my-project:build:production"

},

"fr": {

"browserTarget": "my-project:build:fr"

}

}

}

}

### **Report missing translations**

When a translation is missing, the build succeeds but generates a warning such as Missing translation for message "foo". You can configure the level of warning that is generated by the Angular compiler:

* error: Throw an error. If you are using AOT compilation, the build will fail. If you are using JIT compilation, the app will fail to load.
* warning (default): Show a Missing translation warning in the console or shell.
* ignore: Do nothing.

Specify the warning level in the options section for the build target of your Angular CLI configuration file (angular.json). The following example shows how to set the warning level to error:

angular.json

content\_copy"options": {

...

"i18nMissingTranslation": "error"

}

## **Deploy multiple locales**

If myapp is the directory containing your app's distributable files, you would typically make available different versions for different locales in locale directories such as myapp/fr for the French version and myapp/es for the Spanish version.

The HTML base tag with the href attribute specifies the base URI, or URL, for relative links. If you set the "localize" option in angular.json to true or to an array of locale IDs, the CLI adjusts the base href for each version of the app by adding the locale to the configured "baseHref". You can specify the "baseHref" for each locale in your workspace configuration file (angular.json), as shown in the following example, which sets "baseHref" to an empty string:

angular.json

content\_copy...

"projects": {

"angular.io-example": {

...

"i18n": {

"sourceLocale": "en-US",

"locales": {

"fr": {

"translation": "src/locale/messages.fr.xlf",

"baseHref": ""

}

}

},

"architect": {

...

}

}

You can also use the CLI --baseHref option with [ng build](https://angular.io/cli/build) to declare the base href at compile time.

### **Configuring servers**

Typical deployment of multiple languages serve each language from a different subdirectory. Users are redirected to the preferred language defined in the browser using the Accept-Language HTTP header. If the user has not defined a preferred language, or if the preferred language is not available, then the server falls back to the default language. Users can change the language by navigating to other subdirectories, which often occurs using a menu implemented in the application.

For more information on how to deploy apps to a remote server, see [Deployment](https://angular.io/guide/deployment).

#### Nginx

The following is an example of an Nginx configuration.

content\_copy[http](https://angular.io/api/common/http) {

# Browser preferred language detection (does NOT require AcceptLanguageModule)

map $http\_accept\_language $accept\_language {

~\*^de de;

~\*^fr fr;

~\*^en en;

}

# ...

}

server {

listen 80;

server\_name localhost;

root /www/data;

# Fallback to default language if no preference defined by [browser](https://angular.io/api/animations/browser)

if ($accept\_language ~ "^$") {

set $accept\_language "fr";

}

# Redirect "/" to Angular app in [browser](https://angular.io/api/animations/browser)'s preferred language

rewrite ^/$ /$accept\_language permanent;

# Everything under the Angular app is always redirected to Angular in the correct language

location ~ ^/(fr|de|en) {

try\_files $uri /$1/index.html?$args;

}

# ...

}

#### Apache

The following is an example of an Apache configuration.

content\_copy<VirtualHost \*:80>

ServerName localhost

DocumentRoot /www/data

<Directory "/www/data">

RewriteEngine on

RewriteBase /

RewriteRule ^../index\.html$ - [L]

RewriteCond %{REQUEST\_FILENAME} !-f

RewriteCond %{REQUEST\_FILENAME} !-d

RewriteRule (..) $1/index.html [L]

RewriteCond %{HTTP:Accept-Language} ^de [NC]

RewriteRule ^$ /de/ [R]

RewriteCond %{HTTP:Accept-Language} ^en [NC]

RewriteRule ^$ /en/ [R]

RewriteCond %{HTTP:Accept-Language} !^en [NC]

RewriteCond %{HTTP:Accept-Language} !^de [NC]

RewriteRule ^$ /fr/ [R]

</Directory>

</VirtualHost>

## **Explore the translated example app**

The following tabs show the example app and its translation files:

src/app/app.component.html

src/app/app.component.ts

src/app/app.module.ts

src/main.ts

src/locale/messages.fr.xlf

content\_copy<h1 i18n="User welcome|An introduction header for this sample@@introductionHeader">

Hello i18n!

</h1>

<ng-container i18n>I don't output any element</ng-container>

<br />

<img [src]="logo" i18n-title title="Angular logo" />

<br>

<button (click)="inc(1)">+</button> <button (click)="inc(-1)">-</button>

<span i18n>Updated {minutes, plural, =0 {just now} =1 {one minute ago} other {{{minutes}} minutes ago}}</span>

({{minutes}})

<br><br>

<button (click)="male()">&#9794;</button> <button (click)="female()">&#9792;</button> <button (click)="other()">&#9895;</button>

<span i18n>The author is {gender, select, male {male} female {female} other {other}}</span>

<br><br>

<span i18n>Updated: {minutes, plural,

=0 {just now}

=1 {one minute ago}

other {{{minutes}} minutes ago by {gender, select, male {male} female {female} other {other}}}}

</span>

## **Optional practices**

The following are optional practices that may be required in special cases:

* [Set the source locale manually](https://angular.io/guide/i18n#set-source-manually) by setting the [LOCALE\_ID](https://angular.io/api/core/LOCALE_ID) token.
* [Import global variants of the locale data](https://angular.io/guide/i18n#import-locale) for extra locale data.
* [Manage marked text with custom IDs](https://angular.io/guide/i18n#custom-id) if you require more control over matching translations.

### **Set the source locale manually**

Angular already contains locale data for en-US. The Angular CLI automatically includes the locale data and sets the [LOCALE\_ID](https://angular.io/api/core/LOCALE_ID) value when you use the --localize option with [ng build](https://angular.io/cli/build).

To manually set an app's source locale to one other than the automatic value, follow these steps:

1. Look up the ID for the language-locale combination in [the Angular repository](https://github.com/angular/angular/tree/master/packages/common/locales).
2. Set the [LOCALE\_ID](https://angular.io/api/core/LOCALE_ID) token. The following example sets the value of [LOCALE\_ID](https://angular.io/api/core/LOCALE_ID) to fr (French):

src/app/app.module.ts

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

import { [BrowserModule](https://angular.io/api/platform-browser/BrowserModule) } from '@angular/platform-browser';

import { AppComponent } from '../src/app/app.component';

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

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

declarations: [ AppComponent ],

providers: [ { provide: [LOCALE\_ID](https://angular.io/api/core/LOCALE_ID), useValue: 'fr' } ],

bootstrap: [ AppComponent ]

})

export class AppModule { }

### **Import global variants of the locale data**

Angular will automatically include locale data if you configure the locale using the --localize option with [ng build](https://angular.io/cli/build) CLI command.

The [Angular repository](https://github.com/angular/angular/tree/master/packages/common/locales) files (@angular/common/locales) contain most of the locale data that you need, but some advanced formatting options require additional locale data. Global variants of the locale data are available in [@angular/common/locales/global](https://github.com/angular/angular/tree/master/packages/common/locales/global). The following example imports the global variants for French (fr):

app.module.ts

content\_copyimport '@angular/common/locales/[global](https://angular.io/api/core/global)/fr';

### **Manage marked text with custom IDs**

The Angular extractor generates a file with a translation unit entry for each i18n attribute in a template. As described previously (in [How meanings control text extraction and merging](https://angular.io/guide/i18n#transaction-unit-ids)), Angular assigns each translation unit a unique ID such as the following:

messages.fr.xlf.html

content\_copy<trans-unit id="ba0cc104d3d69bf669f97b8d96a4c5d8d9559aa3" datatype="html">

When you change the translatable text, the extractor generates a new ID for that translation unit. In most cases a text change would also require a change to the translation. Therefore, using a new ID keeps the text change in sync with translations.

However, some translation systems require a specific form or syntax for the ID. To address this requirement, you can mark text with custom IDs. While most developers don't need to use custom IDs, some may want to use IDs that have a unique syntax to convey additional metadata (such as the library, component, or area of the app in which the text appears).

Specify a custom ID in the i18n attribute by using the prefix @@. The following example defines the custom ID introductionHeader:

app/app.component.html

content\_copy<h1 i18n="@@introductionHeader">Hello i18n!</h1>

When you specify a custom ID, the extractor generates a translation unit with the custom ID:

messages.fr.xlf.html

content\_copy<trans-unit id="introductionHeader" datatype="html">

If you change the text, the extractor does not change the ID. As a result, you don't have to take the extra step of updating the translation. The drawback of using custom IDs is that if you change the text, your translation may be out-of-sync with the newly changed source text.

#### Use a custom ID with a description

Use a custom ID in combination with a description and a meaning to further help the translator. The following example includes a description, followed by the custom id:

app/app.component.html

content\_copy<h1 i18n="An introduction header for this sample@@introductionHeader">Hello i18n!</h1>

The following example adds a meaning:

app/app.component.html

content\_copy<h1 i18n="site header|An introduction header for this sample@@introductionHeader">Hello i18n!</h1>

#### Define unique custom IDs

Be sure to define custom IDs that are unique. If you use the same ID for two different text elements, the extraction tool extracts only the first one, and Angular uses its translation in place of both original text elements.

For example, in the following code the same custom ID myId is defined for two different text elements:

content\_copy<h3 i18n="@@myId">Hello</h3>

<!-- ... -->

<p i18n="@@myId">Good bye</p>

The following shows the translation to French:

content\_copy<trans-unit id="myId" datatype="html">

<source>Hello</source>

<target state="new">Bonjour</target>

</trans-unit>

Both elements now use the same translation (Bonjour) because they were defined with the same custom ID:

content\_copy<h3>Bonjour</h3>

<!-- ... -->

<p>Bonjour</p>

# Animations transitions and triggers

You learned the basics of Angular animations in the [introduction](https://angular.io/guide/animations) page.

This guide goes into greater depth on special transition states such as \* (wildcard) and void, and show how these special states are used for elements entering and leaving a view. This chapter also explores multiple animation triggers, animation callbacks, and sequence-based animation using keyframes.

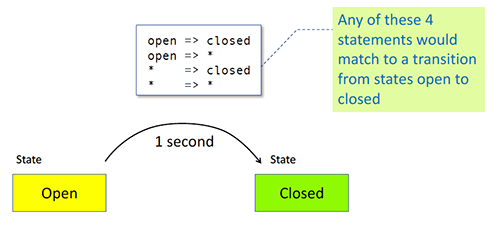
## **Predefined states and wildcard matching**

In Angular, transition states can be defined explicitly through the state() function, or using the predefined \* (wildcard) and void states.

### **Wildcard state**

An asterisk \* or wildcard matches any animation state. This is useful for defining transitions that apply regardless of the HTML element's start or end state.

For example, a transition of open => \* applies when the element's state changes from open to anything else.



The following is another code sample using the wildcard state together with the previous example using the open and closed states. Instead of defining each state-to-state transition pair, any transition to closed takes 1 second, and any transition to open takes 0.5 seconds.

This allows us to add new states without having to include separate transitions for each one.

src/app/open-close.component.ts

content\_copyanimations: [

[trigger](https://angular.io/api/animations/trigger)('openClose', [

// ...

state('open', [style](https://angular.io/api/animations/style)({

height: '200px',

opacity: 1,

backgroundColor: 'yellow'

})),

state('closed', [style](https://angular.io/api/animations/style)({

height: '100px',

opacity: 0.8,

backgroundColor: '#c6ecff'

})),

[transition](https://angular.io/api/animations/transition)('\* => closed', [

[animate](https://angular.io/api/animations/animate)('1s')

]),

[transition](https://angular.io/api/animations/transition)('\* => open', [

[animate](https://angular.io/api/animations/animate)('0.5s')

]),

]),

],

Use a double arrow syntax to specify state-to-state transitions in both directions.

src/app/open-close.component.ts

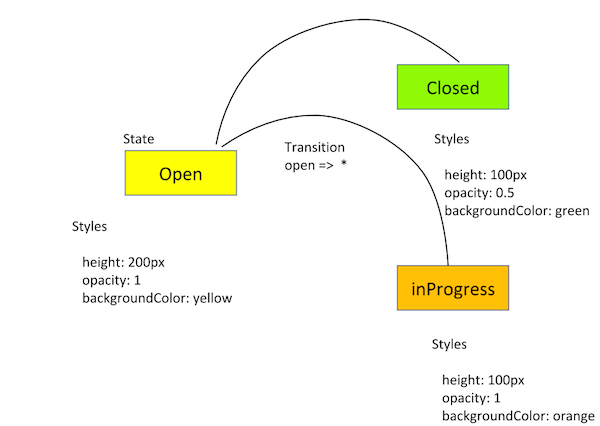
content\_copy[transition](https://angular.io/api/animations/transition)('open <=> closed', [

[animate](https://angular.io/api/animations/animate)('0.5s')

]),

### **Using wildcard state with multiple transition states**

In the two-state button example, the wildcard isn't that useful because there are only two possible states, open and closed. Wildcard states are better when an element in one particular state has multiple potential states that it can change to. If the button can change from open to either closed or something like inProgress, using a wildcard state could reduce the amount of coding needed.



src/app/open-close.component.ts

content\_copyanimations: [

[trigger](https://angular.io/api/animations/trigger)('openClose', [

// ...

state('open', [style](https://angular.io/api/animations/style)({

height: '200px',

opacity: 1,

backgroundColor: 'yellow'

})),

state('closed', [style](https://angular.io/api/animations/style)({

height: '100px',

opacity: 0.8,

backgroundColor: '#c6ecff'

})),

[transition](https://angular.io/api/animations/transition)('open => closed', [

[animate](https://angular.io/api/animations/animate)('1s')

]),

[transition](https://angular.io/api/animations/transition)('closed => open', [

[animate](https://angular.io/api/animations/animate)('0.5s')

]),

[transition](https://angular.io/api/animations/transition)('\* => closed', [

[animate](https://angular.io/api/animations/animate)('1s')

]),

[transition](https://angular.io/api/animations/transition)('\* => open', [

[animate](https://angular.io/api/animations/animate)('0.5s')

]),

[transition](https://angular.io/api/animations/transition)('open <=> closed', [

[animate](https://angular.io/api/animations/animate)('0.5s')

]),

[transition](https://angular.io/api/animations/transition) ('\* => open', [

[animate](https://angular.io/api/animations/animate) ('1s',

[style](https://angular.io/api/animations/style) ({ opacity: '\*' }),

),

]),

[transition](https://angular.io/api/animations/transition)('\* => \*', [

[animate](https://angular.io/api/animations/animate)('1s')

]),

The \* => \* transition applies when any change between two states takes place.

Transitions are matched in the order in which they are defined. Thus, you can apply other transitions on top of the \* => \* (any-to-any) transition. For example, define style changes or animations that would apply just to open => closed, or just to closed => open, and then use \* => \* as a fallback for state pairings that aren't otherwise called out.

To do this, list the more specific transitions before \* => \*.

### **Using wildcards with styles**

Use the wildcard \* with a style to tell the animation to use whatever the current style value is, and animate with that. Wildcard is a fallback value that's used if the state being animated isn't declared within the trigger.

src/app/open-close.component.ts

content\_copy[transition](https://angular.io/api/animations/transition) ('\* => open', [

[animate](https://angular.io/api/animations/animate) ('1s',

[style](https://angular.io/api/animations/style) ({ opacity: '\*' }),

),

]),

### **Void state**

You can use the void state to configure transitions for an element that is entering or leaving a page. See [Animating entering and leaving a view](https://angular.io/guide/transition-and-triggers#enter-leave-view).

### **Combining wildcard and void states**

You can combine wildcard and void states in a transition to trigger animations that enter and leave the page:

* A transition of \* => void applies when the element leaves a view, regardless of what state it was in before it left.
* A transition of void => \* applies when the element enters a view, regardless of what state it assumes when entering.
* The wildcard state \* matches to any state, including void.

## **Animating entering and leaving a view**

This section shows how to animate elements entering or leaving a page.

**Note:** For this example, an element entering or leaving a view is equivalent to being inserted or removed from the DOM.

Now add a new behavior:

* When you add a hero to the list of heroes, it appears to fly onto the page from the left.
* When you remove a hero from the list, it appears to fly out to the right.

src/app/hero-list-enter-leave.component.ts

content\_copyanimations: [

[trigger](https://angular.io/api/animations/trigger)('flyInOut', [

state('in', [style](https://angular.io/api/animations/style)({ transform: 'translateX(0)' })),

[transition](https://angular.io/api/animations/transition)('void => \*', [

[style](https://angular.io/api/animations/style)({ transform: 'translateX(-100%)' }),

[animate](https://angular.io/api/animations/animate)(100)

]),

[transition](https://angular.io/api/animations/transition)('\* => void', [

[animate](https://angular.io/api/animations/animate)(100, [style](https://angular.io/api/animations/style)({ transform: 'translateX(100%)' }))

])

])

]

In the above code, you applied the void state when the HTML element isn't attached to a view.

## **:enter and :leave aliases**

:enter and :leave are aliases for the void => \* and \* => void transitions. These aliases are used by several animation functions.

[transition](https://angular.io/api/animations/transition) ( ':enter', [ ... ] ); // alias for void => \*

[transition](https://angular.io/api/animations/transition) ( ':leave', [ ... ] ); // alias for \* => void

It's harder to target an element that is entering a view because it isn't in the DOM yet. So, use the aliases :enter and :leave to target HTML elements that are inserted or removed from a view.

### **Use of \*ngIf and \*ngFor with :enter and :leave**

The :enter transition runs when any \*[ngIf](https://angular.io/api/common/NgIf) or \*[ngFor](https://angular.io/api/common/NgForOf) views are placed on the page, and :leave runs when those views are removed from the page.

This example has a special trigger for the enter and leave animation called myInsertRemoveTrigger. The HTML template contains the following code.

src/app/insert-remove.component.html

content\_copy<div @myInsertRemoveTrigger \*[ngIf](https://angular.io/api/common/NgIf)="isShown" class="insert-remove-container">

<p>The box is inserted</p>

</div>

In the component file, the :enter transition sets an initial opacity of 0, and then animates it to change that opacity to 1 as the element is inserted into the view.

src/app/insert-remove.component.ts

content\_copy[trigger](https://angular.io/api/animations/trigger)('myInsertRemoveTrigger', [

[transition](https://angular.io/api/animations/transition)(':enter', [

[style](https://angular.io/api/animations/style)({ opacity: 0 }),

[animate](https://angular.io/api/animations/animate)('100ms', [style](https://angular.io/api/animations/style)({ opacity: 1 })),

]),

[transition](https://angular.io/api/animations/transition)(':leave', [

[animate](https://angular.io/api/animations/animate)('100ms', [style](https://angular.io/api/animations/style)({ opacity: 0 }))

])

]),

Note that this example doesn't need to use state().

## **:increment and :decrement in transitions**

The [transition](https://angular.io/api/animations/transition)() function takes additional selector values, :increment and :decrement. Use these to kick off a transition when a numeric value has increased or decreased in value.

**Note:** The following example uses [query](https://angular.io/api/animations/query)() and [stagger](https://angular.io/api/animations/stagger)() methods, which is discussed in the [complex sequences](https://angular.io/guide/complex-animation-sequences#complex-sequence) page.

src/app/hero-list-page.component.ts

content\_copy[trigger](https://angular.io/api/animations/trigger)('filterAnimation', [

[transition](https://angular.io/api/animations/transition)(':enter, \* => 0, \* => -1', []),

[transition](https://angular.io/api/animations/transition)(':increment', [

[query](https://angular.io/api/animations/query)(':enter', [

[style](https://angular.io/api/animations/style)({ opacity: 0, width: '0px' }),

[stagger](https://angular.io/api/animations/stagger)(50, [

[animate](https://angular.io/api/animations/animate)('300ms ease-out', [style](https://angular.io/api/animations/style)({ opacity: 1, width: '\*' })),

]),

], { optional: true })

]),

[transition](https://angular.io/api/animations/transition)(':decrement', [

[query](https://angular.io/api/animations/query)(':leave', [

[stagger](https://angular.io/api/animations/stagger)(50, [

[animate](https://angular.io/api/animations/animate)('300ms ease-out', [style](https://angular.io/api/animations/style)({ opacity: 0, width: '0px' })),

]),

])

]),

]),

## **Boolean values in transitions**

If a trigger contains a boolean value as a binding value, then this value can be matched using a [transition](https://angular.io/api/animations/transition)() expression that compares true and false, or 1 and 0.

src/app/open-close.component.html

content\_copy<div [@openClose]="isOpen ? true : false" class="open-close-container">

</div>

In the code snippet above, the HTML template binds a <div> element to a trigger named openClose with a status expression of isOpen, and with possible values of true and false. This is an alternative to the practice of creating two named states of open and close.

In the component code, in the @[Component](https://angular.io/api/core/Component) metadata under the animations: property, when the state evaluates to true (meaning "open" here), the associated HTML element's height is a wildcard style or default. In this case, use whatever height the element already had before the animation started. When the element is "closed," the element animates to a height of 0, which makes it invisible.

src/app/open-close.component.ts

content\_copyanimations: [

[trigger](https://angular.io/api/animations/trigger)('openClose', [

state('true', [style](https://angular.io/api/animations/style)({ height: '\*' })),

state('false', [style](https://angular.io/api/animations/style)({ height: '0px' })),

[transition](https://angular.io/api/animations/transition)('false <=> true', [animate](https://angular.io/api/animations/animate)(500))

])

],

## **Multiple animation triggers**

You can define more than one animation trigger for a component. You can attach animation triggers to different elements, and the parent-child relationships among the elements affect how and when the animations run.

### **Parent-child animations**

Each time an animation is triggered in Angular, the parent animation always get priority and child animations are blocked. In order for a child animation to run, the parent animation must query each of the elements containing child animations and then allow the animations to run using the [animateChild()](https://angular.io/api/animations/animateChild) function.

#### Disabling an animation on an HTML element

A special animation control binding called @.disabled can be placed on an HTML element to disable animations on that element, as well as any nested elements. When true, the @.disabled binding prevents all animations from rendering.

The code sample below shows how to use this feature.

src/app/open-close.component.html

src/app/open-close.component.ts

content\_copy<div [@.disabled]="isDisabled">

<div [@childAnimation]="isOpen ? 'open' : 'closed'"

class="open-close-container">

<p>The box is now {{ isOpen ? 'Open' : 'Closed' }}!</p>

</div>

</div>

When the @.disabled binding is true, the @childAnimation trigger doesn't kick off.

When an element within an HTML template has animations disabled using the @.disabled host binding, animations are disabled on all inner elements as well. You can't selectively disable multiple animations on a single element.

However, selective child animations can still be run on a disabled parent in one of the following ways:

* A parent animation can use the [query()](https://angular.io/api/animations/query) function to collect inner elements located in disabled areas of the HTML template. Those elements can still animate.
* A subanimation can be queried by a parent and then later animated with the [animateChild](https://angular.io/api/animations/animateChild)() function.

#### Disabling all animations

To disable all animations for an Angular app, place the @.disabled host binding on the topmost Angular component.

src/app/app.component.ts

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

selector: 'app-root',

templateUrl: 'app.component.html',

styleUrls: ['app.component.css'],

animations: [

slideInAnimation

// [animation](https://angular.io/api/animations/animation) triggers go here

]

})

export class AppComponent {

@[HostBinding](https://angular.io/api/core/HostBinding)('@.disabled')

public animationsDisabled = false;

}

**Note:** Disabling animations application-wide is useful during end-to-end (E2E) testing.

## **Animation callbacks**

The animation [trigger](https://angular.io/api/animations/trigger)() function emits callbacks when it starts and when it finishes. The example below features a component that contains an openClose trigger.

src/app/open-close.component.ts

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

selector: 'app-open-close',

animations: [

[trigger](https://angular.io/api/animations/trigger)('openClose', [

// ...

]),

],

templateUrl: 'open-close.component.html',

styleUrls: ['open-close.component.css']

})

export class OpenCloseComponent {

onAnimationEvent( event: [AnimationEvent](https://angular.io/api/animations/AnimationEvent) ) {

}

}

In the HTML template, the animation event is passed back via $event, as @trigger.start and @trigger.done, where [trigger](https://angular.io/api/animations/trigger) is the name of the trigger being used. In this example, the trigger openClose appears as follows.

src/app/open-close.component.html

content\_copy<div [@openClose]="isOpen ? 'open' : 'closed'"

(@openClose.start)="onAnimationEvent($event)"

(@openClose.done)="onAnimationEvent($event)"

class="open-close-container">

</div>

A potential use for animation callbacks could be to cover for a slow API call, such as a database lookup. For example, you could set up the **InProgress** button to have its own looping animation where it pulsates or does some other visual motion while the backend system operation finishes.

Then, another animation can be called when the current animation finishes. For example, the button goes from the inProgress state to the closed state when the API call is completed.

An animation can influence an end user to perceive the operation as faster, even when it isn't. Thus, a simple animation can be a cost-effective way to keep users happy, rather than seeking to improve the speed of a server call and having to compensate for circumstances beyond your control, such as an unreliable network connection.

Callbacks can serve as a debugging tool, for example in conjunction with console.warn() to view the application's progress in a browser's Developer JavaScript Console. The following code snippet creates console log output for the original example, a button with the two states of open and closed.

src/app/open-close.component.ts

content\_copyexport class OpenCloseComponent {

onAnimationEvent( event: [AnimationEvent](https://angular.io/api/animations/AnimationEvent) ) {

// openClose is [trigger](https://angular.io/api/animations/trigger) name in this example

console.warn(`Animation Trigger: ${event.triggerName}`);

// phaseName is start or done

console.warn(`Phase: ${event.phaseName}`);

// in our example, totalTime is 1000 or 1 second

console.warn(`Total time: ${event.totalTime}`);

// in our example, fromState is either open or closed

console.warn(`From: ${event.fromState}`);

// in our example, toState either open or closed

console.warn(`To: ${event.toState}`);

// the HTML element itself, the button in this case

console.warn(`Element: ${event.element}`);

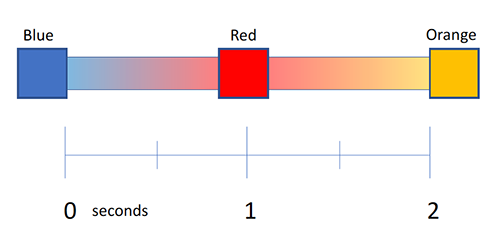
}

}

## **Keyframes**

The previous section features a simple two-state transition. Now create an animation with multiple steps run in sequence using keyframes.

Angular's keyframe() function is similar to keyframes in CSS. Keyframes allow several style changes within a single timing segment. For example, the button, instead of fading, could change color several times over a single 2-second timespan.



The code for this color change might look like this.

src/app/status-slider.component.ts

content\_copy[transition](https://angular.io/api/animations/transition)('\* => active', [

[animate](https://angular.io/api/animations/animate)('2s', [keyframes](https://angular.io/api/animations/keyframes)([

[style](https://angular.io/api/animations/style)({ backgroundColor: 'blue' }),

[style](https://angular.io/api/animations/style)({ backgroundColor: 'red' }),

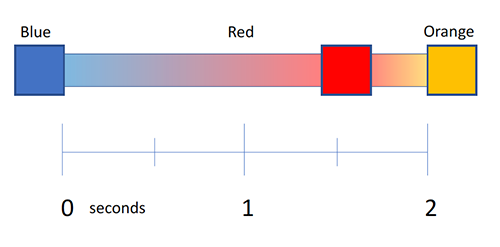
[style](https://angular.io/api/animations/style)({ backgroundColor: 'orange' })

]))

### **Offset**

Keyframes include an offset that defines the point in the animation where each style change occurs. Offsets are relative measures from zero to one, marking the beginning and end of the animation, respectively and should be applied to each of the keyframe's steps if used at least once.

Defining offsets for keyframes is optional. If you omit them, evenly spaced offsets are automatically assigned. For example, three keyframes without predefined offsets receive offsets of 0, 0.5, and 1. Specifying an offset of 0.8 for the middle transition in the above example might look like this.



The code with offsets specified would be as follows.

src/app/status-slider.component.ts

content\_copy[transition](https://angular.io/api/animations/transition)('\* => active', [

[animate](https://angular.io/api/animations/animate)('2s', [keyframes](https://angular.io/api/animations/keyframes)([

[style](https://angular.io/api/animations/style)({ backgroundColor: 'blue', offset: 0}),

[style](https://angular.io/api/animations/style)({ backgroundColor: 'red', offset: 0.8}),

[style](https://angular.io/api/animations/style)({ backgroundColor: '#754600', offset: 1.0})

])),

]),

[transition](https://angular.io/api/animations/transition)('\* => inactive', [

[animate](https://angular.io/api/animations/animate)('2s', [keyframes](https://angular.io/api/animations/keyframes)([

[style](https://angular.io/api/animations/style)({ backgroundColor: '#754600', offset: 0}),

[style](https://angular.io/api/animations/style)({ backgroundColor: 'red', offset: 0.2}),

[style](https://angular.io/api/animations/style)({ backgroundColor: 'blue', offset: 1.0})

]))

]),

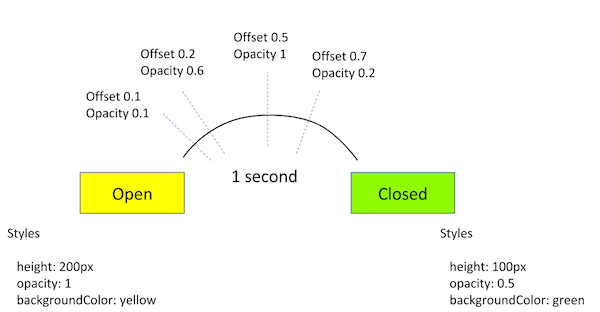
You can combine keyframes with duration, delay, and easing within a single animation.

### **Keyframes with a pulsation**

Use keyframes to create a pulse effect in your animations by defining styles at specific offset throughout the animation.

Here's an example of using keyframes to create a pulse effect:

* The original open and closed states, with the original changes in height, color, and opacity, occurring over a timeframe of 1 second.
* A keyframes sequence inserted in the middle that causes the button to appear to pulsate irregularly over the course of that same 1-second timeframe.



The code snippet for this animation might look like this.

src/app/open-close.component.ts

content\_copy[trigger](https://angular.io/api/animations/trigger)('openClose', [

state('open', [style](https://angular.io/api/animations/style)({

height: '200px',

opacity: 1,

backgroundColor: 'yellow'

})),

state('close', [style](https://angular.io/api/animations/style)({

height: '100px',

opacity: 0.5,

backgroundColor: 'green'

})),

// ...

[transition](https://angular.io/api/animations/transition)('\* => \*', [

[animate](https://angular.io/api/animations/animate)('1s', [keyframes](https://angular.io/api/animations/keyframes) ( [

[style](https://angular.io/api/animations/style)({ opacity: 0.1, offset: 0.1 }),

[style](https://angular.io/api/animations/style)({ opacity: 0.6, offset: 0.2 }),

[style](https://angular.io/api/animations/style)({ opacity: 1, offset: 0.5 }),

[style](https://angular.io/api/animations/style)({ opacity: 0.2, offset: 0.7 })

]))

])

])

### **Animatable properties and units**

Angular's animation support builds on top of web animations, so you can animate any property that the browser considers animatable. This includes positions, sizes, transforms, colors, borders, and more. The W3C maintains a list of animatable properties on its [CSS Transitions](https://www.w3.org/TR/css-transitions-1/) page.

For positional properties with a numeric value, define a unit by providing the value as a string, in quotes, with the appropriate suffix:

* 50 pixels: '50px'
* Relative font size: '3em'
* Percentage: '100%'

If you don't provide a unit when specifying dimension, Angular assumes a default unit of pixels, or px. Expressing 50 pixels as 50 is the same as saying '50px'.

### **Automatic property calculation with wildcards**

Sometimes you don't know the value of a dimensional style property until runtime. For example, elements often have widths and heights that depend on their content and the screen size. These properties are often challenging to animate using CSS.

In these cases, you can use a special wildcard \* property value under [style](https://angular.io/api/animations/style)(), so that the value of that particular style property is computed at runtime and then plugged into the animation.

The following example has a trigger called shrinkOut, used when an HTML element leaves the page. The animation takes whatever height the element has before it leaves, and animates from that height to zero.

src/app/hero-list-auto.component.ts

content\_copyanimations: [

[trigger](https://angular.io/api/animations/trigger)('shrinkOut', [

state('in', [style](https://angular.io/api/animations/style)({ height: '\*' })),

[transition](https://angular.io/api/animations/transition)('\* => void', [

[style](https://angular.io/api/animations/style)({ height: '\*' }),

[animate](https://angular.io/api/animations/animate)(250, [style](https://angular.io/api/animations/style)({ height: 0 }))

])

])

]

### **Keyframes summary**

The [keyframes](https://angular.io/api/animations/keyframes)() function in Angular allows you to specify multiple interim styles within a single transition, with an optional offset to define the point in the animation where each style change occurs.

## **More on Angular animations**

You may also be interested in the following:

* [Introduction to Angular animations](https://angular.io/guide/animations)
* [Complex animation sequences](https://angular.io/guide/complex-animation-sequences)
* [Reusable animations](https://angular.io/guide/reusable-animations)
* [Route transition animations](https://angular.io/guide/route-animations)

# Complex animation sequences

#### Prerequisites

A basic understanding of the following concepts:

* [Introduction to Angular animations](https://angular.io/guide/animations)
* [Transition and triggers](https://angular.io/guide/transition-and-triggers)

So far, we've learned simple animations of single HTML elements. Angular also lets you animate coordinated sequences, such as an entire grid or list of elements as they enter and leave a page. You can choose to run multiple animations in parallel, or run discrete animations sequentially, one following another.

Functions that control complex animation sequences are as follows:

* [query](https://angular.io/api/animations/query)() finds one or more inner HTML elements.
* [stagger](https://angular.io/api/animations/stagger)() applies a cascading delay to animations for multiple elements.
* [group()](https://angular.io/api/animations/group) runs multiple animation steps in parallel.
* [sequence](https://angular.io/api/animations/sequence)() runs animation steps one after another.

## **Animate multiple elements using query() and stagger() functions**

The [query](https://angular.io/api/animations/query)() function allows you to find inner elements within the element that is being animated. This function targets specific HTML elements within a parent component and applies animations to each element individually. Angular intelligently handles setup, teardown, and cleanup as it coordinates the elements across the page.

The [stagger](https://angular.io/api/animations/stagger)() function allows you to define a timing gap between each queried item that is animated and thus animates elements with a delay between them.

The Filter/Stagger tab in the live example shows a list of heroes with an introductory sequence. The entire list of heroes cascades in, with a slight delay from top to bottom.

The following example demonstrates how to use [query](https://angular.io/api/animations/query)() and [stagger](https://angular.io/api/animations/stagger)() functions on the entry of an animated element.

* Use [query](https://angular.io/api/animations/query)() to look for an element entering the page that meets certain criteria.
* For each of these elements, use [style](https://angular.io/api/animations/style)() to set the same initial style for the element. Make it invisible and use transform to move it out of position so that it can slide into place.
* Use [stagger](https://angular.io/api/animations/stagger)() to delay each animation by 30 milliseconds.
* Animate each element on screen for 0.5 seconds using a custom-defined easing curve, simultaneously fading it in and un-transforming it.

src/app/hero-list-page.component.ts

content\_copyanimations: [

[trigger](https://angular.io/api/animations/trigger)('pageAnimations', [

[transition](https://angular.io/api/animations/transition)(':enter', [

[query](https://angular.io/api/animations/query)('.hero, form', [

[style](https://angular.io/api/animations/style)({opacity: 0, transform: 'translateY(-100px)'}),

[stagger](https://angular.io/api/animations/stagger)(-30, [

[animate](https://angular.io/api/animations/animate)('500ms cubic-bezier(0.35, 0, 0.25, 1)', [style](https://angular.io/api/animations/style)({ opacity: 1, transform: 'none' }))

])

])

])

]),

]

})

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

@[HostBinding](https://angular.io/api/core/HostBinding)('@pageAnimations')

public animatePage = true;

heroTotal = -1;

get heroes() { return this.\_heroes; }

private \_heroes: Hero[] = [];

ngOnInit() {

this.\_heroes = HEROES;

}

updateCriteria(criteria: string) {

criteria = criteria ? criteria.trim() : '';

this.\_heroes = HEROES.filter(hero => hero.name.toLowerCase().includes(criteria.toLowerCase()));

const newTotal = this.heroes.length;

if (this.heroTotal !== newTotal) {

this.heroTotal = newTotal;

} else if (!criteria) {

this.heroTotal = -1;

}

}

}

## **Parallel animation using group() function**

You've seen how to add a delay between each successive animation. But you may also want to configure animations that happen in parallel. For example, you may want to animate two CSS properties of the same element but use a different easing function for each one. For this, you can use the animation [group()](https://angular.io/api/animations/group) function.

**Note:** The [group()](https://angular.io/api/animations/group) function is used to group animation steps, rather than animated elements.

In the following example, using groups on both :enter and :leave allow for two different timing configurations. They're applied to the same element in parallel, but run independently.

src/app/hero-list-groups.component.ts (excerpt)

content\_copyanimations: [

[trigger](https://angular.io/api/animations/trigger)('flyInOut', [

state('in', [style](https://angular.io/api/animations/style)({

width: 120,

transform: 'translateX(0)', opacity: 1

})),

[transition](https://angular.io/api/animations/transition)('void => \*', [

[style](https://angular.io/api/animations/style)({ width: 10, transform: 'translateX(50px)', opacity: 0 }),

group([

[animate](https://angular.io/api/animations/animate)('0.3s 0.1s ease', [style](https://angular.io/api/animations/style)({

transform: 'translateX(0)',

width: 120

})),

[animate](https://angular.io/api/animations/animate)('0.3s ease', [style](https://angular.io/api/animations/style)({

opacity: 1

}))

])

]),

[transition](https://angular.io/api/animations/transition)('\* => void', [

group([

[animate](https://angular.io/api/animations/animate)('0.3s ease', [style](https://angular.io/api/animations/style)({

transform: 'translateX(50px)',

width: 10

})),

[animate](https://angular.io/api/animations/animate)('0.3s 0.2s ease', [style](https://angular.io/api/animations/style)({

opacity: 0

}))

])

])

])

]

## **Sequential vs. parallel animations**

Complex animations can have many things happening at once. But what if you want to create an animation involving several animations happening one after the other? Earlier we used [group()](https://angular.io/api/animations/group) to run multiple animations all at the same time, in parallel.

A second function called [sequence](https://angular.io/api/animations/sequence)() lets you run those same animations one after the other. Within [sequence](https://angular.io/api/animations/sequence)(), the animation steps consist of either [style](https://angular.io/api/animations/style)() or [animate](https://angular.io/api/animations/animate)() function calls.

* Use [style](https://angular.io/api/animations/style)() to apply the provided styling data immediately.
* Use [animate](https://angular.io/api/animations/animate)() to apply styling data over a given time interval.

## **Filter animation example**

Let's take a look at another animation on the live example page. Under the Filter/Stagger tab, enter some text into the **Search Heroes** text box, such as Magnet or tornado.

The filter works in real time as you type. Elements leave the page as you type each new letter and the filter gets progressively stricter. The heroes list gradually re-enters the page as you delete each letter in the filter box.

The HTML template contains a trigger called filterAnimation.

src/app/hero-list-page.component.html

content\_copy<ul class="heroes" [@filterAnimation]="heroTotal">

</ul>

The component file contains three transitions.

src/app/hero-list-page.component.ts

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

animations: [

[trigger](https://angular.io/api/animations/trigger)('filterAnimation', [

[transition](https://angular.io/api/animations/transition)(':enter, \* => 0, \* => -1', []),

[transition](https://angular.io/api/animations/transition)(':increment', [

[query](https://angular.io/api/animations/query)(':enter', [

[style](https://angular.io/api/animations/style)({ opacity: 0, width: '0px' }),

[stagger](https://angular.io/api/animations/stagger)(50, [

[animate](https://angular.io/api/animations/animate)('300ms ease-out', [style](https://angular.io/api/animations/style)({ opacity: 1, width: '\*' })),

]),

], { optional: true })

]),

[transition](https://angular.io/api/animations/transition)(':decrement', [

[query](https://angular.io/api/animations/query)(':leave', [

[stagger](https://angular.io/api/animations/stagger)(50, [

[animate](https://angular.io/api/animations/animate)('300ms ease-out', [style](https://angular.io/api/animations/style)({ opacity: 0, width: '0px' })),

]),

])

]),

]),

]

})

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

heroTotal = -1;

}

The animation does the following:

* Ignores any animations that are performed when the user first opens or navigates to this page. The filter narrows what is already there, so it assumes that any HTML elements to be animated already exist in the DOM.
* Performs a filter match for matches.

For each match:

* Hides the element by making it completely transparent and infinitely narrow, by setting its opacity and width to 0.
* Animates in the element over 300 milliseconds. During the animation, the element assumes its default width and opacity.
* If there are multiple matching elements, staggers in each element starting at the top of the page, with a 50-millisecond delay between each element.

## **Animation sequence summary**

Angular functions for animating multiple elements start with [query](https://angular.io/api/animations/query)() to find inner elements, for example gathering all images within a <div>. The remaining functions, [stagger](https://angular.io/api/animations/stagger)(), [group()](https://angular.io/api/animations/group), and [sequence](https://angular.io/api/animations/sequence)(), apply cascades or allow you to control how multiple animation steps are applied.

## **More on Angular animations**

You may also be interested in the following:

* [Introduction to Angular animations](https://angular.io/guide/animations)
* [Transition and triggers](https://angular.io/guide/transition-and-triggers)
* [Reusable animations](https://angular.io/guide/reusable-animations)
* [Route transition animations](https://angular.io/guide/route-animations)

# Reusable animations

This topic provides some examples of how to create reusable animations.

## **Prerequisites**

Before continuing with this topic, you should be familiar with the following:

* [Introduction to Angular animations](https://angular.io/guide/animations)
* [Transition and triggers](https://angular.io/guide/transition-and-triggers)

## **Creating reusable animations**

To create a reusable animation, use the [animation()](https://angular.io/api/animations/animation) method to define an animation in a separate .ts file and declare this animation definition as a const export variable. You can then import and reuse this animation in any of your application components using the [useAnimation()](https://angular.io/api/animations/useAnimation) API.

src/app/animations.ts

content\_copyimport { [animation](https://angular.io/api/animations/animation), [style](https://angular.io/api/animations/style), [animate](https://angular.io/api/animations/animate), [trigger](https://angular.io/api/animations/trigger), [transition](https://angular.io/api/animations/transition), [useAnimation](https://angular.io/api/animations/useAnimation) } from '@angular/animations';

export const transitionAnimation = [animation](https://angular.io/api/animations/animation)([

[style](https://angular.io/api/animations/style)({

height: '{{ height }}',

opacity: '{{ opacity }}',

backgroundColor: '{{ backgroundColor }}'

}),

[animate](https://angular.io/api/animations/animate)('{{ time }}')

]);

In the above code snippet, transAnimation is made reusable by declaring it as an export variable.

**Note:** The height, opacity, backgroundColor, and time inputs are replaced during runtime.

You can also export a part of an animation. For example, the following snippet exports the animation [trigger](https://angular.io/api/animations/trigger).

src/app/animations.1.ts

content\_copyimport { [animation](https://angular.io/api/animations/animation), [style](https://angular.io/api/animations/style), [animate](https://angular.io/api/animations/animate), [trigger](https://angular.io/api/animations/trigger), [transition](https://angular.io/api/animations/transition), [useAnimation](https://angular.io/api/animations/useAnimation) } from '@angular/animations';

export const triggerAnimation = [trigger](https://angular.io/api/animations/trigger)('openClose', [

[transition](https://angular.io/api/animations/transition)('open => closed', [

[useAnimation](https://angular.io/api/animations/useAnimation)(transitionAnimation, {

params: {

height: 0,

opacity: 1,

backgroundColor: 'red',

time: '1s'

}

})

])

]);

From this point, you can import resuable animation variables in your component class. For example, the following code snippet imports the transAnimation variable for use in the [useAnimation](https://angular.io/api/animations/useAnimation)() method.

src/app/open-close.component.ts

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

import { [transition](https://angular.io/api/animations/transition), [trigger](https://angular.io/api/animations/trigger), [useAnimation](https://angular.io/api/animations/useAnimation) } from '@angular/animations';

import { transAnimation } from './animations';

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

selector: 'app-open-close-reusable',

animations: [

[trigger](https://angular.io/api/animations/trigger)('openClose', [

[transition](https://angular.io/api/animations/transition)('open => closed', [

[useAnimation](https://angular.io/api/animations/useAnimation)(transAnimation, {

params: {

height: 0,

opacity: 1,

backgroundColor: 'red',

time: '1s'

}

})

])

])

],

templateUrl: 'open-close.component.html',

styleUrls: ['open-close.component.css']

})

## **More on Angular animations**

You may also be interested in the following:

* [Introduction to Angular animations](https://angular.io/guide/animations)
* [Transition and triggers](https://angular.io/guide/transition-and-triggers)
* [Complex animation Sequences](https://angular.io/guide/complex-animation-sequences)
* [Route transition animations](https://angular.io/guide/route-animations)

# Route transition animations

#### Prerequisites

A basic understanding of the following concepts:

* [Introduction to Angular animations](https://angular.io/guide/animations)
* [Transition and triggers](https://angular.io/guide/transition-and-triggers)
* [Reusable animations](https://angular.io/guide/reusable-animations)

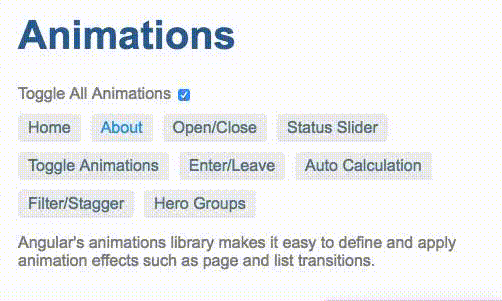
Routing enables users to navigate between different routes in an application. When a user navigates from one route to another, the Angular router maps the URL path to a relevant component and displays its view. Animating this route transition can greatly enhance the user experience.

The Angular router comes with high-level animation functions that let you animate the transitions between views when a route changes. To produce an animation sequence when switching between routes, you need to define nested animation sequences. Start with the top-level component that hosts the view, and nest additional animations in the components that host the embedded views.

To enable routing transition animation, do the following:

1. Import the routing module into the application and create a routing configuration that defines the possible routes.
2. Add a router outlet to tell the Angular router where to place the activated components in the DOM.
3. Define the animation.

Let's illustrate a router transition animation by navigating between two routes, Home and About associated with the HomeComponent and AboutComponent views respectively. Both of these component views are children of the top-most view, hosted by AppComponent. We'll implement a router transition animation that slides in the new view to the right and slides out the old view when the user navigates between the two routes.



## **Route configuration**

To begin, configure a set of routes using methods available in the [RouterModule](https://angular.io/api/router/RouterModule) class. This route configuration tells the router how to navigate.

Use the RouterModule.forRoot method to define a set of routes. Also, import this [RouterModule](https://angular.io/api/router/RouterModule) to the imports array of the main module, AppModule.

**Note:** Use the RouterModule.forRoot method in the root module, AppModule, to register top-level application routes and providers. For feature modules, call the RouterModule.forChild method to register additional routes.

The following configuration defines the possible routes for the application.

src/app/app.module.ts

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

import { [BrowserModule](https://angular.io/api/platform-browser/BrowserModule) } from '@angular/platform-browser';

import { [BrowserAnimationsModule](https://angular.io/api/platform-browser/animations/BrowserAnimationsModule) } from '@angular/platform-browser/animations';

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

import { AppComponent } from './app.component';

import { OpenCloseComponent } from './open-close.component';

import { OpenClosePageComponent } from './open-close-page.component';

import { OpenCloseChildComponent } from './open-close.component.4';

import { ToggleAnimationsPageComponent } from './toggle-animations-page.component';

import { StatusSliderComponent } from './status-slider.component';

import { StatusSliderPageComponent } from './status-slider-page.component';

import { HeroListPageComponent } from './hero-list-page.component';

import { HeroListGroupPageComponent } from './hero-list-group-page.component';

import { HeroListGroupsComponent } from './hero-list-groups.component';

import { HeroListEnterLeavePageComponent } from './hero-list-enter-leave-page.component';

import { HeroListEnterLeaveComponent } from './hero-list-enter-leave.component';

import { HeroListAutoCalcPageComponent } from './hero-list-auto-page.component';

import { HeroListAutoComponent } from './hero-list-auto.component';

import { HomeComponent } from './home.component';

import { AboutComponent } from './about.component';

import { InsertRemoveComponent } from './insert-remove.component';

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

imports: [

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

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

RouterModule.forRoot([

{ path: '', pathMatch: 'full', redirectTo: '/enter-leave' },

{ path: 'open-close', component: OpenClosePageComponent },

{ path: 'status', component: StatusSliderPageComponent },

{ path: 'toggle', component: ToggleAnimationsPageComponent },

{ path: 'heroes', component: HeroListPageComponent, data: {[animation](https://angular.io/api/animations/animation): 'FilterPage'} },

{ path: 'hero-groups', component: HeroListGroupPageComponent },

{ path: 'enter-leave', component: HeroListEnterLeavePageComponent },

{ path: 'auto', component: HeroListAutoCalcPageComponent },

{ path: 'insert-remove', component: InsertRemoveComponent},

{ path: 'home', component: HomeComponent, data: {[animation](https://angular.io/api/animations/animation): 'HomePage'} },

{ path: 'about', component: AboutComponent, data: {[animation](https://angular.io/api/animations/animation): 'AboutPage'} },

])

],

The home and about paths are associated with the HomeComponent and AboutComponent views. The route configuration tells the Angular router to instantiate the HomeComponent and AboutComponent views when the navigation matches the corresponding path.

In addition to path and component, the data property of each route defines the key animation-specific configuration associated with a route. The data property value is passed into AppComponent when the route changes. You can also pass additional data in route configuration that is consumed within the animation. The data property value has to match the transitions defined in the routeAnimation trigger, which we'll define later.

**Note:** The data property names that you use can be arbitrary. For example, the name animation used in the example above is an arbitrary choice.

## **Router outlet**

After configuring the routes, tell the Angular router where to render the views when matched with a route. You can set a router outlet by inserting a <[router-outlet](https://angular.io/api/router/RouterOutlet)> container inside the root AppComponent template.

The <[router-outlet](https://angular.io/api/router/RouterOutlet)> container has an attribute directive that contains data about active routes and their states, based on the data property that we set in the route configuration.

src/app/app.component.html

content\_copy<div [@routeAnimations]="prepareRoute(outlet)">

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

</div>

AppComponent defines a method that can detect when a view changes. The method assigns an animation state value to the animation trigger (@routeAnimation) based on the route configuration data property value. Here's an example of an AppComponent method that detects when a route change happens.

src/app/app.component.ts

content\_copyprepareRoute(outlet: [RouterOutlet](https://angular.io/api/router/RouterOutlet)) {

return outlet && outlet.activatedRouteData && outlet.activatedRouteData.animation;

}

Here, the prepareRoute() method takes the value of the outlet directive (established through #outlet="outlet") and returns a string value representing the state of the animation based on the custom data of the current active route. You can use this data to control which transition to execute for each route.

## **Animation definition**

Animations can be defined directly inside your components. For this example we are defining the animations in a separate file, which allows us to re-use the animations.

The following code snippet defines a reusable animation named slideInAnimation.

src/app/animations.ts

content\_copyexport const slideInAnimation =

[trigger](https://angular.io/api/animations/trigger)('routeAnimations', [

[transition](https://angular.io/api/animations/transition)('HomePage <=> AboutPage', [

[style](https://angular.io/api/animations/style)({ position: 'relative' }),

[query](https://angular.io/api/animations/query)(':enter, :leave', [

[style](https://angular.io/api/animations/style)({

position: 'absolute',

top: 0,

left: 0,

width: '100%'

})

]),

[query](https://angular.io/api/animations/query)(':enter', [

[style](https://angular.io/api/animations/style)({ left: '-100%' })

]),

[query](https://angular.io/api/animations/query)(':leave', [animateChild](https://angular.io/api/animations/animateChild)()),

group([

[query](https://angular.io/api/animations/query)(':leave', [

[animate](https://angular.io/api/animations/animate)('300ms ease-out', [style](https://angular.io/api/animations/style)({ left: '100%' }))

]),

[query](https://angular.io/api/animations/query)(':enter', [

[animate](https://angular.io/api/animations/animate)('300ms ease-out', [style](https://angular.io/api/animations/style)({ left: '0%' }))

])

]),

[query](https://angular.io/api/animations/query)(':enter', [animateChild](https://angular.io/api/animations/animateChild)()),

]),

[transition](https://angular.io/api/animations/transition)('\* <=> FilterPage', [

[style](https://angular.io/api/animations/style)({ position: 'relative' }),

[query](https://angular.io/api/animations/query)(':enter, :leave', [

[style](https://angular.io/api/animations/style)({

position: 'absolute',

top: 0,

left: 0,

width: '100%'

})

]),

[query](https://angular.io/api/animations/query)(':enter', [

[style](https://angular.io/api/animations/style)({ left: '-100%' })

]),

[query](https://angular.io/api/animations/query)(':leave', [animateChild](https://angular.io/api/animations/animateChild)()),

group([

[query](https://angular.io/api/animations/query)(':leave', [

[animate](https://angular.io/api/animations/animate)('200ms ease-out', [style](https://angular.io/api/animations/style)({ left: '100%' }))

]),

[query](https://angular.io/api/animations/query)(':enter', [

[animate](https://angular.io/api/animations/animate)('300ms ease-out', [style](https://angular.io/api/animations/style)({ left: '0%' }))

])

]),

[query](https://angular.io/api/animations/query)(':enter', [animateChild](https://angular.io/api/animations/animateChild)()),

])

]);

The animation definition does several things:

* Defines two transitions. A single trigger can define multiple states and transitions.
* Adjusts the styles of the host and child views to control their relative positions during the transition.
* Uses [query](https://angular.io/api/animations/query)() to determine which child view is entering and which is leaving the host view.

A route change activates the animation trigger, and a transition matching the state change is applied.

**Note:** The transition states must match the data property value defined in the route configuration.

Make the animation definition available in your application by adding the reusable animation (slideInAnimation) to the animations metadata of the AppComponent.

src/app/app.component.ts

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

selector: 'app-root',

templateUrl: 'app.component.html',

styleUrls: ['app.component.css'],

animations: [

slideInAnimation

// [animation](https://angular.io/api/animations/animation) triggers go here

]

})

### **Styling the host and child components**

During a transition, a new view is inserted directly after the old one and both elements appear on screen at the same time. To prevent this, apply additional styling to the host view, and to the removed and inserted child views. The host view must use relative positioning, and the child views must use absolute positioning. Adding styling to the views animates the containers in place, without the DOM moving things around.

src/app/animations.ts

content\_copy[trigger](https://angular.io/api/animations/trigger)('routeAnimations', [

[transition](https://angular.io/api/animations/transition)('HomePage <=> AboutPage', [

[style](https://angular.io/api/animations/style)({ position: 'relative' }),

[query](https://angular.io/api/animations/query)(':enter, :leave', [

[style](https://angular.io/api/animations/style)({

position: 'absolute',

top: 0,

left: 0,

width: '100%'

})

]),

### **Querying the view containers**

Use the [query](https://angular.io/api/animations/query)() method to find and animate elements within the current host component. The [query](https://angular.io/api/animations/query)(":enter") statement returns the view that is being inserted, and [query](https://angular.io/api/animations/query)(":leave") returns the view that is being removed.

Let's assume that we are routing from the Home => About.

src/app/animations.ts (Continuation from above)

content\_copy[query](https://angular.io/api/animations/query)(':enter', [

[style](https://angular.io/api/animations/style)({ left: '-100%' })

]),

[query](https://angular.io/api/animations/query)(':leave', [animateChild](https://angular.io/api/animations/animateChild)()),

group([

[query](https://angular.io/api/animations/query)(':leave', [

[animate](https://angular.io/api/animations/animate)('300ms ease-out', [style](https://angular.io/api/animations/style)({ left: '100%' }))

]),

[query](https://angular.io/api/animations/query)(':enter', [

[animate](https://angular.io/api/animations/animate)('300ms ease-out', [style](https://angular.io/api/animations/style)({ left: '0%' }))

])

]),

[query](https://angular.io/api/animations/query)(':enter', [animateChild](https://angular.io/api/animations/animateChild)()),

]),

[transition](https://angular.io/api/animations/transition)('\* <=> FilterPage', [

[style](https://angular.io/api/animations/style)({ position: 'relative' }),

[query](https://angular.io/api/animations/query)(':enter, :leave', [

[style](https://angular.io/api/animations/style)({

position: 'absolute',

top: 0,

left: 0,

width: '100%'

})

]),

[query](https://angular.io/api/animations/query)(':enter', [

[style](https://angular.io/api/animations/style)({ left: '-100%' })

]),

[query](https://angular.io/api/animations/query)(':leave', [animateChild](https://angular.io/api/animations/animateChild)()),

group([

[query](https://angular.io/api/animations/query)(':leave', [

[animate](https://angular.io/api/animations/animate)('200ms ease-out', [style](https://angular.io/api/animations/style)({ left: '100%' }))

]),

[query](https://angular.io/api/animations/query)(':enter', [

[animate](https://angular.io/api/animations/animate)('300ms ease-out', [style](https://angular.io/api/animations/style)({ left: '0%' }))

])

]),

[query](https://angular.io/api/animations/query)(':enter', [animateChild](https://angular.io/api/animations/animateChild)()),

])

The animation code does the following after styling the views:

* [query](https://angular.io/api/animations/query)(':enter', [style](https://angular.io/api/animations/style)({ left: '-100%' })) matches the view that is added and hides the newly added view by positioning it to the far left.
* Calls [animateChild](https://angular.io/api/animations/animateChild)() on the view that is leaving, to run its child animations.
* Uses group() function to make the inner animations run in parallel.
* Within the group() function:
  + Queries the view that is removed and animates it to slide far to the right.
  + Slides in the new view by animating the view with an easing function and duration.  
    This animation results in the about view sliding from the left to right.
* Calls the [animateChild](https://angular.io/api/animations/animateChild)() method on the new view to run its child animations after the main animation completes.

You now have a basic routable animation that animates routing from one view to another.

## **More on Angular animations**

You may also be interested in the following:

* [Introduction to Angular animations](https://angular.io/guide/animations)
* [Transition and triggers](https://angular.io/guide/transition-and-triggers)
* [Complex animation sequences](https://angular.io/guide/complex-animation-sequences)
* [Reusable animations](https://angular.io/guide/reusable-animations)

# Getting started with service workers

This document explains how to enable Angular service worker support in projects that you created with the [Angular CLI](https://angular.io/cli). It then uses an example to show you a service worker in action, demonstrating loading and basic caching.

#### Prerequisites

A basic understanding of the information in [Introduction to Angular service workers](https://angular.io/guide/service-worker-intro).

## **Adding a service worker to your project**

To set up the Angular service worker in your project, use the CLI command ng add @angular/pwa. It takes care of configuring your application to use service workers by adding the service-worker package along with setting up the necessary support files.

content\_copyng add @angular/pwa --project \*project-name\*

The above command completes the following actions:

1. Adds the @angular/service-worker package to your project.
2. Enables service worker build support in the CLI.
3. Imports and registers the service worker in the application module.
4. Updates the index.html file:
   * Includes a link to add the manifest.webmanifest file.
   * Adds meta tags for theme-color.
5. Installs icon files to support the installed Progressive Web App (PWA).
6. Creates the service worker configuration file called [ngsw-config.json](https://angular.io/guide/service-worker-config), which specifies the caching behaviors and other settings.

Now, build the project:

content\_copyng build

The CLI project is now set up to use the Angular service worker.

## **Service worker in action: a tour**

This section demonstrates a service worker in action, using an example application.

### **Serving with http-server**

Because ng serve does not work with service workers, you must use a separate HTTP server to test your project locally. You can use any HTTP server. The example below uses the [http-server](https://www.npmjs.com/package/http-server) package from npm. To reduce the possibility of conflicts and avoid serving stale content, test on a dedicated port and disable caching.

To serve the directory containing your web files with http-server, run the following command:

content\_copyhttp-server -p 8080 -c-1 dist/<project-name>

### **Initial load**

With the server running, you can point your browser at http://localhost:8080/. Your application should load normally.

**Tip:** When testing Angular service workers, it's a good idea to use an incognito or private window in your browser to ensure the service worker doesn't end up reading from a previous leftover state, which can cause unexpected behavior.

**Note:** If you are not using HTTPS, the service worker will only be registered when accessing the application on localhost.

### **Simulating a network issue**

To simulate a network issue, disable network interaction for your application. In Chrome:

1. Select **Tools** > **Developer Tools** (from the Chrome menu located at the top right corner).
2. Go to the **Network tab**.
3. Check the **Offline box**.

The offline checkbox in the Network tab is checked

Now the application has no access to network interaction.

For applications that do not use the Angular service worker, refreshing now would display Chrome's Internet disconnected page that says "There is no Internet connection".

With the addition of an Angular service worker, the application behavior changes. On a refresh, the page loads normally.

If you look at the Network tab, you can verify that the service worker is active.



Notice that under the "Size" column, the requests state is (from ServiceWorker). This means that the resources are not being loaded from the network. Instead, they are being loaded from the service worker's cache.

### **What's being cached?**

Notice that all of the files the browser needs to render this application are cached. The ngsw-config.json boilerplate configuration is set up to cache the specific resources used by the CLI:

* index.html.
* favicon.ico.
* Build artifacts (JS and CSS bundles).
* Anything under assets.
* Images and fonts directly under the configured outputPath (by default ./dist/<project-name>/) or resourcesOutputPath. See [ng build](https://angular.io/cli/build) for more information about these options.

Pay attention to two key points:

1. The generated ngsw-config.json includes a limited list of cacheable fonts and images extensions. In some cases, you might want to modify the glob pattern to suit your needs.
2. If resourcesOutputPath or assets paths are modified after the generation of configuration file, you need to change the paths manually in ngsw-config.json.

### **Making changes to your application**

Now that you've seen how service workers cache your application, the next step is understanding how updates work. Let's make a change to the application, and watch the service worker install the update:

1. If you're testing in an incognito window, open a second blank tab. This will keep the incognito and the cache state alive during your test.
2. Close the application tab, but not the window. This should also close the Developer Tools.
3. Shut down http-server.
4. Open src/app/app.component.html for editing.
5. Change the text Welcome to {{title}}! to Bienvenue à {{title}}!.
6. Build and run the server again:

content\_copyng build

http-server -p 8080 -c-1 dist/<project-name>

### **Updating your application in the browser**

Now look at how the browser and service worker handle the updated application.

1. Open http://localhost:8080 again in the same window. What happens?



What went wrong? Nothing, actually. The Angular service worker is doing its job and serving the version of the application that it has **installed**, even though there is an update available. In the interest of speed, the service worker doesn't wait to check for updates before it serves the application that it has cached.

If you look at the http-server logs, you can see the service worker requesting /ngsw.json. This is how the service worker checks for updates.

1. Refresh the page.



The service worker installed the updated version of your application in the background, and the next time the page is loaded or reloaded, the service worker switches to the latest version.

## **More on Angular service workers**

You may also be interested in the following:

* [Communicating with service workers](https://angular.io/guide/service-worker-communications).

# App shell

Application shell is a way to render a portion of your application using a route at build time. It can improve the user experience by quickly launching a static rendered page (a skeleton common to all pages) while the browser downloads the full client version and switches to it automatically after the code loads.

This gives users a meaningful first paint of your application that appears quickly because the browser can render the HTML and CSS without the need to initialize any JavaScript.

Learn more in [The App Shell Model](https://developers.google.com/web/fundamentals/architecture/app-shell).

## **Step 1: Prepare the application**

You can do this with the following CLI command:

content\_copyng new my-app --routing

For an existing application, you have to manually add the [RouterModule](https://angular.io/api/router/RouterModule) and defining a <[router-outlet](https://angular.io/api/router/RouterOutlet)> within your application.

## **Step 2: Create the app shell**

Use the CLI to automatically create the application shell.

content\_copyng generate app-shell

For more information about this command see [App shell command](https://angular.io/cli/generate#app-shell-command).

After running this command you will notice that the angular.json configuration file has been updated to add two new targets, with a few other changes.

content\_copy"server": {

"builder": "@angular-devkit/build-angular:server",

"defaultConfiguration": "production",

"options": {

"outputPath": "dist/my-app/server",

"main": "src/main.server.ts",

"tsConfig": "tsconfig.server.json"

},

"configurations": {

"development": {

"outputHashing": "none",

},

"production": {

"outputHashing": "media",

"fileReplacements": [

{

"replace": "src/environments/environment.ts",

"with": "src/environments/environment.prod.ts"

}

],

"sourceMap": false,

"optimization": true

}

}

},

"app-shell": {

"builder": "@angular-devkit/build-angular:app-shell",

"defaultConfiguration": "production",

"options": {

"route": "shell"

},

"configurations": {

"development": {

"browserTarget": "my-app:build:development",

"serverTarget": "my-app:server:development",

},

"production": {

"browserTarget": "my-app:build:production",

"serverTarget": "my-app:server:production"

}

}

}

## **Step 3: Verify the app is built with the shell content**

Use the CLI to build the app-shell target.

content\_copyng run my-app:app-shell:development

Or to use the production configuration.

content\_copyng run my-app:app-shell:production

To verify the build output, open dist/my-app/[browser](https://angular.io/api/animations/browser)/index.html. Look for default text app-shell works! to show that the application shell route was rendered as part of the output.

# Service worker communication

Importing [ServiceWorkerModule](https://angular.io/api/service-worker/ServiceWorkerModule) into your AppModule doesn't just register the service worker, it also provides a few services you can use to interact with the service worker and control the caching of your application.

#### Prerequisites

A basic understanding of the following:

* [Getting Started with Service Workers](https://angular.io/guide/service-worker-getting-started).

## [**SwUpdate**](https://angular.io/api/service-worker/SwUpdate)**service**

The [SwUpdate](https://angular.io/api/service-worker/SwUpdate) service gives you access to events that indicate when the service worker has discovered an available update for your application or when it has activated such an update—meaning it is now serving content from that update to your application.

The [SwUpdate](https://angular.io/api/service-worker/SwUpdate) service supports four separate operations:

* Getting notified of available updates. These are new versions of the application to be loaded if the page is refreshed.
* Getting notified of update activation. This is when the service worker starts serving a new version of the application immediately.
* Asking the service worker to check the server for new updates.
* Asking the service worker to activate the latest version of the application for the current tab.

### **Available and activated updates**

The two update events, available and activated, are Observable properties of [SwUpdate](https://angular.io/api/service-worker/SwUpdate):

log-update.service.ts

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

export class LogUpdateService {

constructor(updates: [SwUpdate](https://angular.io/api/service-worker/SwUpdate)) {

updates.available.subscribe(event => {

console.log('current version is', event.current);

console.log('available version is', event.available);

});

updates.activated.subscribe(event => {

console.log('old version was', event.previous);

console.log('new version is', event.current);

});

}

}

You can use these events to notify the user of a pending update or to refresh their pages when the code they are running is out of date.

### **Checking for updates**

It's possible to ask the service worker to check if any updates have been deployed to the server. The service worker checks for updates during initialization and on each navigation request—that is, when the user navigates from a different address to your application. However, you might choose to manually check for updates if you have a site that changes frequently or want updates to happen on a schedule.

Do this with the checkForUpdate() method:

check-for-update.service.ts

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

import { [SwUpdate](https://angular.io/api/service-worker/SwUpdate) } from '@angular/service-worker';

import { concat, interval } from 'rxjs';

import { first } from 'rxjs/operators';

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

export class CheckForUpdateService {

constructor(appRef: [ApplicationRef](https://angular.io/api/core/ApplicationRef), updates: [SwUpdate](https://angular.io/api/service-worker/SwUpdate)) {

// Allow the app to stabilize first, before starting polling for updates with `interval()`.

const appIsStable$ = appRef.isStable.pipe(first(isStable => isStable === true));

const everySixHours$ = interval(6 \* 60 \* 60 \* 1000);

const everySixHoursOnceAppIsStable$ = concat(appIsStable$, everySixHours$);

everySixHoursOnceAppIsStable$.subscribe(() => updates.checkForUpdate());

}

}

This method returns a Promise which indicates that the update check has completed successfully, though it does not indicate whether an update was discovered as a result of the check. Even if one is found, the service worker must still successfully download the changed files, which can fail. If successful, the available event will indicate availability of a new version of the application.

In order to avoid negatively affecting the initial rendering of the page, [ServiceWorkerModule](https://angular.io/api/service-worker/ServiceWorkerModule) waits for up to 30 seconds by default for the application to stabilize, before registering the ServiceWorker script. Constantly polling for updates, for example, with [setInterval()](https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/setInterval) or RxJS' [interval()](https://rxjs.dev/api/index/function/interval), will prevent the application from stabilizing and the ServiceWorker script will not be registered with the browser until the 30 seconds upper limit is reached.

Note that this is true for any kind of polling done by your application. Check the [isStable](https://angular.io/api/core/ApplicationRef#isStable) documentation for more information.

You can avoid that delay by waiting for the application to stabilize first, before starting to poll for updates, as shown in the example above. Alternatively, you might want to define a different [registration strategy](https://angular.io/api/service-worker/SwRegistrationOptions#registrationStrategy) for the ServiceWorker.

### **Forcing update activation**

If the current tab needs to be updated to the latest application version immediately, it can ask to do so with the activateUpdate() method:

prompt-update.service.ts

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

export class PromptUpdateService {

constructor(updates: [SwUpdate](https://angular.io/api/service-worker/SwUpdate)) {

updates.available.subscribe(event => {

if (promptUser(event)) {

updates.activateUpdate().then(() => document.location.reload());

}

});

}

}

Calling activateUpdate() without reloading the page could break lazy-loading in a currently running app, especially if the lazy-loaded chunks use filenames with hashes, which change every version. Therefore, it is recommended to reload the page once the promise returned by activateUpdate() is resolved.

### **Handling an unrecoverable state**

In some cases, the version of the application used by the service worker to serve a client might be in a broken state that cannot be recovered from without a full page reload.

For example, imagine the following scenario:

* A user opens the application for the first time and the service worker caches the latest version of the application. Let's assume the application's cached assets include index.html, main.<main-hash-1>.js and lazy-chunk.<lazy-hash-1>.js.
* The user closes the application and does not open it for a while.
* After some time, a new version of the application is deployed to the server. This newer version includes the files index.html, main.<main-hash-2>.js and lazy-chunk.<lazy-hash-2>.js (note that the hashes are different now, because the content of the files has changed). The old version is no longer available on the server.
* In the meantime, the user's browser decides to evict lazy-chunk.<lazy-hash-1>.js from its cache. Browsers may decide to evict specific (or all) resources from a cache in order to reclaim disk space.
* The user opens the application again. The service worker serves the latest version known to it at this point, namely the old version (index.html and main.<main-hash-1>.js).
* At some later point, the application requests the lazy bundle, lazy-chunk.<lazy-hash-1>.js.
* The service worker is unable to find the asset in the cache (remember that the browser evicted it). Nor is it able to retrieve it from the server (since the server now only has lazy-chunk.<lazy-hash-2>.js from the newer version).

In the above scenario, the service worker is not able to serve an asset that would normally be cached. That particular application version is broken and there is no way to fix the state of the client without reloading the page. In such cases, the service worker notifies the client by sending an [UnrecoverableStateEvent](https://angular.io/api/service-worker/UnrecoverableStateEvent) event. You can subscribe to [SwUpdate](https://angular.io/api/service-worker/SwUpdate)#unrecoverable to be notified and handle these errors.

handle-unrecoverable-state.service.ts

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

export class HandleUnrecoverableStateService {

constructor(updates: [SwUpdate](https://angular.io/api/service-worker/SwUpdate)) {

updates.unrecoverable.subscribe(event => {

notifyUser(

`An error occurred that we cannot recover from:\n${event.reason}\n\n` +

'Please reload the page.');

});

}

}

## **More on Angular service workers**

You may also be interested in the following:

* [Service Worker Notifications](https://angular.io/guide/service-worker-notifications).

# Service worker notifications

Push notifications are a compelling way to engage users. Through the power of service workers, notifications can be delivered to a device even when your application is not in focus.

The Angular service worker enables the display of push notifications and the handling of notification click events.

When using the Angular service worker, push notification interactions are handled using the [SwPush](https://angular.io/api/service-worker/SwPush) service. To learn more about the native APIs involved see [Push API](https://developer.mozilla.org/en-US/docs/Web/API/Push_API) and [Using the Notifications API](https://developer.mozilla.org/en-US/docs/Web/API/Notifications_API/Using_the_Notifications_API).

#### Prerequisites

We recommend you have a basic understanding of the following:

* [Getting Started with Service Workers](https://angular.io/guide/service-worker-getting-started).

## **Notification payload**

Invoke push notifications by pushing a message with a valid payload. See [SwPush](https://angular.io/api/service-worker/SwPush) for guidance.

In Chrome, you can test push notifications without a backend. Open Devtools -> Application -> Service Workers and use the Push input to send a JSON notification payload.

## **Notification click handling**

The default behavior for the notificationclick event is to close the notification and notify [SwPush.notificationClicks](https://angular.io/api/service-worker/SwPush#notificationClicks).

You can specify an additional operation to be executed on notificationclick by adding an onActionClick property to the data object, and providing a default entry. This is especially useful for when there are no open clients when a notification is clicked.

content\_copy{

"notification": {

"title": "New Notification!",

"data": {

"onActionClick": {

"default": {"operation": "openWindow", "url": "foo"}

}

}

}

}

### **Operations**

The Angular service worker supports the following operations:

* openWindow: Opens a new tab at the specified URL, which is resolved relative to the service worker scope.
* focusLastFocusedOrOpen: Focuses the last focused client. If there is no client open, then it opens a new tab at the specified URL, which is resolved relative to the service worker scope.
* navigateLastFocusedOrOpen: Focuses the last focused client and navigates it to the specified URL, which is resolved relative to the service worker scope. If there is no client open, then it opens a new tab at the specified URL.

If an onActionClick item does not define a url, then the service worker's registration scope is used.

### **Actions**

Actions offer a way to customize how the user can interact with a notification.

Using the actions property, you can define a set of available actions. Each action is represented as an action button that the user can click to interact with the notification.

In addition, using the onActionClick property on the data object, you can tie each action to an operation to be performed when the corresponding action button is clicked:

content\_copy{

"notification": {

"title": "New Notification!",

"actions": [

{"action": "foo", "title": "Open new tab"},

{"action": "bar", "title": "Focus last"},

{"action": "baz", "title": "Navigate last"},

{"action": "qux", "title": "Just notify existing clients"}

],

"data": {

"onActionClick": {

"default": {"operation": "openWindow"},

"foo": {"operation": "openWindow", "url": "/absolute/path"},

"bar": {"operation": "focusLastFocusedOrOpen", "url": "relative/path"},

"baz": {"operation": "navigateLastFocusedOrOpen", "url": "https://other.domain.com/"}

}

}

}

}

If an action does not have a corresponding onActionClick entry, then the notification is closed and [SwPush.notificationClicks](https://angular.io/api/service-worker/SwPush#notificationClicks) is notified on existing clients.

## **More on Angular service workers**

You may also be interested in the following:

* [Service Worker in Production](https://angular.io/guide/service-worker-devops).

# Service worker in production[link](https://angular.io/guide/service-worker-devops#service-worker-in-production)

This page is a reference for deploying and supporting production applications that use the Angular service worker. It explains how the Angular service worker fits into the larger production environment, the service worker's behavior under various conditions, and available resources and fail-safes.

#### Prerequisites

A basic understanding of the following:

* [Service Worker Communication](https://angular.io/guide/service-worker-communications).

## **Service worker and caching of app resources**

Conceptually, you can imagine the Angular service worker as a forward cache or a CDN edge that is installed in the end user's web browser. The service worker's job is to satisfy requests made by the Angular application for resources or data from a local cache, without needing to wait for the network. Like any cache, it has rules for how content is expired and updated.

### **App versions**

In the context of an Angular service worker, a "version" is a collection of resources that represent a specific build of the Angular application. Whenever a new build of the application is deployed, the service worker treats that build as a new version of the application. This is true even if only a single file is updated. At any given time, the service worker may have multiple versions of the application in its cache and it may be serving them simultaneously. For more information, see the [App tabs](https://angular.io/guide/service-worker-devops#tabs) section below.

To preserve application integrity, the Angular service worker groups all files into a version together. The files grouped into a version usually include HTML, JS, and CSS files. Grouping of these files is essential for integrity because HTML, JS, and CSS files frequently refer to each other and depend on specific content. For example, an index.html file might have a <script> tag that references bundle.js and it might attempt to call a function startApp() from within that script. Any time this version of index.html is served, the corresponding bundle.js must be served with it. For example, assume that the startApp() function is renamed to runApp() in both files. In this scenario, it is not valid to serve the old index.html, which calls startApp(), along with the new bundle, which defines runApp().

This file integrity is especially important when lazy loading modules. A JS bundle may reference many lazy chunks, and the filenames of the lazy chunks are unique to the particular build of the application. If a running application at version X attempts to load a lazy chunk, but the server has updated to version X + 1 already, the lazy loading operation will fail.

The version identifier of the application is determined by the contents of all resources, and it changes if any of them change. In practice, the version is determined by the contents of the ngsw.json file, which includes hashes for all known content. If any of the cached files change, the file's hash will change in ngsw.json, causing the Angular service worker to treat the active set of files as a new version.

With the versioning behavior of the Angular service worker, an application server can ensure that the Angular application always has a consistent set of files.

#### Update checks

Every time the user opens or refreshes the application, the Angular service worker checks for updates to the application by looking for updates to the ngsw.json manifest. If an update is found, it is downloaded and cached automatically, and will be served the next time the application is loaded.

### **Resource integrity**

One of the potential side effects of long caching is inadvertently caching an invalid resource. In a normal HTTP cache, a hard refresh or cache expiration limits the negative effects of caching an invalid file. A service worker ignores such constraints and effectively long caches the entire application. Consequently, it is essential that the service worker gets the correct content.

To ensure resource integrity, the Angular service worker validates the hashes of all resources for which it has a hash. Typically for an application created with the [Angular CLI](https://angular.io/cli), this is everything in the dist directory covered by the user's src/ngsw-config.json configuration.

If a particular file fails validation, the Angular service worker attempts to re-fetch the content using a "cache-busting" URL parameter to eliminate the effects of browser or intermediate caching. If that content also fails validation, the service worker considers the entire version of the application to be invalid and it stops serving the application. If necessary, the service worker enters a safe mode where requests fall back on the network, opting not to use its cache if the risk of serving invalid, broken, or outdated content is high.

Hash mismatches can occur for a variety of reasons:

* Caching layers in between the origin server and the end user could serve stale content.
* A non-atomic deployment could result in the Angular service worker having visibility of partially updated content.
* Errors during the build process could result in updated resources without ngsw.json being updated. The reverse could also happen resulting in an updated ngsw.json without updated resources.

#### Unhashed content

The only resources that have hashes in the ngsw.json manifest are resources that were present in the dist directory at the time the manifest was built. Other resources, especially those loaded from CDNs, have content that is unknown at build time or are updated more frequently than the application is deployed.

If the Angular service worker does not have a hash to validate a given resource, it still caches its contents but it honors the HTTP caching headers by using a policy of "stale while revalidate." That is, when HTTP caching headers for a cached resource indicate that the resource has expired, the Angular service worker continues to serve the content and it attempts to refresh the resource in the background. This way, broken unhashed resources do not remain in the cache beyond their configured lifetimes.

### **App tabs**

It can be problematic for an application if the version of resources it's receiving changes suddenly or without warning. See the [Versions](https://angular.io/guide/service-worker-devops#versions) section above for a description of such issues.

The Angular service worker provides a guarantee: a running application will continue to run the same version of the application. If another instance of the application is opened in a new web browser tab, then the most current version of the app is served. As a result, that new tab can be running a different version of the application than the original tab.

It's important to note that this guarantee is **stronger** than that provided by the normal web deployment model. Without a service worker, there is no guarantee that code lazily loaded later in a running application is from the same version as the initial code for the application.

There are a few limited reasons why the Angular service worker might change the version of a running application. Some of them are error conditions:

* The current version becomes invalid due to a failed hash.
* An unrelated error causes the service worker to enter safe mode; that is, temporary deactivation.

The Angular service worker is aware of which versions are in use at any given moment and it cleans up versions when no tab is using them.

Other reasons the Angular service worker might change the version of a running application are normal events:

* The page is reloaded/refreshed.
* The page requests an update be immediately activated using the [SwUpdate](https://angular.io/api/service-worker/SwUpdate) service.

### **Service worker updates**

The Angular service worker is a small script that runs in web browsers. From time to time, the service worker will be updated with bug fixes and feature improvements.

The Angular service worker is downloaded when the application is first opened and when the application is accessed after a period of inactivity. If the service worker has changed, the service worker will be updated in the background.

Most updates to the Angular service worker are transparent to the app—the old caches are still valid and content is still served normally. However, occasionally a bugfix or feature in the Angular service worker requires the invalidation of old caches. In this case, the application will be refreshed transparently from the network.

### **Bypassing the service worker**

In some cases, you may want to bypass the service worker entirely and let the browser handle the request instead. An example is when you rely on a feature that is currently not supported in service workers (for example, [reporting progress on uploaded files](https://github.com/w3c/ServiceWorker/issues/1141)).

To bypass the service worker you can set ngsw-bypass as a request header, or as a query parameter. (The value of the header or query parameter is ignored and can be empty or omitted.)

## **Debugging the Angular service worker**

Occasionally, it may be necessary to examine the Angular service worker in a running state to investigate issues or to ensure that it is operating as designed. Browsers provide built-in tools for debugging service workers and the Angular service worker itself includes useful debugging features.

### **Locating and analyzing debugging information**

The Angular service worker exposes debugging information under the ngsw/ virtual directory. Currently, the single exposed URL is ngsw/state. Here is an example of this debug page's contents:

content\_copyNGSW Debug Info:

Driver version: 13.3.7

Driver state: NORMAL ((nominal))

Latest manifest hash: eea7f5f464f90789b621170af5a569d6be077e5c

Last update check: never

=== [Version](https://angular.io/api/core/Version) eea7f5f464f90789b621170af5a569d6be077e5c ===

Clients: 7b79a015-69af-4d3d-9ae6-95ba90c79486, 5bc08295-aaf2-42f3-a4cc-9e4ef9100f65

=== Idle Task Queue ===

Last update [tick](https://angular.io/api/core/testing/tick): 1s496u

Last update run: never

Task queue:

\* init post-load (update, cleanup)

Debug log:

#### Driver state

The first line indicates the driver state:

content\_copyDriver state: NORMAL ((nominal))

NORMAL indicates that the service worker is operating normally and is not in a degraded state.

There are two possible degraded states:

* EXISTING\_CLIENTS\_ONLY: the service worker does not have a clean copy of the latest known version of the application. Older cached versions are safe to use, so existing tabs continue to run from cache, but new loads of the application will be served from the network. The service worker will try to recover from this state when a new version of the application is detected and installed (that is, when a new ngsw.json is available).
* SAFE\_MODE: the service worker cannot guarantee the safety of using cached data. Either an unexpected error occurred or all cached versions are invalid. All traffic will be served from the network, running as little service worker code as possible.

In both cases, the parenthetical annotation provides the error that caused the service worker to enter the degraded state.

Both states are temporary; they are saved only for the lifetime of the [ServiceWorker instance](https://developer.mozilla.org/en-US/docs/Web/API/ServiceWorkerGlobalScope). The browser sometimes terminates an idle service worker to conserve memory and processor power, and creates a new service worker instance in response to network events. The new instance starts in the NORMAL mode, regardless of the state of the previous instance.

#### Latest manifest hash

content\_copyLatest manifest hash: eea7f5f464f90789b621170af5a569d6be077e5c

This is the SHA1 hash of the most up-to-date version of the application that the service worker knows about.

#### Last update check

content\_copyLast update check: never

This indicates the last time the service worker checked for a new version, or update, of the application. never indicates that the service worker has never checked for an update.

In this example debug file, the update check is currently scheduled, as explained the next section.

#### Version

content\_copy=== [Version](https://angular.io/api/core/Version) eea7f5f464f90789b621170af5a569d6be077e5c ===

Clients: 7b79a015-69af-4d3d-9ae6-95ba90c79486, 5bc08295-aaf2-42f3-a4cc-9e4ef9100f65

In this example, the service worker has one version of the application cached and being used to serve two different tabs. Note that this version hash is the "latest manifest hash" listed above. Both clients are on the latest version. Each client is listed by its ID from the Clients API in the browser.

#### Idle task queue

content\_copy=== Idle Task Queue ===

Last update [tick](https://angular.io/api/core/testing/tick): 1s496u

Last update run: never

Task queue:

\* init post-load (update, cleanup)

The Idle Task Queue is the queue of all pending tasks that happen in the background in the service worker. If there are any tasks in the queue, they are listed with a description. In this example, the service worker has one such task scheduled, a post-initialization operation involving an update check and cleanup of stale caches.

The last update tick/run counters give the time since specific events happened related to the idle queue. The "Last update run" counter shows the last time idle tasks were actually executed. "Last update tick" shows the time since the last event after which the queue might be processed.

#### Debug log

content\_copyDebug log:

Errors that occur within the service worker will be logged here.

### **Developer Tools**

Browsers such as Chrome provide developer tools for interacting with service workers. Such tools can be powerful when used properly, but there are a few things to keep in mind.

* When using developer tools, the service worker is kept running in the background and never restarts. This can cause behavior with Dev Tools open to differ from behavior a user might experience.
* If you look in the Cache Storage viewer, the cache is frequently out of date. Right click the Cache Storage title and refresh the caches.

Stopping and starting the service worker in the Service Worker pane triggers a check for updates.

## **Service Worker Safety**

Like any complex system, bugs or broken configurations can cause the Angular service worker to act in unforeseen ways. While its design attempts to minimize the impact of such problems, the Angular service worker contains several failsafe mechanisms in case an administrator ever needs to deactivate the service worker quickly.

### **Fail-safe**

To deactivate the service worker, remove or rename the ngsw.json file. When the service worker's request for ngsw.json returns a 404, then the service worker removes all of its caches and de-registers itself, essentially self-destructing.

### **Safety Worker**

Also included in the @angular/service-worker NPM package is a small script safety-worker.js, which when loaded will unregister itself from the browser. This script can be used as a last resort to get rid of unwanted service workers already installed on client pages.

It's important to note that you cannot register this worker directly, as old clients with cached state may not see a new index.html which installs the different worker script. Instead, you must serve the contents of safety-worker.js at the URL of the Service Worker script you are trying to unregister, and must continue to do so until you are certain all users have successfully unregistered the old worker. For most sites, this means that you should serve the safety worker at the old Service Worker URL forever.

This script can be used both to deactivate @angular/service-worker as well as any other Service Workers which might have been served in the past on your site.

### **Changing your app's location**

It is important to note that service workers don't work behind redirect. You may have already encountered the error The script resource is behind a redirect, which is disallowed.

This can be a problem if you have to change your application's location. If you setup a redirect from the old location (for example example.com) to the new location (for example www.example.com) the worker will stop working. Also, the redirect won't even trigger for users who are loading the site entirely from Service Worker. The old worker (registered at example.com) tries to update and sends requests to the old location example.com which get redirected to the new location www.example.com and create the error The script resource is behind a redirect, which is disallowed.

To remedy this, you may need to kill the old worker using one of the above techniques ([Fail-safe](https://angular.io/guide/service-worker-devops#fail-safe) or [Safety Worker](https://angular.io/guide/service-worker-devops#safety-worker)).

## **More on Angular service workers**

You may also be interested in the following:

* [Service Worker Configuration](https://angular.io/guide/service-worker-config).

# Service worker configuration

#### Prerequisites

A basic understanding of the following:

* [Service Worker in Production](https://angular.io/guide/service-worker-devops).

The ngsw-config.json configuration file specifies which files and data URLs the Angular service worker should cache and how it should update the cached files and data. The [Angular CLI](https://angular.io/cli) processes the configuration file during ng build. Manually, you can process it with the ngsw-config tool (where <project-name> is the name of the project being built):

content\_copy./node\_modules/.bin/ngsw-config ./dist/<project-name> ./ngsw-config.json [/base/href]

The configuration file uses the JSON format. All file paths must begin with /, which corresponds to the deployment directory—usually dist/<project-name> in CLI projects.

Unless otherwise noted, patterns use a limited glob format:

* \*\* matches 0 or more path segments.
* \* matches 0 or more characters excluding /.
* ? matches exactly one character excluding /.
* The ! prefix marks the pattern as being negative, meaning that only files that don't match the pattern will be included.

Example patterns:

* /\*\*/\*.html specifies all HTML files.
* /\*.html specifies only HTML files in the root.
* !/\*\*/\*.map exclude all sourcemaps.

Each section of the configuration file is described below.

## **appData**

This section enables you to pass any data you want that describes this particular version of the application. The [SwUpdate](https://angular.io/api/service-worker/SwUpdate) service includes that data in the update notifications. Many applications use this section to provide additional information for the display of UI popups, notifying users of the available update.

## **index**

Specifies the file that serves as the index page to satisfy navigation requests. Usually this is /index.html.

## **assetGroups**

Assets are resources that are part of the application version that update along with the application. They can include resources loaded from the page's origin as well as third-party resources loaded from CDNs and other external URLs. As not all such external URLs may be known at build time, URL patterns can be matched.

This field contains an array of asset groups, each of which defines a set of asset resources and the policy by which they are cached.

content\_copy{

"assetGroups": [

{

...

},

{

...

}

]

}

When the ServiceWorker handles a request, it checks asset groups in the order in which they appear in ngsw-config.json. The first asset group that matches the requested resource handles the request.

It is recommended that you put the more specific asset groups higher in the list. For example, an asset group that matches /foo.js should appear before one that matches \*.js.

Each asset group specifies both a group of resources and a policy that governs them. This policy determines when the resources are fetched and what happens when changes are detected.

Asset groups follow the Typescript interface shown here:

content\_copyinterface AssetGroup {

name: string;

installMode?: 'prefetch' | 'lazy';

updateMode?: 'prefetch' | 'lazy';

resources: {

files?: string[];

urls?: string[];

};

cacheQueryOptions?: {

ignoreSearch?: boolean;

};

}

### **name**

A name is mandatory. It identifies this particular group of assets between versions of the configuration.

### **installMode**

The installMode determines how these resources are initially cached. The installMode can be either of two values:

* prefetch tells the Angular service worker to fetch every single listed resource while it's caching the current version of the application. This is bandwidth-intensive but ensures resources are available whenever they're requested, even if the browser is currently offline.
* lazy does not cache any of the resources up front. Instead, the Angular service worker only caches resources for which it receives requests. This is an on-demand caching mode. Resources that are never requested will not be cached. This is useful for things like images at different resolutions, so the service worker only caches the correct assets for the particular screen and orientation.

Defaults to prefetch.

### **updateMode**

For resources already in the cache, the updateMode determines the caching behavior when a new version of the application is discovered. Any resources in the group that have changed since the previous version are updated in accordance with updateMode.

* prefetch tells the service worker to download and cache the changed resources immediately.
* lazy tells the service worker to not cache those resources. Instead, it treats them as unrequested and waits until they're requested again before updating them. An updateMode of lazy is only valid if the installMode is also lazy.

Defaults to the value installMode is set to.

### **resources**

This section describes the resources to cache, broken up into the following groups:

* files lists patterns that match files in the distribution directory. These can be single files or glob-like patterns that match a number of files.
* urls includes both URLs and URL patterns that will be matched at runtime. These resources are not fetched directly and do not have content hashes, but they will be cached according to their HTTP headers. This is most useful for CDNs such as the Google Fonts service.  
  (Negative glob patterns are not supported and *?* will be matched literally; that is, it will not match any character other than *?*.)

### **cacheQueryOptions**

These options are used to modify the matching behavior of requests. They are passed to the browsers Cache#match function. See [MDN](https://developer.mozilla.org/en-US/docs/Web/API/Cache/match) for details. Currently, only the following options are supported:

* ignoreSearch: Ignore query parameters. Defaults to false.

## **dataGroups**

Unlike asset resources, data requests are not versioned along with the application. They're cached according to manually-configured policies that are more useful for situations such as API requests and other data dependencies.

This field contains an array of data groups, each of which defines a set of data resources and the policy by which they are cached.

content\_copy{

"dataGroups": [

{

...

},

{

...

}

]

}

When the ServiceWorker handles a request, it checks data groups in the order in which they appear in ngsw-config.json. The first data group that matches the requested resource handles the request.

It is recommended that you put the more specific data groups higher in the list. For example, a data group that matches /api/foo.json should appear before one that matches /api/\*.json.

Data groups follow this Typescript interface:

content\_copyexport interface DataGroup {

name: string;

urls: string[];

version?: number;

cacheConfig: {

maxSize: number;

maxAge: string;

timeout?: string;

strategy?: 'freshness' | 'performance';

};

cacheQueryOptions?: {

ignoreSearch?: boolean;

};

}

### **name**

Similar to assetGroups, every data group has a name which uniquely identifies it.

### **urls**

A list of URL patterns. URLs that match these patterns are cached according to this data group's policy. Only non-mutating requests (GET and HEAD) are cached.

* Negative glob patterns are not supported.
* ? is matched literally; that is, it matches only the character ?.

### **version**

Occasionally APIs change formats in a way that is not backward-compatible. A new version of the application may not be compatible with the old API format and thus may not be compatible with existing cached resources from that API.

version provides a mechanism to indicate that the resources being cached have been updated in a backwards-incompatible way, and that the old cache entries—those from previous versions—should be discarded.

version is an integer field and defaults to 1.

### **cacheConfig**

This section defines the policy by which matching requests will be cached.

#### maxSize

(required) The maximum number of entries, or responses, in the cache. Open-ended caches can grow in unbounded ways and eventually exceed storage quotas, calling for eviction.

#### maxAge

(required) The maxAge parameter indicates how long responses are allowed to remain in the cache before being considered invalid and evicted. maxAge is a duration string, using the following unit suffixes:

* d: days
* h: hours
* m: minutes
* s: seconds
* u: milliseconds

For example, the string 3d12h will cache content for up to three and a half days.

#### timeout

This duration string specifies the network timeout. The network timeout is how long the Angular service worker will wait for the network to respond before using a cached response, if configured to do so. timeout is a duration string, using the following unit suffixes:

* d: days
* h: hours
* m: minutes
* s: seconds
* u: milliseconds

For example, the string 5s30u will translate to five seconds and 30 milliseconds of network timeout.

#### strategy

The Angular service worker can use either of two caching strategies for data resources.

* performance, the default, optimizes for responses that are as fast as possible. If a resource exists in the cache, the cached version is used, and no network request is made. This allows for some staleness, depending on the maxAge, in exchange for better performance. This is suitable for resources that don't change often; for example, user avatar images.
* freshness optimizes for currency of data, preferentially fetching requested data from the network. Only if the network times out, according to timeout, does the request fall back to the cache. This is useful for resources that change frequently; for example, account balances.

You can also emulate a third strategy, [staleWhileRevalidate](https://developers.google.com/web/fundamentals/instant-and-offline/offline-cookbook/#stale-while-revalidate), which returns cached data (if available), but also fetches fresh data from the network in the background for next time. To use this strategy set strategy to freshness and timeout to 0u in cacheConfig.

This will essentially do the following:

1. Try to fetch from the network first.
2. If the network request does not complete after 0ms (that is, immediately), fall back to the cache (ignoring cache age).
3. Once the network request completes, update the cache for future requests.
4. If the resource does not exist in the cache, wait for the network request anyway.

### **cacheQueryOptions**

See [assetGroups](https://angular.io/guide/service-worker-config#assetgroups) for details.

## **navigationUrls**

This optional section enables you to specify a custom list of URLs that will be redirected to the index file.

### **Handling navigation requests**

The ServiceWorker will redirect navigation requests that don't match any asset or data group to the specified [index file](https://angular.io/guide/service-worker-config#index-file). A request is considered to be a navigation request if:

1. Its [mode](https://developer.mozilla.org/en-US/docs/Web/API/Request/mode) is navigation.
2. It accepts a text/html response (as determined by the value of the Accept header).
3. Its URL matches certain criteria (see below).

By default, these criteria are:

1. The URL must not contain a file extension (that is, a .) in the last path segment.
2. The URL must not contain \_\_.

To configure whether navigation requests are sent through to the network or not, see the [navigationRequestStrategy](https://angular.io/guide/service-worker-config#navigation-request-strategy) section.

### **Matching navigation request URLs**

While these default criteria are fine in most cases, it is sometimes desirable to configure different rules. For example, you may want to ignore specific routes (that are not part of the Angular app) and pass them through to the server.

This field contains an array of URLs and [glob-like](https://angular.io/guide/service-worker-config#glob-patterns) URL patterns that will be matched at runtime. It can contain both negative patterns (that is, patterns starting with !) and non-negative patterns and URLs.

Only requests whose URLs match any of the non-negative URLs/patterns and none of the negative ones will be considered navigation requests. The URL query will be ignored when matching.

If the field is omitted, it defaults to:

content\_copy[

'/\*\*', // Include all URLs.

'!/\*\*/\*.\*', // Exclude URLs to files.

'!/\*\*/\*\_\_\*', // Exclude URLs containing `\_\_` in the last segment.

'!/\*\*/\*\_\_\*/\*\*', // Exclude URLs containing `\_\_` in any other segment.

]

## **navigationRequestStrategy**

This optional property enables you to configure how the service worker handles navigation requests:

content\_copy{

"navigationRequestStrategy": "freshness"

}

Possible values:

* 'performance': The default setting. Serves the specified [index file](https://angular.io/guide/service-worker-config#index-file), which is typically cached.
* 'freshness': Passes the requests through to the network and falls back to the performance behavior when offline. This value is useful when the server redirects the navigation requests elsewhere using an HTTP redirect (3xx status code). Reasons for using this value include:
  + Redirecting to an authentication website when authentication is not handled by the application.
  + Redirecting specific URLs to avoid breaking existing links/bookmarks after a website redesign.
  + Redirecting to a different website, such as a server-status page, while a page is temporarily down.

The freshness strategy usually results in more requests sent to the server, which can increase response latency. It is recommended that you use the default performance strategy whenever possible.

# Background processing using web workers

[Web workers](https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API) allow you to run CPU-intensive computations in a background thread, freeing the main thread to update the user interface. If you find your application performs a lot of computations, such as generating CAD drawings or doing heavy geometrical calculations, using web workers can help increase your application's performance.

The CLI does not support running Angular itself in a web worker.

## **Adding a web worker**

To add a web worker to an existing project, use the Angular CLI ng generate command.

content\_copyng generate web-worker <location>

You can add a web worker anywhere in your application. For example, to add a web worker to the root component, src/app/app.component.ts, run the following command.

content\_copyng generate web-worker app

The command performs the following actions.

* Configures your project to use web workers, if it isn't already.
* Adds the following scaffold code to src/app/app.worker.ts to receive messages.

src/app/app.worker.ts

content\_copyaddEventListener('message', ({ data }) => {

const response = `worker response to ${data}`;

postMessage(response);

});

* Adds the following scaffold code to src/app/app.component.ts to use the worker.

src/app/app.component.ts

content\_copyif (typeof Worker !== 'undefined') {

// Create a new

const worker = new Worker(new URL('./app.worker', import.meta.url));

worker.onmessage = ({ data }) => {

console.log(`page got message: ${data}`);

};

worker.postMessage('hello');

} else {

// Web workers are not supported in this environment.

// You should add a fallback so that your program still executes correctly.

}

After you generate this initial scaffold, you must refactor your code to use the web worker by sending messages to and from the worker.

Some environments or platforms, such as @angular/platform-server used in [Server-side Rendering](https://angular.io/guide/universal), don't support web workers. To ensure that your application will work in these environments, you must provide a fallback mechanism to perform the computations that the worker would otherwise perform.

# Server-side rendering (SSR) with Angular Universal

This guide describes **Angular Universal**, a technology that renders Angular applications on the server.

A normal Angular application executes in the browser, rendering pages in the DOM in response to user actions. Angular Universal executes on the server, generating static application pages that later get bootstrapped on the client. This means that the application generally renders more quickly, giving users a chance to view the application layout before it becomes fully interactive.

For a more detailed look at different techniques and concepts surrounding SSR, check out this [article](https://developers.google.com/web/updates/2019/02/rendering-on-the-web).

You can easily prepare an application for server-side rendering using the [Angular CLI](https://angular.io/guide/glossary#cli). The CLI schematic @nguniversal/express-engine performs the required steps, as described below.

Angular Universal requires an [active LTS or maintenance LTS](https://nodejs.org/about/releases) version of Node.js. See the engines property in the [package.json](https://unpkg.com/browse/@angular/platform-server/package.json) file to learn about the currently supported versions.

**Note:** [Download the finished sample code](https://angular.io/generated/zips/universal/universal.zip), which runs in a [Node.js® Express](https://expressjs.com/) server.

## **Universal tutorial**

The [Tour of Heroes tutorial](https://angular.io/tutorial) is the foundation for this walkthrough.

In this example, the Angular CLI compiles and bundles the Universal version of the application with the [Ahead-of-Time (AOT) compiler](https://angular.io/guide/aot-compiler). A Node.js Express web server compiles HTML pages with Universal based on client requests.

To create the server-side application module, app.server.module.ts, run the following CLI command.

content\_copyng add @nguniversal/express-engine

The command creates the following folder structure.

content\_copysrc/

index.html *app web page*

main.ts *bootstrapper for client app*

main.server.ts *\* bootstrapper for server app*

style.css *styles for the app*

app/ ... *application code*

app.server.module.ts *\* server-side application module*

server.ts *\* express web server*

tsconfig.json *TypeScript base configuration*

tsconfig.app.json *TypeScript* [*browser*](https://angular.io/api/animations/browser) *application configuration*

tsconfig.server.json *TypeScript server application configuration*

tsconfig.spec.json *TypeScript tests configuration*

The files marked with \* are new and not in the original tutorial sample.

### **Universal in action**

To start rendering your application with Universal on your local system, use the following command.

content\_copynpm run dev:ssr

Open a browser and navigate to http://localhost:4200/. You should see the familiar Tour of Heroes dashboard page.

Navigation using routerLinks works correctly because they use the native anchor (<a>) tags. You can go from the Dashboard to the Heroes page and back. You can click a hero on the Dashboard page to display its Details page.

If you throttle your network speed so that the client-side scripts take longer to download (instructions below), you'll notice:

* You can't add or delete a hero.
* The search box on the Dashboard page is ignored.
* The Back and Save buttons on the Details page don't work.

User events other than [routerLink](https://angular.io/api/router/RouterLink) clicks aren't supported. You must wait for the full client application to bootstrap and run, or buffer the events using libraries like [preboot](https://github.com/angular/preboot), which allow you to replay these events once the client-side scripts load.

The transition from the server-rendered application to the client application happens quickly on a development machine, but you should always test your applications in real-world scenarios.

You can simulate a slower network to see the transition more clearly as follows:

1. Open the Chrome Dev Tools and go to the Network tab.
2. Find the [Network Throttling](https://developers.google.com/web/tools/chrome-devtools/network-performance/reference#throttling) dropdown on the far right of the menu bar.
3. Try one of the "3G" speeds.

The server-rendered application still launches quickly but the full client application may take seconds to load.

## **Why use server-side rendering?**

There are three main reasons to create a Universal version of your application.

1. Facilitate web crawlers through [search engine optimization (SEO)](https://static.googleusercontent.com/media/www.google.com/en/webmasters/docs/search-engine-optimization-starter-guide.pdf)
2. Improve performance on mobile and low-powered devices
3. Show the first page quickly with a [first-contentful paint (FCP)](https://developers.google.com/web/tools/lighthouse/audits/first-contentful-paint)

### **Facilitate web crawlers (SEO)**

Google, Bing, Facebook, Twitter, and other social media sites rely on web crawlers to index your application content and make that content searchable on the web. These web crawlers may be unable to navigate and index your highly interactive Angular application as a human user could do.

Angular Universal can generate a static version of your application that is easily searchable, linkable, and navigable without JavaScript. Universal also makes a site preview available since each URL returns a fully rendered page.

### **Improve performance on mobile and low-powered devices**

Some devices don't support JavaScript or execute JavaScript so poorly that the user experience is unacceptable. For these cases, you may require a server-rendered, no-JavaScript version of the application. This version, however limited, may be the only practical alternative for people who otherwise couldn't use the application at all.

### **Show the first page quickly**

Displaying the first page quickly can be critical for user engagement. Pages that load faster perform better, [even with changes as small as 100ms](https://web.dev/shopping-for-speed-on-ebay/). Your application may have to launch faster to engage these users before they decide to do something else.

With Angular Universal, you can generate landing pages for the application that look like the complete application. The pages are pure HTML, and can display even if JavaScript is disabled. The pages don't handle browser events, but they do support navigation through the site using [routerLink](https://angular.io/guide/router-reference#router-link).

In practice, you'll serve a static version of the landing page to hold the user's attention. At the same time, you'll load the full Angular application behind it. The user perceives near-instant performance from the landing page and gets the full interactive experience after the full application loads.

## **Universal web servers**

A Universal web server responds to application page requests with static HTML rendered by the [Universal template engine](https://angular.io/guide/universal#universal-engine). The server receives and responds to HTTP requests from clients (usually browsers), and serves static assets such as scripts, CSS, and images. It may respond to data requests, either directly or as a proxy to a separate data server.

The sample web server for this guide is based on the popular [Express](https://expressjs.com/) framework.

**Note:** Any web server technology can serve a Universal application as long as it can call Universal's [renderModule](https://angular.io/api/platform-server/renderModule)() function. The principles and decision points discussed here apply to any web server technology.

Universal applications use the Angular platform-server package (as opposed to platform-browser), which provides server implementations of the DOM, XMLHttpRequest, and other low-level features that don't rely on a browser.

The server ([Node.js Express](https://expressjs.com/) in this guide's example) passes client requests for application pages to the NgUniversal ngExpressEngine. Under the hood, this calls Universal's [renderModule](https://angular.io/api/platform-server/renderModule)() function, while providing caching and other helpful utilities.

The [renderModule](https://angular.io/api/platform-server/renderModule)() function takes as inputs a template HTML page (usually index.html), an Angular module containing components, and a route that determines which components to display. The route comes from the client's request to the server.

Each request results in the appropriate view for the requested route. The [renderModule](https://angular.io/api/platform-server/renderModule)() function renders the view within the <app> tag of the template, creating a finished HTML page for the client.

Finally, the server returns the rendered page to the client.

### **Working around the browser APIs**

Because a Universal application doesn't execute in the browser, some of the browser APIs and capabilities may be missing on the server.

For example, server-side applications can't reference browser-only global objects such as window, document, navigator, or location.

Angular provides some injectable abstractions over these objects, such as [Location](https://angular.io/api/common/Location) or [DOCUMENT](https://angular.io/api/common/DOCUMENT); it may substitute adequately for these APIs. If Angular doesn't provide it, it's possible to write new abstractions that delegate to the browser APIs while in the browser and to an alternative implementation while on the server (aka shimming).

Similarly, without mouse or keyboard events, a server-side application can't rely on a user clicking a button to show a component. The application must determine what to render based solely on the incoming client request. This is a good argument for making the application [routable](https://angular.io/guide/router).

### **Universal template engine**

The important bit in the server.ts file is the ngExpressEngine() function.

server.ts

content\_copy// Our Universal express-engine (found @ https://github.com/angular/universal/tree/master/modules/express-engine)

server.engine('html', ngExpressEngine({

bootstrap: AppServerModule,

}));

The ngExpressEngine() function is a wrapper around Universal's [renderModule](https://angular.io/api/platform-server/renderModule)() function which turns a client's requests into server-rendered HTML pages. It accepts an object with the following properties:

* bootstrap: The root [NgModule](https://angular.io/api/core/NgModule) or [NgModule](https://angular.io/api/core/NgModule) factory to use for bootstraping the application when rendering on the server. For the example app, it is AppServerModule. It's the bridge between the Universal server-side renderer and the Angular application.
* extraProviders: This is optional and lets you specify dependency providers that apply only when rendering the application on the server. You can do this when your application needs information that can only be determined by the currently running server instance.

The ngExpressEngine() function returns a Promise callback that resolves to the rendered page. It's up to the engine to decide what to do with that page. This engine's Promise callback returns the rendered page to the web server, which then forwards it to the client in the HTTP response.

**Note:** These wrappers help hide the complexity of the [renderModule](https://angular.io/api/platform-server/renderModule)() function. There are more wrappers for different backend technologies at the [Universal repository](https://github.com/angular/universal).

### **Filtering request URLs**

NOTE: The basic behavior described below is handled automatically when using the NgUniversal Express schematic. This is helpful when trying to understand the underlying behavior or replicate it without using the schematic.

The web server must distinguish app page requests from other kinds of requests.

It's not as simple as intercepting a request to the root address /. The browser could ask for one of the application routes such as /dashboard, /heroes, or /detail:12. In fact, if the application were only rendered by the server, every application link clicked would arrive at the server as a navigation URL intended for the router.

Fortunately, application routes have something in common: their URLs lack file extensions. (Data requests also lack extensions but they're easy to recognize because they always begin with /api.) All static asset requests have a file extension (such as main.js or /node\_modules/zone.js/bundles/zone.umd.js).

Because we use routing, we can easily recognize the three types of requests and handle them differently.

1. **Data request**: request URL that begins /api.
2. **App navigation**: request URL with no file extension.
3. **Static asset**: all other requests.

A Node.js Express server is a pipeline of middleware that filters and processes requests one after the other. You configure the Node.js Express server pipeline with calls to server.get() like this one for data requests.

server.ts (data URL)

content\_copy// TODO: implement data requests securely

server.get('/api/\*\*', (req, res) => {

res.status(404).send('data requests are not yet supported');

});

**Note:** This sample server doesn't handle data requests.

The tutorial's "in-memory web API" module, a demo and development tool, intercepts all HTTP calls and simulates the behavior of a remote data server. In practice, you would remove that module and register your web API middleware on the server here.

The following code filters for request URLs with no extensions and treats them as navigation requests.

server.ts (navigation)

content\_copy// All regular routes use the Universal engine

server.get('\*', (req, res) => {

res.render(indexHtml, { req, providers: [{ provide: [APP\_BASE\_HREF](https://angular.io/api/common/APP_BASE_HREF), useValue: req.baseUrl }] });

});

### **Serving static files safely**

A single server.use() treats all other URLs as requests for static assets such as JavaScript, image, and style files.

To ensure that clients can only download the files that they are permitted to see, put all client-facing asset files in the /dist folder and only honor requests for files from the /dist folder.

The following Node.js Express code routes all remaining requests to /dist, and returns a 404 - NOT FOUND error if the file isn't found.

server.ts (static files)

content\_copy// Serve [static](https://angular.io/api/upgrade/static) files from /[browser](https://angular.io/api/animations/browser)

server.get('\*.\*', express.static(distFolder, {

maxAge: '1y'

}));

### **Using absolute URLs for HTTP (data) requests on the server**

The tutorial's HeroService and HeroSearchService delegate to the Angular [HttpClient](https://angular.io/api/common/http/HttpClient) module to fetch application data. These services send requests to relative URLs such as api/heroes. In a server-side rendered app, HTTP URLs must be absolute (for example, https://my-server.com/api/heroes). This means that the URLs must be somehow converted to absolute when running on the server and be left relative when running in the browser.

If you are using one of the @nguniversal/\*-engine packages (such as @nguniversal/express-engine), this is taken care for you automatically. You don't need to do anything to make relative URLs work on the server.

If, for some reason, you are not using an @nguniversal/\*-engine package, you may need to handle it yourself.

The recommended solution is to pass the full request URL to the options argument of [renderModule()](https://angular.io/api/platform-server/renderModule) or [renderModuleFactory()](https://angular.io/api/platform-server/renderModuleFactory) (depending on what you use to render AppServerModule on the server). This option is the least intrusive as it does not require any changes to the application. Here, "request URL" refers to the URL of the request as a response to which the application is being rendered on the server. For example, if the client requested https://my-server.com/dashboard and you are rendering the application on the server to respond to that request, options.url should be set to https://my-server.com/dashboard.

Now, on every HTTP request made as part of rendering the application on the server, Angular can correctly resolve the request URL to an absolute URL, using the provided options.url.

### **Useful scripts**

* npm run dev:ssr

This command is similar to [ng serve](https://angular.io/cli/serve), which offers live reload during development, but uses server-side rendering. The application will run in watch mode and refresh the browser after every change. This command is slower than the actual ng serve command.

* ng build && ng run app-name:server

This command builds both the server script and the application in production mode. Use this command when you want to build the project for deployment.

* npm run serve:ssr

This command starts the server script for serving the application locally with server-side rendering. It uses the build artifacts created by ng run build:ssr, so make sure you have run that command as well.

Note that serve:ssr is not intended to be used to serve your application in production, but only for testing the server-side rendered application locally.

* npm run prerender

This script can be used to prerender an application's pages. You can read more about prerendering [here](https://angular.io/guide/prerendering).

# Prerendering static pages

Angular Universal allows you to prerender the pages of your application. Prerendering is the process where a dynamic page is processed at build time generating static HTML.

## **How to prerender a page**

To prerender a static page make sure to add SSR capabilities to your application. For more information see the [universal guide](https://angular.io/guide/universal). Once SSR is added, run the following command:

content\_copynpm run prerender

### **Build options for prerendering**

When you add prerendering to your application, the following build options are available:

* browserTarget: Specify the target to build.
* serverTarget: Specify the Server target to use for prerendering the application.
* routes: Define an array of additional routes to prerender.
* guessRoutes: Whether builder should extract routes and guess which paths to render. Defaults to true.
* routesFile: Specify a file that contains a list of all routes to prerender, separated by newlines. This option is useful if you have a large number of routes.
* numProcesses: Specify the number of CPUs to be used while running the prerendering command.

### **Prerendering dynamic routes**

You can prerender dynamic routes. An example of a dynamic route is product/:id, where id is dynamically provided.

To prerender dynamic routes, choose one from the following options:

* Provide additional routes in the command line
* Provide routes using a file
* Prerender specific routes

#### Provide additional routes in the command line

While running the prerender command, you can provide additional routes. For example:

content\_copyng run <app-name>:prerender --routes /product/1 /product/2

#### Providing additonal routes using a file

You can provide routes using a file to generate static pages. This method is useful if you have a large number of routes to generate, such as product details for an e-commerce application, which may come from an external source (Database or CMS).

To provide routes using a file, use the --routes-file option with the name of a .txt file containing the routes.

For example, you could generate this file by using a script to extract IDs from a database and save them to a routes.txt file:

routes.txt

content\_copy/products/1

/products/555

When your .txt file is ready, run the following command to prerender the static files with dynamic values:

content\_copyng run <app-name>:prerender --routes-file routes.txt

#### Prerendering specific routes

You can also pass specific routes to the prerender command. If you choose this option, make sure to disable the guessRoutes option.

content\_copyng run <app-name>:prerender --no-guess-routes --routes /product/1 /product/1

# Security

This topic describes Angular's built-in protections against common web-application vulnerabilities and attacks such as cross-site scripting attacks. It doesn't cover application-level security, such as authentication and authorization.

For more information about the attacks and mitigations described below, see [OWASP Guide Project](https://www.owasp.org/index.php/Category:OWASP_Guide_Project).

You can run the [live example](https://angular.io/generated/live-examples/security/stackblitz.html) / [download example](https://angular.io/generated/zips/security/security.zip) in Stackblitz and download the code from there.

REPORTING VULNERABILITIES

To report vulnerabilities in Angular itself, email us at [security@angular.io](mailto:security@angular.io).

For more information about how Google handles security issues, see [Google's security philosophy](https://www.google.com/about/appsecurity/).

BEST PRACTICES

* **Keep current with the latest Angular library releases.** We regularly update the Angular libraries, and these updates may fix security defects discovered in previous versions. Check the Angular [change log](https://github.com/angular/angular/blob/master/CHANGELOG.md) for security-related updates.
* **Don't modify your copy of Angular.** Private, customized versions of Angular tend to fall behind the current version and may not include important security fixes and enhancements. Instead, share your Angular improvements with the community and make a pull request.
* **Avoid Angular APIs marked in the documentation as “**Security Risk**.”** For more information, see the [Trusting safe values](https://angular.io/guide/security#bypass-security-apis) section of this page.

## **Preventing cross-site scripting (XSS)**

[Cross-site scripting (XSS)](https://en.wikipedia.org/wiki/Cross-site_scripting) enables attackers to inject malicious code into web pages. Such code can then, for example, steal user data (in particular, login data) or perform actions to impersonate the user. This is one of the most common attacks on the web.

To block XSS attacks, you must prevent malicious code from entering the DOM (Document Object Model). For example, if attackers can trick you into inserting a <script> tag in the DOM, they can run arbitrary code on your website. The attack isn't limited to <script> tags—many elements and properties in the DOM allow code execution, for example, <img onerror="..."> and <a href="javascript:...">. If attacker-controlled data enters the DOM, expect security vulnerabilities.

### **Angular’s cross-site scripting security model**

To systematically block XSS bugs, Angular treats all values as untrusted by default. When a value is inserted into the DOM from a template binding, or interpolation, Angular sanitizes and escapes untrusted values. If a value was already sanitized outside of Angular and is considered safe, you can communicate this to Angular by marking the [value as trusted](https://angular.io/guide/security#bypass-security-apis).

Unlike values to be used for rendering, Angular templates are considered trusted by default, and should be treated as executable code. Never generate templates by concatenating user input and template syntax. Doing this would enable attackers to [inject arbitrary code](https://en.wikipedia.org/wiki/Code_injection) into your application. To prevent these vulnerabilities, always use the default [AOT template compiler](https://angular.io/guide/security#offline-template-compiler) in production deployments.

### **Sanitization and security contexts**

Sanitization is the inspection of an untrusted value, turning it into a value that's safe to insert into the DOM. In many cases, sanitization doesn't change a value at all. Sanitization depends on context: a value that's harmless in CSS is potentially dangerous in a URL.

Angular defines the following security contexts:

* **HTML** is used when interpreting a value as HTML, for example, when binding to innerHtml.
* **Style** is used when binding CSS into the [style](https://angular.io/api/animations/style) property.
* **URL** is used for URL properties, such as <a href>.
* **Resource URL** is a URL that will be loaded and executed as code, for example, in <script src>.

Angular sanitizes untrusted values for HTML, styles, and URLs; sanitizing resource URLs isn't possible because they contain arbitrary code. In development mode, Angular prints a console warning when it has to change a value during sanitization.

### **Sanitization example**

The following template binds the value of htmlSnippet, once by interpolating it into an element's content, and once by binding it to the innerHTML property of an element:

src/app/inner-html-binding.component.html

content\_copy<h3>Binding innerHTML</h3>

<p>Bound value:</p>

<p class="e2e-inner-html-interpolated">{{htmlSnippet}}</p>

<p>Result of binding to innerHTML:</p>

<p class="e2e-inner-html-bound" [innerHTML]="htmlSnippet"></p>

Interpolated content is always escaped—the HTML isn't interpreted and the browser displays angle brackets in the element's text content.

For the HTML to be interpreted, bind it to an HTML property such as innerHTML. But binding a value that an attacker might control into innerHTML normally causes an XSS vulnerability. For example, one could execute JavaScript in a following way:

src/app/inner-html-binding.component.ts (class)

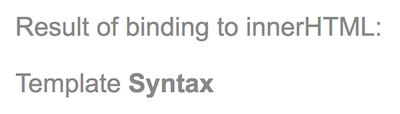
content\_copyexport class InnerHtmlBindingComponent {

// For example, a user/attacker-controlled value from a URL.

htmlSnippet = 'Template <script>alert("0wned")</script> <b>Syntax</b>';

}

Angular recognizes the value as unsafe and automatically sanitizes it, which removes the script element but keeps safe content such as the <b> element.



### **Direct use of the DOM APIs and explicit sanitization calls**

Built-in browser DOM APIs don't automatically protect you from security vulnerabilities. For example, document, the node available through [ElementRef](https://angular.io/api/core/ElementRef), and many third-party APIs contain unsafe methods. In the same way, if you interact with other libraries that manipulate the DOM, you likely won't have the same automatic sanitization as with Angular interpolations. Avoid directly interacting with the DOM and instead use Angular templates where possible.

For cases where this is unavoidable, use the built-in Angular sanitization functions. Sanitize untrusted values with the [DomSanitizer.sanitize](https://angular.io/api/platform-browser/DomSanitizer#sanitize) method and the appropriate [SecurityContext](https://angular.io/api/core/SecurityContext). That function also accepts values that were marked as trusted using the bypassSecurityTrust... functions, and will not sanitize them, as [described below](https://angular.io/guide/security#bypass-security-apis).

### **Trusting safe values**

Sometimes applications genuinely need to include executable code, display an <iframe> from some URL, or construct potentially dangerous URLs. To prevent automatic sanitization in any of these situations, you can tell Angular that you inspected a value, checked how it was generated, and made sure it will always be secure. But be careful. If you trust a value that might be malicious, you are introducing a security vulnerability into your application. If in doubt, find a professional security reviewer.

To mark a value as trusted, inject [DomSanitizer](https://angular.io/api/platform-browser/DomSanitizer) and call one of the following methods:

* bypassSecurityTrustHtml
* bypassSecurityTrustScript
* bypassSecurityTrustStyle
* bypassSecurityTrustUrl
* bypassSecurityTrustResourceUrl

Remember, whether a value is safe depends on context, so choose the right context for your intended use of the value. Imagine that the following template needs to bind a URL to a javascript:alert(...) call:

src/app/bypass-security.component.html (URL)

content\_copy<h4>An untrusted URL:</h4>

<p><a class="e2e-dangerous-url" [href]="dangerousUrl">Click me</a></p>

<h4>A trusted URL:</h4>

<p><a class="e2e-trusted-url" [href]="trustedUrl">Click me</a></p>

Normally, Angular automatically sanitizes the URL, disables the dangerous code, and in development mode, logs this action to the console. To prevent this, mark the URL value as a trusted URL using the bypassSecurityTrustUrl call:

src/app/bypass-security.component.ts (trust-url)

content\_copyconstructor(private sanitizer: [DomSanitizer](https://angular.io/api/platform-browser/DomSanitizer)) {

// javascript: URLs are dangerous if attacker controlled.

// Angular sanitizes them in data binding, but you can

// explicitly tell Angular to trust this value:

this.dangerousUrl = 'javascript:alert("Hi there")';

this.trustedUrl = sanitizer.bypassSecurityTrustUrl(this.dangerousUrl);



If you need to convert user input into a trusted value, use a component method. The following template allows users to enter a YouTube video ID and load the corresponding video in an <iframe>. The <iframe src> attribute is a resource URL security context, because an untrusted source can, for example, smuggle in file downloads that unsuspecting users could execute. So call a method on the component to construct a trusted video URL, which causes Angular to allow binding into <iframe src>:

src/app/bypass-security.component.html (iframe)

content\_copy<h4>Resource URL:</h4>

<p>Showing: {{dangerousVideoUrl}}</p>

<p>Trusted:</p>

<iframe class="e2e-iframe-trusted-src" width="640" height="390" [src]="videoUrl"></iframe>

<p>Untrusted:</p>

<iframe class="e2e-iframe-untrusted-src" width="640" height="390" [src]="dangerousVideoUrl"></iframe>

src/app/bypass-security.component.ts (trust-video-url)

content\_copyupdateVideoUrl(id: string) {

// Appending an ID to a YouTube URL is safe.

// Always make sure to construct [SafeValue](https://angular.io/api/platform-browser/SafeValue) objects as

// close as possible to the input data so

// that it's easier to check if the value is safe.

this.dangerousVideoUrl = 'https://www.youtube.com/embed/' + id;

this.videoUrl =

this.sanitizer.bypassSecurityTrustResourceUrl(this.dangerousVideoUrl);

}

### **Content security policy**

Content Security Policy (CSP) is a defense-in-depth technique to prevent XSS. To enable CSP, configure your web server to return an appropriate Content-Security-Policy HTTP header. Read more about content security policy at the [Web Fundamentals guide](https://developers.google.com/web/fundamentals/security/csp) on the Google Developers website.

### **Use the AOT template compiler**

The AOT template compiler prevents a whole class of vulnerabilities called template injection, and greatly improves application performance. The AOT template compiler is the default compiler used by Angular CLI applications, and you should use it in all production deployments.

An alternative to the AOT compiler is the JIT compiler which compiles templates to executable template code within the browser at runtime. Angular trusts template code, so dynamically generating templates and compiling them, in particular templates containing user data, circumvents Angular's built-in protections and is a security anti-pattern. For information about dynamically constructing forms in a safe way, see the [Dynamic Forms](https://angular.io/guide/dynamic-form) guide.

### **Server-side XSS protection**

HTML constructed on the server is vulnerable to injection attacks. Injecting template code into an Angular application is the same as injecting executable code into the application: it gives the attacker full control over the application. To prevent this, use a templating language that automatically escapes values to prevent XSS vulnerabilities on the server. Don't generate Angular templates on the server side using a templating language; doing this carries a high risk of introducing template-injection vulnerabilities.

## **HTTP-level vulnerabilities**

Angular has built-in support to help prevent two common HTTP vulnerabilities, cross-site request forgery (CSRF or XSRF) and cross-site script inclusion (XSSI). Both of these must be mitigated primarily on the server side, but Angular provides helpers to make integration on the client side easier.

### **Cross-site request forgery**

In a cross-site request forgery (CSRF or XSRF), an attacker tricks the user into visiting a different web page (such as evil.com) with malignant code that secretly sends a malicious request to the application's web server (such as example-bank.com).

Assume the user is logged into the application at example-bank.com. The user opens an email and clicks a link to evil.com, which opens in a new tab.

The evil.com page immediately sends a malicious request to example-bank.com. Perhaps it's a request to transfer money from the user's account to the attacker's account. The browser automatically sends the example-bank.com cookies (including the authentication cookie) with this request.

If the example-bank.com server lacks XSRF protection, it can't tell the difference between a legitimate request from the application and the forged request from evil.com.

To prevent this, the application must ensure that a user request originates from the real application, not from a different site. The server and client must cooperate to thwart this attack.

In a common anti-XSRF technique, the application server sends a randomly generated authentication token in a cookie. The client code reads the cookie and adds a custom request header with the token in all subsequent requests. The server compares the received cookie value to the request header value and rejects the request if the values are missing or don't match.

This technique is effective because all browsers implement the same origin policy. Only code from the website on which cookies are set can read the cookies from that site and set custom headers on requests to that site. That means only your application can read this cookie token and set the custom header. The malicious code on evil.com can't.

Angular's [HttpClient](https://angular.io/api/common/http/HttpClient) has built-in support for the client-side half of this technique. Read about it more in the [HttpClient guide](https://angular.io/guide/http#security-xsrf-protection).

For information about CSRF at the Open Web Application Security Project (OWASP), see [Cross-Site Request Forgery (CSRF)](https://owasp.org/www-community/attacks/csrf) and [Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet](https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html). The Stanford University paper [Robust Defenses for Cross-Site Request Forgery](https://seclab.stanford.edu/websec/csrf/csrf.pdf) is a rich source of detail.

See also Dave Smith's easy-to-understand [talk on XSRF at AngularConnect 2016](https://www.youtube.com/watch?v=9inczw6qtpY).

### **Cross-site script inclusion (XSSI)**

Cross-site script inclusion, also known as JSON vulnerability, can allow an attacker's website to read data from a JSON API. The attack works on older browsers by overriding native JavaScript object constructors, and then including an API URL using a <script> tag.

This attack is only successful if the returned JSON is executable as JavaScript. Servers can prevent an attack by prefixing all JSON responses to make them non-executable, by convention, using the well-known string ")]}',\n".

Angular's [HttpClient](https://angular.io/api/common/http/HttpClient) library recognizes this convention and automatically strips the string ")]}',\n" from all responses before further parsing.

For more information, see the XSSI section of this [Google web security blog post](https://security.googleblog.com/2011/05/website-security-for-webmasters.html).

## **Auditing Angular applications**

Angular applications must follow the same security principles as regular web applications, and must be audited as such. Angular-specific APIs that should be audited in a security review, such as the [bypassSecurityTrust](https://angular.io/guide/security#bypass-security-apis) methods, are marked in the documentation as security sensitive.

# Accessibility in Angular

The web is used by a wide variety of people, including those who have visual or motor impairments. A variety of assistive technologies are available that make it much easier for these groups to interact with web-based software applications. In addition, designing an application to be more accessible generally improves the user experience for all users.

For an in-depth introduction to issues and techniques for designing accessible applications, see the [Accessibility](https://developers.google.com/web/fundamentals/accessibility/#what_is_accessibility) section of the Google's [Web Fundamentals](https://developers.google.com/web/fundamentals/).

This page discusses best practices for designing Angular applications that work well for all users, including those who rely on assistive technologies.

For the sample application that this page describes, see the [live example](https://angular.io/generated/live-examples/accessibility/stackblitz.html) / [download example](https://angular.io/generated/zips/accessibility/accessibility.zip).

## **Accessibility attributes**

Building accessible web experience often involves setting [ARIA attributes](https://developers.google.com/web/fundamentals/accessibility/semantics-aria) to provide semantic meaning where it might otherwise be missing. Use [attribute binding](https://angular.io/guide/attribute-binding) template syntax to control the values of accessibility-related attributes.

When binding to ARIA attributes in Angular, you must use the attr. prefix, as the ARIA specification depends specifically on HTML attributes rather than properties of DOM elements.

content\_copy<!-- Use attr. when binding to an ARIA attribute -->

<button [attr.aria-label]="myActionLabel">...</button>

Note that this syntax is only necessary for attribute bindings. Static ARIA attributes require no extra syntax.

content\_copy<!-- Static ARIA attributes require no extra syntax -->

<button aria-label="Save document">...</button>

By convention, HTML attributes use lowercase names (tabindex), while properties use camelCase names (tabIndex).

See the [Binding syntax](https://angular.io/guide/binding-syntax#html-attribute-vs-dom-property) guide for more background on the difference between attributes and properties.

## **Angular UI components**

The [Angular Material](https://material.angular.io/) library, which is maintained by the Angular team, is a suite of reusable UI components that aims to be fully accessible. The [Component Development Kit (CDK)](https://material.angular.io/cdk/categories) includes the a11y package that provides tools to support various areas of accessibility. For example:

* LiveAnnouncer is used to announce messages for screen-reader users using an aria-live region. See the W3C documentation for more information on [aria-live regions](https://www.w3.org/WAI/PF/aria-1.1/states_and_properties#aria-live).
* The cdkTrapFocus directive traps Tab-key focus within an element. Use it to create accessible experience for components like modal dialogs, where focus must be constrained.

For full details of these and other tools, see the [Angular CDK accessibility overview](https://material.angular.io/cdk/a11y/overview).

### **Augmenting native elements**

Native HTML elements capture a number of standard interaction patterns that are important to accessibility. When authoring Angular components, you should re-use these native elements directly when possible, rather than re-implementing well-supported behaviors.

For example, instead of creating a custom element for a new variety of button, you can create a component that uses an attribute selector with a native <button> element. This most commonly applies to <button> and <a>, but can be used with many other types of element.

You can see examples of this pattern in Angular Material: [MatButton](https://github.com/angular/components/blob/50d3f29b6dc717b512dbd0234ce76f4ab7e9762a/src/material/button/button.ts#L67-L69), [MatTabNav](https://github.com/angular/components/blob/50d3f29b6dc717b512dbd0234ce76f4ab7e9762a/src/material/tabs/tab-nav-bar/tab-nav-bar.ts#L139), [MatTable](https://github.com/angular/components/blob/50d3f29b6dc717b512dbd0234ce76f4ab7e9762a/src/material/table/table.ts#L22).

### **Using containers for native elements**

Sometimes using the appropriate native element requires a container element. For example, the native <input> element cannot have children, so any custom text entry components need to wrap an <input> with additional elements. While you might just include the <input> in your custom component's template, this makes it impossible for users of the component to set arbitrary properties and attributes to the input element. Instead, you can create a container component that uses content projection to include the native control in the component's API.

You can see [MatFormField](https://material.angular.io/components/form-field/overview) as an example of this pattern.

## **Case study: Building a custom progress bar**

The following example shows how to make a progress bar accessible by using host binding to control accessibility-related attributes.

* The component defines an accessibility-enabled element with both the standard HTML attribute role, and ARIA attributes. The ARIA attribute aria-valuenow is bound to the user's input.

src/app/progress-bar.component.ts

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

/\*\*

\* Example progressbar component.

\*/

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

selector: 'app-example-progressbar',

template: `<div class="bar" [style.width.%]="value"></div>`,

styleUrls: ['./progress-bar.component.css'],

host: {

// Sets the role for this component to "progressbar"

role: 'progressbar',

// Sets the minimum and maximum values for the progressbar role.

'aria-valuemin': '0',

'aria-valuemax': '100',

// Binding that updates the current value of the progressbar.

'[attr.aria-valuenow]': 'value',

}

})

export class ExampleProgressbarComponent {

/\*\* Current value of the progressbar. \*/

@[Input](https://angular.io/api/core/Input)() value = 0;

}

* In the template, the aria-label attribute ensures that the control is accessible to screen readers.

src/app/app.component.html

content\_copy<label>

Enter an example progress value

<input type="number" [min](https://angular.io/api/forms/MinValidator)="0" [max](https://angular.io/api/forms/MaxValidator)="100"

[value]="progress" (input)="setProgress($event)">

</label>

<!-- The user of the progressbar sets an aria-label to communicate what the progress means. -->

<app-example-progressbar [value]="progress" aria-label="Example of a progress bar">

</app-example-progressbar>

## **Routing and focus management**

Tracking and controlling [focus](https://developers.google.com/web/fundamentals/accessibility/focus/) in a UI is an important consideration in designing for accessibility. When using Angular routing, you should decide where page focus goes upon navigation.

To avoid relying solely on visual cues, you need to make sure your routing code updates focus after page navigation. Use the [NavigationEnd](https://angular.io/api/router/NavigationEnd) event from the [Router](https://angular.io/api/router/Router) service to know when to update focus.

The following example shows how to find and focus the main content header in the DOM after navigation.

content\_copyrouter.events.pipe(filter(e => e instanceof [NavigationEnd](https://angular.io/api/router/NavigationEnd))).subscribe(() => {

const mainHeader = document.querySelector('#main-content-header')

if (mainHeader) {

mainHeader.focus();

}

});

In a real application, the element that receives focus will depend on your specific application structure and layout. The focused element should put users in a position to immediately move into the main content that has just been routed into view. You should avoid situations where focus returns to the body element after a route change.

## **Additional resources**

* [Accessibility - Google Web Fundamentals](https://developers.google.com/web/fundamentals/accessibility)
* [ARIA specification and authoring practices](https://www.w3.org/TR/wai-aria/)
* [Material Design - Accessibility](https://material.io/design/usability/accessibility.html)
* [Smashing Magazine](https://www.smashingmagazine.com/search/?q=accessibility)
* [Inclusive Components](https://inclusive-components.design/)
* [Accessibility Resources and Code Examples](https://dequeuniversity.com/resources/)
* [W3C - Web Accessibility Initiative](https://www.w3.org/WAI/people-use-web/)
* [Rob Dodson A11ycasts](https://www.youtube.com/watch?v=HtTyRajRuyY)
* [Angular ESLint](https://github.com/angular-eslint/angular-eslint#functionality) provides linting rules that can help you make sure your code meets accessibility standards.

Books

* "A Web for Everyone: Designing Accessible User Experiences", Sarah Horton and Whitney Quesenbery
* "Inclusive Design Patterns", Heydon Pickering

# Keeping your Angular projects up-to-date

Just like Web and the entire web ecosystem, Angular is continuously improving. Angular balances continuous improvement with a strong focus on stability and making updates easy. Keeping your Angular application up-to-date enables you to take advantage of leading-edge new features, as well as optimizations and bug fixes.

This document contains information and resources to help you keep your Angular applications and libraries up-to-date.

For information about our versioning policy and practices—including support and deprecation practices, as well as the release schedule—see [Angular versioning and releases](https://angular.io/guide/releases).

If you are currently using AngularJS, see [Upgrading from AngularJS](https://angular.io/guide/upgrade). AngularJS is the name for all v1.x versions of Angular.

## **Getting notified of new releases**

To be notified when new releases are available, follow [@angular](https://twitter.com/angular) on Twitter or subscribe to the [Angular blog](https://blog.angular.io/).

## **Learning about new features**

What's new? What's changed? We share the most important things you need to know on the Angular blog in [release announcements](https://blog.angular.io/tagged/release%20notes).

To review a complete list of changes, organized by version, see the [Angular change log](https://github.com/angular/angular/blob/master/CHANGELOG.md).

## **Checking your version of Angular**

To check your application's version of Angular: From within your project directory, use the ng version command.

## **Finding the current version of Angular**

The most recent stable released version of Angular appears in the [Angular documentation](https://angular.io/docs) at the bottom of the left side navigation. For example, stable (v5.2.9).

You can also find the most current version of Angular by using the CLI command [ng update](https://angular.io/cli/update). By default, [ng update](https://angular.io/cli/update)(without additional arguments) lists the updates that are available to you.

## **Updating your environment and apps**

To make updating easy, we provide complete instructions in the interactive [Angular Update Guide](https://update.angular.io/).

The Angular Update Guide provides customized update instructions, based on the current and target versions that you specify. It includes basic and advanced update paths, to match the complexity of your applications. It also includes troubleshooting information and any recommended manual changes to help you get the most out of the new release.

For simple updates, the CLI command [ng update](https://angular.io/cli/update) is all you need. Without additional arguments, [ng update](https://angular.io/cli/update) lists the updates that are available to you and provides recommended steps to update your application to the most current version.

[Angular Versioning and Releases](https://angular.io/guide/releases#versioning) describes the level of change that you can expect based a release's version number. It also describes supported update paths.

## **Resource summary**

* Release announcements: [Angular blog - release announcements](https://blog.angular.io/tagged/release%20notes)
* Release announcements (older): [Angular blog - announcements about releases prior to August 2017](https://blog.angularjs.org/search?q=available&by-date=true)
* Release details: [Angular change log](https://github.com/angular/angular/blob/master/CHANGELOG.md)
* Update instructions: [Angular Update Guide](https://update.angular.io/)
* Update command reference: [Angular CLI ng update command reference](https://angular.io/cli/update)
* Versioning, release, support, and deprecation practices: [Angular versioning and releases](https://angular.io/guide/releases)

# Property binding best practices

By following a few guidelines, you can use property binding in a way that helps you minimize bugs and keep your code readable.

See the [live example](https://angular.io/generated/live-examples/property-binding/stackblitz.html) / [download example](https://angular.io/generated/zips/property-binding/property-binding.zip) for a working example containing the code snippets in this guide.

## **Avoid side effects**

Evaluation of a template expression should have no visible side effects. Use the syntax for template expressions to help avoid side effects. In general, the correct syntax prevents you from assigning a value to anything in a property binding expression. The syntax also prevents you from using increment and decrement operators.

### **An example of producing side effects**

If you had an expression that changed the value of something else that you were binding to, that change of value would be a side effect. Angular might or might not display the changed value. If Angular does detect the change, it throws an error.

As a best practice, use only properties and methods that return values.

## **Return the proper type**

A template expression should evaluate to the type of value that the target property expects. For example, return a string if the target property expects a string, a number if it expects a number, or an object if it expects an object.

### **Passing in a string**

In the following example, the childItem property of the ItemDetailComponent expects a string.

src/app/app.component.html

content\_copy<app-item-detail [childItem]="parentItem"></app-item-detail>

You can confirm this expectation by looking in the ItemDetailComponent where the @[Input](https://angular.io/api/core/Input)() type is string:

src/app/item-detail/item-detail.component.ts (setting the @Input() type)

content\_copy@[Input](https://angular.io/api/core/Input)() childItem = '';

The parentItem in AppComponent is a string, which means that the expression, parentItem within [childItem]="parentItem", evaluates to a string.

src/app/app.component.ts

content\_copyparentItem = 'lamp';

If parentItem were some other type, you would need to specify childItem @[Input](https://angular.io/api/core/Input)() as that type as well.

### **Passing in an object**

In this example, ItemListComponent is a child component of AppComponent and the items property expects an array of objects.

src/app/app.component.html

content\_copy<app-item-list [items]="currentItems"></app-item-list>

In the ItemListComponent the @[Input](https://angular.io/api/core/Input)(), items, has a type of Item[].

src/app/item-list.component.ts

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

Notice that Item is an object that it has two properties; an id and a name.

src/app/item.ts

content\_copyexport interface Item {

id: number;

name: string;

}

In app.component.ts, currentItems is an array of objects in the same shape as the Item object in items.ts, with an id and a name.

src/app.component.ts

content\_copycurrentItems = [{

id: 21,

name: 'phone'

}];

By supplying an object in the same shape, you satisfy the expectations of items when Angular evaluates the expression currentItems.

# Lazy-loading feature modules

By default, NgModules are eagerly loaded, which means that as soon as the application loads, so do all the NgModules, whether or not they are immediately necessary. For large applications with lots of routes, consider lazy loading—a design pattern that loads NgModules as needed. Lazy loading helps keep initial bundle sizes smaller, which in turn helps decrease load times.

For the final sample application with two lazy-loaded modules that this page describes, see the [live example](https://angular.io/generated/live-examples/lazy-loading-ngmodules/stackblitz.html) / [download example](https://angular.io/generated/zips/lazy-loading-ngmodules/lazy-loading-ngmodules.zip).

## **Lazy loading basics**

This section introduces the basic procedure for configuring a lazy-loaded route. For a step-by-step example, see the [step-by-step setup](https://angular.io/guide/lazy-loading-ngmodules#step-by-step) section on this page.

To lazy load Angular modules, use loadChildren (instead of component) in your AppRoutingModule routes configuration as follows.

AppRoutingModule (excerpt)

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

{

path: 'items',

loadChildren: () => import('./items/items.module').then(m => m.ItemsModule)

}

];

In the lazy-loaded module's routing module, add a route for the component.

Routing module for lazy loaded module (excerpt)

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

{

path: '',

component: ItemsComponent

}

];

Also be sure to remove the ItemsModule from the AppModule. For step-by-step instructions on lazy loading modules, continue with the following sections of this page.

## **Step-by-step setup**

There are two main steps to setting up a lazy-loaded feature module:

1. Create the feature module with the CLI, using the --route flag.
2. Configure the routes.

### **Set up an app**

If you don’t already have an app, you can follow the steps below to create one with the CLI. If you already have an app, skip to [Configure the routes](https://angular.io/guide/lazy-loading-ngmodules#config-routes). Enter the following command where customer-app is the name of your app:

content\_copyng new customer-app --routing

This creates an application called customer-app and the --routing flag generates a file called app-routing.module.ts, which is one of the files you need for setting up lazy loading for your feature module. Navigate into the project by issuing the command cd customer-app.

The --routing option requires Angular/CLI version 8.1 or higher. See [Keeping Up to Date](https://angular.io/guide/updating).

### **Create a feature module with routing**

Next, you’ll need a feature module with a component to route to. To make one, enter the following command in the terminal, where customers is the name of the feature module. The path for loading the customers feature modules is also customers because it is specified with the --route option:

content\_copyng generate module customers --route customers --module app.module

This creates a customers folder having the new lazy-loadable feature module CustomersModule defined in the customers.module.ts file and the routing module CustomersRoutingModule defined in the customers-routing.module.ts file. The command automatically declares the CustomersComponent and imports CustomersRoutingModule inside the new feature module.

Because the new module is meant to be lazy-loaded, the command does NOT add a reference to the new feature module in the application's root module file, app.module.ts. Instead, it adds the declared route, customers to the routes array declared in the module provided as the --module option.

src/app/app-routing.module.ts

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

{

path: 'customers',

loadChildren: () => import('./customers/customers.module').then(m => m.CustomersModule)

}

];

Notice that the lazy-loading syntax uses loadChildren followed by a function that uses the browser's built-in import('...') syntax for dynamic imports. The import path is the relative path to the module.

STRING-BASED LAZY LOADING

In Angular version 8, the string syntax for the loadChildren route specification [was deprecated](https://angular.io/guide/deprecations#loadchildren-string-syntax) in favor of the import() syntax. However, you can opt into using string-based lazy loading (loadChildren: './path/to/module#Module') by including the lazy-loaded routes in your tsconfig file, which includes the lazy-loaded files in the compilation.

By default the CLI will generate projects with stricter file inclusions intended to be used with the import() syntax.

### **Add another feature module**

Use the same command to create a second lazy-loaded feature module with routing, along with its stub component.

content\_copyng generate module orders --route orders --module app.module

This creates a new folder called orders containing the OrdersModule and OrdersRoutingModule, along with the new OrdersComponent source files. The orders route, specified with the --route option, is added to the routes array inside the app-routing.module.ts file, using the lazy-loading syntax.

src/app/app-routing.module.ts

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

{

path: 'customers',

loadChildren: () => import('./customers/customers.module').then(m => m.CustomersModule)

},

{

path: 'orders',

loadChildren: () => import('./orders/orders.module').then(m => m.OrdersModule)

}

];

### **Set up the UI**

Though you can type the URL into the address bar, a navigation UI is easier for the user and more common. Replace the default placeholder markup in app.component.html with a custom nav so you can easily navigate to your modules in the browser:

src/app/app.component.html

content\_copy<h1>

{{title}}

</h1>

<button [routerLink](https://angular.io/api/router/RouterLink)="/customers">Customers</button>

<button [routerLink](https://angular.io/api/router/RouterLink)="/orders">Orders</button>

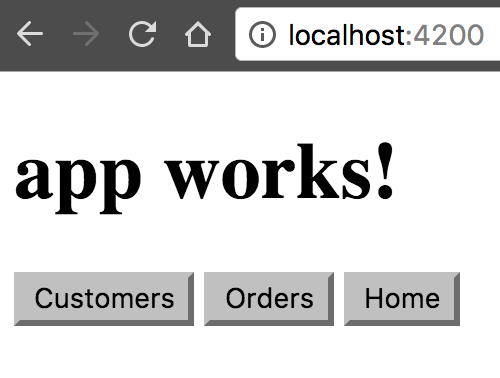
<button [routerLink](https://angular.io/api/router/RouterLink)="">Home</button>

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

To see your application in the browser so far, enter the following command in the terminal window:

content\_copyng serve

Then go to localhost:4200 where you should see “customer-app” and three buttons.



These buttons work, because the CLI automatically added the routes to the feature modules to the routes array in app.module.ts.

### **Imports and route configuration**

The CLI automatically added each feature module to the routes map at the application level. Finish this off by adding the default route. In the app-routing.module.ts file, update the routes array with the following:

src/app/app-routing.module.ts

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

{

path: 'customers',

loadChildren: () => import('./customers/customers.module').then(m => m.CustomersModule)

},

{

path: 'orders',

loadChildren: () => import('./orders/orders.module').then(m => m.OrdersModule)

},

{

path: '',

redirectTo: '',

pathMatch: 'full'

}

];

The first two paths are the routes to the CustomersModule and the OrdersModule. The final entry defines a default route. The empty path matches everything that doesn't match an earlier path.

### **Inside the feature module**

Next, take a look at the customers.module.ts file. If you’re using the CLI and following the steps outlined in this page, you don’t have to do anything here.

src/app/customers/customers.module.ts

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

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

import { CustomersRoutingModule } from './customers-routing.module';

import { CustomersComponent } from './customers.component';

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

imports: [

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

CustomersRoutingModule

],

declarations: [CustomersComponent]

})

export class CustomersModule { }

The customers.module.ts file imports the customers-routing.module.ts and customers.component.ts files. CustomersRoutingModule is listed in the @[NgModule](https://angular.io/api/core/NgModule) imports array giving CustomersModule access to its own routing module. CustomersComponent is in the declarations array, which means CustomersComponent belongs to the CustomersModule.

The app-routing.module.ts then imports the feature module, customers.module.ts using JavaScript's dynamic import.

The feature-specific route definition file customers-routing.module.ts imports its own feature component defined in the customers.component.ts file, along with the other JavaScript import statements. It then maps the empty path to the CustomersComponent.

src/app/customers/customers-routing.module.ts

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

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

import { CustomersComponent } from './customers.component';

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

{

path: '',

component: CustomersComponent

}

];

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

imports: [RouterModule.forChild(routes)],

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

})

export class CustomersRoutingModule { }

The path here is set to an empty string because the path in AppRoutingModule is already set to customers, so this route in the CustomersRoutingModule, is already within the customers context. Every route in this routing module is a child route.

The other feature module's routing module is configured similarly.

src/app/orders/orders-routing.module.ts (excerpt)

content\_copyimport { OrdersComponent } from './orders.component';

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

{

path: '',

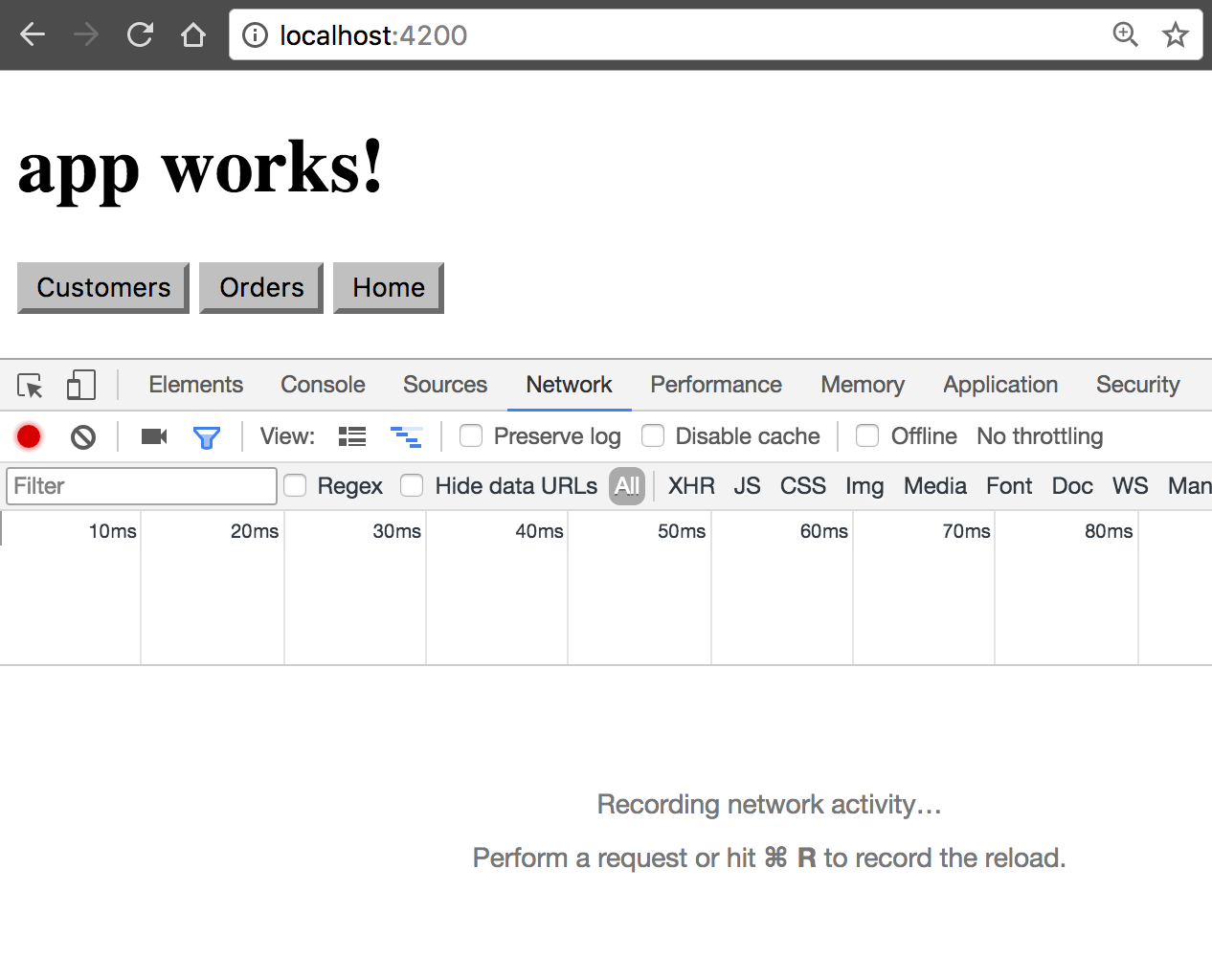
component: OrdersComponent

}

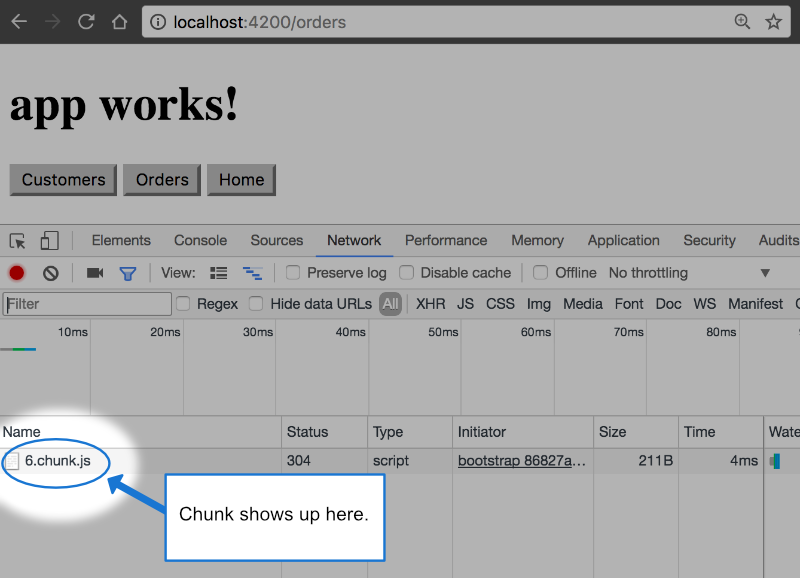
];

### **Verify lazy loading**

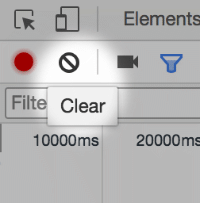
You can check to see that a module is indeed being lazy loaded with the Chrome developer tools. In Chrome, open the developer tools by pressing Cmd+Option+i on a Mac or Ctrl+Shift+j on a PC and go to the Network Tab.



Click on the Orders or Customers button. If you see a chunk appear, everything is wired up properly and the feature module is being lazy loaded. A chunk should appear for Orders and for Customers but will only appear once for each.



To see it again, or to test after working in the project, clear everything out by clicking the circle with a line through it in the upper left of the Network Tab:



Then reload with Cmd+r or Ctrl+r, depending on your platform.

## **forRoot() and forChild()**

You might have noticed that the CLI adds RouterModule.forRoot(routes) to the AppRoutingModule imports array. This lets Angular know that the AppRoutingModule is a routing module and forRoot() specifies that this is the root routing module. It configures all the routes you pass to it, gives you access to the router directives, and registers the [Router](https://angular.io/api/router/Router) service. Use forRoot() only once in the application, inside the AppRoutingModule.

The CLI also adds RouterModule.forChild(routes) to feature routing modules. This way, Angular knows that the route list is only responsible for providing additional routes and is intended for feature modules. You can use forChild() in multiple modules.

The forRoot() method takes care of the global injector configuration for the Router. The forChild() method has no injector configuration. It uses directives such as [RouterOutlet](https://angular.io/api/router/RouterOutlet) and [RouterLink](https://angular.io/api/router/RouterLink). For more information, see the [forRoot() pattern](https://angular.io/guide/singleton-services#forRoot) section of the [Singleton Services](https://angular.io/guide/singleton-services) guide.

## **Preloading**

Preloading improves UX by loading parts of your application in the background. You can preload modules or component data.

### **Preloading modules**

Preloading modules improves UX by loading parts of your application in the background so users don't have to wait for the elements to download when they activate a route.

To enable preloading of all lazy loaded modules, import the [PreloadAllModules](https://angular.io/api/router/PreloadAllModules) token from the Angular router.

AppRoutingModule (excerpt)

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

Still in the AppRoutingModule, specify your preloading strategy in forRoot().

AppRoutingModule (excerpt)

content\_copyRouterModule.forRoot(

appRoutes,

{

preloadingStrategy: [PreloadAllModules](https://angular.io/api/router/PreloadAllModules)

}

)

### **Preloading component data**

To preload component data, you can use a resolver. Resolvers improve UX by blocking the page load until all necessary data is available to fully display the page.

#### Resolvers

Create a resolver service. With the CLI, the command to generate a service is as follows:

content\_copyng generate service

In the newly-created service, implement the [Resolve](https://angular.io/api/router/Resolve) interface provided by the @angular/router package:

Resolver service (excerpt)

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

...

/\* An interface that represents your data model \*/

export interface Crisis {

id: number;

name: string;

}

export class CrisisDetailResolverService implements [Resolve](https://angular.io/api/router/Resolve) {

resolve(route: [ActivatedRouteSnapshot](https://angular.io/api/router/ActivatedRouteSnapshot), state: [RouterStateSnapshot](https://angular.io/api/router/RouterStateSnapshot)): Observable {

// your logic goes here

}

}

Import this resolver into your module's routing module.

Feature module's routing module (excerpt)

content\_copyimport { CrisisDetailResolverService } from './crisis-detail-resolver.service';

Add a resolve object to the component's route configuration.

Feature module's routing module (excerpt)

content\_copy{

path: '/your-path',

component: YourComponent,

resolve: {

crisis: CrisisDetailResolverService

}

}

In the component's constructor, inject an instance of the [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) class that represents the current route.

Component's constructor (excerpt)

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

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

class YourComponent {

constructor(private route: [ActivatedRoute](https://angular.io/api/router/ActivatedRoute)) {}

}

Use the injected instance of the [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) class to access data associated with a given route.

Component's ngOnInit lifecycle hook (excerpt)

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

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

class YourComponent {

constructor(private route: [ActivatedRoute](https://angular.io/api/router/ActivatedRoute)) {}

ngOnInit() {

this.route.data

.subscribe(data => {

const crisis: Crisis = data.crisis;

// ...

});

}

}

For more information with a working example, see the [routing tutorial section on preloading](https://angular.io/guide/router-tutorial-toh#preloading-background-loading-of-feature-areas).

## **Troubleshooting lazy-loading modules**

A common error when lazy-loading modules is importing common modules in multiple places within an application. You can test for this condition by first generating the module using the Angular CLI and including the --route route-name parameter, where route-name is the name of your module. Next, generate the module without the --route parameter. If the Angular CLI generates an error when you use the --route parameter, but runs correctly without it, you may have imported the same module in multiple places.

Remember, many common Angular modules should be imported at the base of your application.

For more information on Angular Modules, see [NgModules](https://angular.io/guide/ngmodules).

## **More on NgModules and routing**

You may also be interested in the following:

* [Routing and Navigation](https://angular.io/guide/router).
* [Providers](https://angular.io/guide/providers).
* [Types of Feature Modules](https://angular.io/guide/module-types).
* [Route-level code-splitting in Angular](https://web.dev/route-level-code-splitting-in-angular/)
* [Route preloading strategies in Angular](https://web.dev/route-preloading-in-angular/)

# Optimizing client app size with lightweight injection tokens

This page provides a conceptual overview of a dependency injection technique that is recommended for library developers. Designing your library with lightweight injection tokens helps optimize the bundle size of client applications that use your library.

You can manage the dependency structure among your components and injectable services to optimize bundle size by using [tree-shakable providers](https://angular.io/guide/architecture-services#introduction-to-services-and-dependency-injection). This normally ensures that if a provided component or service is never actually used by the app, the compiler can eliminate its code from the bundle.

However, due to the way Angular stores injection tokens, it is possible that such an unused component or service can end up in the bundle anyway. This page describes a dependency-injection design pattern that supports proper tree-shaking by using lightweight injection tokens.

The lightweight injection token design pattern is especially important for library developers. It ensures that when an application uses only some of your library's capabilities, the unused code can be eliminated from the client's application bundle.

When an application uses your library, there might be some services that your library supplies which the client application doesn't use. In this case, the application developer should expect that service to be tree-shaken, and not contribute to the size of the compiled application. Because the application developer cannot know about or remedy a tree-shaking problem in the library, it is the responsibility of the library developer to do so. To prevent the retention of unused components, your library should use the lightweight injection token design pattern.

## **When tokens are retained**

To better explain the condition under which token retention occurs, consider a library that provides a library-card component, which contains a body and can contain an optional header.

content\_copy<lib-card>

<lib-header>...</lib-header>

</lib-card>

In a likely implementation, the <lib-card> component uses @[ContentChild](https://angular.io/api/core/ContentChild)() or @[ContentChildren](https://angular.io/api/core/ContentChildren)() to obtain <lib-header> and <lib-body>, as in the following.

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

selector: 'lib-header',

...,

})

class LibHeaderComponent {}

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

selector: 'lib-card',

...,

})

class LibCardComponent {

@[ContentChild](https://angular.io/api/core/ContentChild)(LibHeaderComponent)

header: LibHeaderComponent|null = null;

}

Because <lib-header> is optional, the element can appear in the template in its minimal form, <lib-card></lib-card>. In this case, <lib-header> is not used and you would expect it to be tree-shaken, but that is not what happens. This is because LibCardComponent actually contains two references to the LibHeaderComponent.

@[ContentChild](https://angular.io/api/core/ContentChild)(LibHeaderComponent) header: LibHeaderComponent;

* One of these reference is in the type position-- that is, it specifies LibHeaderComponent as a type: header: LibHeaderComponent;.
* The other reference is in the value position-- that is, LibHeaderComponent is the value of the @[ContentChild](https://angular.io/api/core/ContentChild)() parameter decorator: @[ContentChild](https://angular.io/api/core/ContentChild)(LibHeaderComponent).

The compiler handles token references in these positions differently.

* The compiler erases type position references after conversion from TypeScript, so they have no impact on tree-shaking.
* The compiler must retain value position references at runtime, which prevents the component from being tree-shaken.

In the example, the compiler retains the LibHeaderComponent token that occurs in the value position, which prevents the referenced component from being tree-shaken, even if the application developer does not actually use <lib-header> anywhere. If LibHeaderComponent is large (code, template, and styles), including it unnecessarily can significantly increase the size of the client application.

## **When to use the lightweight injection token pattern**

The tree-shaking problem arises when a component is used as an injection token. There are two cases when that can happen.

* The token is used in the value position of a [content query](https://angular.io/guide/lifecycle-hooks#using-aftercontent-hooks).
* The token is used as a type specifier for constructor injection.

In the following example, both uses of the OtherComponent token cause retention of OtherComponent (that is, prevent it from being tree-shaken when it is not used).

content\_copyclass MyComponent {

constructor(@[Optional](https://angular.io/api/core/Optional)() other: OtherComponent) {}

@[ContentChild](https://angular.io/api/core/ContentChild)(OtherComponent)

other: OtherComponent|null;

}

Although tokens used only as type specifiers are removed when converted to JavaScript, all tokens used for dependency injection are needed at runtime. These effectively change constructor(@[Optional](https://angular.io/api/core/Optional)() other: OtherComponent) to constructor(@[Optional](https://angular.io/api/core/Optional)() @[Inject](https://angular.io/api/core/Inject)(OtherComponent) other). The token is now in a value position, and causes the tree shaker to retain the reference.

For all services, a library should use [tree-shakable providers](https://angular.io/guide/architecture-services#introduction-to-services-and-dependency-injection), providing dependencies at the root level rather than in component constructors.

## **Using lightweight injection tokens**

The lightweight injection token design pattern consists of using a small abstract class as an injection token, and providing the actual implementation at a later stage. The abstract class is retained (not tree-shaken), but it is small and has no material impact on the application size.

The following example shows how this works for the LibHeaderComponent.

content\_copyabstract class LibHeaderToken {}

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

selector: 'lib-header',

providers: [

{provide: LibHeaderToken, useExisting: LibHeaderComponent}

]

...,

})

class LibHeaderComponent extends LibHeaderToken {}

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

selector: 'lib-card',

...,

})

class LibCardComponent {

@[ContentChild](https://angular.io/api/core/ContentChild)(LibHeaderToken) header: LibHeaderToken|null = null;

}

In this example, the LibCardComponent implementation no longer refers to LibHeaderComponent in either the type position or the value position. This allows full tree shaking of LibHeaderComponent to take place. The LibHeaderToken is retained, but it is only a class declaration, with no concrete implementation. It is small and does not materially impact the application size when retained after compilation.

Instead, LibHeaderComponent itself implements the abstract LibHeaderToken class. You can safely use that token as the provider in the component definition, allowing Angular to correctly inject the concrete type.

To summarize, the lightweight injection token pattern consists of the following.

1. A lightweight injection token that is represented as an abstract class.
2. A component definition that implements the abstract class.
3. Injection of the lightweight pattern, using @[ContentChild](https://angular.io/api/core/ContentChild)() or @[ContentChildren](https://angular.io/api/core/ContentChildren)().
4. A provider in the implementation of the lightweight injection token which associates the lightweight injection token with the implementation.

### **Use the lightweight injection token for API definition**

A component that injects a lightweight injection token might need to invoke a method in the injected class. Because the token is now an abstract class, and the injectable component implements that class, you must also declare an abstract method in the abstract lightweight injection token class. The implementation of the method (with all of its code overhead) resides in the injectable component that can be tree-shaken. This allows the parent to communicate with the child (if it is present) in a type-safe manner.

For example, the LibCardComponent now queries LibHeaderToken rather than LibHeaderComponent. The following example shows how the pattern allows LibCardComponent to communicate with the LibHeaderComponent without actually referring to LibHeaderComponent.

content\_copyabstract class LibHeaderToken {

abstract doSomething(): void;

}

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

selector: 'lib-header',

providers: [

{provide: LibHeaderToken, useExisting: LibHeaderComponent}

]

...,

})

class LibHeaderComponent extends LibHeaderToken {

doSomething(): void {

// Concrete implementation of `doSomething`

}

}

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

selector: 'lib-card',

...,

})

class LibCardComponent implement [AfterContentInit](https://angular.io/api/core/AfterContentInit) {

@[ContentChild](https://angular.io/api/core/ContentChild)(LibHeaderToken)

header: LibHeaderToken|null = null;

ngAfterContentInit(): void {

this.header && this.header.doSomething();

}

}

In this example the parent queries the token to obtain the child component, and stores the resulting component reference if it is present. Before calling a method in the child, the parent component checks to see if the child component is present. If the child component has been tree-shaken, there is no runtime reference to it, and no call to its method.

### **Naming your lightweight injection token**

Lightweight injection tokens are only useful with components. The Angular style guide suggests that you name components using the "Component" suffix. The example "LibHeaderComponent" follows this convention.

To maintain the relationship between the component and its token while still distinguishing between them, the recommended style is to use the component base name with the suffix "Token" to name your lightweight injection tokens: "LibHeaderToken".

# Browser support

Angular supports most recent browsers. This includes the following specific versions:

|  |  |
| --- | --- |
| **Browser** | **Supported versions** |
| Chrome | latest |
| Firefox | latest and extended support release (ESR) |
| Edge | 2 most recent major versions |
| IE | 11 \*deprecated, see the [*deprecations guide*](https://angular.io/guide/deprecations#internet-explorer-11) |
| Safari | 2 most recent major versions |
| iOS | 2 most recent major versions |
| Android | Q (10.0), Pie (9.0), Oreo (8.0), Nougat (7.0) |

Angular's continuous integration process runs unit tests of the framework on all of these browsers for every pull request, using [Sauce Labs](https://saucelabs.com/) and [BrowserStack](https://www.browserstack.com/).

## **Configuring Angular CLI for compatibility with IE11**

While Angular supports all browsers listed above, in order to improve the build times and output, Angular CLI applications don't support IE11 by default.

Angular CLI uses [browserlist](https://github.com/browserslist/browserslist) to configure browser support for applications.

You can enable the IE11 support by following the instructions in the .browserslistrc file at the root of your project.

## **Polyfills**

Angular is built on the latest standards of the web platform. Targeting such a wide range of browsers is challenging because they do not support all features of modern browsers. You compensate by loading polyfill scripts ("polyfills") for the browsers that you must support. The [table below](https://angular.io/guide/browser-support#polyfill-libs) identifies most of the polyfills you might need.

The suggested polyfills are the ones that run full Angular applications. You may need additional polyfills to support features not covered by this list. Note that polyfills cannot magically transform an old, slow browser into a modern, fast one.

In Angular CLI version 8 and higher, applications are built using differential loading, a strategy where the CLI builds two separate bundles as part of your deployed application.

* The first bundle contains modern ES2015 syntax, takes advantage of built-in support in modern browsers, ships less polyfills, and results in a smaller bundle size.
* The second bundle contains code in the old ES5 syntax, along with all necessary polyfills. This results in a larger bundle size, but supports older browsers.

This strategy allows you to continue to build your web application to support multiple browsers, but only load the necessary code that the browser needs. For more information about how this works, see [Differential Loading](https://angular.io/guide/deployment#differential-loading) in the [Deployment guide](https://angular.io/guide/deployment).

## **Enabling polyfills with CLI projects**

The [Angular CLI](https://angular.io/cli) provides support for polyfills. If you are not using the CLI to create your projects, see [Polyfill instructions for non-CLI users](https://angular.io/guide/browser-support#non-cli).

When you create a project with the ng new command, a src/polyfills.ts configuration file is created as part of your project folder. This file incorporates the mandatory and many of the optional polyfills as JavaScript import statements.

* The npm packages for the mandatory polyfills (such as zone.js) are installed automatically for you when you create your project with ng new, and their corresponding import statements are already enabled in the src/polyfills.ts configuration file.
* If you need an optional polyfill, you must install its npm package, then uncomment or create the corresponding import statement in the src/polyfills.ts configuration file.

For example, if you need the optional [web animations polyfill](https://caniuse.com/web-animation), you could install it with npm, using the following command (or the yarn equivalent):

content\_copy# install the optional web animations polyfill

npm install --save web-animations-js

You can then add the import statement in the src/polyfills.ts file. For many polyfills, you can un-comment the corresponding import statement in the file, as in the following example.

src/polyfills.ts

content\_copy/\*\*

\* Required to support Web Animations `@angular/platform-browser/animations`.

\* Needed for: All but Chrome, Firefox and Opera. https://caniuse.com/web-animation

\*\*/

import 'web-animations-js'; // Run `npm install --save web-animations-js`.

If the polyfill you want is not already in polyfills.ts file, add the import statement by hand.

### **Optional browser features to polyfill**

Some features of Angular may require additional polyfills.

|  |  |  |
| --- | --- | --- |
| **Feature** | **Polyfill** | **Browsers (Desktop & Mobile)** |
| [AnimationBuilder](https://angular.io/api/animations/AnimationBuilder) (Standard animation support does not require polyfills.) | [Web Animations](https://angular.io/guide/browser-support#web-animations) | If AnimationBuilder is used, enables scrubbing support for IE/Edge and Safari. (Chrome and Firefox support this natively). |
| [NgClass](https://angular.io/api/common/NgClass) on SVG elements | [classList](https://angular.io/guide/browser-support#classlist) | IE 11 |

### **Suggested polyfills**

The following polyfills are used to test the framework itself. They are a good starting point for an application.

|  |  |  |
| --- | --- | --- |
| **Polyfill** | **License** | **Size\*** |
| [classList](https://github.com/eligrey/classList.js) | Public domain | 1KB |
| [Web Animations](https://github.com/web-animations/web-animations-js) | Apache | 14.8KB |

\* Figures are for minified and gzipped code, computed with the [closure compiler](https://closure-compiler.appspot.com/home).

## **Polyfills for non-CLI users**

If you are not using the CLI, add your polyfill scripts directly to the host web page (index.html).

For example:

src/index.html

content\_copy<!-- pre-zone polyfills -->

<script src="node\_modules/core-js/client/shim.min.js"></script>

<script src="node\_modules/web-animations-js/web-animations.min.js"></script>

<script>

/\*\*

\* you can configure some zone flags which can disable zone interception for some

\* asynchronous activities to improve startup performance - use these options only

\* if you know what you are doing as it could result in hard to trace down bugs..

\*/

// \_\_Zone\_disable\_requestAnimationFrame = true; // disable patch requestAnimationFrame

// \_\_Zone\_disable\_on\_property = true; // disable patch onProperty such as onclick

// \_\_zone\_symbol\_\_UNPATCHED\_EVENTS = ['scroll', 'mousemove']; // disable patch specified eventNames

/\*

\* in IE/Edge developer tools, the addEventListener will also be wrapped by zone.js

\* with the following flag, it will bypass `zone.js` patch for IE/Edge

\*/

// \_\_Zone\_enable\_cross\_context\_check = true;

</script>

<!-- zone.js required by Angular -->

<script src="node\_modules/zone.js/bundles/zone.umd.js"></script>

<!-- application polyfills -->