



One framework.  
Mobile & desktop.

DOCS



## DEVELOP ACROSS ALL PLATFORMS

Learn one way to build applications with Angular and reuse your code and abilities to build apps for any deployment target. For web, mobile web, native mobile and native desktop.

All content was originally produced by the [Angular Team](#) under the "Creative Commons License, Attribution 4.0 International (CC BY 4.0)".  
[https://wiki.creativecommons.org/wiki/License\\_Versions#Detailed\\_attribution\\_comparison\\_chart](https://wiki.creativecommons.org/wiki/License_Versions#Detailed_attribution_comparison_chart)

No warranties are given. The license may not give you all of the permissions necessary for your intended use.

For example, other rights such as [publicity, privacy, or moral rights](#) may limit how you use the material.

You are free to:

Share – copy and redistribute the material in any medium or format

Adapt – remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Angular is a platform and framework for building client applications in HTML and TypeScript. Angular is written in TypeScript. It implements core and optional functionality as a set of TypeScript libraries that you import into your apps.

The basic building blocks of an Angular application are NgModules, which provide a compilation context for components. NgModules collect related code into functional sets; an Angular app is defined by a set of NgModules. An app always has at least a root module that enables bootstrapping, and typically has many more feature modules.

- Components define views, which are sets of screen elements that Angular can choose among and modify according to your program logic and data.
- Components use services, which provide specific functionality not directly related to views. Service providers can be injected into components as dependencies, making your code modular, reusable, and efficient.

Both components and services are simply classes, with decorators that mark their type and provide metadata that tells Angular how to use them.

- The metadata for a component class associates it with a template that defines a view. A template combines ordinary HTML with Angular directives and binding markup that allow Angular to modify the HTML before rendering it for display.
- The metadata for a service class provides the information Angular needs to make it available to components through dependency injection (DI).

An app's components typically define many views, arranged hierarchically. Angular provides the [Router](#) service to help you define navigation paths among views. The router provides sophisticated in-browser navigational capabilities.

## Modules

Angular NgModules differ from and complement JavaScript (ES2015) modules. An NgModule declares a compilation context for a set of components that is dedicated to an application domain, a workflow, or a closely related set of capabilities. An NgModule can associate its components with related code, such as services, to form functional units.

Every Angular app has a root module, conventionally named AppModule, which provides the bootstrap mechanism that launches the application. An app typically contains many functional modules.

Like JavaScript modules, NgModules can import functionality from other NgModules, and allow their own functionality to be exported and used by other NgModules. For example, to use the router service in your app, you import the [RouterModule](#).

Organizing your code into distinct functional modules helps in managing development of complex applications, and in designing for reusability. In addition, this technique lets you take advantage of lazy-loading—that is, loading modules on demand—to minimize the amount of code that needs to be loaded at startup.

For a more detailed discussion, see [Introduction to modules](#).

## Components

Every Angular application has at least one component, the root component that connects a component hierarchy with the page document object model (DOM). Each component defines a class that contains application data and logic, and is associated with an HTML template that defines a view to be displayed in a target environment.

The `@Component()` decorator identifies the class immediately below it as a component, and provides the template and related component-specific metadata.

Decorators are functions that modify JavaScript classes. Angular defines a number of decorators that attach specific kinds of metadata to classes, so that the system knows what those classes mean and how they should work.

[Learn more about decorators on the web.](#)

## Templates, directives, and data binding

A template combines HTML with Angular markup that can modify HTML elements before they are displayed. Template directives provide program logic, and binding markup connects your application data and the DOM. There are two types of data binding:

- Event binding lets your app respond to user input in the target environment by updating your application data.
- Property binding lets you interpolate values that are computed from your application data into the HTML.

Before a view is displayed, Angular evaluates the directives and resolves the binding syntax in the template to modify the HTML elements and the DOM, according to your program data and logic. Angular supports two-way data binding, meaning that changes in the DOM, such as user choices, are also reflected in your program data.

Your templates can use pipes to improve the user experience by transforming values for display. For example, use pipes to display dates and currency values that are appropriate for a user's locale. Angular provides predefined pipes for common transformations, and you can also define your own pipes.

For a more detailed discussion of these concepts, see [Introduction to components](#).

## Services and dependency injection

For data or logic that isn't associated with a specific view, and that you want to share across components, you create a service class. A service class definition is immediately preceded by the `@Injectable()` decorator. The decorator provides the metadata that allows your service to be injected into client components as a dependency.

Dependency injection (DI) lets you keep your component classes lean and efficient. They don't fetch data from the server, validate user input, or log directly to the console; they delegate such tasks to services.

For a more detailed discussion, see [Introduction to services and DI](#).

## Routing

The Angular [Router](#) NgModule provides a service that lets you define a navigation path among the different application states and view hierarchies in your app. It is modeled on the familiar browser navigation conventions:

- Enter a URL in the address bar and the browser navigates to a corresponding page.
- Click links on the page and the browser navigates to a new page.
- Click the browser's back and forward buttons and the browser navigates backward and forward through the history of pages you've seen.

The router maps URL-like paths to views instead of pages. When a user performs an action, such as clicking a link, that would load a new page in the browser, the router intercepts the browser's behavior, and shows or hides view hierarchies.

If the router determines that the current application state requires particular functionality, and the module that defines it hasn't been loaded, the router can lazy-load the module on demand.

The router interprets a link URL according to your app's view navigation rules and data state. You can navigate to new views when the user clicks a button or selects from a drop box, or in response to some other stimulus from any source. The router logs activity in the browser's history, so the back and forward buttons work as well.

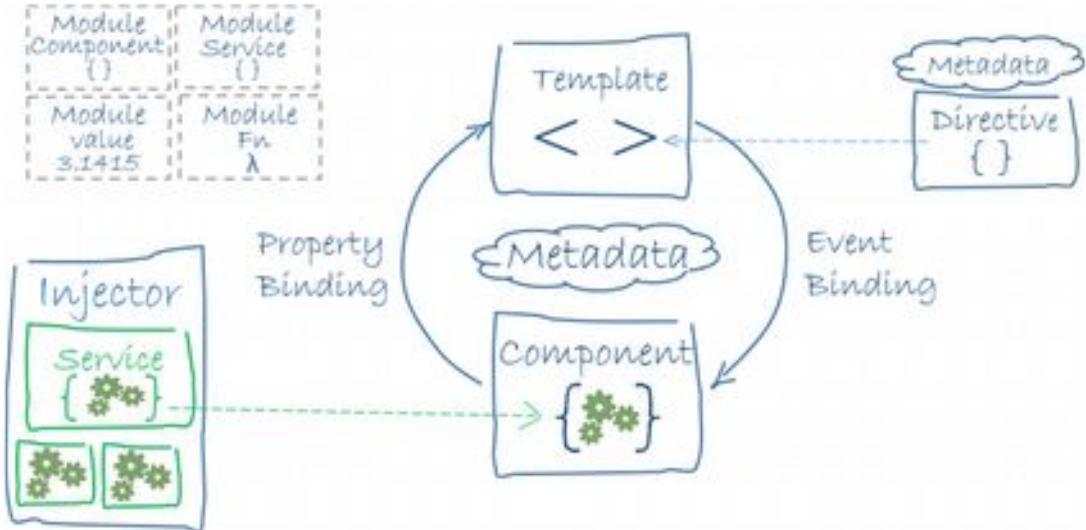
To define navigation rules, you associate navigation paths with your components. A path uses a URL-like syntax that integrates your program data, in much the same way that template syntax integrates your views with your program data. You can then apply program logic to choose which views to show or to hide, in response to user input and your own access rules.

For a more detailed discussion, see [Routing and navigation](#).

---

## What's next

You've learned the basics about the main building blocks of an Angular application. The following diagram shows how these basic pieces are related.



- Together, a component and template define an Angular view.
- A decorator on a component class adds the metadata, including a pointer to the associated template.
- Directives and binding markup in a component's template modify views based on program data and logic.
- The dependency injector provides services to a component, such as the router service that lets you define navigation among views.

Each of these subjects is introduced in more detail in the following pages.

- [Introduction to Modules](#)
- [Introduction to Components](#)
- [Templates and views](#)
- [Component metadata](#)
- [Data binding](#)
- [Directives](#)
- [Pipes](#)
- [Introduction to services and dependency injection](#)

Note that the code referenced on these pages is available as a [live example / download example](#).

When you're familiar with these fundamental building blocks, you can explore them in more detail in the documentation. To learn about more tools and techniques that are available to help you build and deploy Angular applications, see [Next steps: tools and techniques](#).

Angular apps are modular and Angular has its own modularity system called NgModules. NgModules are containers for a cohesive block of code dedicated to an application domain, a workflow, or a closely related set of capabilities. They can contain components, service providers, and other code files whose scope is defined by the containing NgModule. They can

import functionality that is exported from other NgModules, and export selected functionality for use by other NgModules.

Every Angular app has at least one NgModule class, [the root module](#), which is conventionally named AppModule and resides in a file named app.module.ts. You launch your app by bootstrapping the root NgModule.

While a small application might have only one NgModule, most apps have many more feature modules. The root NgModule for an app is so named because it can include child NgModules in a hierarchy of any depth.

## NgModule metadata

An NgModule is defined by a class decorated with [@NgModule\(\)](#). The [@NgModule\(\)](#) decorator is a function that takes a single metadata object, whose properties describe the module. The most important properties are as follows.

- [declarations](#): The [components](#), directives, and pipes that belong to this NgModule.
- [exports](#): The subset of declarations that should be visible and usable in the component templates of other NgModules.
- [imports](#): Other modules whose exported classes are needed by component templates declared in this NgModule.
- [providers](#): Creators of [services](#) that this NgModule contributes to the global collection of services; they become accessible in all parts of the app. (You can also specify providers at the component level, which is often preferred.)
- [bootstrap](#): The main application view, called the root component, which hosts all other app views. Only the root NgModule should set the [bootstrap](#) property.

Here's a simple root NgModule definition.

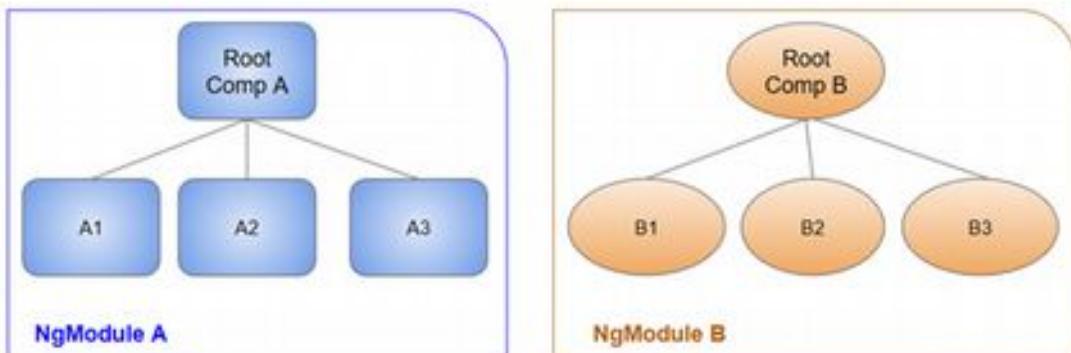
src/app/app.module.ts

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
@NgModule({
  imports:      [ BrowserModule ],
  providers:    [ Logger ],
  declarations: [ AppComponent ],
  exports:      [ AppComponent ],
  bootstrap:   [ AppComponent ]
})
export class AppModule { }
```

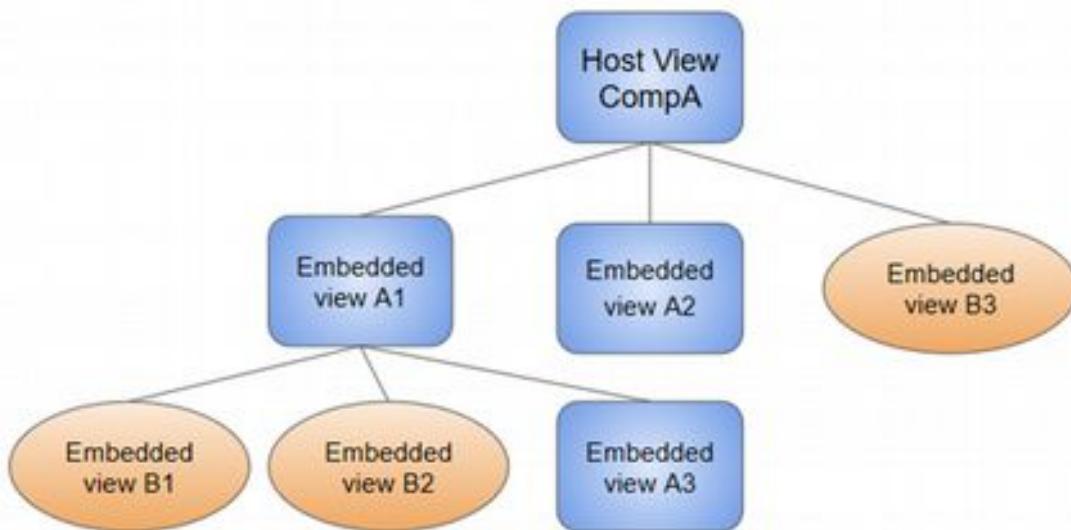
The export property of AppComponent is included here for illustration; it isn't actually necessary in this example. A root NgModule has no reason to export anything because other modules don't need to import the root NgModule.

## NgModules and components

NgModules provide a compilation context for their components. A root NgModule always has a root component that is created during bootstrap, but any NgModule can include any number of additional components, which can be loaded through the router or created through the template. The components that belong to an NgModule share a compilation context.



A component and its template together define a view. A component can contain a view hierarchy, which allows you to define arbitrarily complex areas of the screen that can be created, modified, and destroyed as a unit. A view hierarchy can mix views defined in components that belong to different NgModules. This is often the case, especially for UI libraries.



When you create a component, it's associated directly with a single view, called the host view. The host view can be the root of a view hierarchy, which can contain embedded views, which are in turn the host views of other components. Those components can be in the same NgModule, or can be imported from other NgModules. Views in the tree can be nested to any depth.

**\*\*Note:\*\*** The hierarchical structure of views is a key factor in the way Angular detects and responds to changes in the DOM and app data.

## NgModules and JavaScript modules

The NgModule system is different from and unrelated to the JavaScript (ES2015) module system for managing collections of JavaScript objects. These are complementary module systems that you can use together to write your apps.

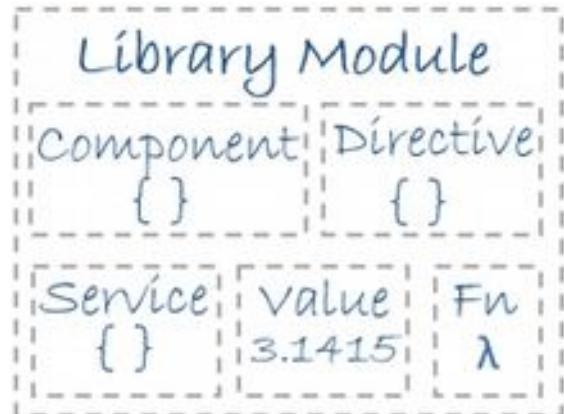
In JavaScript each file is a module and all objects defined in the file belong to that module. The module declares some objects to be public by marking them with the `export` key word. Other JavaScript modules use import statements to access public objects from other modules.

```
import { NgModule }      from '@angular/core';
import { AppComponent } from './app.component';

export class AppModule { }
```

[Learn more about the JavaScript module system on the web.](#)

## Angular libraries



Angular loads as a collection of JavaScript modules. You can think of them as library modules. Each Angular library name begins with the `@angular` prefix. Install them with the npm package manager and import parts of them with JavaScript import statements.

For example, import Angular's [Component](#) decorator from the `@angular/core` library like this.

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

You also import NgModules from Angular libraries using JavaScript import statements. For example, the following code imports the [BrowserModule](#) NgModule from the platform-browser library.

```
import { BrowserModule } from '@angular/platform-browser';
```

In the example of the simple root module above, the application module needs material from within [BrowserModule](#). To access that material, add it to the [@NgModule](#) metadata [imports](#) like this.

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

In this way you're using the Angular and JavaScript module systems together. Although it's easy to confuse the two systems, which share the common vocabulary of "imports" and "exports", you will become familiar with the different contexts in which they are used.

Learn more from the [NgModules](#) guide.

A component controls a patch of screen called a view. For example, individual components define and control each of the following views from the [Tutorial](#):

- The app root with the navigation links.
- The list of heroes.
- The hero editor.

You define a component's application logic—what it does to support the view—inside a class. The class interacts with the view through an API of properties and methods.

For example, HeroListComponent has a heroes property that holds an array of heroes. Its selectHero() method sets a selectedHero property when the user clicks to choose a hero from that list. The component acquires the heroes from a service, which is a TypeScript [parameter property](#) on the constructor. The service is provided to the component through the dependency injection system.

```
src/app/hero-list.component.ts (class)

export class HeroListComponent implements OnInit {
  heroes: Hero[];
  selectedHero: Hero;

  constructor(private service: HeroService) {}

  ngOnInit() {
    this.heroes = this.service.getHeroes();
  }

  selectHero(hero: Hero) { this.selectedHero = hero; }
}
```

Angular creates, updates, and destroys components as the user moves through the application. Your app can take action at each moment in this lifecycle through optional [lifecycle hooks](#), like ngOnInit().

## Component metadata



The [@Component](#) decorator identifies the class immediately below it as a component class, and specifies its metadata. In the example code below, you can see that `HeroListComponent` is just a class, with no special Angular notation or syntax at all. It's not a component until you mark it as one with the [@Component](#) decorator.

The metadata for a component tells Angular where to get the major building blocks that it needs to create and present the component and its view. In particular, it associates a template with the component, either directly with inline code, or by reference. Together, the component and its template describe a view.

In addition to containing or pointing to the template, the [@Component](#) metadata configures, for example, how the component can be referenced in HTML and what services it requires.

Here's an example of basic metadata for `HeroListComponent`.

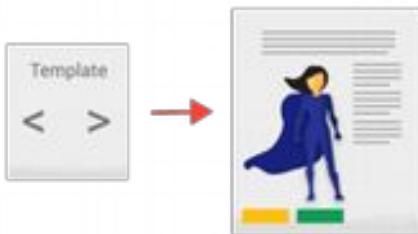
```
src/app/hero-list.component.ts (metadata)
```

```
@Component({
  selector: 'app-hero-list',
  templateUrl: './hero-list.component.html',
  providers: [ HeroService ]
})
export class HeroListComponent implements OnInit {
  /* . . . */
}
```

This example shows some of the most useful [@Component](#) configuration options:

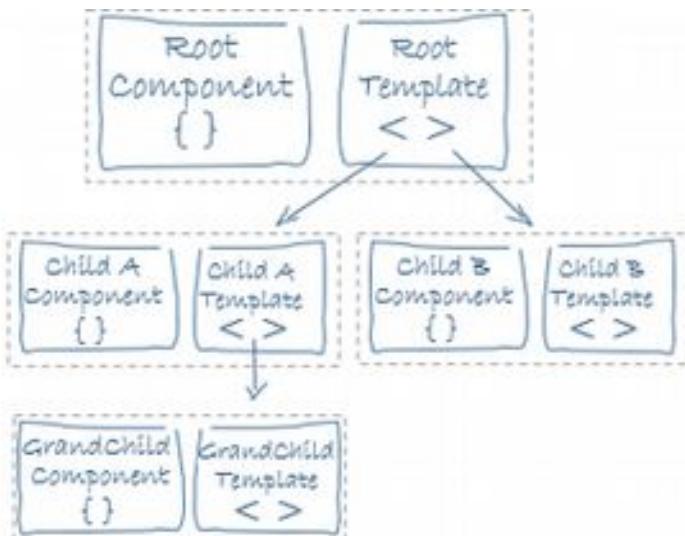
- **selector:** A CSS selector that tells Angular to create and insert an instance of this component wherever it finds the corresponding tag in template HTML. For example, if an app's HTML contains `<app-hero-list></app-hero-list>`, then Angular inserts an instance of the `HeroListComponent` view between those tags.
- **templateUrl:** The module-relative address of this component's HTML template. Alternatively, you can provide the HTML template inline, as the value of the [template](#) property. This template defines the component's host view.
- **providers:** An array of [providers](#) for services that the component requires. In the example, this tells Angular how to provide the `HeroService` instance that the component's constructor uses to get the list of heroes to display.

## Templates and views



You define a component's view with its companion template. A template is a form of HTML that tells Angular how to render the component.

Views are typically arranged hierarchically, allowing you to modify or show and hide entire UI sections or pages as a unit. The template immediately associated with a component defines that component's host view. The component can also define a view hierarchy, which contains embedded views, hosted by other components.



A view hierarchy can include views from components in the same NgModule, but it also can (and often does) include views from components that are defined in different NgModules.

## Template syntax

A template looks like regular HTML, except that it also contains Angular [template syntax](#), which alters the HTML based on your app's logic and the state of app and DOM data. Your template can use data binding to coordinate the app and DOM data, pipes to transform data before it is displayed, and directives to apply app logic to what gets displayed.

For example, here is a template for the Tutorial's HeroListComponent.

src/app/hero-list.component.html

```
<h2>Hero List</h2>
```

```
<p><i>Pick a hero from the list</i></p>
```

```
<ul>
<li *ngFor="let hero of heroes" (click)="selectHero(hero)">
  {{hero.name}}
</li>
</ul>
```

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

This template uses typical HTML elements like `<h2>` and `<p>`, and also includes Angular template-syntax elements, `*ngFor`, `{{hero.name}}`, `(click)`, `[hero]`, and `<app-hero-detail>`. The template-syntax elements tell Angular how to render the HTML to the screen, using program logic and data.

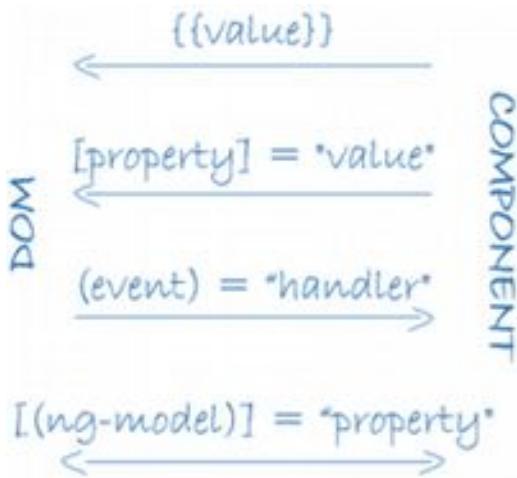
- The `*ngFor` directive tells Angular to iterate over a list.
- `{{hero.name}}`, `(click)`, and `[hero]` bind program data to and from the DOM, responding to user input. See more about [data binding](#) below.
- The `<app-hero-detail>` tag in the example is an element that represents a new component, `HeroDetailComponent`.
- `HeroDetailComponent` (code not shown) defines the hero-detail child view of `HeroListComponent`. Notice how custom components like this mix seamlessly with native HTML in the same layouts.

## Data binding

Without a framework, you would be responsible for pushing data values into the HTML controls and turning user responses into actions and value updates. Writing such push and pull logic by hand is tedious, error-prone, and a nightmare to read, as any experienced jQuery programmer can attest.

Angular supports two-way data binding, a mechanism for coordinating the parts of a template with the parts of a component. Add binding markup to the template HTML to tell Angular how to connect both sides.

The following diagram shows the four forms of data binding markup. Each form has a direction: to the DOM, from the DOM, or both.



This example from the HeroListComponent template uses three of these forms.

src/app/hero-list.component.html (binding)

```
<li>{{hero.name}}</li>
<app-hero-detail [hero]="selectedHero"></app-hero-detail>
<li (click)="selectHero(hero)"></li>
```

- The `{{hero.name}}` [interpolation](#) displays the component's `hero.name` property value within the `<li>` element.
- The `[hero]` [property binding](#) passes the value of `selectedHero` from the parent `HeroListComponent` to the `hero` property of the child `HeroDetailComponent`.
- The `(click)` [event binding](#) calls the component's `selectHero` method when the user clicks a hero's name.

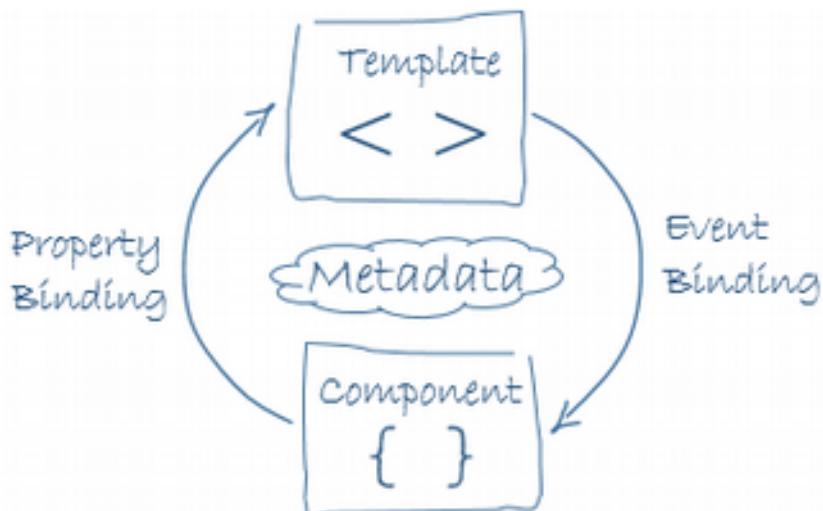
Two-way data binding (used mainly in [template-driven forms](#)) combines property and event binding in a single notation. Here's an example from the `HeroDetailComponent` template that uses two-way data binding with the `ngModel` directive.

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

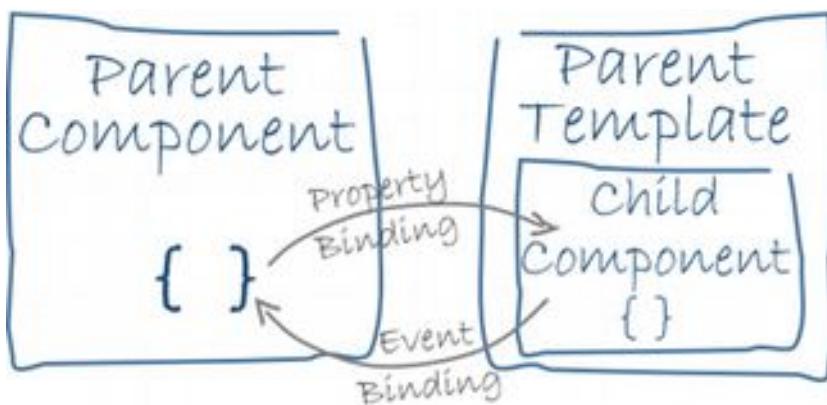
```
<input [(ngModel)]="hero.name">
```

In two-way binding, a data property value flows to the input box from the component as with property binding. The user's changes also flow back to the component, resetting the property to the latest value, as with event binding.

Angular processes all data bindings once for each JavaScript event cycle, from the root of the application component tree through all child components.



Data binding plays an important role in communication between a template and its component, and is also important for communication between parent and child components.



## Pipes

Angular pipes let you declare display-value transformations in your template HTML. A class with the [@Pipe](#) decorator defines a function that transforms input values to output values for display in a view.

Angular defines various pipes, such as the [date](#) pipe and [currency](#) pipe; for a complete list, see the [Pipes API list](#). You can also define new pipes.

To specify a value transformation in an HTML template, use the [pipe operator \(|\)](#).

```
 {{interpolated_value | pipe_name}}
```

You can chain pipes, sending the output of one pipe function to be transformed by another pipe function. A pipe can also take arguments that control how it performs its transformation. For example, you can pass the desired format to the datepipe.

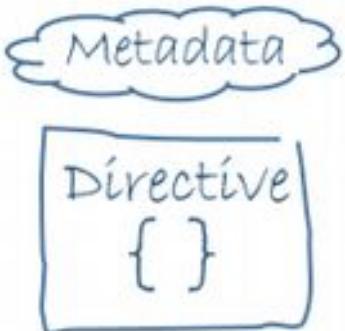
```
<!-- Default format: output 'Jun 15, 2015'-->
```

```
<p>Today is {{today | date}}</p>
```

```
<!-- fullDate format: output 'Monday, June 15, 2015'-->
```

```
<p>The date is {{today | date:'fullDate'}}</p>  
  
<!-- shortTime format: output '9:43 AM'-->  
<p>The time is {{today | date:'shortTime'}}</p>
```

## Directives



Angular templates are dynamic. When Angular renders them, it transforms the DOM according to the instructions given by directives. A directive is a class with a [@Directive\(\)](#) decorator.

A component is technically a directive. However, components are so distinctive and central to Angular applications that Angular defines the [@Component\(\)](#) decorator, which extends the [@Directive\(\)](#) decorator with template-oriented features.

In addition to components, there are two other kinds of directives: structural and attribute. Angular defines a number of directives of both kinds, and you can define your own using the [@Directive\(\)](#) decorator.

Just as for components, the metadata for a directive associates the decorated class with a selector element that you use to insert it into HTML. In templates, directives typically appear within an element tag as attributes, either by name or as the target of an assignment or a binding.

### Structural directives

Structural directives alter layout by adding, removing, and replacing elements in the DOM. The example template uses two built-in structural directives to add application logic to how the view is rendered.

src/app/hero-list.component.html (structural)

```
<li *ngFor="let hero of heroes"></li>  
<app-hero-detail *ngIf="selectedHero"></app-hero-detail>
```

- [\\*ngFor](#) is an iterative; it tells Angular to stamp out one `<li>` per hero in the heroes list.
- [\\*ngIf](#) is a conditional; it includes the HeroDetail component only if a selected hero exists.

## Attribute directives

Attribute directives alter the appearance or behavior of an existing element. In templates they look like regular HTML attributes, hence the name.

The [ngModel](#) directive, which implements two-way data binding, is an example of an attribute directive. [ngModel](#) modifies the behavior of an existing element (typically `<input>`) by setting its `display value` property and responding to change events.

```
src/app/hero-detail.component.html (ngModel)
```

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

Angular has more pre-defined directives that either alter the layout structure (for example, [ngSwitch](#)) or modify aspects of DOM elements and components (for example, [ngStyle](#) and [ngClass](#)).

Learn more in the [Attribute Directives](#) and [Structural Directives](#) guides.

Service is a broad category encompassing any value, function, or feature that an app needs. A service is typically a class with a narrow, well-defined purpose. It should do something specific and do it well.

Angular distinguishes components from services to increase modularity and reusability. By separating a component's view-related functionality from other kinds of processing, you can make your component classes lean and efficient.

Ideally, a component's job is to enable the user experience and nothing more. A component should present properties and methods for data binding, in order to mediate between the view (rendered by the template) and the application logic (which often includes some notion of a model).

A component can delegate certain tasks to services, such as fetching data from the server, validating user input, or logging directly to the console. By defining such processing tasks in an injectable service class, you make those tasks available to any component. You can also make your app more adaptable by injecting different providers of the same kind of service, as appropriate in different circumstances.

Angular doesn't enforce these principles. Angular does help you follow these principles by making it easy to factor your application logic into services and make those services available to components through dependency injection.

## Service examples

Here's an example of a service class that logs to the browser console.

```
src/app/logger.service.ts (class)
```

```
export class Logger {  
  log(msg: any) { console.log(msg); }  
}
```

```
error(msg: any) { console.error(msg); }
warn(msg: any) { console.warn(msg); }
}
```

Services can depend on other services. For example, here's a HeroService that depends on the Logger service, and also uses BackendService to get heroes. That service in turn might depend on the [HttpClient](#) service to fetch heroes asynchronously from a server.

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

```
export class HeroService {
  private heroes: Hero[] = [];

  constructor(
    private backend: BackendService,
    private logger: Logger) { }

  getHeroes() {
    this.backend.getAll(Hero).then( (heroes: Hero[]) => {
      this.logger.log(`Fetched ${heroes.length} heroes.`);
      this.heroes.push(...heroes); // fill cache
    });
    return this.heroes;
  }
}
```

## Dependency injection (DI)



DI is wired into the Angular framework and used everywhere to provide new components with the services or other things they need. Components consume services; that is, you can inject a service into a component, giving the component access to that service class.

To define a class as a service in Angular, use the [@Injectable\(\)](#) decorator to provide the metadata that allows Angular to inject it into a component as a dependency.

Similarly, use the [@Injectable\(\)](#) decorator to indicate that a component or other class (such as another service, a pipe, or an NgModule) has a dependency.

- The injector is the main mechanism. Angular creates an application-wide injector for you during the bootstrap process, and additional injectors as needed. You don't have to create injectors.

- An injector creates dependencies, and maintains a container of dependency instances that it reuses if possible.
- A provider is an object that tell an injector how to obtain or create a dependency.

For any dependency that you need in your app, you must register a provider with the app's injector, so that the injector can use the provider to create new instances. For a service, the provider is typically the service class itself.

A dependency doesn't have to be a service—it could be a function, for example, or a value.

When Angular creates a new instance of a component class, it determines which services or other dependencies that component needs by looking at the constructor parameter types. For example, the constructor of HeroListComponent needs HeroService.

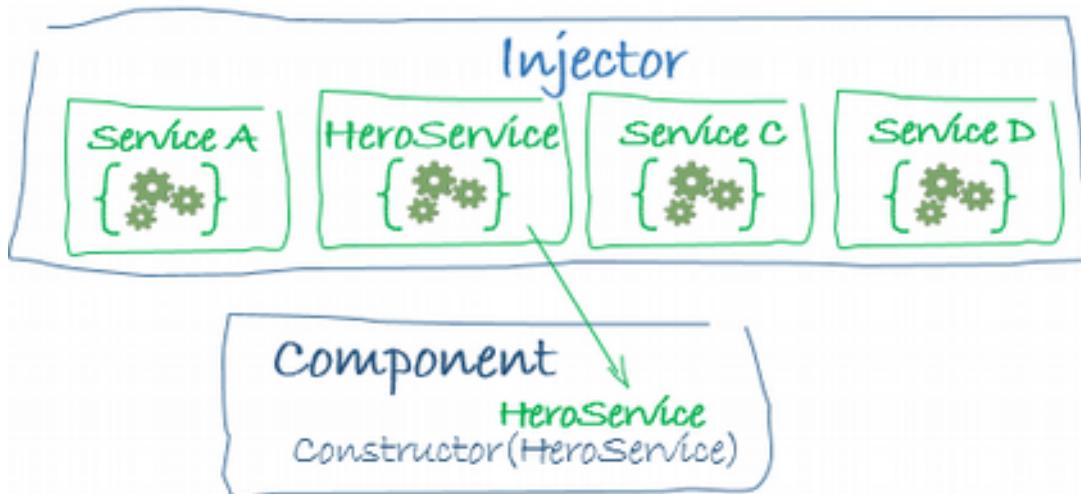
`src/app/hero-list.component.ts (constructor)`

```
constructor(private service: HeroService) { }
```

When Angular discovers that a component depends on a service, it first checks if the injector has any existing instances of that service. If a requested service instance doesn't yet exist, the injector makes one using the registered provider, and adds it to the injector before returning the service to Angular.

When all requested services have been resolved and returned, Angular can call the component's constructor with those services as arguments.

The process of HeroService injection looks something like this.



## Providing services

You must register at least one provider of any service you are going to use. The provider can be part of the service's own metadata, making that service available everywhere, or you can register providers with specific modules or components. You register providers in the metadata of the service (in the `@Injectable()` decorator), or in the `@NgModule()` or `@Component()` metadata

- By default, the Angular CLI command `ng generate service` registers a provider with the root injector for your service by including provider metadata in the `@Injectable()` decorator. The tutorial uses this method to register the provider of `HeroService` class definition.
- ```
@Injectable({
  providedIn: 'root',
})
```
- When you provide the service at the root level, Angular creates a single, shared instance of `HeroService` and injects it into any class that asks for it. Registering the provider in the `@Injectable()` metadata also allows Angular to optimize an app by removing the service from the compiled app if it isn't used.
- When you register a provider with a [specific NgModule](#), the same instance of a service is available to all components in that NgModule. To register at this level, use the `providers` property of the `@NgModule()` decorator,
- ```
@NgModule({
  providers: [
    BackendService,
    Logger
  ],
  ...
})
```
- When you register a provider at the component level, you get a new instance of the service with each new instance of that component. At the component level, register a service provider in the `providers` property of the `@Component()` metadata.
- `src/app/hero-list.component.ts` (component providers)
- ```
@Component({
  selector: 'app-hero-list',
  templateUrl: './hero-list.component.html',
  providers: [ HeroService ]
})
```

For more detailed information, see the [Dependency Injection](#) section.

After you understand the basic Angular building blocks, you can begin to learn more about the features and tools that are available to help you develop and deliver Angular applications. Here are some key features.

## Responsive programming tools

- [Lifecycle hooks](#): Tap into key moments in the lifetime of a component, from its creation to its destruction, by implementing the lifecycle hook interfaces.
- [Observables and event processing](#): How to use observables with components and services to publish and subscribe to messages of any type, such as user-interaction events and asynchronous operation results.

## Client-server interaction tools

- [HTTP](#): Communicate with a server to get data, save data, and invoke server-side actions with an HTTP client.
- [Server-side Rendering](#): Angular Universal generates static application pages on the server through server-side rendering (SSR). This allows you to run your Angular app on the server in order to improve performance and show the first page quickly on mobile and low-powered devices, and also facilitate web crawlers.
- [Service Workers](#): Use a service worker to reduce dependency on the network significantly improving the user experience.

## Domain-specific libraries

- [Animations](#): Use Angular's animation library to animate component behavior without deep knowledge of animation techniques or CSS.
- [Forms](#): Support complex data entry scenarios with HTML-based validation and dirty checking.

## Support for the development cycle

- [Testing platform](#): Run unit tests on your application parts as they interact with the Angular framework.
- [Internationalization](#): Make your app available in multiple languages with Angular's internationalization (i18n) tools.
- [Compilation](#): Angular provides just-in-time (JIT) compilation for the development environment, and ahead-of-time (AOT) compilation for the production environment.
- [Security guidelines](#): Learn about Angular's built-in protections against common web-app vulnerabilities and attacks such as cross-site scripting attacks.

## Setup and deployment tools

- [Setup for local development](#): Set up a new project for development with QuickStart.
- [Installation](#): The [Angular CLI](#), Angular applications, and Angular itself depend on features and functionality provided by libraries that are available as [npm](#) packages.
- [TypeScript configuration](#): TypeScript is the primary language for Angular application development.
- [Browser support](#): Make your apps compatible across a wide range of browsers.
- [Deployment](#): Learn techniques for deploying your Angular application to a remote server.

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

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

The final UI looks like this:



The [live example](#) / [download example](#) demonstrates all of the syntax and code snippets described in this page.

## Showing component properties with interpolation

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

Follow the [Getting Started](#) instructions for creating a new project named displaying-data.

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

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

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

src/app/app.component.ts

```
1. import { Component } from '@angular/core';
2.
3. @Component({
4.   selector: 'app-root',
5.   template: `
6.     <h1>{{title}}</h1>
7.     <h2>My favorite hero is: {{myHero}}</h2>
8.   `
9. })
```

```
10.export class AppComponent {  
11. title = 'Tour of Heroes';  
12. myHero = 'Windstorm';  
13.}
```

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

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

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

```
template: `  
<h1>{{title}}</h1>  
<h2>My favorite hero is: {{myHero}}</h2>  
`
```

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

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

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

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

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

src/index.html (body)

```
<body>  
<app-root></app-root>  
</body>
```

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

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

# Tour of Heroes

## My favorite hero is: Windstorm

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

### Template inline or template file?

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

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

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

By default, the Angular CLI generates components with a template file. You can override that with:

```
ng generate component hero -it
```

### Constructor or variable initialization?

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

```
export class AppCtorComponent {  
  title: string;  
  myHero: string;  
  
  constructor() {  
    this.title = 'Tour of Heroes';  
    this.myHero = 'Windstorm';  
  }  
}
```

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

## Showing an array property with \*ngFor

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

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

```
export class AppComponent {  
  title = 'Tour of Heroes';  
  heroes = ['Windstorm', 'Bombasto', 'Magneta', 'Tornado'];  
  myHero = this.heroes[0];  
}
```

Now use the Angular [ngFor](#) directive in the template to display each item in the heroes list.

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

```
template: `  
<h1>{{title}}</h1>  
<h2>My favorite hero is: {{myHero}}</h2>  
<p>Heroes:</p>  
<ul>  
  <li *ngFor="let hero of heroes">  
    {{ hero }}  
  </li>  
</ul>  
`
```

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

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

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

Don't forget the leading asterisk (\*) in [\\*ngFor](#). It is an essential part of the syntax. For more information, see the [Template Syntax](#) page.

Notice the hero in the [ngFor](#) double-quoted instruction; it is an example of a template input variable. Read more about template input variables in the [microsyntax](#) section of the [Template Syntax](#) page.

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

In this case, `ngFor` is displaying an array, but `ngFor` can repeat items for any `iterable` object. Now the heroes appear in an unordered list.

# Tour of Heroes

My favorite hero is: Windstorm

Heroes:

- Windstorm
- Bombasto
- Magneta
- Tornado

## Creating a class for the data

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

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

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

ng generate class hero

With the following code:

```
src/app/hero.ts
```

```
export class Hero {  
  constructor(  
    public id: number,  
    public name: string) {}  
}
```

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

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

Consider the first parameter:

```
src/app/hero.ts (id)
```

```
public id: number,
```

That brief syntax does a lot:

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

## Using the Hero class

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

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

```
heroes = [  
  new Hero(1, 'Windstorm'),  
  new Hero(13, 'Bombasto'),  
  new Hero(15, 'Magneta'),  
  new Hero(20, 'Tornado')  
];  
myHero = this.heroes[0];
```

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

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

```
template: `  
<h1>{{title}}</h1>  
<h2>My favorite hero is: {{myHero.name}}</h2>  
<p>Heroes:</p>  
<ul>  
  <li *ngFor="let hero of heroes">  
    {{ hero.name }}  
  </li>  
</ul>
```

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

## Conditional display with NgIf

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

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

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

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

```
<p *ngIf="heroes.length > 3">There are many heroes!</p>
```

Don't forget the leading asterisk (\*) in [\\*ngIf](#). It is an essential part of the syntax. Read more about [ngIf](#) and \* in the [ngIf section](#) of the [Template Syntax](#) page.

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

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

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

## Summary

Now you know how to use:

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

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

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

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

Many code snippets illustrate the points and concepts, all of them available in the [Template Syntax Live Code / download example](#).

## HTML in templates

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

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

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

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

---

## Interpolation ( `{{...}}` )

You met the double-curly braces of interpolation, `{{` and `}`}, early in your Angular education.

src/app/app.component.html

```
<p>My current hero is {{currentHero.name}}</p>
```

You use interpolation to weave calculated strings into the text between HTML element tags and within attribute assignments.

src/app/app.component.html

```
<h3>
  {{title}}
  
</h3>
```

The text between the braces is often the name of a component property. Angular replaces that name with the string value of the corresponding component property. In the example above, Angular evaluates the title and heroImageUrl properties and "fills in the blanks", first displaying a bold application title and then a heroic image.

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

src/app/app.component.html

```
<!-- "The sum of 1 + 1 is 2" -->
<p>The sum of 1 + 1 is {{1 + 1}}</p>
```

The expression can invoke methods of the host component such as `getVal()`, seen here:

`src/app/app.component.html`

```
<!-- "The sum of 1 + 1 is not 4" -->
<p>The sum of 1 + 1 is not {{1 + 1 + getVal()}}</p>
```

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

You appear to be inserting the result between element tags and assigning it to attributes. It's convenient to think so, and you rarely suffer for this mistake. Though this is not exactly true. Interpolation is a special syntax that Angular converts into a [property binding](#), as is explained [below](#).

But first, let's take a closer look at template expressions and statements.

---

## Template expressions

A template expression produces a value. Angular executes the expression and assigns it to a property of a binding target; the target might be an HTML element, a component, or a directive.

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

You write these template expressions in a language that looks like JavaScript. Many JavaScript expressions are legal template expressions, but not all.

JavaScript expressions that have or promote side effects are prohibited, including:

- assignments (`=, +=, -=, ...`)
- `new`
- chaining expressions with `;` or `,`
- increment and decrement operators (`++` and `--`)

Other notable differences from JavaScript syntax include:

- no support for the bitwise operators `|` and `&`
- new [template expression operators](#), such as `|, ?, and !`.

## Expression context

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

```
src/app/app.component.html
```

```
  {{title}}  
  <span [hidden]="isUnchanged">changed</span>
```

An expression may also refer to properties of the template's context such as a [template input variable](#) (let hero) or a [template reference variable](#) (#heroInput).

```
src/app/app.component.html
```

```
<div *ngFor="let hero of heroes">{{hero.name}}</div>  
<input #heroInput> {{heroInput.value}}
```

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

The previous example presents such a name collision. The component has a hero property and the `*ngFor` defines a herotemplate variable. The hero in `{{hero.name}}` refers to the template input variable, not the component's property.

Template expressions cannot refer to anything in the global namespace (except undefined). They can't refer to window or [document](#). They can't call console.log or Math.max. They are restricted to referencing members of the expression context.

## Expression guidelines

Template expressions can make or break an application. Please follow these guidelines:

- [No visible side effects](#)
- [Quick execution](#)
- [Simplicity](#)
- [Idempotence](#)

The only exceptions to these guidelines should be in specific circumstances that you thoroughly understand.

### No visible side effects

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

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

## Quick execution

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

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

## Simplicity

Although it's possible to write quite complex template expressions, you should avoid them.

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

## Idempotence

An [idempotent](#) expression is ideal because it is free of side effects and improves Angular's change detection performance.

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

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

---

## Template statements

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

src/app/app.component.html

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

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

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

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

However, certain JavaScript syntax is not allowed:

- new
- increment and decrement operators, ++ and --
- operator assignment, such as += and -=
- the bitwise operators | and &
- the [template expression operators](#)

## Statement context

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

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

src/app/app.component.html

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

The statement context may also refer to properties of the template's own context. In the following examples, the template \$event object, a [template input variable](#) (let hero), and a [template reference variable](#) (#heroForm) are passed to an event handling method of the component.

src/app/app.component.html

```
<button (click)="onSave($event)">Save</button>
<button *ngFor="let hero of heroes" (click)="deleteHero(hero)">{{hero.name}}</button>
<form #heroForm (ngSubmit)="onSubmit(heroForm)"> ... </form>
```

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

Template statements cannot refer to anything in the global namespace. They can't refer to window or [document](#). They can't call console.log or Math.max.

## Statement guidelines

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

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

---

# Binding syntax: An overview

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

Angular provides many kinds of data binding. This guide covers most of them, after a high-level view of Angular data binding and its syntax.

Binding types can be grouped into three categories distinguished by the direction of data flow: from the source-to-view, from view-to-source, and in the two-way sequence: view-to-source-to-view:

Data direction	Syntax	Type
		Interpolation
One-way		Property
from data source	<code>  {{expression}}</code> <code>  [target]="expression"</code> <code>  bind-target="expression"</code>	Attribute
to view target		Class
		Style
One-way		
from view target	<code>  (target)="statement"</code> <code>  on-target="statement"</code>	Event
to data source		
Two-way	<code>  [(target)]="expression"</code> <code>  bindon-target="expression"</code>	Two-way

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

The target name is the name of a property. It may look like the name of an attribute but it never is. To appreciate the difference, you must develop a new way to think about template HTML.

## A new mental model

With all the power of data binding and the ability to extend the HTML vocabulary with custom markup, it is tempting to think of template HTML as HTML Plus.

It really is HTML Plus. But it's also significantly different than the HTML you're used to. It requires a new mental model.

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

```
src/app/app.component.html
```

```
<div class="special">Mental Model</div>

<button disabled>Save</button>
```

You still create a structure and initialize attribute values this way in Angular templates.

Then you learn to create new elements with components that encapsulate HTML and drop them into templates as if they were native HTML elements.

```
src/app/app.component.html
```

```
<!-- Normal HTML -->
<div class="special">Mental Model</div>
<!-- Wow! A new element! -->
<app-hero-detail></app-hero-detail>
```

That's HTML Plus.

Then you learn about data binding. The first binding you meet might look like this:

```
src/app/app.component.html
```

```
<!-- Bind button disabled state to `isUnchanged` property -->
<button [disabled]="isUnchanged">Save</button>
```

You'll get to that peculiar bracket notation in a moment. Looking beyond it, your intuition suggests that you're binding to the button's disabled attribute and setting it to the current value of the component's isUnchanged property.

Your intuition is incorrect! Your everyday HTML mental model is misleading. In fact, once you start data binding, you are no longer working with HTML attributes. You aren't setting attributes. You are setting the properties of DOM elements, components, and directives.

## HTML attribute vs. DOM property

The distinction between an HTML attribute and a DOM property is crucial to understanding how Angular binding works.

Attributes are defined by HTML. Properties are defined by the DOM (Document Object Model).

- A few HTML attributes have 1:1 mapping to properties. id is one example.
- Some HTML attributes don't have corresponding properties. colspan is one example.

- Some DOM properties don't have corresponding attributes. `textContent` is one example.
- Many HTML attributes appear to map to properties ... but not in the way you might think!

That last category is confusing until you grasp this general rule:

Attributes initialize DOM properties and then they are done. Property values can change; attribute values can't.

For example, when the browser renders `<input type="text" value="Bob">`, it creates a corresponding DOM node with a `value` property initialized to "Bob".

When the user enters "Sally" into the input box, the DOM element `value` property becomes "Sally". But the HTML `value` attribute remains unchanged as you discover if you ask the input element about that attribute: `input.getAttribute('value')` returns "Bob".

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

The `disabled` attribute is another peculiar example. A button's `disabled` property is `false` by default so the button is enabled. When you add the `disabled` attribute, its presence alone initializes the button's `disabled` property to `true` so the button is disabled.

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

Setting the button's `disabled` property (say, with an Angular binding) disables or enables the button. The value of the property matters.

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

This fact bears repeating: Template binding works with properties and events, not attributes.

## A WORLD WITHOUT ATTRIBUTES

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

With this model firmly in mind, read on to learn about binding targets.

## Binding targets

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

Type	Target	Examples
------	--------	----------

	Element property	src/app/app.component.html
Property	Component property	<img [src]="herolImageUrl"> <app-hero-detail [hero]="currentHero"></app-hero-detail>
	Directive property	<div [ <a href="#">ngClass</a> ]="{{'special': isSpecial}}></div>
Event	Element event	src/app/app.component.html
	Component event	<button (click)="onSave()">Save</button> <app-hero-detail (deleteRequest)="deleteHero()"></app-hero-detail>
	Directive event	<div (myClick)="clicked=\$event" clickable>click me</div>
Two-way	Event and property	src/app/app.component.html
		<input [( <a href="#">ngModel</a> )]= "name">
Attribute	Attribute (the exception)	src/app/app.component.html
		<button [attr.aria-label]="help">help</button>
Class	class property	src/app/app.component.html
		<div [class.special]="isSpecial">Special</div>
Style	<a href="#">style</a> property	src/app/app.component.html
		<button [style.color]="isSpecial ? 'red' : 'green'">

With this broad view in mind, you're ready to look at binding types in detail.

---

## Property binding ( [property] )

Write a template property binding to set a property of a view element. The binding sets the property to the value of a [template expression](#).

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

src/app/app.component.html

```
<img [src]="herolImageUrl">
```

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

```
src/app/app.component.html
```

```
<button [disabled]="isUnchanged">Cancel is disabled</button>
```

Another is setting a property of a directive:

```
src/app/app.component.html
```

```
<div [ngClass]="classes">[ngClass] binding to the classes property</div>
```

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

```
src/app/app.component.html
```

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

## One-way in

People often describe property binding as one-way data binding because it flows a value in one direction, from a component's data property into a target element property.

You cannot use property binding to pull values out of the target element. You can't bind to a property of the target element to read it. You can only set it.

Similarly, you cannot use property binding to call a method on the target element.

If the element raises events, you can listen to them with an [event binding](#).

If you must read a target element property or call one of its methods, you'll need a different technique. See the API reference for [ViewChild](#) and [ContentChild](#).

## Binding target

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

```
src/app/app.component.html
```

```
<img [src]="herolImageUrl">
```

Some people prefer the bind- prefix alternative, known as the canonical form:

```
src/app/app.component.html
```

```

```

The target name is always the name of a property, even when it appears to be the name of something else. You see src and may think it's the name of an attribute. No. It's the name of an image element property.

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

src/app/app.component.html

```
<div [ngClass]="classes">[ngClass] binding to the classes property</div>
```

Technically, Angular is matching the name to a directive [input](#), one of the property names listed in the directive's inputs array or a property decorated with [@Input\(\)](#). Such inputs map to the directive's own properties.

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

## Avoid side effects

As mentioned previously, evaluation of a template expression should have no visible side effects. The expression language itself does its part to keep you safe. You can't assign a value to anything in a property binding expression nor use the increment and decrement operators.

Of course, the expression might invoke a property or method that has side effects. Angular has no way of knowing that or stopping you.

The expression could call something like `getFoo()`. Only you know what `getFoo()` does. If `getFoo()` changes something and you happen to be binding to that something, you risk an unpleasant experience. Angular may or may not display the changed value. Angular may detect the change and throw a warning error. In general, stick to data properties and to methods that return values and do no more.

## Return the proper type

The template expression should evaluate to the type of value expected by the target property. Return a string if the target property expects a string. Return a number if the target property expects a number. Return an object if the target property expects an object.

The `hero` property of the `HeroDetail` component expects a `Hero` object, which is exactly what you're sending in the property binding:

src/app/app.component.html

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

## Remember the brackets

The brackets tell Angular to evaluate the template expression. If you omit the brackets, Angular treats the string as a constant and initializes the target property with that string. It does not evaluate the string!

Don't make the following mistake:

```
src/app/app.component.html
```

```
<!-- ERROR: HeroDetailComponent.hero expects a
  Hero object, not the string "currentHero" -->
<app-hero-detail hero="currentHero"></app-hero-detail>
```

## One-time string initialization

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

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

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

```
src/app/app.component.html
```

```
<app-hero-detail prefix="You are my" [hero]="currentHero"></app-hero-detail>
```

The [hero] binding, on the other hand, remains a live binding to the component's currentHero property.

## Property binding or interpolation?

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

```
src/app/app.component.html
```

```
<p> is the <i>property bound</i> image.</p>
```

```
<p><span>"{{title}}> is the <i>interpolated</i> title.</span></p>
<p>"<span [innerHTML]="title"></span>" is the <i>property bound</i> title.</p>
```

Interpolation is a convenient alternative to property binding in many cases.

When rendering data values as strings, there is no technical reason to prefer one form to the other. You lean toward readability, which tends to favor interpolation. You suggest establishing coding style rules and choosing the form that both conforms to the rules and feels most natural for the task at hand.

When setting an element property to a non-string data value, you must use property binding.

## Content security

Imagine the following malicious content.

```
src/app/app.component.ts
```

```
evilTitle = 'Template <script>alert("evil never sleeps")</script>Syntax';
```

Fortunately, Angular data binding is on alert for dangerous HTML. It sanitizes the values before displaying them. It will not allow HTML with script tags to leak into the browser, neither with interpolation nor property binding.

```
src/app/app.component.html
```

```
<!--
```

Angular generates warnings for these two lines as it sanitizes them

WARNING: sanitizing HTML stripped some content (see <http://g.co/ng/security#xss>).

```
-->
```

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

```
<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.

"Template <script>alert("evil never sleeps")</script>Syntax" is the *interpolated* evil title.

"Template Syntax" is the *property bound* evil title.

---

## Attribute, class, and style bindings

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

### Attribute binding

You can set the value of an attribute directly with an attribute binding.

This is the only exception to the rule that a binding sets a target property. This is the only binding that creates and sets an attribute.

This guide stresses repeatedly that setting an element property with a property binding is always preferred to setting the attribute with a string. Why does Angular offer attribute binding?

You must use attribute binding when there is no element property to bind.

Consider the ARIA, SVG, and table span attributes. They are pure attributes. They do not correspond to element properties, and they do not set element properties. There are no property targets to bind to.

This fact becomes painfully obvious when you write something like this.

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

And you get this error:

Template 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. It has the `"colspan"` attribute, but interpolation and property binding can set only properties, not attributes.

You need attribute bindings to create and bind to such attributes.

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

Bind `[attr.colspan]` to a calculated value:

src/app/app.component.html

```
<table border=1>
  <!-- expression calculates colspan=2 -->
  <tr><td [attr.colspan]="1 + 1">One-Two</td></tr>

  <!-- ERROR: There is no `colspan` property to set!
    <tr><td colspan="{{1 + 1}}>Three-Four</td></tr>
  -->

  <tr><td>Five</td><td>Six</td></tr>
</table>
```

Here's how the table renders:

One-Two

Five Six

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

src/app/app.component.html

```
<!-- create and set an aria attribute for assistive technology -->
<button [attr.aria-label]="actionName">{{actionName}} with Aria</button>
```

---

## Class binding

You can add and remove CSS class names from an element's `class` attribute with a class binding.

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

The following examples show how to add and remove the application's "special" class with class bindings. Here's how to set the attribute without binding:

```
src/app/app.component.html
```

```
<!-- standard class attribute setting -->
<div class="bad curly special">Bad curly special</div>
```

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

```
src/app/app.component.html
```

```
<!-- reset/override all class names with a binding -->
<div class="bad curly special"
  [class]="badCurly">Bad curly</div>
```

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

```
src/app/app.component.html
```

```
<!-- toggle the "special" class on/off with a property -->
<div [class.special]="isSpecial">The class binding is special</div>
```

```
<!-- binding to `class.special` trumps the class attribute -->
<div class="special"
  [class.special]!="isSpecial">This one is not so special</div>
```

While this is a fine way to toggle a single class name, the [NgClass directive](#) is usually preferred when managing multiple class names at the same time.

---

## Style binding

You can set inline styles with a style binding.

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

```
src/app/app.component.html
```

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

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

```
src/app/app.component.html
```

```
<button [style.fontSize.em]="isSpecial ? 3 : 1" >Big</button>
<button [style.fontSize.%] ="!isSpecial ? 150 : 50" >Small</button>
```

While this is a fine way to set a single style, the [NgStyle directive](#) is generally preferred when setting several inline styles at the same time.

Note that a style property name can be written in either [dash-case](#), as shown above, or [camelCase](#), such as `fontSize`.

---

## Event binding ( (event) )

The bindings directives you've met so far flow data in one direction: from a component to an element.

Users don't just stare at the screen. They enter text into input boxes. They pick items from lists. They click buttons. Such user actions may result in a flow of data in the opposite direction: from an element to a component.

The only way to know about a user action is to listen for certain events such as keystrokes, mouse movements, clicks, and touches. You declare your interest in user actions through Angular event binding.

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

```
src/app/app.component.html
```

```
<button (click)="onSave()">Save</button>
```

### Target event

A name between parentheses — for example, `(click)` — identifies the target event. In the following example, the target is the button's click event.

```
src/app/app.component.html
```

```
<button (click)="onSave()">Save</button>
```

Some people prefer the `on-` prefix alternative, known as the canonical form:

```
src/app/app.component.html
```

```
<button on-click="onSave()">On Save</button>
```

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

```
src/app/app.component.html
```

```
<!-- 'myClick' is an event on the custom 'ClickDirective' -->
<div (myClick)="clickMessage=$event" clickable>click with myClick</div>
```

The myClick directive is further described in the section on [aliasing input/output properties](#).

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

## \$event and event handling statements

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

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

The binding conveys information about the event, including data values, through an event object named \$event.

The shape of the event object is determined by the target event. If the target event is a native DOM element event, then \$event is a [DOM event object](#), with properties such as [target](#) and [target.value](#).

Consider this example:

```
src/app/app.component.html
```

```
<input [value]="currentHero.name"
       (input)="currentHero.name=$event.target.value" >
```

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

To update the name property, the changed text is retrieved by following the path [\\$event.target.value](#).

If the event belongs to a directive (recall that components are directives), \$event has whatever shape the directive decides to produce.

## Custom events with EventEmitter

Directives typically raise custom events with an Angular [EventEmitter](#). The directive creates an [EventEmitter](#) and exposes it as a property. The directive calls [EventEmitter.emit\(payload\)](#) to fire an event, passing in a message payload, which can be

anything. Parent directives listen for the event by binding to this property and accessing the payload through the \$event object.

Consider a HeroDetailComponent that presents hero information and responds to user actions. Although the HeroDetailComponent has a delete button it doesn't know how to delete the hero itself. The best it can do is raise an event reporting the user's delete request.

Here are the pertinent excerpts from that HeroDetailComponent:

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

template: `

```
<div>
  
    {{prefix}} {{hero?.name}}
  </span>
  <button (click)="delete()">Delete</button>
</div>`
```

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

```
// This component makes a request but it can't actually delete a hero.
deleteRequest = new EventEmitter<Hero>();
```

```
delete() {
  this.deleteRequest.emit(this.hero);
}
```

The component defines a deleteRequest property that returns an EventEmitter. When the user clicks delete, the component invokes the delete() method, telling the EventEmitter to emit a Hero object.

Now imagine a hosting parent component that binds to the HeroDetailComponent's deleteRequest event.

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

```
<app-hero-detail (deleteRequest)="deleteHero($event)" [hero]="currentHero"></app-hero-detail>
```

When the deleteRequest event fires, Angular calls the parent component's deleteHero method, passing the hero-to-delete(emitted by HeroDetail) in the \$event variable.

## Template statements have side effects

The deleteHero method has a side effect: it deletes a hero. Template statement side effects are not just OK, but expected.

Deleting the hero updates the model, perhaps triggering other changes including queries and saves to a remote server. These changes percolate through the system and are ultimately displayed in this and other views.

---

## Two-way binding ( [(...)] )

You often want to both display a data property and update that property when the user makes changes.

On the element side that takes a combination of setting a specific element property and listening for an element change event.

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

[( )] = BANANA IN A BOX

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

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

src/app/sizer.component.ts

```
1. import { Component, EventEmitter, Input, Output } from '@angular/core';
2.
3. @Component({
4.   selector: 'app-sizer',
5.   template: `
6.     <div>
7.       <button (click)="dec()" title="smaller">-</button>
8.       <button (click)="inc()" title="bigger">+</button>
9.       <label [style.fontSize.px]="size">FontSize: {{size}}px</label>
10.    </div>
11.  `)
12. export class SizerComponent {
13.   @Input() size: number | string;
14.   @Output() sizeChange = new EventEmitter<number>();
15.
16.   dec() { this.resize(-1); }
17.   inc() { this.resize(+1); }
18.
19.   resize(delta: number) {
20.     this.size = Math.min(40, Math.max(8, +this.size + delta));
```

```
21. this.sizeChange.emit(this.size);
22. }
23.}
```

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

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

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

```
<app-sizer [(size)]="fontSizePx"></app-sizer>
<div [style.fontSize.px]="fontSizePx">Resizable Text</div>
```

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

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

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

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

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

Clearly the two-way binding syntax is a great convenience compared to separate property and event bindings.

It would be convenient to use two-way binding with HTML form elements like <input> and <select>. However, no native HTML element follows the x value and xChange event pattern.

Fortunately, the Angular [NgModel](#) directive is a bridge that enables two-way binding to form elements.

---

## Built-in directives

Earlier versions of Angular included over seventy built-in directives. The community contributed many more, and countless private directives have been created for internal applications.

You don't need many of those directives in Angular. You can often achieve the same results with the more capable and expressive Angular binding system. Why create a directive to handle a click when you can write a simple binding such as this?

```
src/app/app.component.html
```

```
<button (click)="onSave()">Save</button>
```

You still benefit from directives that simplify complex tasks. Angular still ships with built-in directives; just not as many. You'll write your own directives, just not as many.

This segment reviews some of the most frequently used built-in directives, classified as either [attribute directives](#) or [structural directives](#).

---

## Built-in attribute directives

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

Many details are covered in the [Attribute Directives](#) guide. Many NgModules such as the [RouterModule](#) and the [FormsModule](#) define their own attribute directives. This section is an introduction to the most commonly used attribute directives:

- [NgClass](#) - add and remove a set of CSS classes
  - [NgStyle](#) - add and remove a set of HTML styles
  - [NgModel](#) - two-way data binding to an HTML form element
- 

### NgClass

You typically control how elements appear by adding and removing CSS classes dynamically. You can bind to the [ngClass](#) to add or remove several classes simultaneously.

A [class binding](#) is a good way to add or remove a single class.

```
src/app/app.component.html
```

```
<!-- toggle the "special" class on/off with a property -->
<div [class.special]="isSpecial">The class binding is special</div>
```

To add or remove many CSS classes at the same time, the [NgClass](#) directive may be the better choice.

Try binding [ngClass](#) to a key:value control object. Each key of the object is a CSS class name; its value is true if the class should be added, false if it should be removed.

Consider a `setCurrentClasses` component method that sets a `component` property, `currentClasses` with an object that adds or removes three classes based on the true/false state of three other component properties:

```
src/app/app.component.ts
```

```
currentClasses: {};
setCurrentClasses() {
// CSS classes: added/removed per current state of component properties
this.currentClasses = {
  'saveable': this.canSave,
  'modified': !this.isUnchanged,
  'special': this.isSpecial
};
}
```

Adding an [ngClass](#) property binding to currentClasses sets the element's classes accordingly:

src/app/app.component.html

```
<div [ngClass]="currentClasses">This div is initially saveable, unchanged, and special</div>
```

It's up to you to call setCurrentClasses(), both initially and when the dependent properties change.

---

## NgStyle

You can set inline styles dynamically, based on the state of the component. With [NgStyle](#) you can set many inline styles simultaneously.

A [style binding](#) is an easy way to set a single style value.

src/app/app.component.html

```
<div [style.fontSize]="isSpecial ? 'x-large' : 'smaller'">
  This div is x-large or smaller.
</div>
```

To set many inline styles at the same time, the [NgStyle](#) directive may be the better choice.

Try binding [ngStyle](#) to a key:value control object. Each key of the object is a style name; its value is whatever is appropriate for that style.

Consider a setCurrentStyles component method that sets a component property, currentStyles with an object that defines three styles, based on the state of three other component properties:

src/app/app.component.ts

```
currentStyles: {};
setCurrentStyles() {
// CSS styles: set per current state of component properties
this.currentStyles = {
  'font-style': this.canSave ? 'italic' : 'normal',
  'font-weight': !this.isUnchanged ? 'bold' : 'normal',
```

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

Adding an [ngStyle](#) property binding to currentStyles sets the element's styles accordingly:

src/app/app.component.html

```
<div [ngStyle]="currentStyles">  
  This div is initially italic, normal weight, and extra large (24px).  
</div>
```

It's up to you to call setCurrentStyles(), both initially and when the dependent properties change.

---

## NgModel - Two-way binding to form elements with [(ngModel)]

When developing data entry forms, you often both display a data property and update that property when the user makes changes.

Two-way data binding with the [NgModel](#) directive makes that easy. Here's an example:

src/app/app.component.html (NgModel-1)

```
<input [(ngModel)]="currentHero.name">
```

### FormsModule is required to use ngModel

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

Here's how to import the [FormsModule](#) to make [\[\(ngModel\)\]](#) available.

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

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

```
/* Other imports */
```

```
@NgModule({  
  imports: [  
    BrowserModule,  
    FormsModule // <--- import into the NgModule  
,  
  /* Other module metadata */
```

```
})
export class AppModule { }
```

### Inside [(ngModel)]

Looking back at the name binding, note that you could have achieved the same result with separate bindings to the <input> element's value property and input event.

src/app/app.component.html

```
<input [value]="currentHero.name"
       (input)="currentHero.name=$event.target.value" >
```

That's cumbersome. Who can remember which element property to set and which element event emits user changes? How do you extract the currently displayed text from the input box so you can update the data property? Who wants to look that up each time?

That [ngModel](#) directive hides these onerous details behind its own [ngModel](#) input and ngModelChange output properties.

src/app/app.component.html

```
<input
  [ngModel]="currentHero.name"
  (ngModelChange)="currentHero.name=$event">
```

The [ngModel](#) data property sets the element's value property and the ngModelChange event property listens for changes to the element's value.

The details are specific to each kind of element and therefore the [NgModel](#) directive only works for an element supported by a [ControlValueAccessor](#) that adapts an element to this protocol. The <input> box is one of those elements. Angular provides value accessors for all of the basic HTML form elements and the [Forms](#) guide shows how to bind to them.

You can't apply [([ngModel](#))] to a non-form native element or a third-party custom component until you write a suitable value accessor, a technique that is beyond the scope of this guide.

You don't need a value accessor for an Angular component that you write because you can name the value and event properties to suit Angular's basic [two-way binding syntax](#) and skip [NgModel](#) altogether. The [sizer shown above](#) is an example of this technique.

Separate [ngModel](#) bindings is an improvement over binding to the element's native properties. You can do better.

You shouldn't have to mention the data property twice. Angular should be able to capture the component's data property and set it with a single declaration, which it can with the [([ngModel](#))] syntax:

src/app/app.component.html

```
<input [(ngModel)]="currentHero.name">
```

Is `[(ngModel)]` all you need? Is there ever a reason to fall back to its expanded form?

The `[(ngModel)]` syntax can only set a data-bound property. If you need to do something more or something different, you can write the expanded form.

The following contrived example forces the input value to uppercase:

src/app/app.component.html

```
<input  
  [ngModel]="currentHero.name"  
  (ngModelChange)="setUppercaseName($event)">
```

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

## NgModel Binding

### Result: Hercules

Hercules	without NgModel
Hercules	<code>[(ngModel)]</code>
Hercules	<code>bindon-ngModel</code>
Hercules	<code>(ngModelChange) = "...firstName=\$event"</code>
Hercules	<code>(ngModelChange) = "setUpperCaseFirstName(\$event)"</code>

## 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.

The deep details of structural directives are covered in the [Structural Directives](#) guide where you'll learn:

- why you [prefix the directive name with an asterisk \(\\*\)](#).
- to use [`<ng-container>`](#) to group elements when there is no suitable host element for the directive.
- how to write your own structural directive.
- that you can only apply [one structural directive](#) to an element.

This section is an introduction to the common structural directives:

- [NgIf](#) - conditionally add or remove an element from the DOM
- [NgSwitch](#) - a set of directives that switch among alternative views
- [NgForOf](#) - repeat a template for each item in a list

## NgIf

You can add or remove an element from the DOM by applying an [NgIf](#) directive to that element (called the host element). Bind the directive to a condition expression like [isActive](#) in this example.

src/app/app.component.html

```
<app-hero-detail *ngIf="isActive"></app-hero-detail>
```

Don't forget the asterisk (\*) in front of [ngIf](#).

When the [isActive](#) expression returns a truthy value, [NgIf](#) adds the HeroDetailComponent to the DOM. When the expression is falsy, [NgIf](#) removes the HeroDetailComponent from the DOM, destroying that component and all of its sub-components.

### Show/hide is not the same thing

You can control the visibility of an element with a [class](#) or [style](#) binding:

src/app/app.component.html

```
<!-- isSpecial is true -->
<div [class.hidden]="!isSpecial">Show with class</div>
<div [class.hidden]="isSpecial">Hide with class</div>
```

```
<!-- HeroDetail is in the DOM but hidden -->
<app-hero-detail [class.hidden]="isSpecial"></app-hero-detail>
```

```
<div [style.display]="isSpecial ? 'block' : 'none'">Show with style</div>
<div [style.display]="isSpecial ? 'none' : 'block'">Hide with style</div>
```

Hiding an element is quite different from removing an element with [NgIf](#).

When you hide an element, that element and all of its descendants remain in the DOM. All components for those elements stay in memory and Angular may continue to check for changes. You could be holding onto considerable computing resources and degrading performance, for something the user can't see.

When [NgIf](#) is false, Angular removes the element and its descendants from the DOM. It destroys their components, potentially freeing up substantial resources, resulting in a more responsive user experience.

The show/hide technique is fine for a few elements with few children. You should be wary when hiding large component trees; [NgIf](#) may be the safer choice.

### Guard against null

The [ngIf](#) directive is often used to guard against null. Show/hide is useless as a guard. Angular will throw an error if a nested expression tries to access a property of null.

Here we see [NgIf](#) guarding two <div>s. The currentHero name will appear only when there is a currentHero. The nullHero will never be displayed.

src/app/app.component.html

```
<div *ngIf="currentHero">Hello, {{currentHero.name}}</div>
<div *ngIf="nullHero">Hello, {{nullHero.name}}</div>
```

See also the [safe navigation operator](#) described below.

---

## NgForOf

[NgForOf](#) is a repeater directive — a way to present a list of items. You define a block of HTML that defines how a single item should be displayed. You tell Angular to use that block as a template for rendering each item in the list.

Here is an example of [NgForOf](#) applied to a simple <div>:

src/app/app.component.html

```
<div *ngFor="let hero of heroes">{{hero.name}}</div>
```

You can also apply an [NgForOf](#) to a component element, as in this example:

src/app/app.component.html

```
<app-hero-detail *ngFor="let hero of heroes" [hero]="hero"></app-hero-detail>
```

Don't forget the asterisk (\*) in front of [ngFor](#).

The text assigned to [\\*ngFor](#) is the instruction that guides the repeater process.

### \*ngFor microsyntax

The string assigned to [\\*ngFor](#) is not a [template expression](#). It's a microsyntax — a little language of its own that Angular interprets. The string "let hero of heroes" means:

Take each hero in the heroes array, store it in the local hero looping variable, and make it available to the templated HTML for each iteration.

Angular translates this instruction into a <ng-template> around the host element, then uses this template repeatedly to create a new set of elements and bindings for each hero in the list.

Learn about the microsyntax in the [Structural Directives](#) guide.

## Template input variables

The let keyword before hero creates a template input variable called hero. The [NgForOf](#) directive iterates over the heroesarray returned by the parent component's heroes property and sets hero to the current item from the array during each iteration.

You reference the hero input variable within the [NgForOf](#) host element (and within its descendants) to access the hero's properties. Here it is referenced first in an interpolation and then passed in a binding to the hero property of the <hero-detail> component.

src/app/app.component.html

```
<div *ngFor="let hero of heroes">{{hero.name}}</div>
<app-hero-detail *ngFor="let hero of heroes" [hero]="hero"></app-hero-detail>
```

Learn more about template input variables in the [Structural Directives](#) guide.

### \*ngFor with index

The index property of the [NgForOf](#) directive context returns the zero-based index of the item in each iteration. You can capture the index in a template input variable and use it in the template.

The next example captures the index in a variable named i and displays it with the hero name like this.

src/app/app.component.html

```
<div *ngFor="let hero of heroes; let i=index">{{i + 1}} - {{hero.name}}</div>
```

NgFor is implemented by the [NgForOf](#) directive. Read more about the other [NgForOf](#) context values such as last, [even](#), and [odd](#) in the [NgForOf API reference](#).

### \*ngFor with trackBy

The [NgForOf](#) directive may perform poorly, especially with large lists. A small change to one item, an item removed, or an item added can trigger a cascade of DOM manipulations.

For example, re-querying the server could reset the list with all new hero objects.

Most, if not all, are previously displayed heroes. You know this because the id of each hero hasn't changed. But Angular sees only a fresh list of new object references. It has no choice but to tear down the old DOM elements and insert all new DOM elements.

Angular can avoid this churn with trackBy. Add a method to the component that returns the value [NgForOf](#) should track. In this case, that value is the hero's id.

src/app/app.component.ts

```
trackByHeroes(index: number, hero: Hero): number { return hero.id; }
```

In the microsyntax expression, set trackBy to this method.

src/app/app.component.html

```
<div *ngFor="let hero of heroes; trackBy: trackByHeroes">
  {{hero.id}} {{hero.name}}
</div>
```

Here is an illustration of the trackBy effect. "Reset heroes" creates new heroes with the same hero.ids. "Change ids" creates new heroes with new hero.ids.

- With no trackBy, both buttons trigger complete DOM element replacement.
- With trackBy, only changing the id triggers element replacement.

#### \*ngFor trackBy

Reset heroes 

Change ids

Clear counts

without trackBy

```
(0) Hercules  
(1) Mr. Nice  
(2) Narco  
(3) Windstorm  
(4) Magneta
```

with trackBy

```
(0) Hercules  
(1) Mr. Nice  
(2) Narco  
(3) Windstorm  
(4) Magneta
```

---

## The NgSwitch directives

NgSwitch is like the JavaScript switch statement. It can display one element from among several possible elements, based on a switch condition. Angular puts only the selected element into the DOM.

NgSwitch is actually a set of three, cooperating directives: [NgSwitch](#), [NgSwitchCase](#), and [NgSwitchDefault](#) as seen in this example.

src/app/app.component.html

```
<div [ngSwitch]="currentHero.emotion">  
  <app-happy-hero *ngSwitchCase="happy" [hero]="currentHero"></app-happy-hero>  
  <app-sad-hero   *ngSwitchCase="sad"    [hero]="currentHero"></app-sad-hero>
```

```
<app-confused-hero *ngSwitchCase="confused" [hero]="currentHero"></app-confused-hero>
<app-unknown-hero *ngSwitchDefault [hero]="currentHero"></app-unknown-hero>
</div>
```

Pick your favorite hero

Hercules  Mr. Nice  Narco  Windstorm  Magneta

Wow. You like Hercules. What a happy hero ... just like you.

[NgSwitch](#) is the controller directive. Bind it to an expression that returns the switch value. The emotion value in this example is a string, but the switch value can be of any type.

Bind to [\[ngSwitch\]](#). You'll get an error if you try to set [\\*ngSwitch](#) because [NgSwitch](#) is an attribute directive, not a structural directive. It changes the behavior of its companion directives. It doesn't touch the DOM directly.

Bind to [\\*ngSwitchCase](#) and [\\*ngSwitchDefault](#). The [NgSwitchCase](#) and [NgSwitchDefault](#) directives are structural directives because they add or remove elements from the DOM.

- [NgSwitchCase](#) adds its element to the DOM when its bound value equals the switch value.
- [NgSwitchDefault](#) adds its element to the DOM when there is no selected [NgSwitchCase](#).

The switch directives are particularly useful for adding and removing component elements. This example switches among four "emotional hero" components defined in the hero-switch.components.ts file. Each component has a hero [input property](#) which is bound to the currentHero of the parent component.

Switch directives work as well with native elements and web components too. For example, you could replace the <confused-hero> switch case with the following.

src/app/app.component.html

```
<div *ngSwitchCase="confused">Are you as confused as {{currentHero.name}}?</div>
```

---

## Template reference variables (#var)

A template reference variable is often a reference to a DOM element within a template. It can also be a reference to an Angular component or directive or a [web component](#).

Use the hash symbol (#) to declare a reference variable. The #phone declares a phone variable on an <input> element.

```
src/app/app.component.html
```

```
<input #phone placeholder="phone number">
```

You can refer to a template reference variable anywhere in the template. The phone variable declared on this `<input>` is consumed in a `<button>` on the other side of the template

```
src/app/app.component.html
```

```
<input #phone placeholder="phone number">
```

```
<!-- lots of other elements -->
```

```
<!-- phone refers to the input element; pass its `value` to an event handler -->
```

```
<button (click)="callPhone(phone.value)">Call</button>
```

## How a reference variable gets its value

In most cases, Angular sets the reference variable's value to the element on which it was declared. In the previous example, phone refers to the phone number `<input>` box. The phone button click handler passes the input value to the component's `callPhone` method. But a directive can change that behavior and set the value to something else, such as itself. The [NgForm](#) directive does that.

The following is a simplified version of the form example in the [Forms](#) guide.

```
src/app/hero-form.component.html
```

```
<form (ngSubmit)="onSubmit(heroForm) #heroForm="ngForm">
  <div class="form-group">
    <label for="name">Name
      <input class="form-control" name="name" required [(ngModel)]="hero.name">
    </label>
  </div>
  <button type="submit" [disabled]="!heroForm.form.valid">Submit</button>
</form>
<div [hidden]="!heroForm.form.valid">
  {{submitMessage}}
</div>
```

A template reference variable, `heroForm`, appears three times in this example, separated by a large amount of HTML. What is the value of `heroForm`?

If Angular hadn't taken it over when you imported the [FormsModule](#), it would be the [HTMLFormElement](#). The `heroForm` is actually a reference to an Angular [NgForm](#) directive with the ability to track the value and validity of every control in the form.

The native <form> element doesn't have a form property. But the [NgForm](#) directive does, which explains how you can disable the submit button if the heroForm.form.valid is invalid and pass the entire form control tree to the parent component's onSubmit method.

## Template reference variable warning notes

A template reference variable (#phone) is not the same as a template input variable (let phone) such as you might see in an [\\*ngFor](#). Learn the difference in the [Structural Directives](#) guide.

The scope of a reference variable is the entire template. Do not define the same variable name more than once in the same template. The runtime value will be unpredictable.

You can use the ref- prefix alternative to #. This example declares the fax variable as ref-fax instead of #fax.

src/app/app.component.html

```
<input ref-fax placeholder="fax number">
<button (click)="callFax(fax.value)">Fax</button>
```

---

## Input and Output properties

An Input property is a settable property annotated with an [@Input](#) decorator. Values flow into the property when it is data bound with a [property binding](#).

An Output property is an observable property annotated with an [@Output](#) decorator. The property almost always returns an Angular [EventEmitter](#). Values flow out of the component as events bound with an [event binding](#).

You can only bind to another component or directive through its Input and Output properties.

Remember that all components are directives.

The following discussion refers to components for brevity and because this topic is mostly a concern for component authors.

## Discussion

You are usually binding a template to its own component class. In such binding expressions, the component's property or method is to the right of the (=).

src/app/app.component.html

```
<img [src]="iconUrl"/>
<button (click)="onSave()">Save</button>
```

The iconUrl and onSave are members of the AppComponent class. They are not decorated with [@Input\(\)](#) or [@Output](#). Angular does not object.

You can always bind to a public property of a component in its own template. It doesn't have to be an Input or Outputproperty

A component's class and template are closely coupled. They are both parts of the same thing. Together they are the component. Exchanges between a component class and its template are internal implementation details.

## Binding to a different component

You can also bind to a property of a different component. In such bindings, the other component's property is to the left of the (=).

In the following example, the AppComponent template binds AppComponent class members to properties of the HeroDetailComponent whose selector is 'app-hero-detail'.

src/app/app.component.html

```
<app-hero-detail [hero]="currentHero" (deleteRequest)="deleteHero($event)">
</app-hero-detail>
```

The Angular compiler may reject these bindings with errors like this one:

Uncaught Error: Template parse errors:

Can't bind to 'hero' since it isn't [a](#) known property of 'app-hero-detail'

You know that HeroDetailComponent has hero and deleteRequest properties. But the Angular compiler refuses to recognize them.

The Angular compiler won't bind to properties of a different component unless they are Input or Output properties.

There's a good reason for this rule.

It's OK for a component to bind to its own properties. The component author is in complete control of those bindings.

But other components shouldn't have that kind of unrestricted access. You'd have a hard time supporting your component if anyone could bind to any of its properties. Outside components should only be able to bind to the component's public binding API.

Angular asks you to be explicit about that API. It's up to you to decide which properties are available for binding by external components.

## TypeScript public doesn't matter

You can't use the TypeScript public and private access modifiers to shape the component's public binding API.

All data bound properties must be TypeScript public properties. Angular never binds to a TypeScript privateproperty.

Angular requires some other way to identify properties that outside components are allowed to bind to. That other way is the `@Input()` and `@Output()` decorators.

## Declaring Input and Output properties

In the sample for this guide, the bindings to `HeroDetailComponent` do not fail because the data bound properties are annotated with `@Input()` and `@Output()` decorators.

src/app/hero-detail.component.ts

```
@Input() hero: Hero;  
@Output() deleteRequest = new EventEmitter<Hero>();
```

Alternatively, you can identify members in the inputs and outputs arrays of the directive metadata, as in this example:

src/app/hero-detail.component.ts

```
@Component({  
  inputs: ['hero'],  
  outputs: ['deleteRequest'],  
})
```

## Input or output?

Input properties usually receive data values. Output properties expose event producers, such as `EventEmitter` objects.

The terms input and output reflect the perspective of the target directive.

```
<hero-detail [hero]="currentHero" (deleteRequest)="deleteHero($event)">
```

`HeroDetailComponent.hero` is an input property from the perspective of `HeroDetailComponent` because data flows into that property from a template binding expression.

`HeroDetailComponent.deleteRequest` is an output property from the perspective of `HeroDetailComponent` because events stream out of that property and toward the handler in a template binding statement.

## Aliasing input/output properties

Sometimes the public name of an input/output property should be different from the internal name.

This is frequently the case with [attribute directives](#). Directive consumers expect to bind to the name of the directive. For example, when you apply a directive with a `myClick` selector to a `<div>` tag, you expect to bind to an event property that is also called `myClick`.

src/app/app.component.html

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

However, the directive name is often a poor choice for the name of a property within the directive class. The directive name rarely describes what the property does. The myClick directive name is not a good name for a property that emits click messages.

Fortunately, you can have a public name for the property that meets conventional expectations, while using a different name internally. In the example immediately above, you are actually binding through the myClick alias to the directive's own clicksproperty.

You can specify the alias for the property name by passing it into the input/output decorator like this:

src/app/click.directive.ts

```
@Output('myClick') clicks = new EventEmitter<string>(); // @Output(alias) propertyName  
= ...
```

You can also alias property names in the inputs and outputs arrays. You write a colon-delimited (:) string with the directive property name on the left and the public alias on the right:

src/app/click.directive.ts

```
@Directive({  
  outputs: ['clicks:myClick'] // propertyName:alias  
})
```

---

## Template expression operators

The template expression language employs a subset of JavaScript syntax supplemented with a few special operators for specific scenarios. The next sections cover two of these operators: pipe and safe navigation operator.

### The pipe operator (|)

The result of an expression might require some transformation before you're ready to use it in a binding. For example, you might display a number as a currency, force text to uppercase, or filter a list and sort it.

Angular [pipes](#) are a good choice for small transformations such as these. Pipes are simple functions that accept an input value and return a transformed value. They're easy to apply within template expressions, using the pipe operator (|):

src/app/app.component.html

```
<div>Title through uppercase pipe: {{title | uppercase}}</div>
```

The pipe operator passes the result of an expression on the left to a pipe function on the right.

You can chain expressions through multiple pipes:

src/app/app.component.html

```
<!-- Pipe chaining: convert title to uppercase, then to lowercase -->
```

```
<div>
```

Title through a pipe chain:

```
{{title | uppercase | lowercase}}
```

```
</div>
```

And you can also apply parameters to a pipe:

src/app/app.component.html

```
<!-- pipe with configuration argument => "February 25, 1970" -->
```

```
<div>Birthdate: {{currentHero?.birthdate | date:'longDate'}}</div>
```

The json pipe is particularly helpful for debugging bindings:

src/app/app.component.html (pipes-json)

```
<div>{{currentHero | json}}</div>
```

The generated output would look something like this

```
{ "id": 0, "name": "Hercules", "emotion": "happy",  
"birthdate": "1970-02-25T08:00:00.000Z",  
"url": "http://www.imdb.com/title/tt0065832/",  
"rate": 325 }
```

---

## The safe navigation operator ( ?. ) and null property paths

The Angular safe navigation operator (?.) is a fluent and convenient way to guard against null and undefined values in property paths. Here it is, protecting against a view render failure if the currentHero is null.

src/app/app.component.html

The current hero's name is {{currentHero?.name}}

What happens when the following data bound title property is null?

src/app/app.component.html

The title is {{title}}

The view still renders but the displayed value is blank; you see only "The title is" with nothing after it. That is reasonable behavior. At least the app doesn't crash.

Suppose the template expression involves a property path, as in this next example that displays the name of a null hero.

The null hero's name is {{nullHero.name}}

JavaScript throws a null reference error, and so does Angular:

TypeError: Cannot [read](#) property 'name' of null in [null].

Worse, the entire view disappears.

This would be reasonable behavior if the hero property could never be null. If it must never be null and yet it is null, that's a programming error that should be caught and fixed. Throwing an exception is the right thing to do.

On the other hand, null values in the property path may be OK from time to time, especially when the data are null now and will arrive eventually.

While waiting for data, the view should render without complaint, and the null property path should display as blank just as the title property does.

Unfortunately, the app crashes when the currentHero is null.

You could code around that problem with [\\*ngIf](#).

src/app/app.component.html

```
<!--No hero, div not displayed, no error -->
```

```
<div *ngIf="nullHero">The null hero's name is {{nullHero.name}}</div>
```

You could try to chain parts of the property path with &&, knowing that the expression bails out when it encounters the first null.

src/app/app.component.html

```
The null hero's name is {{nullHero && nullHero.name}}
```

These approaches have merit but can be cumbersome, especially if the property path is long. Imagine guarding against a null somewhere in a long property path such as a.b.c.d.

The Angular safe navigation operator (?) is a more fluent and convenient way to guard against nulls in property paths. The expression bails out when it hits the first null value. The display is blank, but the app keeps rolling without errors.

src/app/app.component.html

```
<!-- No hero, no problem! -->
```

```
The null hero's name is {{nullHero?.name}}
```

It works perfectly with long property paths such as [a?.b?.c?.d](#).

---

## The non-null assertion operator ( ! )

As of Typescript 2.0, you can enforce [strict null checking](#) with the --strictNullChecks flag. TypeScript then ensures that no variable is unintentionally null or undefined.

In this mode, typed variables disallow null and undefined by default. The type checker throws an error if you leave a variable unassigned or try to assign null or undefined to a variable whose type disallows null and undefined.

The type checker also throws an error if it can't determine whether a variable will be null or undefined at runtime. You may know that can't happen but the type checker doesn't know. You tell the type checker that it can't happen by applying the post-fix [non-null assertion operator \(!\)](#).

The Angular non-null assertion operator (!) serves the same purpose in an Angular template.

For example, after you use [\\*ngIf](#) to check that hero is defined, you can assert that hero properties are also defined.

src/app/app.component.html

```
<!--No hero, no text -->
<div *ngIf="hero">
  The hero's name is {{hero!.name}}
</div>
```

When the Angular compiler turns your template into TypeScript code, it prevents TypeScript from reporting that hero.name might be null or undefined.

Unlike the [safe navigation operator](#), the non-null assertion operator does not guard against null or undefined. Rather it tells the TypeScript type checker to suspend strict null checks for a specific property expression.

You'll need this template operator when you turn on strict null checks. It's optional otherwise.

[back to top](#)

---

## The \$any type cast function (\$any( <expression> ))

Sometimes a binding expression will be reported as a type error and it is not possible or difficult to fully specify the type. To silence the error, you can use the \$any cast function to cast the expression to [the any type](#).

src/app/app.component.html

```
<!-- Accessing an undeclared member -->
<div>
  The hero's marker is {{$any(hero).marker}}
</div>
```

In this example, when the Angular compiler turns your template into TypeScript code, it prevents TypeScript from reporting that marker is not a member of the Hero interface.

The `$any` cast function can be used in conjunction with this to allow access to undeclared members of the component.

src/app/app.component.html

```
<!-- Accessing an undeclared member -->
<div>
  Undeclared members is {{$any(this).member}}
</div>
```

The `$any` cast function can be used anywhere in a binding expression where a method call is valid.

## Summary

You've completed this survey of template syntax. Now it's time to put that knowledge to work on your own components and directives.

User actions such as clicking a link, pushing a button, and entering text raise DOM events. This page explains how to bind those events to component event handlers using the Angular event binding syntax.

Run the [live example](#) / [download example](#).

## Binding to user input events

You can use [Angular event bindings](#) to respond to any [DOM event](#). Many DOM events are triggered by user input. Binding to these events provides a way to get input from the user.

To bind to a DOM event, surround the DOM event name in parentheses and assign a quoted [template statement](#) to it.

The following example shows an event binding that implements a click handler:

src/app/click-me.component.ts

```
<button (click)="onClickMe()">Click me!</button>
```

The `(click)` to the left of the equals sign identifies the button's click event as the target of the binding. The text in quotes to the right of the equals sign is the template statement, which responds to the click event by calling the component's `onClickMe` method.

When writing a binding, be aware of a template statement's execution context. The identifiers in a template statement belong to a specific context object, usually the Angular component controlling the template. The example above shows a single line of HTML, but that HTML belongs to a larger component:

src/app/click-me.component.ts

```

@Component({
  selector: 'app-click-me',
  template: `
    <button (click)="onClickMe()">Click me!</button>
    {{clickMessage}}
  `})
export class ClickMeComponent {
  clickMessage = "";

  onClickMe() {
    this.clickMessage = 'You are my hero!';
  }
}

```

When the user clicks the button, Angular calls the onClickMe method from ClickMeComponent.

## Get user input from the \$event object

DOM events carry a payload of information that may be useful to the component. This section shows how to bind to the keyup event of an input box to get the user's input after each keystroke.

The following code listens to the keyup event and passes the entire event payload (\$event) to the component event handler.

src/app/keyup.components.ts (template v.1)

```

template: `
<input (keyup)="onKey($event)">
<p>{{values}}</p>
`
```

When a user presses and releases a key, the keyup event occurs, and Angular provides a corresponding DOM event object in the \$event variable which this code passes as a parameter to the component's onKey() method.

src/app/keyup.components.ts (class v.1)

```

export class KeyUpComponent_v1 {
  values = "";

  onKey(event: any) { // without type info
    this.values += event.target.value + ' | ';
  }
}
```

The properties of an \$event object vary depending on the type of DOM event. For example, a mouse event includes different information than an input box editing event.

All [standard DOM event objects](#) have a `target` property, a reference to the element that raised the event. In this case, `target` refers to the [`<input>` element](#) and `event.target.value` returns the current contents of that element.

After each call, the `onKey()` method appends the contents of the input box value to the list in the component's `values` property, followed by a separator character (`|`). The [interpolation](#) displays the accumulating input box changes from the `values` property.

Suppose the user enters the letters "abc", and then backspaces to remove them one by one. Here's what the UI displays:

`a | ab | abc | ab | a | |`

## Give me some keys!



Alternatively, you could accumulate the individual keys themselves by substituting `event.key` for `event.target.value` in which case the same user input would produce:

`a | b | c | backspace | backspace | backspace |`

## Type the \$event

The example above casts the `$event` as an any type. That simplifies the code at a cost. There is no type information that could reveal properties of the event object and prevent silly mistakes.

The following example rewrites the method with types:

```
src/app/keyup.components.ts (class v.1 - typed )  
  
export class KeyUpComponent_v1 {  
  values = "";  
  
  onKey(event: KeyboardEvent) { // with type info  
    this.values += (<HTMLInputElement>event.target).value + ' | ';  
  }  
}
```

The `$event` is now a specific `KeyboardEvent`. Not all elements have a `value` property so it casts `target` to an `input` element. The `OnKey` method more clearly expresses what it expects from the template and how it interprets the event.

## Passing \$event is a dubious practice

Typing the event object reveals a significant objection to passing the entire DOM event into the method: the component has too much awareness of the template details. It can't extract information without knowing more than it should about the HTML implementation. That breaks the separation of concerns between the template (what the user sees) and the component (how the application processes user data).

The next section shows how to use template reference variables to address this problem.

## Get user input from a template reference variable

There's another way to get the user data: use Angular [template reference variables](#). These variables provide direct access to an element from within the template. To declare a template reference variable, precede an identifier with a hash (or pound) character (#).

The following example uses a template reference variable to implement a keystroke loopback in a simple template.

src/app/loop-back.component.ts

```
@Component({
  selector: 'app-loop-back',
  template: `
    <input #box (keyup)="0">
    <p>{{box.value}}</p>
  `
})
export class LoopbackComponent { }
```

The template reference variable named box, declared on the `<input>` element, refers to the `<input>` element itself. The code uses the box variable to get the input element's value and display it with interpolation between `<p>` tags.

The template is completely self contained. It doesn't bind to the component, and the component does nothing.

Type something in the input box, and watch the display update with each keystroke.

**keyup loop-back component**



This won't work at all unless you bind to an event.

Angular updates the bindings (and therefore the screen) only if the app does something in response to asynchronous events, such as keystrokes. This example code binds the

keyup event to the number 0, the shortest template statement possible. While the statement does nothing useful, it satisfies Angular's requirement so that Angular will update the screen.

It's easier to get to the input box with the template reference variable than to go through the \$event object. Here's a rewrite of the previous keyup example that uses a template reference variable to get the user's input.

src/app/keyup.components.ts (v2)

```
@Component({
  selector: 'app-key-up2',
  template: `
    <input #box (keyup)="onKey(box.value)">
    <p>{{values}}</p>
  `
})
export class KeyUpComponent_v2 {
  values = '';
  onKey(value: string) {
    this.values += value + ' | ';
  }
}
```

A nice aspect of this approach is that the component gets clean data values from the view. It no longer requires knowledge of the \$event and its structure.

## Key event filtering (with key.enter)

The (keyup) event handler hears every keystroke. Sometimes only the Enter key matters, because it signals that the user has finished typing. One way to reduce the noise would be to examine every \$event.keyCode and take action only when the key is Enter.

There's an easier way: bind to Angular's keyup.enter pseudo-event. Then Angular calls the event handler only when the user presses Enter.

src/app/keyup.components.ts (v3)

```
@Component({
  selector: 'app-key-up3',
  template: `
    <input #box (keyup.enter)="onEnter(box.value)">
    <p>{{value}}</p>
  `
})
export class KeyUpComponent_v3 {
  value = '';
```

```
onEnter(value: string) { this.value = value; }
}
```

Here's how it works.

**Type away! Press [enter] when done**

## On blur

In the previous example, the current state of the input box is lost if the user mouses away and clicks elsewhere on the page without first pressing Enter. The component's value property is updated only when the user presses Enter.

To fix this issue, listen to both the Enter key and the blur event.

src/appkeyup.components.ts (v4)

```
@Component({
  selector: 'app-key-up4',
  template: `
    <input #box
      (keyup.enter)="update(box.value)"
      (blur)="update(box.value)">

    <p>{{value}}</p>
  `
})
export class KeyUpComponent_v4 {
  value = '';
  update(value: string) { this.value = value; }
}
```

## Put it all together

The previous page showed how to [display data](#). This page demonstrated event binding techniques.

Now, put it all together in a micro-app that can display a list of heroes and add new heroes to the list. The user can add a hero by typing the hero's name in the input box and clicking Add.

## Little Tour of Heroes



Add

- Windstorm
- Bombasto
- Magneta
- Tornado

Below is the "Little Tour of Heroes" component.

src/app/little-tour.component.ts

```
@Component({
  selector: 'app-little-tour',
  template: `
    <input #newHero
      (keyup.enter)="addHero(newHero.value)"
      (blur)="addHero(newHero.value); newHero.value=' '>

    <button (click)="addHero(newHero.value)">Add</button>

    <ul><li *ngFor="let hero of heroes">{{hero}}</li></ul>
  `

})
export class LittleTourComponent {
  heroes = ['Windstorm', 'Bombasto', 'Magneta', 'Tornado'];
  addHero(newHero: string) {
    if (newHero) {
      this.heroes.push(newHero);
    }
  }
}
```

## Observations

- Use template variables to refer to elements — The newHero template variable refers to the `<input>` element. You can reference newHero from any sibling or child of the `<input>` element.

- Pass values, not elements — Instead of passing the newHero into the component's addHero method, get the input box value and pass that to addHero.
- Keep template statements simple — The (blur) event is bound to two JavaScript statements. The first statement calls addHero. The second statement, newHero.value="", clears the input box after a new hero is added to the list.

## Source code

Following is all the code discussed in this page.

`click-me.component.ts`

`keyup.components.ts`

`loop-back.component.ts`

`little-tour.component.ts`

```

1. import { Component } from '@angular/core';
2.
3. @Component({
4.   selector: 'app-click-me',
5.   template: `
6.     <button (click)="onClickMe()">Click me!</button>
7.     {{clickMessage}}
8.   `)
9. export class ClickMeComponent {
10.   clickMessage = "";
11.
12.   onClickMe() {
13.     this.clickMessage = 'You are my hero!';
14.   }
15. }
```

## Summary

You have mastered the basic primitives for responding to user input and gestures.

These techniques are useful for small-scale demonstrations, but they quickly become verbose and clumsy when handling large amounts of user input. Two-way data binding is a more elegant and compact way to move values between data entry fields and model properties. The next page, [Forms](#), explains how to write two-way bindings with [NgModel](#).

A component has a lifecycle managed by Angular.

Angular creates it, renders it, creates and renders its children, checks it when its data-bound properties change, and destroys it before removing it from the DOM.

Angular offers lifecycle hooks that provide visibility into these key life moments and the ability to act when they occur.

A directive has the same set of lifecycle hooks.

## Component lifecycle hooks overview

Directive and component instances have a lifecycle as Angular creates, updates, and destroys them. Developers can tap into key moments in that lifecycle by implementing one or more of the lifecycle hook interfaces in the Angular core library.

Each interface has a single hook method whose name is the interface name prefixed with ng. For example, the [OnInit](#) interface has a hook method named ngOnInit() that Angular calls shortly after creating the component:

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

```
export class PeekABoo implements OnInit {
  constructor(private logger: LoggerService) { }

  // implement OnInit's `ngOnInit` method
  ngOnInit() { this.logIt(`OnInit`); }
```

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

No directive or component will implement all of the lifecycle hooks. Angular only calls a directive/component hook method if it is defined.

## Lifecycle sequence

After creating a component/directive by calling its constructor, Angular calls the lifecycle hook methods in the following sequence at specific moments:

Hook	Purpose and Timing
ngOnChanges()	Respond when Angular (re)sets data-bound input properties. The method receives a <a href="#">SimpleChanges</a> object of current and previous property values.
ngOnInit()	Called before ngOnInit() and whenever one or more data-bound input properties change.
	Initialize the directive/component after Angular first displays the data-bound properties and sets the directive/component's input

	properties.
	Called once, after the first <code>ngOnChanges()</code> .
	Detect and act upon changes that Angular can't or won't detect on its own.
<code>ngDoCheck()</code>	Called during every change detection run, immediately after <code>ngOnChanges()</code> and <code>ngOnInit()</code> .
<code>ngAfterContentInit()</code>	Respond after Angular projects external content into the component's view / the view that a directive is in.
	Called once after the first <code>ngDoCheck()</code> .
<u><code>ngAfterContentChecked()</code></u> )	Respond after Angular checks the content projected into the directive/component.
<code>ngAfterViewInit()</code>	Called after the <code>ngAfterContentInit()</code> and every subsequent <code>ngDoCheck()</code> .
	Respond after Angular initializes the component's views and child views / the view that a directive is in.
<u><code>ngAfterViewChecked()</code></u>	Called once after the first <u><code>ngAfterContentChecked()</code></u> .
	Respond after Angular checks the component's views and child views / the view that a directive is in.
<u><code>ngOnDestroy()</code></u>	Called after the <code>ngAfterViewInit</code> and every subsequent <u><code>ngAfterContentChecked()</code></u> .
	Cleanup just before Angular destroys the directive/component. Unsubscribe Observables and detach event handlers to avoid memory leaks.
	Called just before Angular destroys the directive/component.

## Interfaces are optional (technically)

The interfaces are optional for JavaScript and Typescript developers from a purely technical perspective. The JavaScript language doesn't have interfaces. Angular can't see TypeScript interfaces at runtime because they disappear from the transpiled JavaScript.

Fortunately, they aren't necessary. You don't have to add the lifecycle hook interfaces to directives and components to benefit from the hooks themselves.

Angular instead inspects directive and component classes and calls the hook methods if they are defined. Angular finds and calls methods like `ngOnInit()`, with or without the interfaces.

Nonetheless, it's good practice to add interfaces to TypeScript directive classes in order to benefit from strong typing and editor tooling.

## Other Angular lifecycle hooks

Other Angular sub-systems may have their own lifecycle hooks apart from these component hooks.

3rd party libraries might implement their hooks as well in order to give developers more control over how these libraries are used.

## Lifecycle examples

The [live example](#) / [download example](#) demonstrates the lifecycle hooks in action through a series of exercises presented as components under the control of the root `AppComponent`.

They follow a common pattern: a parent component serves as a test rig for a child component that illustrates one or more of the lifecycle hook methods.

Here's a brief description of each exercise:

Component	Description
<a href="#">Peek-a-boo</a>	Demonstrates every lifecycle hook. Each hook method writes to the on-screen log.
<a href="#">Spy</a>	Directives have lifecycle hooks too. A <code>SpyDirective</code> can log when the element it spies upon is created or destroyed using the <code>ngOnInit</code> and <code>ngOnDestroy</code> hooks.
<a href="#">OnChanges</a>	This example applies the <code>SpyDirective</code> to a <code>&lt;div&gt;</code> in an <code>ngFor</code> hero repeater managed by the parent <code>SpyComponent</code> .
<a href="#">DoCheck</a>	See how Angular calls the <code>ngOnChanges()</code> hook with a <code>changes</code> object every time one of the component input properties changes. Shows how to interpret the <code>changes</code> object.
<a href="#">AfterView</a>	Implements an <code>ngDoCheck()</code> method with custom change detection. See how often Angular calls this hook and watch it post changes to a log.
<a href="#">AfterContent</a>	Shows what Angular means by a view. Demonstrates the <code>ngAfterViewInit</code> and <code>ngAfterViewChecked</code> hooks.

ngAfterContentInit and ngAfterContentChecked hooks.

Demonstrates a combination of a component and a directive each with its own hooks.

Counter	In this example, a CounterComponent logs a change (via ngOnChanges) every time the parent component increments its input counter property. Meanwhile, the SpyDirective from the previous example is applied to the CounterComponent log where it watches log entries being created and destroyed.
---------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

The remainder of this page discusses selected exercises in further detail.

## Peek-a-boo: all hooks

The PeekABooComponent demonstrates all of the hooks in one component.

You would rarely, if ever, implement all of the interfaces like this. The peek-a-boo exists to show how Angular calls the hooks in the expected order.

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

```
-- Lifecycle Hook Log --
#1 name is not known at construction
#2 OnChanges: name initialized to "Windstorm"
#3 OnInit
#4 DoCheck
#5 AfterContentInit
#6 AfterContentChecked
#7 AfterViewInit
#8 AfterViewChecked
#9 DoCheck
#10 AfterContentChecked
#11 AfterViewChecked
#12 DoCheck
#13 AfterContentChecked
#14 AfterViewChecked
#15 OnDestroy
```

The sequence of log messages follows the prescribed hook calling order: [OnChanges](#), [OnInit](#), [DoCheck](#) (3x), [AfterContentInit](#), [AfterContentChecked](#) (3x), [AfterViewInit](#), [AfterViewChecked](#) (3x), and [.onDestroy](#).

The constructor isn't an Angular hook per se. The log confirms that input properties (the name property in this case) have no assigned values at construction.

Had the user clicked the Update Hero button, the log would show another [OnChanges](#) and two more triplets of [DoCheck](#), [AfterContentChecked](#) and [AfterViewChecked](#). Clearly these three hooks fire often. Keep the logic in these hooks as lean as possible!

The next examples focus on hook details.

## Spying OnInit and OnDestroy

Go undercover with these two spy hooks to discover when an element is initialized or destroyed.

This is the perfect infiltration job for a directive. The heroes will never know they're being watched.

Kidding aside, pay attention to two key points:

1. Angular calls hook methods for directives as well as components.
2. A spy directive can provide insight into a DOM object that you cannot change directly.  
Obviously you can't touch the implementation of a native <div>. You can't modify a third party component either. But you can watch both with a directive.

The sneaky spy directive is simple, consisting almost entirely of `ngOnInit()` and `ngOnDestroy()` hooks that log messages to the parent via an injected `LoggerService`.

`src/app/spy.directive.ts`

```
// Spy on any element to which it is applied.
// Usage: <div mySpy>...</div>
@Directive({selector: '[mySpy]'})
export class SpyDirective implements OnInit, OnDestroy {

  constructor(private logger: LoggerService) { }

  ngOnInit() { this.logIt('onInit'); }

  ngOnDestroy() { this.logIt('onDestroy'); }

  private logIt(msg: string) {
    this.logger.log(`Spy #${nextId++} ${msg}`);
  }
}
```

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

src/app/spy.component.html

```
<div *ngFor="let hero of heroes" mySpy class="heroes">  
  {{hero}}  
</div>
```

Each spy's birth and death marks the birth and death of the attached hero <div> with an entry in the Hook Log as seen here:



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.

The ngOnInit() and ngOnDestroy() methods have more vital roles to play in real applications.

## OnInit()

Use ngOnInit() for two main reasons:

1. To perform complex initializations shortly after construction.
2. To set up the component after Angular sets the input properties.

Experienced developers agree that components should be cheap and safe to construct.

Misko Hevery, Angular team lead, [explains why](#) you should avoid complex constructor logic.

Don't 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. Constructors should do no more than set the initial local variables to simple values.

An `ngOnInit()` is a good place for a component to fetch its initial data. The [Tour of Heroes Tutorial](#) guide shows how.

Remember also that a directive's data-bound input properties are not set until after construction. That's a problem if you need to initialize the directive based on those properties. They'll have been set when `ngOnInit()` runs.

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

You can count on Angular to call the `ngOnInit()` method soon after creating the component. That's where the heavy initialization logic belongs.

## OnDestroy()

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

This is the time to notify another part of the application that the component is going away.

This is the place to free resources that won't be garbage collected automatically. Unsubscribe from Observables and DOM events. Stop interval timers. Unregister all callbacks that this directive registered with global or application services. You risk memory leaks if you neglect to do so.

## OnChanges()

Angular calls its `ngOnChanges()` method whenever it detects changes to input properties of the component (or directive). This example monitors the [OnChanges hook](#).

`on-changes.component.ts` (excerpt)

```
ngOnChanges(changes: SimpleChanges) {
  for (let propName in changes) {
    let chng = changes[propName];
    let cur = JSON.stringify(chng.currentValue);
    let 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](#) 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`

```
@Input() hero: Hero;  
@Input() power: string;
```

The host OnChangesParentComponent binds to them like this:

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

```
<on-changes [hero]="hero" [power]="power"></on-changes>
```

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

The screenshot shows a web application interface titled "OnChanges". At the top, there are two input fields: "Power:" with the value "sing" and "Hero.name:" with the value "Windstorm". Below the inputs is a "Reset Log" button. A yellow bar displays the message "Windstorm can sing". Underneath the bar, a log section is titled "-- Change Log --" and contains the entries "hero: currentValue = {"name": "Windstorm"}, previousValue = {}" and "power: currentValue = "sing", previousValue = {}". At the bottom left, there is a link "back to top".

The log entries appear as the string value of the power property changes. But the ngOnChanges does not catch changes to hero.name. That's surprising at first.

Angular only calls the hook when the value of the input property changes. The value of the hero property is the reference to the hero object. Angular doesn't care that the hero's own name property changed. The hero object reference didn't change so, from Angular's perspective, there is no change to report!

## DoCheck()

Use the [DoCheck](#) hook to detect and act upon changes that Angular doesn't catch on its own.

Use this method to detect a change that Angular overlooked.

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

```

DoCheckComponent (ngDoCheck)
ngDoCheck() {

  if (this.hero.name !== this.oldHeroName) {
    this.changeDetected = true;
    this.changeLog.push(`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: 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.
    let count = this.noChangeCount += 1;
    let noChangeMsg = `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](#) is called. The results are illuminating:

## DoCheck

```
Power: sing  
Hero.name: Windstorm
```

Reset Log

Windstorm can sing

-- Change Log --

```
OnChanges: hero: currentValue = {"name": "Windstorm"}, previousValue = {}  
OnChanges: power: currentValue = "sing", previousValue = ""  
DoCheck: Hero name changed to "Windstorm" from ""  
DoCheck: Power changed to "sing" from ""  
DoCheck called 26x when no change to hero or power
```

While the `ngDoCheck()` hook can detect when the hero's name has changed, it has a frightful cost. 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. Mere mousing into another `<input>` triggers a call. Relatively few calls reveal actual changes to pertinent data. Clearly our implementation must be very lightweight or the user experience suffers.

## AfterView

The AfterView sample explores the `AfterViewInit()` and `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>`:

ChildComponent

```
@Component({  
  selector: 'app-child-view',
```

```
template: '<input [(ngModel)]="hero">'  
}  
export class ChildViewComponent {  
  hero = 'Magneta';  
}
```

The AfterViewComponent displays this child view within its template:

AfterViewComponent (template)

```
template: `  
<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 via the property decorated with [@ViewChild](#).

AfterViewComponent (class excerpts)

```
export class AfterViewComponent implements AfterViewChecked, AfterViewInit {  
  private prevHero = "";
```

```
// Query for a VIEW child of type `ChildViewComponent`  
@ViewChild(ChildViewComponent) viewChild: ChildViewComponent;
```

```
ngAfterViewInit() {  
  // viewChild is set after the view has been initialized  
  this.logIt('AfterViewInit');  
  this.doSomething();  
}
```

```
ngAfterViewChecked() {  
  // viewChild is updated after the view has been checked  
  if (this.prevHero === this.viewChild.hero) {  
    this.logIt('AfterViewChecked (no change)');  
  } else {  
    this.prevHero = this.viewChild.hero;  
    this.logIt('AfterViewChecked');  
    this.doSomething();  
  }  
}  
// ...  
}
```

## Abide by the unidirectional data flow rule

The doSomething() method updates the screen when the hero name exceeds 10 characters.

AfterViewComponent (doSomething)

```
// This surrogate for real business logic sets the `comment`  
private doSomething() {  
  let c = this.viewChild.hero.length > 10 ? `That's a long name` : "  
  if (c !== this.comment) {  
    // Wait a tick because the component's view has already been checked  
    this.logger.tick_then(() => this.comment = c);  
  }  
}
```

Why does the doSomething() method wait a tick before updating comment?

Angular's unidirectional data flow rule forbids updates to the view after it has been composed. Both of these hooks fire after the component's view has been composed.

Angular throws an error if the hook updates the component's data-bound comment property immediately (try it!). The LoggerService.tick\_then() postpones the log update for one turn of the browser's JavaScript cycle and that's just long enough.

Here's AfterView in action:

## AfterView

-- child view begins --

Magneta

-- child view ends --

-- AfterView Logs --

Reset

AfterView constructor: no child view

AfterViewInit: Magneta child view

AfterViewChecked: Magneta child view

[back to top](#)

Notice that Angular frequently calls [AfterViewChecked\(\)](#), often when there are no changes of interest. Write lean hook methods to avoid performance problems.

## AfterContent

The AfterContent sample explores the [AfterContentInit\(\)](#) and [AfterContentChecked\(\)](#) hooks that Angular calls after Angular projects external content into the component.

### Content projection

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.

AngularJS developers know this technique as transclusion.

Consider this variation on the [previous AfterView](#) example. This time, instead of including the child view within the template, it imports the content from the AfterContentComponent's parent. Here's the parent's template:

AfterContentParentComponent (template excerpt)

```
`<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.

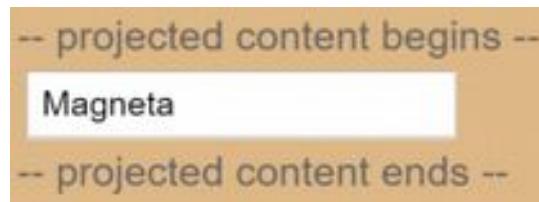
Now look at the component's template:

AfterContentComponent (template)

template:

```
<div>-- projected content begins --</div>
  <ng-content></ng-content>
<div>-- projected content ends --</div>`
```

The `<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.



The telltale signs of content projection are twofold:

- HTML between component element tags.
- The presence of `<ng-content>` tags in the component's template.

## AfterContent hooks

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

- The AfterView hooks concern [ViewChildren](#), the child components whose element tags appear within the component's template.
- The AfterContent hooks concern [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 via the property decorated with [@ContentChild](#).

AfterContentComponent (class excerpts)

```
export class AfterContentComponent implements AfterContentChecked, AfterContentInit {
  private prevHero = "";
```

```

comment = "";

// Query for a CONTENT child of type `ChildComponent`
@ContentChild(ChildComponent) contentChild: ChildComponent;

ngAfterContentInit() {
  // contentChild is set after the content has been initialized
  this.logIt('AfterContentInit');
  this.doSomething();
}

ngAfterContentChecked() {
  // contentChild is updated after the content has been checked
  if (this.prevHero === this.contentChild.hero) {
    this.logIt('AfterContentChecked (no change)');
  } else {
    this.prevHero = this.contentChild.hero;
    this.logIt('AfterContentChecked');
    this.doSomething();
  }
}
// ...
}

```

## No unidirectional flow worries with AfterContent

This component's doSomething() method update's the component's data-bound comment property immediately. There's no [need to wait](#).

Recall that 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 to modify the host view.

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

See the [live example](#) / [download example](#).

## Pass data from parent to child with input binding

HeroChildComponent has two input properties, typically adorned with [@Input](#) decorations.  
 component-interaction/src/app/hero-child.component.ts

1. import {[Component](#), [Input](#)} from '@angular/core';

```

2.
3. import { Hero } from './hero';
4.
5. @Component({
6.   selector: 'app-hero-child',
7.   template: `
8.     <h3>{{hero.name}} says:</h3>
9.     <p>I, {{hero.name}}, am at your service, {{masterName}}.</p>
10.    `
11.  })
12.export class HeroChildComponent {
13.   @Input() hero: Hero;
14.   @Input('master') masterName: string;
15. }

```

The second `@Input` aliases the child component property name `masterName` as 'master'.

The `HeroParentComponent` nests the child `HeroChildComponent` inside an `*ngFor` 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`

```

1. import { Component } from '@angular/core';
2.
3. import { HEROES } from './hero';
4.
5. @Component({
6.   selector: 'app-hero-parent',
7.   template: `
8.     <h2>{{master}} controls {{heroes.length}} heroes</h2>
9.     <app-hero-child *ngFor="let hero of heroes"
10.       [hero]="hero"
11.       [master]="master">
12.     </app-hero-child>
13.    `
14.  })
15.export class HeroParentComponent {
16.   heroes = HEROES;
17.   master = 'Master';
18. }

```

The running application displays three heroes:

## Master controls 3 heroes

**Mr. IQ says:**

I, Mr. IQ, am at your service, Master.

**Magneta says:**

I, Magneta, am at your service, Master.

**Bombasto says:**

I, Bombasto, am at your service, Master.

### Test it

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

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

```
1. // ...
2. let _heroNames = ['Mr. IQ', 'Magneta', 'Bombasto'];
3. let _masterName = 'Master';
4.
5. it('should pass properties to children properly', function () {
6.   let parent = element.all(by.tagName('app-hero-parent')).get(0);
7.   let heroes = parent.all(by.tagName('app-hero-child'));
8.
9.   for (let i = 0; i < _heroNames.length; i++) {
10.     let childTitle = heroes.get(i).element(by.tagName('h3')).getText();
11.     let childDetail = heroes.get(i).element(by.tagName('p')).getText();
12.     expect(childTitle).toEqual(_heroNames[i] + ' says:');
13.     expect(childDetail).toContain(_masterName);
14.   }
15. });
16.// ...
```

[Back to 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

```
1. import { Component, Input } from '@angular/core';
2.
3. @Component({
4.   selector: 'app-name-child',
5.   template: '<h3>"{{name}}"</h3>'
6. })
7. export class NameChildComponent {
8.   private _name = "";
9.
10. @Input()
11. set name(name: string) {
12.   this._name = (name && name.trim()) || '<no name set>';
13. }
14.
15. get name(): string { return this._name; }
16.}
```

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

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

```
1. import { Component } from '@angular/core';
2.
3. @Component({
4.   selector: 'app-name-parent',
5.   template: `
6.     <h2>Master controls {{names.length}} names</h2>
7.     <app-name-child *ngFor="let name of names" [name]="name"></app-name-child>
8.   `
9. })
10.export class NameParentComponent {
11. // Displays 'Mr. IQ', '<no name set>', 'Bombasto'
12. names = ['Mr. IQ', ' ', ' Bombasto '];
13.}
```

# Master controls 3 names

"Mr. IQ"

"<no name set>"

"Bombasto"

## Test it

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

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

```
1. // ...
2. it('should display trimmed, non-empty names', function () {
3.   let _nonEmptyNameIndex = 0;
4.   let _nonEmptyName = "Mr. IQ";
5.   let parent = element.all(by.tagName('app-name-parent')).get(0);
6.   let hero = parent.all(by.tagName('app-name-child')).get(_nonEmptyNameIndex);
7.
8.   let displayName = hero.element(by.tagName('h3')).getText();
9.   expect(displayName).toEqual(_nonEmptyName);
10.});
11.
12.it('should replace empty name with default name', function () {
13. let _emptyNameIndex = 1;
14. let _defaultName = "<no name set>";
15. let parent = element.all(by.tagName('app-name-parent')).get(0);
16. let hero = parent.all(by.tagName('app-name-child')).get(_emptyNameIndex);
17.
18. let displayName = hero.element(by.tagName('h3')).getText();
19. expect(displayName).toEqual(_defaultName);
20.});
21.// ...
```

[Back to top](#)

## Intercept input property changes with ngOnChanges()

Detect and act upon changes to input property values with the ngOnChanges() method of the [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](#) 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`

```
1. import { Component, Input, OnChanges, SimpleChange } from '@angular/core';
2.
3. @Component({
4.   selector: 'app-version-child',
5.   template: `
6.     <h3>Version {{major}}.{{minor}}</h3>
7.     <h4>Change log:</h4>
8.     <ul>
9.       <li *ngFor="let change of changeLog">{{change}}</li>
10.    </ul>
11.  `
12.})
13.export class VersionChildComponent implements OnChanges {
14.   @Input() major: number;
15.   @Input() minor: number;
16.   changeLog: string[] = [];
17.
18.   ngOnChanges(changes: {[propKey: string]: SimpleChange}) {
19.     let log: string[] = [];
20.     for (let propName in changes) {
21.       let changedProp = changes[propName];
22.       let to = JSON.stringify(changedProp.currentValue);
23.       if (changedProp.isFirstChange()) {
24.         log.push(`Initial value of ${propName} set to ${to}`);
25.       } else {
26.         let from = JSON.stringify(changedProp.previousValue);
27.         log.push(`${propName} changed from ${from} to ${to}`);
28.       }
29.     }
30.     this.changeLog.push(log.join(', '));
31.   }
32.}
```

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

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

```
1. import { Component } from '@angular/core';
2.
3. @Component({
4.   selector: 'app-version-parent',
5.   template: `
6.     <h2>Source code version</h2>
7.     <button (click)="newMinor()">New minor version</button>
8.     <button (click)="newMajor()">New major version</button>
9.     <app-version-child [major]="major" [minor]="minor"></app-version-child>
10.    `
11.  })
12. export class VersionParentComponent {
13.   major = 1;
14.   minor = 23;
15.
16.   newMinor() {
17.     this.minor++;
18.   }
19.
20.   newMajor() {
21.     this.major++;
22.     this.minor = 0;
23.   }
24. }
```

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

## Source code version

New minor version    New major version

Version 1.23

Change log:

- Initial value of major set to 1, Initial value of minor set to 23

## 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

```
1. // ...
2. // Test must all execute in this exact order
3. it('should set expected initial values', function () {
4.   let actual = getActual();
5.
6.   let initialLabel = 'Version 1.23';
7.   let initialLog = 'Initial value of major set to 1, Initial value of minor set to 23';
8.
9.   expect(actual.label).toBe(initialLabel);
10.  expect(actual.count).toBe(1);
11.  expect(actual.logs.get(0).getText()).toBe(initialLog);
12.});
13.
14.it('should set expected values after clicking \'Minor\' twice', function () {
15.  let repoTag = element(by.tagName('app-version-parent'));
16.  let newMinorButton = repoTag.all(by.tagName('button')).get(0);
17.
18.  newMinorButton.click().then(function() {
19.    newMinorButton.click().then(function() {
20.      let actual = getActual();
21.
22.      let labelAfter2Minor = 'Version 1.25';
23.      let logAfter2Minor = 'minor changed from 24 to 25';
24.
25.      expect(actual.label).toBe(labelAfter2Minor);
26.      expect(actual.count).toBe(3);
27.      expect(actual.logs.get(2).getText()).toBe(logAfter2Minor);
28.    });
29.  });
30.});
31.
32.it('should set expected values after clicking \'Major\' once', function () {
33.  let repoTag = element(by.tagName('app-version-parent'));
34.  let newMajorButton = repoTag.all(by.tagName('button')).get(1);
35.
36.  newMajorButton.click().then(function() {
37.    let actual = getActual();
```

```

38.
39. let labelAfterMajor = 'Version 2.0';
40. let logAfterMajor = 'major changed from 1 to 2, minor changed from 25 to 0';
41.
42. expect(actual.label).toBe(labelAfterMajor);
43. expect(actual.count).toBe(4);
44. expect(actual.logs.get(3).getText()).toBe(logAfterMajor);
45. });
46.});
47.
48.function getActual() {
49. let versionTag = element(by.tagName('app-version-child'));
50. let label = versionTag.element(by.tagName('h3')).getText();
51. let ul = versionTag.element((by.tagName('ul')));
52. let logs = ul.all(by.tagName('li'));
53.
54. return {
55. label: label,
56. logs: logs,
57. count: logs.count()
58. };
59.}
60./ ...

```

[Back to top](#)

## Parent listens for child event

The child component exposes an [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](#) property is an output property, typically adorned with an [@Output decoration](#) as seen in this VoterComponent:

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

```

1. import { Component, EventEmitter, Input, Output } from '@angular/core';
2.
3. @Component({
4. selector: 'app-voter',
5. template: `
6.   <h4>{{name}}</h4>
7.   <button (click)="vote(true)" [disabled]="didVote">Agree</button>
8.   <button (click)="vote(false)" [disabled]="didVote">Disagree</button>
9. `

```

```

10.})
11.export class VoterComponent {
12. @Input() name: string;
13. @Output() voted = new EventEmitter<boolean>();
14. didVote = false;
15.
16. vote(agreed: boolean) {
17.   this.voted.emit(agreed);
18.   this.didVote = true;
19. }
20.}

```

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

```

1. import { Component } from '@angular/core';
2.
3. @Component({
4. selector: 'app-vote-taker',
5. template: `
6. <h2>Should mankind colonize the Universe?</h2>
7. <h3>Agree: {{agreed}}, Disagree: {{disagreed}}</h3>
8. <app-voter *ngFor="let voter of voters"
9.   [name]="voter"
10.  (voted)="onVoted($event)">
11. </app-voter>
12. `
13.})
14.export class VoteTakerComponent {
15. agreed = 0;
16. disagreed = 0;
17. voters = ['Mr. IQ', 'Ms. Universe', 'Bombasto'];
18.
19. onVoted(agreed: boolean) {
20.   agreed ? this.agreed++ : this.disagreed++;
21. }
22.}

```

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

# Should mankind colonize the Universe?

Agree: 0, Disagree: 0

Mr. IQ

Ms. Universe

Bombasto

## Test it

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

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

```
1. // ...
2. it('should not emit the event initially', function () {
3.   let voteLabel = element(by.tagName('app-vote-taker'))
4.   .element(by.tagName('h3')).getText();
5.   expect(voteLabel).toBe('Agree: 0, Disagree: 0');
6. });
7.
8. it('should process Agree vote', function () {
9.   let agreeButton1 = element.all(by.tagName('app-voter')).get(0)
10.  .all(by.tagName('button')).get(0);
11.  agreeButton1.click().then(function() {
12.    let voteLabel = element(by.tagName('app-vote-taker'))
13.    .element(by.tagName('h3')).getText();
14.    expect(voteLabel).toBe('Agree: 1, Disagree: 0');
15.  });
16.});
17.
18.it('should process Disagree vote', function () {
19. let agreeButton1 = element.all(by.tagName('app-voter')).get(1)
```

```
20. .all(by.tagName('button')).get(1);
21. agreeButton1.click().then(function() {
22.   let voteLabel = element(by.tagName('app-vote-taker'))
23.   .element(by.tagName('h3')).getText();
24.   expect(voteLabel).toBe('Agree: 1, Disagree: 1');
25. });
26.});
27.// ...
```

[Back to top](#)

## Parent interacts with child via 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

```
1. import { Component, OnDestroy, OnInit } from '@angular/core';
2.
3. @Component({
4.   selector: 'app-countdown-timer',
5.   template: '<p>{{message}}</p>'
6. })
7. export class CountdownTimerComponent implements OnInit, OnDestroy {
8.
9.   intervalId = 0;
10.  message = '';
11.  seconds = 11;
12.
13.  clearTimer() { clearInterval(this.intervalId); }
14.
15.  ngOnInit() { this.start(); }
16.  ngOnDestroy() { this.clearTimer(); }
17.
18.  start() { this.countDown(); }
19.  stop() {
20.    this.clearTimer();
21.    this.message = `Holding at T-${this.seconds} seconds`;
22.  }
```

```

23.
24. private countDown() {
25.   this.clearTimer();
26.   this.intervalId = window.setInterval(() => {
27.     this.seconds -= 1;
28.     if (this.seconds === 0) {
29.       this.message = 'Blast off!';
30.     } else {
31.       if (this.seconds < 0) { this.seconds = 10; } // reset
32.       this.message = `T-${this.seconds} seconds and counting`;
33.     }
34.   }, 1000);
35. }
36.

```

The CountdownLocalVarParentComponent that hosts the timer component is as follows:

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

```

1. import { Component }           from '@angular/core';
2. import { CountdownTimerComponent } from './countdown-timer.component';
3.
4. @Component({
5.   selector: 'app-countdown-parent-lv',
6.   template: `
7.     <h3>Countdown to Liftoff (via local variable)</h3>
8.     <button (click)="timer.start()">Start</button>
9.     <button (click)="timer.stop()">Stop</button>
10.    <div class="seconds">{{timer.seconds}}</div>
11.    <app-countdown-timer #timer></app-countdown-timer>
12.  `,
13.   styleUrls: ['./assets/demo.css']
14. })
15.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 secondsproperty.

Here we see the parent and child working together.

## Countdown to Liftoff

[Start](#) [Stop](#)

10

T-10 seconds and counting

### 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](#)

```
1. // ...
2. it('timer and parent seconds should match', function () {
3.   let parent = element(by.tagName(parentTag));
4.   let message = parent.element(by.tagName('app-countdown-timer')).getText();
5.   browser.sleep(10); // give `seconds` a chance to catchup with `message`
6.   let seconds = parent.element(by.className('seconds')).getText();
7.   expect(message).toContain(seconds);
8. });
9.
10.it('should stop the countdown', function () {
11. let parent = element(by.tagName(parentTag));
12. let stopButton = parent.all(by.tagName('button')).get(1);
13.
14. stopButton.click().then(function() {
15.   let message = parent.element(by.tagName('app-countdown-timer')).getText();
16.   expect(message).toContain('Holding');
17. });
18.});
19.// ...
```

[Back to top](#)

## Parent calls an @ViewChild()

The local variable approach is simple and 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 an instance of the parent component class must read or write child component values or must call child component 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](#) example. Neither its appearance nor its behavior will change. The child [CountdownTimerComponent](#) 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`

```
1. import { AfterViewInit, ViewChild } from '@angular/core';
2. import { Component }           from '@angular/core';
3. import { CountdownTimerComponent } from './countdown-timer.component';
4.
5. @Component({
6.   selector: 'app-countdown-parent-vc',
7.   template: `
8.     <h3>Countdown to Liftoff (via ViewChild)</h3>
9.     <button (click)="start()">Start</button>
10.    <button (click)="stop()">Stop</button>
11.    <div class="seconds">{{ seconds() }}</div>
12.    <app-countdown-timer></app-countdown-timer>
13.  `,
14.   styleUrls: ['./assets/demo.css']
15. })
16. export class CountdownViewChildParentComponent implements AfterViewInit {
17.
18.   @ViewChild(CountdownTimerComponent)
19.   private timerComponent: CountdownTimerComponent;
20.
21.   seconds() { return 0; }
22.
23.   ngAfterViewInit() {
24.     // Redefine `seconds()` to get from the `CountdownTimerComponent.seconds` ...
25.     // but wait a tick first to avoid one-time devMode
26.     // unidirectional-data-flow-violation error
27.     setTimeout(() => this.seconds = () => this.timerComponent.seconds, 0);
28.   }
29.
```

```
30. start() { this.timerComponent.start(); }  
31. stop() { this.timerComponent.stop(); }  
32.}
```

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](#) decorator and the [AfterViewInit](#) lifecycle hook.

Next, inject the child CountdownTimerComponent into the private timerComponent property via the [@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 app 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](#) as before.

[Back to top](#)

## Parent and children communicate via 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

1. `import { Injectable } from '@angular/core';`
2. `import { Subject } from 'rxjs';`
- 3.

```

4. @Injectable()
5. export class MissionService {
6.
7.   // Observable string sources
8.   private missionAnnouncedSource = new Subject<string>();
9.   private missionConfirmedSource = new Subject<string>();
10.
11.  // Observable string streams
12.  missionAnnounced$ = this.missionAnnouncedSource.asObservable();
13.  missionConfirmed$ = this.missionConfirmedSource.asObservable();
14.
15.  // Service message commands
16.  announceMission(mission: string) {
17.    this.missionAnnouncedSource.next(mission);
18.  }
19.
20.  confirmMission(astronaut: string) {
21.    this.missionConfirmedSource.next(astronaut);
22.  }
23.}

```

The MissionControlComponent both provides the instance of the service that it shares with its children (through the providersmetadata array) and injects that instance into itself through its constructor:

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

```

1. import { Component }      from '@angular/core';
2.
3. import { MissionService } from './mission.service';
4.
5. @Component({
6.   selector: 'app-mission-control',
7.   template: `
8.     <h2>Mission Control</h2>
9.     <button (click)="announce()">Announce mission</button>
10.    <app-astronaut *ngFor="let astronaut of astronauts">
11.      [astronaut]="astronaut">
12.    </app-astronaut>
13.    <h3>History</h3>
14.    <ul>
15.      <li *ngFor="let event of history">{{event}}</li>
16.    </ul>
17.  `,

```

```

18. providers: [MissionService]
19.})
20.export class MissionControlComponent {
21. astronauts = ['Lovell', 'Swigert', 'Haise'];
22. history: string[] = [];
23. missions = ['Fly to the moon!',
24.     'Fly to mars!',
25.     'Fly to Vegas!'];
26. nextMission = 0;
27.
28. constructor(private missionService: MissionService) {
29.   missionService.missionConfirmed$.subscribe(
30.     astronaut => {
31.       this.history.push(` ${astronaut} confirmed the mission`);
32.     });
33. }
34.
35. announce() {
36.   let mission = this.missions[this.nextMission++];
37.   this.missionService.announceMission(mission);
38.   this.history.push(`Mission "${mission}" announced`);
39.   if (this.nextMission >= this.missions.length) { this.nextMission = 0; }
40. }
41.}

```

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

```

1. import { Component, Input, OnDestroy } from '@angular/core';
2.
3. import { MissionService } from './mission.service';
4. import { Subscription } from 'rxjs';
5.
6. @Component({
7.   selector: 'app-astronaut',
8.   template: `
9.     <p>
10.       {{astronaut}}: <strong>{{mission}}</strong>
11.       <button
12.         (click)="confirm()"
13.         [disabled]="!announced || confirmed">

```

```

14.    Confirm
15.  </button>
16. </p>
17. `
18.})
19.export class AstronautComponent implements OnDestroy {
20. @Input\(\) astronaut: string;
21. mission = '<no mission announced>';
22. confirmed = false;
23. announced = false;
24. subscription: Subscription;
25.
26. constructor(private missionService: MissionService) {
27.   this.subscription = missionService.missionAnnounced$.subscribe(
28.     mission => {
29.       this.mission = mission;
30.       this.announced = true;
31.       this.confirmed = false;
32.     });
33. }
34.
35. confirm() {
36.   this.confirmed = true;
37.   this.missionService.confirmMission(this.astronaut);
38. }
39.
40. ngOnDestroy() {
41.   // prevent memory leak when component destroyed
42.   this.subscription.unsubscribe();
43. }
44.}

```

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 app because the lifetime of a AstronautComponent is the same as the lifetime of the app 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:

# Mission Control

Announce mission

Lovell: <no mission announced>

Swigert: <no mission announced>

Haise: <no mission announced>

## History

### 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

```
1. // ...
2. it('should announce a mission', function () {
3.   let missionControl = element(by.tagName('app-mission-control'));
4.   let announceButton = missionControl.all(by.tagName("button")).get(0);
5.   announceButton.click().then(function () {
6.     let history = missionControl.all(by.tagName('li'));
7.     expect(history.count()).toBe(1);
8.     expect(history.get(0).getText()).toMatch(/Mission.* announced/);
9.   });
10.});
11.
12.it('should confirm the mission by Lovell', function () {
13. testConfirmMission(1, 2, 'Lovell');
14.});
15.
16.it('should confirm the mission by Haise', function () {
17. testConfirmMission(3, 3, 'Haise');
18.});
```

```
19.  
20.it('should confirm the mission by Swigert', function () {  
21. testConfirmMission(2, 4, 'Swigert');  
22.});  
23.  
24.function testConfirmMission(buttonIndex: number, expectedLogCount: number, astron  
aut: string) {  
25. let _confirmedLog = ' confirmed the mission';  
26. let missionControl = element(by.tagName('app-mission-control'));  
27. let confirmButton = missionControl.all(by.tagName('button')).get(buttonIndex);  
28. confirmButton.click().then(function () {  
29. let history = missionControl.all(by.tagName('li'));  
30. expect(history.count()).toBe(expectedLogCount);  
31. expect(history.get(expectedLogCount - 1).getText()).toBe(astronaut  
+ _confirmedLog);  
32.});  
33.}  
34.// ...
```

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](#) / [download example](#) 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

```
@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` 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](#) page on the [W3C](#) 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`

```
: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

```
:host(.active) {  
  border-width: 3px;  
}
```

## **: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

```
: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.

Use the `/deep/` shadow-piercing descendant combinator to force a style down through the child component tree into all the child component views. The `/deep/` combinator works to any depth of nested components, and it applies to both the view children and content children of the component.

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`

```
:host /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 [Controlling view encapsulation](#) section.

The shadow-piercing descendant combinator is deprecated and [support is being removed from major browsers](#) 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](#) decorator.

Each string in the array defines some CSS for this component.

`src/app/hero-app.component.ts` (CSS inline)

```
@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 CLI defines an empty styles array when you create the component with the --inline-style flag.

```
ng 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](#) decorator:

```
src/app/hero-app.component.ts (CSS in file)
```

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

```
@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](#).

The CLI creates an empty styles file for you by default and references that file in the component's generated [styleUrls](#).

```
ng generate component hero-app
```

## Template inline styles

You can embed CSS styles directly into the HTML template by putting them inside [<style>](#) tags.

```
src/app/hero-controls.component.ts
```

1. [@Component](#){
2. selector: 'app-hero-controls',
3. [template](#): `
4. [<style>](#)

```
5. button {
6.   background-color: white;
7.   border: 1px solid #777;
8. }
9. </style>
10. <h3>Controls</h3>
11. <button (click)="activate()">Activate</button>
12. `
13.})
```

## Template link tags

You can also write `<link>` tags into the component's HTML template.

src/app/hero-team.component.ts

```
1. @Component({
2. selector: 'app-hero-team',
3. template: `
4.   <!-- We must use a relative URL so that the AOT compiler can find the stylesheet -->
5.   <link rel="stylesheet" href="../assets/hero-team.component.css">
6.   <h3>Team</h3>
7.   <ul>
8.     <li *ngFor="let member of hero.team">
9.       {{member}}
10.    </li>
11.   </ul>
12. })
```

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 [CLI documentation](#).

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](#) on the [MDN site](#).

In this case, the URL is relative to the CSS file into which you're importing.

src/app/hero-details.component.css (excerpt)

```
/* 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 [CLI documentation](#) to learn more.

## Non-CSS style files

If you're building with the CLI, you can write style files in [sass](#), [less](#), or [stylus](#) and specify those files in the [@Component.styleUrls](#) metadata with the appropriate extensions (.scss, .less, .styl) as in the following example:

```
@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 [CLI documentation](#).

Style strings added to the [@Component.styles](#) array must be written in CSS because the CLI cannot apply a preprocessor to inline styles.

## View encapsulation

As discussed earlier, 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](#) on the [MDN](#) site) 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.
- [Native](#) view encapsulation uses a now deprecated version of the browser's native shadow DOM implementation - [learn about the changes](#).

- [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 [Appendix 1](#).
- [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 is essentially the same as pasting the component's styles into the HTML.

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

```
src/app/quest-summary.component.ts
```

```
// warning: few browsers support shadow DOM encapsulation at this time
encapsulation: ViewEncapsulation.Native
```

[ShadowDom](#) view encapsulation only works on browsers that have native support for shadow DOM (see [Shadow DOM v1](#) on the [Can I use](#) site). 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:

```
<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 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:

```
[_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.

Angular elements are Angular components packaged as custom elements, a web standard for defining new HTML elements in a framework-agnostic way.

[Custom elements](#) are a Web Platform feature currently supported by Chrome, Opera, and Safari, and available in other browsers through polyfills (see [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 (also called Web Components), which maps an instantiable JavaScript class to an HTML tag.

The `@angular/elements` package exports a [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](#).

## 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 app
- Transforming a component to a custom element provides an easy path to creating dynamic HTML content in your Angular app. HTML content that you add directly to the DOM in an Angular app is normally displayed without Angular processing, unless you

define a dynamic component, adding your own code to connect the HTML tag to your app data, and participate in change detection. With a custom element, all of that wiring is taken care of automatically.

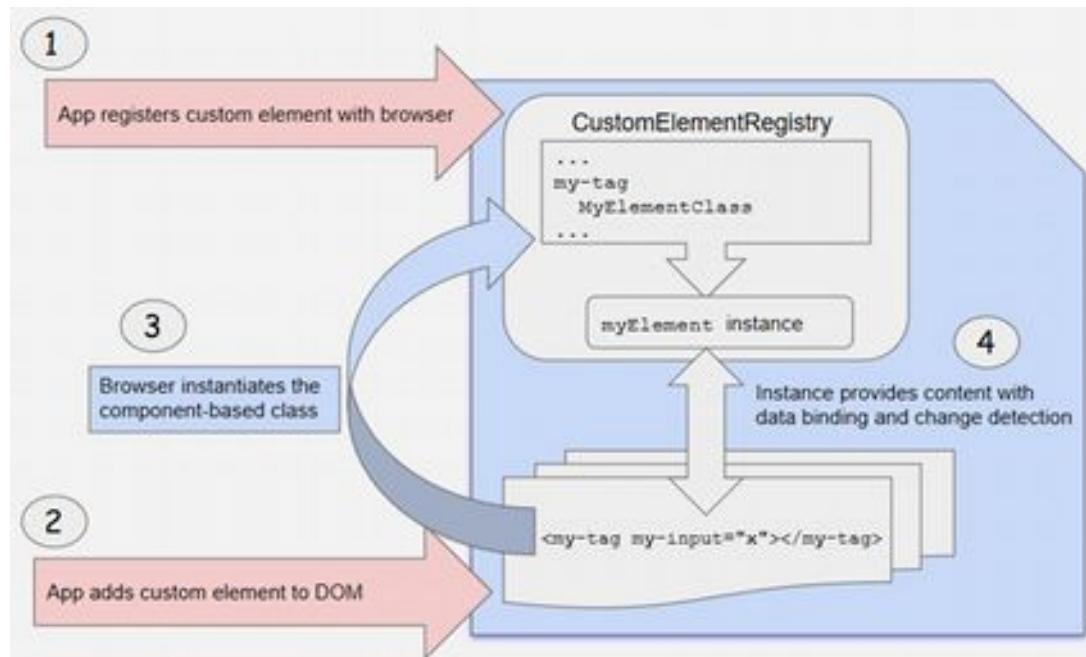
- Content-rich applications
- If you have a content-rich app, such as the Angular app that presents this documentation, custom elements let you give your content providers sophisticated Angular functionality without requiring knowledge of Angular. For example, an Angular guide like this one is added 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\(\)](#) 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:

```
<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.

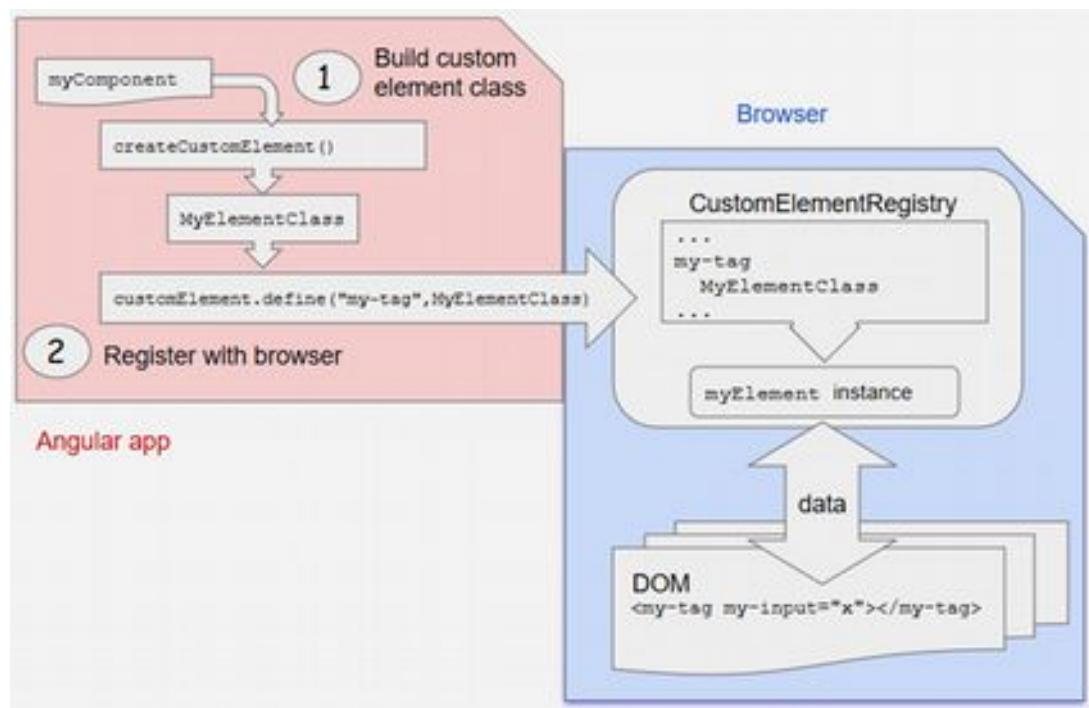


# Transforming components to custom elements

Angular provides the [createCustomElement\(\)](#) 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](#) interface, and creates a constructor class that is configured to produce a self-bootstrapping instance of your component.

Use a JavaScript function, `customElements.define()`, to register the configured constructor and its associated custom-element tag with the browser's `CustomElementRegistry`. When the browser encounters the tag for the registered element, it uses the constructor to create a custom-element instance.



## 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('myInputProp') inputProp`, the corresponding custom element defines an attribute `my-input-prop`.

- Component outputs are dispatched as HTML [Custom Events](#), with the name of the custom event matching the output name. For example, for a component with `@Output()` `valueChanged = new 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('myClick') clicks = new EventEmitter<string>()`; results in dispatch events with the name "myClick".

For more information, see Web Component documentation for [Creating custom events](#).

## Browser support for custom elements

The recently-developed [custom elements](#) Web Platform feature is currently supported natively in a number of browsers. Support is pending or planned in other browsers.

### Browser Custom Element Support

Chrome Supported natively.

Opera Supported natively.

Safari Supported natively.

Firefox Set the `dom.webcomponents.enabled` and `dom.webcomponents.customelements.enabled` preferences to true. Planned to be enabled by default in version 63.

Edge Working on an implementation.

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 `tsconfig.json`. 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](#) to automatically set up your project with the correct polyfill: `ng add @angular/elements --name=*``your_project_name`\*

- For more information about polyfills, see [polyfill documentation](#).
- For more information about Angular browser support, see [Browser Support](#).

## Example: A Popup Service

Previously, when you wanted to add a component to an app at runtime, you had to define a dynamic component. The app module would have to list your dynamic component under `entryComponents`, so that the app wouldn't expect it to be present at startup, 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](#).

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 app.)

The Popup Service example app (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 `entryComponents` list, to exclude it from compilation and avoid startup warnings or errors.
- `app.component.ts` defines the app's root component, which uses the `PopupService` to add the pop-up to the DOM at run time. When the app 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`

```
1. import { Component, EventEmitter, Input, Output } from '@angular/core';
2. import { animate, state, style, transition, trigger } from '@angular/animations';
3.
4. @Component({
5.   selector: 'my-popup',
6.   template: `
7.     <span>Popup: {{message}}</span>
8.     <button (click)="closed.next()">&#x2716;</button>
9.   `,
10.  host: {
11.    '[@state]': 'state',
12.  },
13.  animations: [
```

```

14. trigger('state', [
15.   state('opened', style({transform: 'translateY(0%)'})),
16.   state('void, closed', style({transform: 'translateY(100%)', opacity: 0})),
17.   transition('* => *', animate('100ms ease-in')),
18. ])
19. ],
20. styles: [
21.   :host {
22.     position: absolute;
23.     bottom: 0;
24.     left: 0;
25.     right: 0;
26.     background: #009cff;
27.     height: 48px;
28.     padding: 16px;
29.     display: flex;
30.     justify-content: space-between;
31.     align-items: center;
32.     border-top: 1px solid black;
33.     font-size: 24px;
34.   }
35.
36.   button {
37.     border-radius: 50%;
38.   }
39. `]
40.})
41.export class PopupComponent {
42. private state: 'opened' | 'closed' = 'closed';
43.
44. @Input()
45. set message(message: string) {
46.   this._message = message;
47.   this.state = 'opened';
48. }
49. get message(): string { return this._message; }
50. _message: string;
51.
52. @Output()
53. closed = new EventEmitter();
54.}

```

You can download the full code for the example [here](#).

## 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` (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:

```
@Component(...)  
class MyDialog {  
  @Input() content: string;  
}
```

The most straight forward 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` and `WithProperties` types (both exported from `@angular/elements`):

```
const aDialog = document.createElement('my-  
dialog') as NgElement & 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.):

```
declare global {  
  interface HTMLElementTagNameMap {  
    'my-dialog': NgElement & WithProperties<{content: string}>;  
  }  
}
```

```
'my-other-element': NgElement & WithProperties<{foo: 'bar'}>;  
...  
}  
}
```

Now, TypeScript can infer the correct type the same way it does for built-in elements:

```
document.createElement('div')           //--> HTMLDivElement (built-in element)  
document.querySelector('foo')          //--> Element      (unknown element)  
document.createElement('my-dialog')    //--> NgElement & WithProperties<{content:  
string}> (custom element)  
document.querySelector('my-other-element') //--> NgElement & WithProperties<{foo: 'bar'}>  
   (custom element)
```

Component templates are not always fixed. An application may need to load new components at runtime.

This cookbook shows you how to use [ComponentFactoryResolver](#) to add components dynamically.

See the [live example](#) / [download example](#) 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
```

```
import { Directive, ViewContainerRef } from '@angular/core';
```

```
@Directive({  
  selector: '[ad-host]',
```

```
})
export class AdDirective {
  constructor(public viewContainerRef: ViewContainerRef) { }
}
```

AdDirective injects [ViewContainerRef](#) to gain access to the view container of the element that will host the dynamically added component.

In the [@Directive](#) decorator, notice the selector name, ad-host; 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](#) decorator's [template](#) property as a template string.

The <ng-template> element is where you apply the directive you just made. To apply the AdDirective, recall the selector from ad.directive.ts, ad-host. Apply that to <ng-template> without the square brackets. Now Angular knows where to dynamically load components.

src/app/ad-banner.component.ts (template)

```
template:
<div class="ad-banner">
  <h3>Advertisements</h3>
  <ng-template ad-host></ng-template>
</div>
```

The <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)

```

export class AdBannerComponent implements OnInit, OnDestroy {
  @Input() ads: AdItem[];
  currentAdIndex = -1;
  @ViewChild(AdDirective) adHost: AdDirective;
  interval: any;

  constructor(private componentFactoryResolver: ComponentFactoryResolver) { }

  ngOnInit() {
    this.loadComponent();
    this.getAds();
  }

  ngOnDestroy() {
    clearInterval(this.interval);
  }

  loadComponent() {
    this.currentAdIndex = (this.currentAdIndex + 1) % this.ads.length;
    let adItem = this.ads[this.currentAdIndex];

    let componentFactory
    = this.componentFactoryResolver.resolveComponentFactory(adItem.component);

    let viewContainerRef = this.adHost.viewContainerRef;
    viewContainerRef.clear();

    let componentRef = viewContainerRef.createComponent(componentFactory);
    (<AdComponent>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](#) to resolve a [ComponentFactory](#) for each specific component. The [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](#) 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](#).

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.

### Selector references

Generally, the Angular compiler generates a [ComponentFactory](#) for any component referenced in a template. However, there are no selector references in the templates for dynamically loaded components since they load at runtime.

To ensure that the compiler still generates a factory, add dynamically loaded components to the [NgModule](#)'s entryComponents array:

src/app/app.module.ts (entry components)

```
entryComponents: [ HeroJobAdComponent, HeroProfileComponent ],
```

## 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

1. import { [Component](#), [Input](#) } from '@angular/core';
- 2.
3. import { AdComponent } from './ad.component';
- 4.
5. [@Component\({](#)

```
6. template: ``
7.   <div class="job-ad">
8.     <h4>{{data.headline}}</h4>
9.
10.    {{data.body}}
11.  </div>
12. `
13.`)
14.export class HeroJobAdComponent implements AdComponent {
15. @Input() data: any;
16.
17.}
```

## Final ad banner

The final ad banner looks like this:

An Attribute directive changes the appearance or behavior of a DOM element.

Try the [Attribute Directive example](#) / [download example](#).

## Directives overview

There are three kinds of directives in Angular:

1. Components—directives with a template.
2. Structural directives—change the DOM layout by adding and removing DOM elements.
3. Attribute directives—change the appearance or behavior of an element, component, or another directive.

Components are the most common of the three directives. You saw a component for the first time in the [Getting Started](#).

Structural Directives change the structure of the view. Two examples are [NgFor](#) and [NgIf](#). Learn about them in the [Structural Directives](#) guide.

Attribute directives are used as attributes of elements. The built-in [NgStyle](#) directive in the [Template Syntax](#) guide, for example, can change several element styles at the same time.

## Build a simple attribute directive

An attribute directive minimally requires building a controller class annotated with [@Directive](#), which specifies the selector that identifies the attribute. The controller class implements the desired directive behavior.

This page demonstrates building a simple appHighlight attribute directive to set an element's background color when the user hovers over that element. You can apply it like this:

```
src/app/app.component.html (applied)
```

```
<p appHighlight>Highlight me!</p>
```

## Write the directive code

Create the directive class file in a terminal window with this CLI command.

```
ng generate directive highlight
```

The CLI creates src/app/highlight.directive.ts, a corresponding test file (.../spec.ts, and declares the directive class in the root AppModule.

Directives must be declared in [Angular Modules](#) in the same manner as components.

The generated src/app/highlight.directive.ts is as follows:

```
src/app/highlight.directive.ts
```

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

```
@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  constructor() { }
}
```

The imported [Directive](#) symbol provides the Angular the [@Directive](#) decorator.

The [@Directive](#) decorator's lone configuration property specifies the directive's [CSS attribute selector](#), [appHighlight].

It's the brackets ([] that make it an attribute selector. Angular locates each element in the template that has an attribute named appHighlight and applies the logic of this directive to that element.

The attribute selector pattern explains the name of this kind of directive.

## Why not "highlight"?

Though highlight would be a more concise selector than appHighlight and it would work, the best practice is to prefix selector names to ensure they don't conflict with standard HTML attributes. This also reduces the risk of colliding with third-party directive names. The CLI added the app prefix for you.

Make sure you do not prefix the highlight directive name with ng because that prefix is reserved for Angular and using it could cause bugs that are difficult to diagnose.

After the `@Directive` metadata comes the directive's controller class, called `HighlightDirective`, which contains the (currently empty) logic for the directive. Exporting `HighlightDirective` makes the directive accessible.

Now edit the generated `src/app/highlight.directive.ts` to look as follows:

```
src/app/highlight.directive.ts
```

```
import { Directive, ElementRef } from '@angular/core';
```

```
@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  constructor(el: ElementRef) {
    el.nativeElement.style.backgroundColor = 'yellow';
  }
}
```

The import statement specifies an additional `ElementRef` symbol from the Angular core library:

You use the `ElementRef` in the directive's constructor to `inject` a reference to the host DOM element, the element to which you applied `appHighlight`.

`ElementRef` grants direct access to the host DOM element through its `nativeElement` property.

This first implementation sets the background color of the host element to yellow.

## Apply the attribute directive

To use the new `HighlightDirective`, add a paragraph (`<p>`) element to the template of the root `AppComponent` and apply the directive as an attribute.

```
src/app/app.component.html
```

```
<p appHighlight>Highlight me!</p>
```

Now run the application to see the `HighlightDirective` in action.

```
ng serve
```

To summarize, Angular found the `appHighlight` attribute on the host `<p>` element. It created an instance of the `HighlightDirective` class and injected a reference to the `<p>` element into the directive's constructor which sets the `<p>` element's background style to yellow.

## Respond to user-initiated events

Currently, appHighlight simply sets an element color. The directive could be more dynamic. It could detect when the user mouses into or out of the element and respond by setting or clearing the highlight color.

Begin by adding [HostListener](#) to the list of imported symbols.

```
src/app/highlight.directive.ts (imports)
```

```
import { Directive, ElementRef, HostListener } from '@angular/core';
```

Then add two eventhandlers that respond when the mouse enters or leaves, each adorned by the [HostListener](#) decorator.

```
src/app/highlight.directive.ts (mouse-methods)
```

```
@HostListener('mouseenter') onMouseEnter() {
  this.highlight('yellow');
}
```

```
@HostListener('mouseleave') onMouseLeave() {
  this.highlight(null);
}
```

```
private highlight(color: string) {
  this.el.nativeElement.style.backgroundColor = color;
}
```

The [@HostListener](#) decorator lets you subscribe to events of the DOM element that hosts an attribute directive, the `<p>` in this case.

Of course you could reach into the DOM with standard JavaScript and attach event listeners manually. There are at least three problems with that approach:

1. You have to write the listeners correctly.
2. The code must detach the listener when the directive is destroyed to avoid memory leaks.
3. Talking to DOM API directly isn't a best practice.

The handlers delegate to a helper method that sets the color on the host DOM element, `el`.

The helper method, `highlight`, was extracted from the constructor. The revised constructor simply declares the injected `el`: [ElementRef](#).

```
src/app/highlight.directive.ts (constructor)
```

```
constructor(private el: ElementRef) { }
```

Here's the updated directive in full:

src/app/highlight.directive.ts

```
1. import { Directive, ElementRef, HostListener } from '@angular/core';
2.
3. @Directive({
4.   selector: '[appHighlight]'
5. })
6. export class HighlightDirective {
7.   constructor(private el: ElementRef) { }
8.
9.   @HostListener('mouseenter') onMouseEnter() {
10.   this.highlight('yellow');
11. }
12.
13. @HostListener('mouseleave') onMouseLeave() {
14.   this.highlight(null);
15. }
16.
17. private highlight(color: string) {
18.   this.el.nativeElement.style.backgroundColor = color;
19. }
20.}
```

Run the app and confirm that the background color appears when the mouse hovers over the p and disappears as it moves out.

## Highlight me!

### Pass values into the directive with an @Input data binding

Currently the highlight color is hard-coded within the directive. That's inflexible. In this section, you give the developer the power to set the highlight color while applying the directive.

Begin by adding `Input` to the list of symbols imported from `@angular/core`.

src/app/highlight.directive.ts (imports)

```
content_copyimport { Directive, ElementRef, HostListener, Input } from '@angular/core';
```

Add a `highlightColor` property to the directive class like this:

src/app/highlight.directive.ts (highlightColor)

```
code">@Input() highlightColor: string;
```

## Binding to an `@Input` property

Notice the `@Input` decorator. It adds metadata to the class that makes the directive's `highlightColor` property available for binding.

It's called an input property because data flows from the binding expression into the directive. Without that input metadata, Angular rejects the binding; see [below](#) for more about that.

Try it by adding the following directive binding variations to the `AppComponent` template:

`src/app/app.component.html` (excerpt)

```
<p appHighlight highlightColor="yellow">Highlighted in yellow</p>
<p appHighlight [highlightColor]=""orange"">Highlighted in orange</p>
```

Add a color property to the `AppComponent`.

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

```
export class AppComponent {
  color = 'yellow';
}
```

Let it control the highlight color with a property binding.

`src/app/app.component.html` (excerpt)

```
<p appHighlight [highlightColor]="color">Highlighted with parent component's color</p>
```

That's good, but it would be nice to simultaneously apply the directive and set the color in the same attribute like this.

`src/app/app.component.html` (color)

```
<p [appHighlight]="color">Highlight me!</p>
```

The `[appHighlight]` attribute binding both applies the highlighting directive to the `<p>` element and sets the directive's `highlightColor` property with a property binding. You're re-using the directive's attribute selector (`[appHighlight]`) to do both jobs. That's a crisp, compact syntax.

You'll have to rename the directive's `highlightColor` property to `appHighlight` because that's now the color property binding name.

`src/app/highlight.directive.ts` (renamed to match directive selector)

```
@Input() appHighlight: string;
```

This is disagreeable. The word, `appHighlight`, is a terrible property name and it doesn't convey the property's intent.

## Bind to an `@Input` alias

Fortunately you can name the directive property whatever you want and alias it for binding purposes.

Restore the original property name and specify the selector as the alias in the argument to `@Input`.

src/app/highlight.directive.ts (color property with alias)

```
@Input('appHighlight') highlightColor: string;
```

Inside the directive the property is known as `highlightColor`. Outside the directive, where you bind to it, it's known as `appHighlight`.

You get the best of both worlds: the property name you want and the binding syntax you want:

src/app/app.component.html (color)

```
<p [appHighlight]="color">Highlight me!</p>
```

Now that you're binding via the alias to the `highlightColor`, modify the `onMouseEnter()` method to use that property. If someone neglects to bind to `appHighlightColor`, highlight the host element in red:

src/app/highlight.directive.ts (mouse enter)

```
@HostListener('mouseenter') onMouseEnter() {
  this.highlight(this.highlightColor || 'red');
}
```

Here's the latest version of the directive class.

src/app/highlight.directive.ts (excerpt)

```
import { Directive, ElementRef, HostListener, Input } from '@angular/core';
```

```
@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
```

```
  constructor(private el: ElementRef) { }
```

```
  @Input('appHighlight') highlightColor: string;
```

```
  @HostListener('mouseenter') onMouseEnter() {
    this.highlight(this.highlightColor || 'red');
  }
```

```
  @HostListener('mouseleave') onMouseLeave() {
    this.highlight(null);
  }
```

```
private highlight(color: string) {  
  this.el.nativeElement.style.backgroundColor = color;  
}  
}
```

## Write a harness to try it

It may be difficult to imagine how this directive actually works. In this section, you'll turn AppComponent into a harness that lets you pick the highlight color with a radio button and bind your color choice to the directive.

Update app.component.html as follows:

src/app/app.component.html (v2)

```
<h1>My First Attribute Directive</h1>
```

```
<h4>Pick a highlight color</h4>
```

```
<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>
```

Revise the AppComponent.color so that it has no initial value.

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

```
export class AppComponent {  
  color: string;  
}
```

Here are the harness and directive in action.

# My First Attribute Directive

## Pick a highlight color

Green  Yellow  Cyan

Highlight me!

## Bind to a second property

This highlight directive has a single customizable property. In a real app, it may need more.

At the moment, the default color—the color that prevails until the user picks a highlight color—is hard-coded as "red". Let the template developer set the default color.

Add a second input property to `HighlightDirective` called `defaultColor`:

```
src/app/highlight.directive.ts (defaultColor)
```

```
@Input() defaultColor: string;
```

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)
```

```
@HostListener('mouseenter') onMouseEnter() {
  this.highlight(this.highlightColor || this.defaultColor || 'red');
}
```

How do you bind to a second property when you're already binding to the `appHighlight` attribute name?

As with components, you can add as many directive property bindings as you need by stringing them along in the template. The developer should be able to write the following template HTML to both bind to the `AppComponent.color` and fall back to "violet" as the default color.

```
src/app/app.component.html (defaultColor)
```

```
<p [appHighlight]="color" defaultColor="violet">
  Highlight me too!
</p>
```

Angular knows that the `defaultColor` binding belongs to the `HighlightDirective` because you made it public with the `@Input` decorator.

Here's how the harness should work when you're done coding.

# My First Attribute Directive

Pick a highlight color

Green  Yellow  Cyan

Highlight me! **no default-color binding**

Highlight me too! **with 'violet' default-color binding**

## Summary

This page covered how to:

- [Build an attribute directive](#) that modifies the behavior of an element.
- [Apply the directive](#) to an element in a template.
- [Respond to events](#) that change the directive's behavior.
- [Bind values to the directive](#).

The final source code follows:

app/app.component.ts

app/app.component.html

app/highlight.directive.ts

app/app.module.ts

main.ts

index.html

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

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  color: string;
}
```

You can also experience and download the [Attribute Directive example](#) / [download example](#).

## Appendix: Why add `@Input`?

In this demo, the `highlightColor` property is an input property of the `HighlightDirective`. You've seen it applied without an alias:

```
src/app/highlight.directive.ts (color)
```

```
@Input\(\) highlightColor: string;
```

You've seen it with an alias:

```
src/app/highlight.directive.ts (color)
```

```
@Input\('appHighlight'\) highlightColor: string;
```

Either way, the `@Input` decorator tells Angular that this property is public and available for binding by a parent component. Without `@Input`, Angular refuses to bind to the property.

You've bound template HTML to component properties before and never used `@Input`. What's different?

The difference is a matter of trust. Angular treats a component's template as belonging to the component. The component and its template trust each other implicitly. Therefore, the component's own template may bind to any property of that component, with or without the `@Input` decorator.

But a component or directive shouldn't blindly trust other components and directives. The properties of a component or directive are hidden from binding by default. They are private from an Angular binding perspective. When adorned with the `@Input` decorator, the property becomes public from an Angular binding perspective. Only then can it be bound by some other component or directive.

You can tell if `@Input` is needed by the position of the property name in a binding.

- When it appears in the template expression to the right of the equals (=), it belongs to the template's component and does not require the `@Input` decorator.
- When it appears in square brackets ([ ]) to the left of the equals (=), the property belongs to some other component or directive; that property must be adorned with the `@Input` decorator.

Now apply that reasoning to the following example:

```
src/app/app.component.html (color)
```

```
<p [appHighlight]="color">Highlight me!</p>
```

- The `color` property in the expression on the right belongs to the template's component. The template and its component trust each other. The `color` property doesn't require the `@Input` decorator.

- The appHighlight property on the left refers to an aliased property of the HighlightDirective, not a property of the template's component. There are trust issues. Therefore, the directive property must carry the `@Input` decorator.

This guide looks at how Angular manipulates the DOM with structural directives and how you can write your own structural directives to do the same thing.

Try the [live example](#) / [download example](#).

## What are structural directives?

Structural directives are responsible for HTML layout. They shape or reshape the DOM's structure, typically by adding, removing, or manipulating elements.

As with other directives, you apply a structural directive to a host element. The directive then does whatever it's supposed to do with that host element and its descendants.

Structural directives are easy to recognize. An asterisk (\*) precedes the directive attribute name as in this example.

src/app/app.component.html (ngif)

```
<div *ngIf="hero" class="name">{{hero.name}}</div>
```

No brackets. No parentheses. Just `*ngIf` set to a string.

You'll learn in this guide that the [asterisk \(\\*\) is a convenience notation](#) and the string is a [microsyntax](#) rather than the usual[template expression](#). Angular desugars this notation into a marked-up `<ng-template>` that surrounds the host element and its descendants. Each structural directive does something different with that template.

Three of the common, built-in structural directives—[NgIf](#), [NgFor](#), and [NgSwitch...](#)—are described in the [Template Syntax](#) guide and seen in samples throughout the Angular documentation. Here's an example of them in a template:

src/app/app.component.html (built-in)

```
<div *ngIf="hero" class="name">{{hero.name}}</div>
```

```
<ul>
```

```
  <li *ngFor="let hero of heroes">{{hero.name}}</li>
```

```
</ul>
```

```
<div [ngSwitch]="hero?.emotion">
```

```
  <app-happy-hero *ngSwitchCase="happy" [hero]="hero"></app-happy-hero>
```

```
  <app-sad-hero *ngSwitchCase="sad" [hero]="hero"></app-sad-hero>
```

```
  <app-confused-hero *ngSwitchCase="app-confused" [hero]="hero"></app-confused-hero>
```

```
  <app-unknown-hero *ngSwitchDefault [hero]="hero"></app-unknown-hero>
```

```
</div>
```

This guide won't repeat how to use them. But it does explain how they work and how to [write your own](#) structural directive.

## DIRECTIVE SPELLING

Throughout this guide, you'll see a directive spelled in both UpperCamelCase and lowerCamelCase. Already you've seen [NgIf](#) and [ngIf](#). There's a reason. [NgIf](#) refers to the directive class; [ngIf](#) refers to the directive's attribute name.

A directive class is spelled in UpperCamelCase ([NgIf](#)). A directive's attribute name is spelled in lowerCamelCase ([ngIf](#)). The guide refers to the directive class when talking about its properties and what the directive does. The guide refers to the attribute name when describing how you apply the directive to an element in the HTML template.

There are two other kinds of Angular directives, described extensively elsewhere: (1) components and (2) attribute directives.

A component manages a region of HTML in the manner of a native HTML element. Technically it's a directive with a template.

An [attribute directive](#) changes the appearance or behavior of an element, component, or another directive. For example, the built-in [NgStyle](#) directive changes several element styles at the same time.

You can apply many attribute directives to one host element. You can [only apply one](#) structural directive to a host element.

## NgIf case study

[NgIf](#) is the simplest structural directive and the easiest to understand. It takes a boolean expression and makes an entire chunk of the DOM appear or disappear.

src/app/app.component.html (ngif-true)

```
<p *ngIf="true">  
  Expression is true and ngIf is true.  
  This paragraph is in the DOM.  
</p>  
<p *ngIf="false">  
  Expression is false and ngIf is false.  
  This paragraph is not in the DOM.  
</p>
```

The [ngIf](#) directive doesn't hide elements with CSS. It adds and removes them physically from the DOM. Confirm that fact using browser developer tools to inspect the DOM.

```
<p _ngcontent-c0>
  Expression is true and ngIf is true.
  This paragraph is in the DOM.
</p>
<!--bindings={
  "ng-reflect-ng-if": "false"
}-->
```

The top paragraph is in the DOM. The bottom, disused paragraph is not; in its place is a comment about "bindings" (more about that [later](#)).

When the condition is false, [NgIf](#) removes its host element from the DOM, detaches it from DOM events (the attachments that it made), detaches the component from Angular change detection, and destroys it. The component and DOM nodes can be garbage-collected and free up memory.

## Why remove rather than hide?

A directive could hide the unwanted paragraph instead by setting its display style to none.

src/app/app.component.html (display-none)

```
<p [style.display]=""block"">
  Expression sets display to "block".
  This paragraph is visible.
</p>
<p [style.display]=""none"">
  Expression sets display to "none".
  This paragraph is hidden but still in the DOM.
</p>
```

While invisible, the element remains in the DOM.

```
<p _ngcontent-fwv-0 style="display: block;">
  Expression sets display to "block" .
  This paragraph is visible.
</p>
<p _ngcontent-fwv-0 style="display: none;">
  "
  Expression sets display to "none" .
  This paragraph is hidden but still in the DOM.
  "
</p>
```

The difference between hiding and removing doesn't matter for a simple paragraph. It does matter when the host element is attached to a resource intensive component. Such a component's behavior continues even when hidden. The component stays attached to its

DOM element. It keeps listening to events. Angular keeps checking for changes that could affect data bindings. Whatever the component was doing, it keeps doing.

Although invisible, the component—and all of its descendant components—tie up resources. The performance and memory burden can be substantial, responsiveness can degrade, and the user sees nothing.

On the positive side, showing the element again is quick. The component's previous state is preserved and ready to display. The component doesn't re-initialize—an operation that could be expensive. So hiding and showing is sometimes the right thing to do.

But in the absence of a compelling reason to keep them around, your preference should be to remove DOM elements that the user can't see and recover the unused resources with a structural directive like [NgIf](#).

These same considerations apply to every structural directive, whether built-in or custom. Before applying a structural directive, you might want to pause for a moment to consider the consequences of adding and removing elements and of creating and destroying components.

## The asterisk (\*) prefix

Surely you noticed the asterisk (\*) prefix to the directive name and wondered why it is necessary and what it does.

Here is `*ngIf` displaying the hero's name if hero exists.

src/app/app.component.html (asterisk)

```
<div *ngIf="hero" class="name">{{hero.name}}</div>
```

The asterisk is "syntactic sugar" for something a bit more complicated. Internally, Angular translates the `*ngIf` attribute into a `<ng-template>` element, wrapped around the host element, like this.

src/app/app.component.html (ngif-template)

```
<ng-template [ngIf]="hero">
<div class="name">{{hero.name}}</div>
</ng-template>
```

- The `*ngIf` directive moved to the `<ng-template>` element where it became a property binding, `[ngIf]`.
- The rest of the `<div>`, including its class attribute, moved inside the `<ng-template>` element.

The first form is not actually rendered, only the finished product ends up in the DOM.

```
<!--bindings={  
  "ng-reflect-ng-if": "[object Object]"  
}-->  
<div _ngcontent-c0>Mr. Nice</div>
```

Angular consumed the `<ng-template>` content during its actual rendering and replaced the `<ng-template>` with a diagnostic comment.

The [NgFor](#) and [NgSwitch...](#) directives follow the same pattern.

## Inside \*ngFor

Angular transforms the `*ngFor` in similar fashion from asterisk (\*) syntax to `<ng-template>` element.

Here's a full-featured application of NgFor, written both ways:

src/app/app.component.html (inside-ngfor)

```
<div *ngFor="let hero of heroes; let i=index; let odd=odd;  
trackBy: trackById" [class.odd]="odd">  
  {{i}} {{hero.name}}  
</div>  
  
<ng-template ngFor let-hero [ngForOf]="heroes" let-i="index" let-odd="odd"  
[ngForTrackBy]="trackById">  
  <div [class.odd]="odd">{{i}} {{hero.name}}</div>  
</ng-template>
```

This is manifestly more complicated than [ngIf](#) and rightly so. The NgFor directive has more features, both required and optional, than the [NgIf](#) shown in this guide. At minimum NgFor needs a looping variable (let hero) and a list (heroes).

You enable these features in the string assigned to [ngFor](#), which you write in Angular's [microsyntax](#).

Everything outside the [ngFor](#) string stays with the host element (the `<div>`) as it moves inside the `<ng-template>`. In this example, the `[ngClass]="odd"` stays on the `<div>`.

## Microsyntax

The Angular microsyntax lets you configure a directive in a compact, friendly string. The microsyntax parser translates that string into attributes on the `<ng-template>`:

- The `let` keyword declares a [template input variable](#) that you 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 microsyntax parser takes `of` and `trackBy`, title-cases them (`of` -> `Of`, `trackBy` -> `TrackBy`), and prefixes them with the directive's attribute name (`ngFor`), yielding the

names [ngForOf](#) and [ngForTrackBy](#). Those are the names of two NgFor input properties. That's how the directive learns that the list is heroes and the track-by function is [trackById](#).

- As the NgFor directive loops through the list, it sets and resets properties of its own context object. These properties include index and [odd](#) and a special property named \$implicit.
- The let-i and let-odd variables were defined as let i=index and let [odd=odd](#). Angular sets them to the current value of the context's index and [odd](#) properties.
- The context property for let-hero wasn't specified. Its intended source is 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.
- The [API guide](#) describes additional NgFor directive properties and context properties.
- NgFor is implemented by the [NgForOf](#) directive. Read more about additional [NgForOf](#) directive properties and context properties [NgForOf API reference](#).

These microsyntax mechanisms are available to you when you write your own structural directives. Studying the [source code for NgIf](#) and [NgForOf](#) is a great way to learn more.

## Template input variable

A template input variable is a variable whose value you can reference within a single instance of the template. There are several such variables in this example: hero, i, and [odd](#). All are preceded by the keyword let.

A template input variable is not the same as a [template reference variable](#), neither semantically nor syntactically.

You declare a template input variable using the let keyword (let hero). 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.

You declare a template reference variable by prefixing the variable name with # (#var). A reference variable refers to its attached element, component or directive. It can be accessed anywhere in the entire template.

Template input and reference variable names have their own namespaces. The hero in let hero is never the same variable as the hero declared as #hero.

## One structural directive per host element

Someday you'll want to repeat a block of HTML but only when a particular condition is true. You'll try to put both an \*[ngFor](#) and an \*[ngIf](#) on the same host element. Angular won't let you. You may apply only one structural directive to an element.

The reason is simplicity. Structural directives can do complex things with the host element and its descendants. When two directives lay claim to the same host element, which one takes precedence? Which should go first, the [NgIf](#) or the NgFor? Can the [NgIf](#) cancel the

effect of the NgFor? If so (and it seems like it should be so), how should Angular generalize the ability to cancel for other structural directives?

There are no easy answers to these questions. Prohibiting multiple structural directives makes them moot. There's an easy solution for this use case: put the `*ngIf` on a container element that wraps the `*ngFor` element. One or both elements can be an `ng-container` so you don't have to introduce extra levels of HTML.

## Inside NgSwitch directives

The Angular NgSwitch is actually a set of cooperating directives: `NgSwitch`, `NgSwitchCase`, and `NgSwitchDefault`.

Here's an example.

src/app/app.component.html (ngswitch)

```
<div [ngSwitch]="hero?.emotion">
  <app-happy-hero *ngSwitchCase="happy" [hero]="hero"></app-happy-hero>
  <app-sad-hero *ngSwitchCase="sad" [hero]="hero"></app-sad-hero>
  <app-confused-hero *ngSwitchCase="app-confused" [hero]="hero"></app-confused-hero>
  <app-unknown-hero *ngSwitchDefault [hero]="hero"></app-unknown-hero>
</div>
```

The switch value assigned to `NgSwitch` (`hero.emotion`) determines which (if any) of the switch cases are displayed.

`NgSwitch` itself is not a structural directive. It's an attribute directive that controls the behavior of the other two switch directives. That's why you write `[ngSwitch]`, never `*ngSwitch`.

`NgSwitchCase` and `NgSwitchDefault` are structural directives. You attach them to elements using the asterisk (\*) prefix notation. An `NgSwitchCase` displays its host element when its value matches the switch value. The `NgSwitchDefault` displays its host element when no sibling `NgSwitchCase` matches the switch value.

The element to which you apply a directive is its host element. The `<happy-hero>` is the host element for the happy `*ngSwitchCase`. The `<unknown-hero>` is the host element for the `*ngSwitchDefault`.

As with other structural directives, the `NgSwitchCase` and `NgSwitchDefault` can be desugared into the `<ng-template>` element form.

src/app/app.component.html (ngswitch-template)

```
<div [ngSwitch]="hero?.emotion">
  <ng-template [ngSwitchCase]="happy">
    <app-happy-hero [hero]="hero"></app-happy-hero>
  </ng-template>
  <ng-template [ngSwitchCase]="sad">
```

```
<app-sad-hero [hero]="hero"></app-sad-hero>
</ng-template>
<ng-template [ngSwitchCase]="'confused'">
  <app-confused-hero [hero]="hero"></app-confused-hero>
</ng-template >
<ng-template ngSwitchDefault>
  <app-unknown-hero [hero]="hero"></app-unknown-hero>
</ng-template>
</div>
```

## Prefer the asterisk (\*) syntax.

The asterisk (\*) syntax is more clear than the desugared form. Use [`<ng-container>`](#) when there's no single element to host the directive.

While there's rarely a good reason to apply a structural directive in template attribute or element form, it's still important to know that Angular creates a `<ng-template>` and to understand how it works. You'll refer to the `<ng-template>` when you [write your own structural directive](#).

## The `<ng-template>`

The `<ng-template>` is an Angular element for rendering HTML. It is never displayed directly. In fact, before rendering the view, Angular replaces the `<ng-template>` and its contents with a comment.

If there is no structural directive and you merely wrap some elements in a `<ng-template>`, those elements disappear. That's the fate of the middle "Hip!" in the phrase "Hip! Hip! Hooray!".

src/app/app.component.html (template-tag)

```
<p>Hip!</p>
<ng-template>
  <p>Hip!</p>
</ng-template>
<p>Hooray!</p>
```

Angular erases the middle "Hip!", leaving the cheer a bit less enthusiastic.

```
<p _ngcontent-c0>Hip!</p>
<!---->
<p _ngcontent-c0>Hooray!</p>
```

Hip!  
Hooray!

A structural directive puts a `<ng-template>` to work as you'll see when you [write your own structural directive](#).

## Group sibling elements with `<ng-container>`

There's often a root element that can and should host the structural directive. The list element (`<li>`) is a typical host element of an `NgFor` repeater.

src/app/app.component.html (ngfor-li)

```
<li *ngFor="let hero of heroes">{{hero.name}}</li>
```

When there isn't a host element, you can usually wrap the content in a native HTML container element, such as a `<div>`, and attach the directive to that wrapper.

src/app/app.component.html (ngif)

```
<div *ngIf="hero" class="name">{{hero.name}}</div>
```

Introducing another container element—typically a `<span>` or `<div>`—to group the elements under a single root is usually harmless. Usually ... but not always.

The grouping element may break the template appearance because CSS styles neither expect nor accommodate the new layout. For example, suppose you have the following paragraph layout.

src/app/app.component.html (ngif-span)

```
<p>
  I turned the corner
  <span *ngIf="hero">
    and saw {{hero.name}}. I waved
  </span>
  and continued on my way.
</p>
```

You also have a CSS style rule that happens to apply to a `<span>` within a `<p>`aragraph.

src/app/app.component.css (p-span)

```
p span { color: red; font-size: 70%; }
```

The constructed paragraph renders strangely.

I turned the corner and saw Mr. Nice. I waved and continued on my way.

The `p span` style, intended for use elsewhere, was inadvertently applied here.

Another problem: some HTML elements require all immediate children to be of a specific type. For example, the `<select>`element requires `<option>` children. You can't wrap the options in a conditional `<div>` or a `<span>`.

When you try this,

```
src/app/app.component.html (select-span)
```

```
<div>
  Pick your favorite hero
  (<label><input type="checkbox" checked (change)="showSad = !showSad">show
  sad</label>)
</div>
<select [(ngModel)]="hero">
  <span *ngFor="let h of heroes">
    <span *ngIf="showSad || h.emotion !== 'sad'">
      <option [ngValue]="h">{{h.name}} ({{h.emotion}})</option>
    </span>
  </span>
</select>
```

the drop down is empty.

Pick your favorite hero, who is  not sad  
▼

The browser won't display an `<option>` within a `<span>`.

### <ng-container> to the rescue

The Angular `<ng-container>` is a grouping element that doesn't interfere with styles or layout because Angular doesn't put it in the DOM.

Here's the conditional paragraph again, this time using `<ng-container>`.

```
src/app/app.component.html (ngif-ngcontainer)
```

```
<p>
  I turned the corner
  <ng-container *ngIf="hero">
    and saw {{hero.name}}. I waved
  </ng-container>
  and continued on my way.
</p>
```

It renders properly.

I turned the corner and saw Mr. Nice. I waved and continued on my way.

Now conditionally exclude a select `<option>` with `<ng-container>`.

```
src/app/app.component.html (select-ngcontainer)
```

```
<div>
  Pick your favorite hero
  (<label><input type="checkbox" checked (change)="showSad = !showSad">show
  sad</label>)
```

```

</div>
<select [(ngModel)]="hero">
  <ng-container *ngFor="let h of heroes">
    <ng-container *ngIf="showSad || h.emotion !== 'sad'">
      <option [ngValue]="h">{{h.name}} ({{h.emotion}})</option>
    </ng-container>
  </ng-container>
</select>

```

The drop down works properly.



The `<ng-container>` is a syntax element recognized by the Angular parser. It's not a directive, component, class, or interface. It's more like the curly braces in a JavaScript if-block:

```

if (someCondition) {
  statement1;
  statement2;
  statement3;
}

```

Without those braces, JavaScript would only execute the first statement when you intend to conditionally execute all of them as a single block. The `<ng-container>` satisfies a similar need in Angular templates.

## Write a structural directive

In this section, you write an `UnlessDirective` structural directive that does the opposite of `NgIf`. `NgIf` displays the template content when the condition is true. `UnlessDirective` displays the content when the condition is false.

`src/app/app.component.html (appUnless-1)`

```
<p *appUnless="condition">Show this sentence unless the condition is true.</p>
```

Creating a directive is similar to creating a component.

- Import the `Directive` decorator (instead of the `Component` decorator).
- Import the `Input`, `TemplateRef`, and `ViewContainerRef` symbols; you'll need them for any structural directive.
- Apply the decorator to the directive class.
- Set the CSS attribute selector that identifies the directive when applied to an element in a template.

Here's how you might begin:

```
src/app/unless.directive.ts (skeleton)
```

```
import { Directive, Input, TemplateRef, ViewContainerRef } from '@angular/core';
```

```
@Directive({ selector: '[appUnless]' })
export class UnlessDirective {
```

```
}
```

The directive's selector is typically the directive's attribute name in square brackets, [appUnless]. The brackets define a CSS[attribute selector](#).

The directive attribute name should be spelled in lowerCamelCase and begin with a prefix. Don't use ng. That prefix belongs to Angular. Pick something short that fits you or your company. In this example, the prefix is app.

The directive class name ends in [Directive](#) per the [style guide](#). Angular's own directives do not.

## TemplateRef and ViewContainerRef

A simple structural directive like this one creates an [embedded view](#) from the Angular-generated <ng-template> and inserts that view in a [view container](#) adjacent to the directive's original <p> host element.

You'll acquire the <ng-template> contents with a [TemplateRef](#) and access the view container through a [ViewContainerRef](#).

You inject both in the directive constructor as private variables of the class.

```
src/app/unless.directive.ts (ctor)
```

```
constructor(
  private templateRef: TemplateRef<any>,
  private viewContainer: ViewContainerRef) { }
```

## The appUnless property

The directive consumer expects to bind a true/false condition to [appUnless]. That means the directive needs an appUnlessproperty, decorated with [@Input](#)

Read about [@Input](#) in the [Template Syntax](#) guide.

```
src/app/unless.directive.ts (set)
```

```
@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. Because the `appUnless` property does work, it needs a setter.

- If the condition is falsy and the view hasn't been created previously, tell the view container to create the embedded view from the template.
- If the condition is truthy and the view is currently displayed, clear the container which also destroys the view.

Nobody reads the `appUnless` property so it doesn't need a getter.

The completed directive code looks like this:

src/app/unless.directive.ts (excerpt)

```
import { Directive, Input, TemplateRef, ViewContainerRef } from '@angular/core';

/**
 * Add the template content to the DOM unless the condition is true.
 */
@Directive({ selector: '[appUnless]' })
export class UnlessDirective {
  private hasView = false;

  constructor(
    private templateRef: TemplateRef<any>,
    private viewContainer: ViewContainerRef) { }

  @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;
    }
  }
}
```

Add this directive to the `declarations` array of the AppModule.

Then create some HTML to try it.

src/app/app.component.html (appUnless)

```
<p *appUnless="condition" class="unless_a">
```

(A) This paragraph is displayed because the condition is false.

```
</p>
```

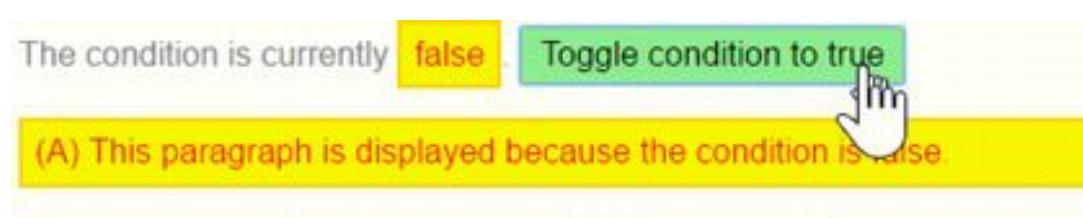
```
<p *appUnless="!condition" class="unless_b">
```

(B) Although the condition is true,

this paragraph is displayed because appUnless is set to false.

```
</p>
```

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 is removed and the bottom (B) paragraph appears.



## Summary

You can both try and download the source code for this guide in the [live example](#) / [download example](#).

Here is the source from the src/app/ folder.

app.component.ts

app.component.html

app.component.css

app.module.ts

hero.ts

hero-switch.components.ts

unless.directive.ts

1. import { [Component](#) } from '@angular/core';
- 2.
3. import { Hero, heroes } from './hero';
- 4.
5. [@Component](#)({
6. selector: 'app-root',
7. templateUrl: './app.component.html',
8. [styleUrls](#): [ './app.component.css' ]

```
9. })
10.export class AppComponent {
11. heroes = heroes;
12. hero = this.heroes[0];
13.
14. condition = false;
15. logs: string[] = [];
16. showSad = true;
17. status = 'ready';
18.
19. trackById(index: number, hero: Hero): number { return hero.id; }
20.}
```

You learned

- that structural directives manipulate HTML layout.
- to use [`<ng-container>`](#) as a grouping element when there is no suitable host element.
- that the Angular desugars [asterisk \(\\*\) syntax](#) into a `<ng-template>`.
- how that works for the [NgIf](#), NgFor and [NgSwitch](#) built-in directives.
- about the [microsyntax](#) that expands into a `<ng-template>`.
- to write a [custom structural directive](#), UnlessDirective.

Every application starts out with what seems like a simple task: get data, transform them, and show them to users. Getting data could be as simple as creating a local variable or as complex as streaming data over a WebSocket.

Once data arrives, you could push their raw `toString` values directly to the view, but that rarely makes for a good user experience. For example, in most use cases, users prefer to see a date in a simple format like April 15, 1988 rather than the raw string format Fri Apr 15 1988 00:00:00 GMT-0700 (Pacific Daylight Time).

Clearly, some values benefit from a bit of editing. You may notice that you desire many of the same transformations repeatedly, both within and across many applications. You can almost think of them as styles. In fact, you might like to apply them in your HTML templates as you do styles.

Introducing Angular pipes, a way to write display-value transformations that you can declare in your HTML.

You can run the [live example](#) / [download example](#) in Stackblitz and download the code from there.

## Using pipes

A pipe takes in data as input and transforms it to a desired output. In this page, you'll use pipes to transform a component's `birthday` property into a human-friendly date.

```
src/app/hero-birthday1.component.ts

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

@Component({
  selector: 'app-hero-birthday',
  template: `<p>The hero's birthday is {{ birthday | date }}</p>`
})
export class HeroBirthdayComponent {
  birthday = new Date(1988, 3, 15); // April 15, 1988
}
```

Focus on the component's template.

```
src/app/app.component.html
```

```
<p>The hero's birthday is {{ birthday | date }}</p>
```

Inside the interpolation expression, you flow the component's birthday value through the [pipe operator](#) ( | ) to the [Date pipe](#) function on the right. All pipes work this way.

## Built-in pipes

Angular comes with a stock of pipes such as [DatePipe](#), [UpperCasePipe](#), [LowerCasePipe](#), [CurrencyPipe](#), and [PercentPipe](#). They are all available for use in any template.

Read more about these and many other built-in pipes in the [pipes topics](#) of the [API Reference](#); filter for entries that include the word "pipe".

Angular doesn't have a [FilterPipe](#) or an [OrderByPipe](#) for reasons explained in the [Appendix](#) of this page.

## Parameterizing a pipe

A pipe can accept any number of optional parameters to fine-tune its output. To add parameters to a pipe, follow the pipe name with a colon ( : ) and then the parameter value (such as currency:'EUR'). If the pipe accepts multiple parameters, separate the values with colons (such as slice:1:5)

Modify the birthday template to give the date pipe a format parameter. After formatting the hero's April 15th birthday, it renders as 04/15/88:

```
src/app/app.component.html
```

```
<p>The hero's birthday is {{ birthday | date:"MM/dd/yy" }}</p>
```

The parameter value can be any valid template expression, (see the [Template expressions](#) section of the [Template Syntax](#) page) such as a string literal or a component

property. In other words, you can control the format through a binding the same way you control the birthday value through a binding.

Write a second component that binds the pipe's format parameter to the component's format property. Here's the template for that component:

src/app/hero-birthday2.component.ts (template)

template:

```
<p>The hero's birthday is {{ birthday | date:format }}</p>
<button (click)="toggleFormat()">Toggle Format</button>
```

You also added a button to the template and bound its click event to the component's toggleFormat() method. That method toggles the component's format property between a short form ('shortDate') and a longer form ('fullDate').

src/app/hero-birthday2.component.ts (class)

```
export class HeroBirthday2Component {
  birthday = new Date(1988, 3, 15); // April 15, 1988
  toggle = true; // start with true == shortDate

  get format() { return this.toggle ? 'shortDate' : 'fullDate'; }
  toggleFormat() { this.toggle = !this.toggle; }
}
```

As you click the button, the displayed date alternates between "04/15/1988" and "Friday, April 15, 1988".

The hero's birthday is 4/15/1988

[Toggle Format](#)

Read more about the [DatePipe](#) format options in the [Date Pipe](#) API Reference page.

## Chaining pipes

You can chain pipes together in potentially useful combinations. In the following example, to display the birthday in uppercase, the birthday is chained to the [DatePipe](#) and on to the [UpperCasePipe](#). The birthday displays as APR 15, 1988.

src/app/app.component.html

The chained hero's birthday is  
{{ birthday | date | uppercase}}

This example—which displays FRIDAY, APRIL 15, 1988—chains the same pipes as above, but passes in a parameter to date as well.

```
src/app/app.component.html
```

```
The chained hero's birthday is  
{ { birthday | date:'fullDate' | uppercase}}
```

## Custom pipes

You can write your own custom pipes. Here's a custom pipe named `ExponentialStrengthPipe` that can boost a hero's powers:

```
src/app/exponential-strength.pipe.ts
```

```
import { Pipe, 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({name: 'exponentialStrength'})  
export class ExponentialStrengthPipe implements PipeTransform {  
  transform(value: number, exponent: string): number {  
    let exp = parseFloat(exponent);  
    return Math.pow(value, isNaN(exp) ? 1 : exp);  
  }  
}
```

This pipe definition reveals the following key points:

- A pipe is a class decorated with pipe metadata.
- The pipe class implements the `PipeTransform` interface's `transform` method that accepts an input value followed by optional parameters and returns the transformed value.
- There will be one additional argument to the `transform` method for each parameter passed to the pipe. Your pipe has one such parameter: the exponent.
- To tell Angular that this is a pipe, you apply the `@Pipe` decorator, which you import from the core Angular library.
- The `@Pipe` decorator allows you to define the pipe name that you'll use within template expressions. It must be a valid JavaScript identifier. Your pipe's name is `exponentialStrength`.

## The PipeTransform interface

The transform method is essential to a pipe. The [PipeTransform](#) interface defines that method and guides both tooling and the compiler. Technically, it's optional; Angular looks for and executes the transform method regardless.

Now you need a component to demonstrate the pipe.

```
src/app/power-booster.component.ts

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

@Component({
  selector: 'app-power-booster',
  template: `
    <h2>Power Booster</h2>
    <p>Super power boost: {{2 | exponentialStrength: 10}}</p>
  `
})
export class PowerBoosterComponent { }
```

## Power Booster

Super power boost: 1024

Note the following:

- You use your custom pipe the same way you use built-in pipes.
- You must include your pipe in the [declarations](#) array of the AppModule.

### REMEMBER THE DECLARATIONS ARRAY

You must register custom pipes. If you don't, Angular reports an error. Angular CLI's generator registers the pipe automatically.

To probe the behavior in the [live example](#) / [download example](#), change the value and optional exponent in the template.

## Power Boost Calculator

It's not much fun updating the template to test the custom pipe. Upgrade the example to a "Power Boost Calculator" that combines your pipe and two-way data binding with [ngModel](#).

```
src/app/power-boost-calculator.component.ts
```

1. import { Component } from '@angular/core';
- 2.

```
3. @Component({
4.   selector: 'app-power-boost-calculator',
5.   template: `
6.     <h2>Power Boost Calculator</h2>
7.     <div>Normal power: <input [(ngModel)]="power"></div>
8.     <div>Boost factor: <input [(ngModel)]="factor"></div>
9.     <p>
10.       Super Hero Power: {{power | exponentialStrength: factor}}
11.     </p>
12.   `
13. })
14. export class PowerBoostCalculatorComponent {
15.   power = 5;
16.   factor = 1;
17. }
```

## Power Boost Calculator

Normal power: 5

Boost factor: 1

Super Hero Power: 5

## Pipes and change detection

Angular looks for changes to data-bound values through a change detection process that runs after every DOM event: every keystroke, mouse move, timer tick, and server response. This could be expensive. Angular strives to lower the cost whenever possible and appropriate.

Angular picks a simpler, faster change detection algorithm when you use a pipe.

### No pipe

In the next example, the component uses the default, aggressive change detection strategy to monitor and update its display of every hero in the heroes array. Here's the template:

src/app/flying-heroes.component.html (v1)

New hero:

```
<input type="text" #box
  (keyup.enter)="addHero(box.value); box.value=''"
  placeholder="hero name">
<button (click)="reset()">Reset</button>
<div *ngFor="let hero of heroes">
```

```
  {{hero.name}}  
</div>
```

The companion component class provides heroes, adds heroes into the array, and can reset the array.

src/app/flying-heroes.component.ts (v1)

```
export class FlyingHeroesComponent {  
  heroes: any[] = [];  
  canFly = true;  
  constructor() { this.reset(); }  
  
  addHero(name: string) {  
    name = name.trim();  
    if (!name) { return; }  
    let hero = {name, canFly: this.canFly};  
    this.heroes.push(hero);  
  }  
  
  reset() { this.heroes = HEROES.slice(); }  
}
```

You can add heroes and Angular updates the display when you do. If you click the reset button, Angular replaces heroes with a new array of the original heroes and updates the display. If you added the ability to remove or change a hero, Angular would detect those changes and update the display as well.

## FlyingHeroesPipe

Add a FlyingHeroesPipe to the `*ngFor` repeater that filters the list of heroes to just those heroes who can fly.

src/app/flying-heroes.component.html (flyers)

```
<div *ngFor="let hero of (heroes | flyingHeroes)">  
  {{hero.name}}  
</div>
```

Here's the FlyingHeroesPipe implementation, which follows the pattern for custom pipes described earlier.

src/app/flying-heroes.pipe.ts

```
import { Pipe, PipeTransform } from '@angular/core';  
  
import { Flyer } from './heroes';
```

```
@Pipe({ name: 'flyingHeroes' })
export class FlyingHeroesPipe implements PipeTransform {
  transform(allHeroes: Flyer[]) {
    return allHeroes.filter(hero => hero.canFly);
  }
}
```

Notice the odd behavior in the [live example](#) / [download example](#): when you add flying heroes, none of them are displayed under "Heroes who fly."

Although you're not getting the behavior you want, Angular isn't broken. It's just using a different change-detection algorithm that ignores changes to the list or any of its items.

Notice how a hero is added:

```
src/app/flying-heroes.component.ts
this.heroes.push(hero);
```

You add the hero into the heroes array. The reference to the array hasn't changed. It's the same array. That's all Angular cares about. From its perspective, same array, no change, no display update.

To fix that, create an array with the new hero appended and assign that to heroes. This time Angular detects that the array reference has changed. It executes the pipe and updates the display with the new array, which includes the new flying hero.

If you mutate the array, no pipe is invoked and the display isn't updated; if you replace the array, the pipe executes and the display is updated. The Flying Heroes application extends the code with checkbox switches and additional displays to help you experience these effects.

## Flying Heroes

New flying hero:   can fly

Mutate array

### Heroes who fly (piped)

Windstorm  
Tornado

### All Heroes (no pipe)

Windstorm  
Bombasto  
Magneto  
Tornado

---

Replacing the array is an efficient way to signal Angular to update the display. When do you replace the array? When the data change. That's an easy rule to follow in this example where the only way to change the data is by adding a hero.

More often, you don't know when the data have changed, especially in applications that mutate data in many ways, perhaps in application locations far away. A component in such an application usually can't know about those changes. Moreover, it's unwise to distort the component design to accommodate a pipe. Strive to keep the component class independent of the HTML. The component should be unaware of pipes.

For filtering flying heroes, consider an impure pipe.

## Pure and impure pipes

There are two categories of pipes: pure and impure. Pipes are pure by default. Every pipe you've seen so far has been pure. You make a pipe impure by setting its pure flag to false. You could make the FlyingHeroesPipe impure like this:

src/app/flying-heroes.pipe.ts

```
@Pipe({  
  name: 'flyingHeroesImpure',
```

```
pure: false  
})
```

Before doing that, understand the difference between pure and impure, starting with a pure pipe.

## Pure pipes

Angular executes a pure pipe only when it detects a pure change to the input value. A pure change is either a change to a primitive input value (String, Number, Boolean, Symbol) or a changed object reference (Date, Array, Function, Object).

Angular ignores changes within (composite) objects. It won't call a pure pipe if you change an input month, add to an input array, or update an input object property.

This may seem restrictive but it's also fast. An object reference check is fast—much faster than a deep check for differences—so Angular can quickly determine if it can skip both the pipe execution and a view update.

For this reason, a pure pipe is preferable when you can live with the change detection strategy. When you can't, you can use the impure pipe.

Or you might not use a pipe at all. It may be better to pursue the pipe's purpose with a property of the component, a point that's discussed later in this page.

## Impure pipes

Angular executes an impure pipe during every component change detection cycle. An impure pipe is called often, as often as every keystroke or mouse-move.

With that concern in mind, implement an impure pipe with great care. An expensive, long-running pipe could destroy the user experience.

### An impure FlyingHeroesPipe

A flip of the switch turns the FlyingHeroesPipe into a FlyingHeroesImpurePipe. The complete implementation is as follows:

```
FlyingHeroesImpurePipe  
  
FlyingHeroesPipe  
  
@Pipe({  
  name: 'flyingHeroesImpure',  
  pure: false  
})  
export class FlyingHeroesImpurePipe extends FlyingHeroesPipe {}
```

You inherit from FlyingHeroesPipe to prove the point that nothing changed internally. The only difference is the pure flag in the pipe metadata.

This is a good candidate for an impure pipe because the transform function is trivial and fast.

src/app/flying-heroes.pipe.ts (filter)

```
return allHeroes.filter(hero => hero.canFly);
```

You can derive a FlyingHeroesImpureComponent from FlyingHeroesComponent.

src/app/flying-heroes-impure.component.html (excerpt)

```
<div *ngFor="let hero of (heroes | flyingHeroesImpure)">  
  {{hero.name}}  
</div>
```

The only substantive change is the pipe in the template. You can confirm in the [live example](#) / [download example](#) that the flying heroes display updates as you add heroes, even when you mutate the heroes array.

## The impure AsyncPipe

The Angular [AsyncPipe](#) is an interesting example of an impure pipe. The [AsyncPipe](#) accepts a Promise or Observable as input and subscribes to the input automatically, eventually returning the emitted values.

The [AsyncPipe](#) is also stateful. The pipe maintains a subscription to the input Observable and keeps delivering values from that Observable as they arrive.

This next example binds an Observable of message strings ([message\\$](#)) to a view with the [async](#) pipe.

src/app/hero-async-message.component.ts

```
1. import { Component } from '@angular/core';  
2.  
3. import { Observable, interval } from 'rxjs';  
4. import { map, take } from 'rxjs/operators';  
5.  
6. @Component({  
7.   selector: 'app-hero-message',  
8.   template: `  
9.     <h2>Async Hero Message and AsyncPipe</h2>  
10.    <p>Message: {{ message\$ | async }}</p>  
11.    <button (click)="resend()">Resend</button>,  
12.  `)  
13. export class HeroAsyncMessageComponent {  
14.   message\$: Observable<string>;  
15.     
16.   private messages = [  
17.     'You are my hero!',
```

```
18. 'You are the best hero!',  
19. 'Will you be my hero?'  
20. ];  
21.  
22. constructor() { this.resend(); }  
23.  
24. resend() {  
25.   this.message$ = interval(500).pipe(  
26.     map(i => this.messages[i]),  
27.     take(this.messages.length)  
28.   );  
29. }  
30.}
```

The Async pipe saves boilerplate in the component code. The component doesn't have to subscribe to the async data source, extract the resolved values and expose them for binding, and have to unsubscribe when it's destroyed (a potent source of memory leaks).

## An impure caching pipe

Write one more impure pipe, a pipe that makes an HTTP request.

Remember that impure pipes are called every few milliseconds. If you're not careful, this pipe will punish the server with requests.

In the following code, the pipe only calls the server when the request URL changes and it caches the server response. The code uses the [Angular http](#) client to retrieve data:

src/app/fetch-json.pipe.ts

```
1. import { HttpClient }      from '@angular/common/http';  
2. import { Pipe, PipeTransform } from '@angular/core';  
3.  
4. @Pipe({  
5.   name: 'fetch',  
6.   pure: false  
7. })  
8. export class FetchJsonPipe implements PipeTransform {  
9.   private cachedData: any = null;  
10.  private cachedUrl = '';  
11.  
12.  constructor(private http: HttpClient) {}  
13.  
14.  transform(url: string): any {  
15.    if (url !== this.cachedUrl) {  
16.      this.cachedData = null;
```

```
17. this.cachedUrl = url;
18. this.http.get(url).subscribe(result => this.cachedData = result);
19. }
20.
21. return this.cachedData;
22. }
23.}
```

Now demonstrate it in a harness component whose template defines two bindings to this pipe, both requesting the heroes from the heroes.json file.

src/app/hero-list.component.ts

```
1. import { Component } from '@angular/core';
2.
3. @Component({
4. selector: 'app-hero-list',
5. template: `
6. <h2>Heroes from JSON File</h2>
7.
8. <div *ngFor="let hero of ('assets/heroes.json' | fetch) ">
9.   {{hero.name}}
10. </div>
11.
12. <p>Heroes as JSON:
13.   {{'assets/heroes.json' | fetch | json}}
14. </p>
15.})
16.export class HeroListComponent { }
```

The component renders as the following:

## Heroes from JSON File

Windstorm  
Bombasto  
Magneto  
Tornado

Heroes as JSON: [ { "name": "Windstorm" }, {  
"name": "Bombasto" }, { "name": "Magneto" }, {  
"name": "Tornado" } ]

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.
- Each pipe instance only calls the server once.

## JsonPipe

In the previous code sample, the second fetch pipe binding demonstrates more pipe chaining. It displays the same hero data in JSON format by chaining through to the built-in [JsonPipe](#).

### DEBUGGING WITH THE JSON PIPE

The [JsonPipe](#) provides an easy way to diagnosis a mysteriously failing data binding or inspect an object for future binding.

## Pure pipes and pure functions

A pure pipe uses pure functions. Pure functions process inputs and return values without detectable side effects. Given the same input, they should always return the same output.

The pipes discussed earlier in this page are implemented with pure functions. The built-in [DatePipe](#) is a pure pipe with a pure function implementation. So are the [ExponentialStrengthPipe](#) and [FlyingHeroesPipe](#). A few steps back, you reviewed the [FlyingHeroesImpurePipe](#)—an impure pipe with a pure function.

But always implement a pure pipe with a pure function. Otherwise, you'll see many console errors regarding expressions that changed after they were checked.

## Next steps

Pipes are a great way to encapsulate and share common display-value transformations. Use them like styles, dropping them into your template's expressions to enrich the appeal and usability of your views.

Explore Angular's inventory of built-in pipes in the [API Reference](#). Try writing a custom pipe and perhaps contributing it to the community.

## Appendix: No FilterPipe or OrderByPipe

Angular doesn't provide pipes for filtering or sorting lists. Developers familiar with AngularJS know these as filter and orderBy. There are no equivalents in Angular.

This isn't an oversight. Angular doesn't offer such pipes because they perform poorly and prevent aggressive minification. Both filter and orderBy require parameters that reference object properties. Earlier in this page, you learned that such pipes must be [impure](#) and that Angular calls impure pipes in almost every change-detection cycle.

Filtering and especially sorting are expensive operations. The user experience can degrade severely for even moderate-sized lists when Angular calls these pipe methods many times per second. filter and orderBy have often been abused in AngularJS apps, leading to complaints that Angular itself is slow. That charge is fair in the indirect sense that AngularJS prepared this performance trap by offering filter and orderBy in the first place.

The minification hazard is also compelling, if less obvious. Imagine a sorting pipe applied to a list of heroes. The list might be sorted by hero name and planet of origin properties in the following way:

```
<!-- NOT REAL CODE! -->
<div *ngFor="let hero of heroes | orderBy:'name,planet'"></div>
```

You identify the sort fields by text strings, expecting the pipe to reference a property value by indexing (such as hero['name']). Unfortunately, aggressive minification manipulates the Hero property names so that Hero.name and Hero.planet become something like Hero.a and Hero.b. Clearly hero['name'] doesn't work.

While some may not care to minify this aggressively, the Angular product shouldn't prevent anyone from minifying aggressively. Therefore, the Angular team decided that everything Angular provides will minify safely.

The Angular team and many experienced Angular developers strongly recommend moving filtering and sorting logic into the component itself. The component can expose a filteredHeroes or sortedHeroes property and take control over when and how often to execute the supporting logic. Any capabilities that you would have put in a pipe and shared across the app can be written in a filtering/sorting service and injected into the component.

If these performance and minification considerations don't apply to you, you can always create your own such pipes (similar to the [FlyingHeroesPipe](#)) or find them in the community.

Handling user input with forms is the cornerstone of many common applications. Applications use forms to enable users log in, to update a profile, to enter sensitive information, and to perform many other data-entry tasks.

Angular provides two different approaches to handling user input through forms: reactive and template-driven. Both capture user input events from the view, validate the user input, create a form model and data model to update, and provide a way to track changes.

Reactive and template-driven forms differ, however, in how they do the work of processing and managing forms and form data. Each offers different advantages.

In general:

- Reactive forms are more robust: they are more scalable, reusable, and testable. If forms are a key part of your application, or you're already using reactive patterns for building your application, use reactive forms.
- Template-driven forms are useful for adding a simple form to an app, such as an email list signup form. They are easy to add to an app, but they do not scale as well as

reactive forms. If you have very basic form requirements and logic that can be managed solely in the template, use template-driven forms.

This guide provides information to help you decide which approach works best for your situation. It introduces the common building blocks used by both approaches. It also summarizes the key differences between the two approaches, and demonstrates those differences in the context of setup, data flow, and testing.

Note: For complete information about each kind of form, see the [Reactive Forms](#) and [Template-driven Forms](#) guides.

## Key differences

The table below summarizes the key differences between reactive and template-driven forms.

	REACTIVE	TEMPLATE-DRIVEN
Setup (form model)	More explicit, created in the component class.	Less explicit, created by the directives.
Data model	Structured	Unstructured
Predictability	Synchronous	Asynchronous
Form validation	Functions	Directives
Mutability	Immutable	Mutable
Scalability	Low-level API access	Abstraction on top of APIs

## Common foundation

Both reactive and template-driven forms share underlying building blocks.

- A [FormControl](#) instance that tracks the value and validation status of an individual form control.
- A [FormGroup](#) instance that tracks the same values and status for a collection of form controls.
- A [FormArray](#) instance that tracks the same values and status for an array of form controls.
- A [ControlValueAccessor](#) that creates a bridge between Angular [FormControl](#) instances and native DOM elements.

How these control instances are created and managed with reactive and template-driven forms is introduced in the [form model setup](#) section below and detailed further in the [data flow section](#) of this guide.

## Setup: The form model

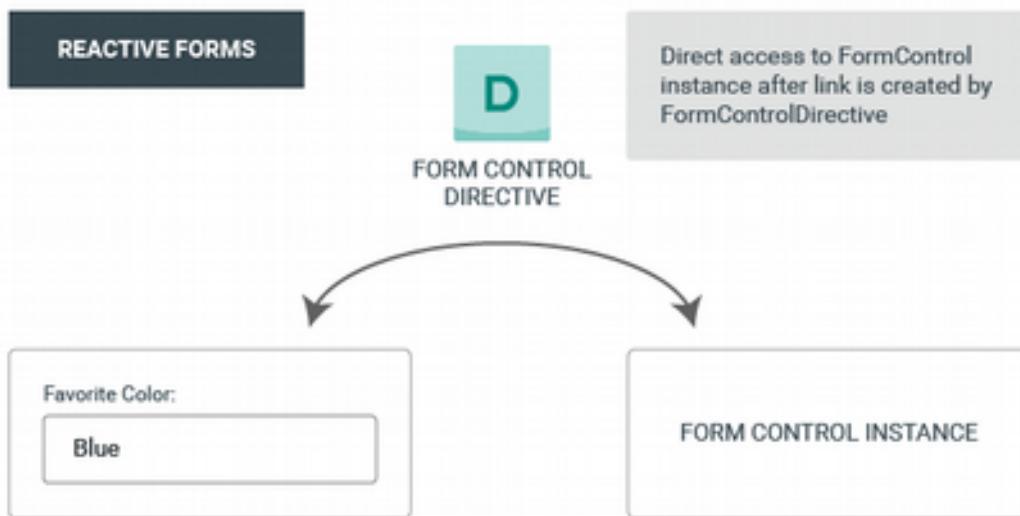
Reactive and template-driven forms both use a form model to track value changes between Angular forms and form input elements. The examples below show how the form model is defined and created.

### Setup in reactive forms

Here is a component with an input field for a single control implemented using reactive forms.

```
1. import { Component } from '@angular/core';
2. import { FormControl } from '@angular/forms';
3.
4. @Component({
5.   selector: 'app-reactive-favorite-color',
6.   template: `
7.     Favorite Color: <input type="text" [FormControl]="favoriteColorControl">
8.   `
9. })
10.export class FavoriteColorComponent {
11. favoriteColorControl = new FormControl("");
12.}
```

The source of truth provides the value and status of the form element at a given point in time. In reactive forms, the form model is source of truth. The form model in the above example is the [FormControl](#) instance.



With reactive forms, the form model is explicitly defined in the component class. The reactive form directive (in this case, [FormControlDirective](#)) then links the existing form control instance to a specific form element in the view using a value accessor (instance of [ControlValueAccessor](#)).

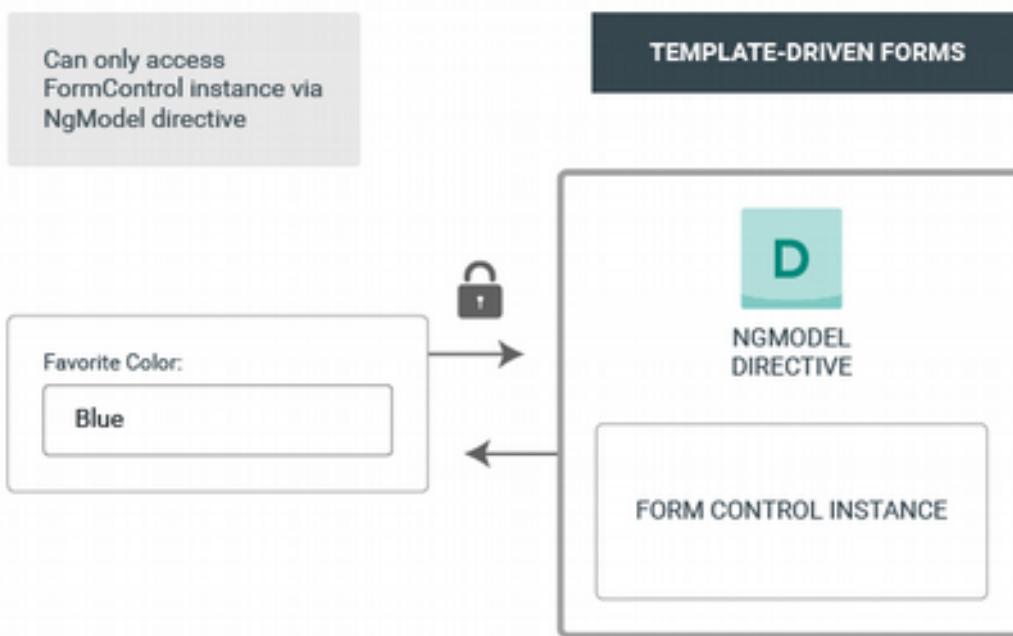
## Setup in template-driven forms

Here is the same component with an input field for a single control implemented using template-driven forms.

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

@Component({
  selector: 'app-template-favorite-color',
  template: `
    Favorite Color: <input type="text" [(ngModel)]="favoriteColor">
  `
})
export class FavoriteColorComponent {
  favoriteColor = '';
}
```

In template-driven forms, the source of truth is the template.



The abstraction of the form model promotes simplicity over structure. The template-driven form directive [NgModel](#) is responsible for creating and managing the form control instance for a given form element. It is less explicit, but you no longer have direct control over the form model.

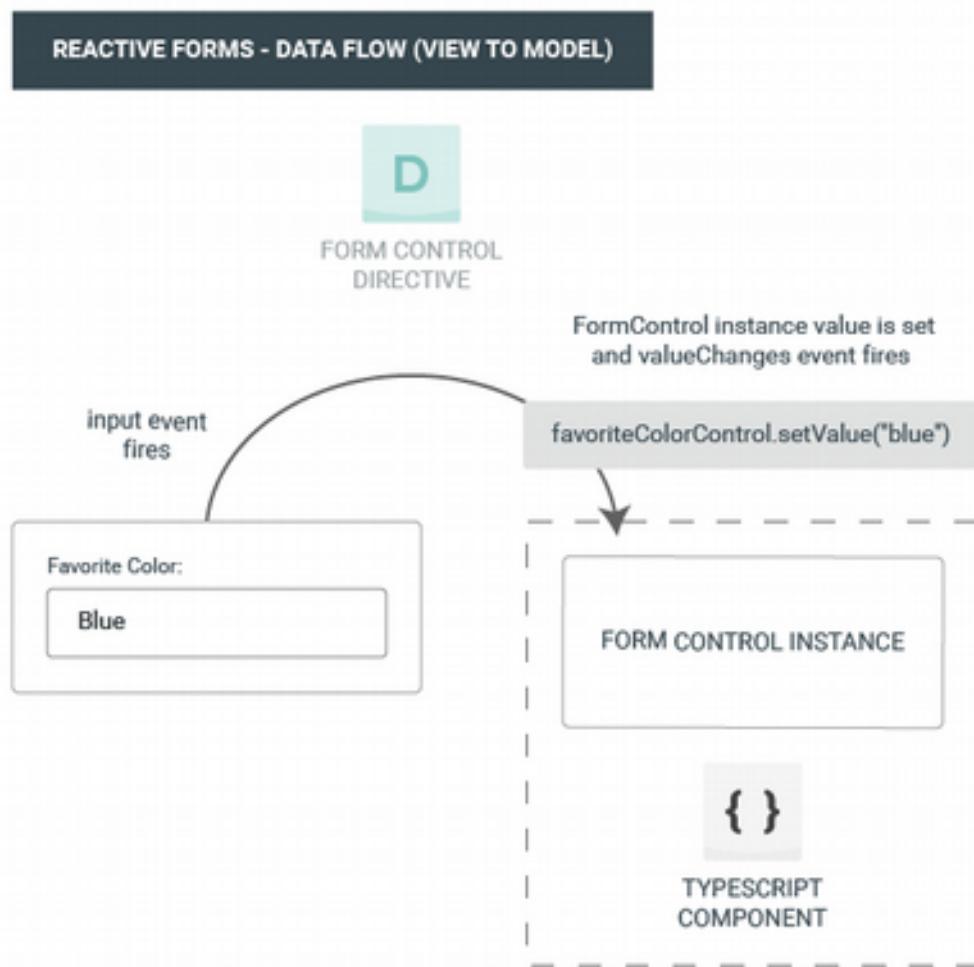
## Data flow in forms

When building forms in Angular, it's important to understand how the framework handles data flowing from the user or from programmatic changes. Reactive and template-driven forms

follow two different strategies when handling form input. The data flow examples below begin with the favorite color input field example from above, and they show how changes to favorite color are handled in reactive forms compared to template-driven forms.

## Data flow in reactive forms

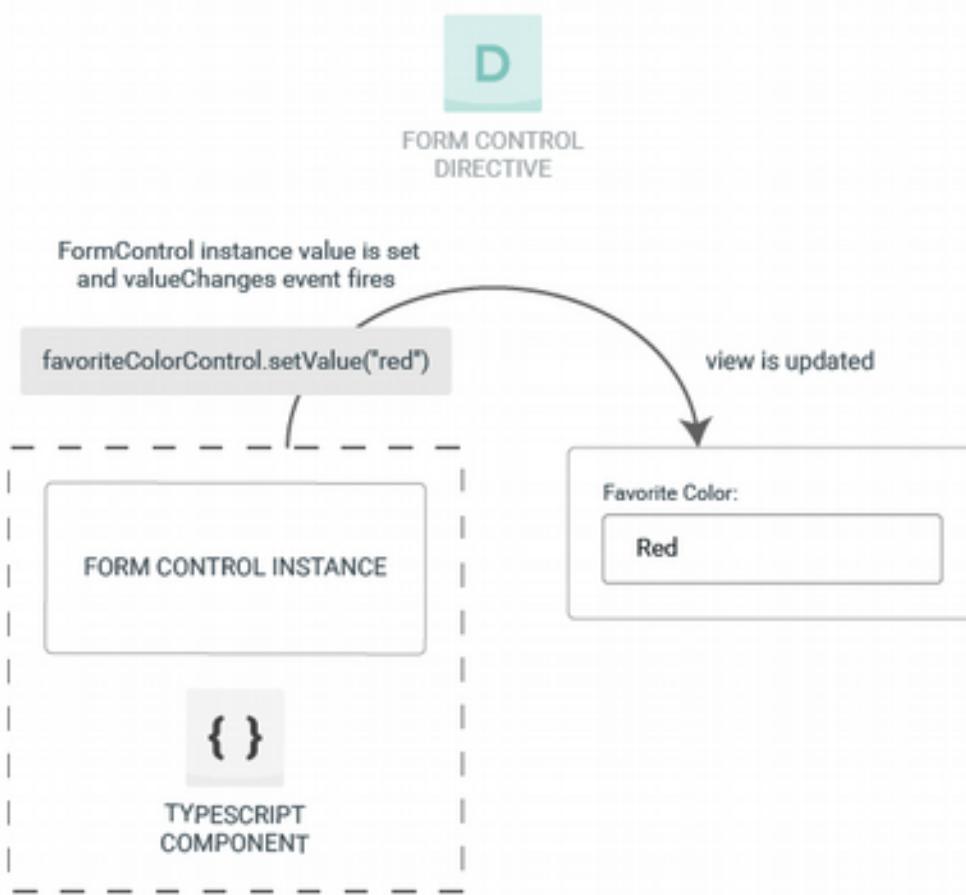
As described above, in reactive forms each form element in the view is directly linked to a form model ([FormControl](#) instance). Updates from the view to model and model to view are synchronous and not dependent on the UI rendered. The diagrams below use the same favorite color example to demonstrate how data flows when an input field's value is changed from the view and then from the model.



The steps below outline the view to model data flow.

1. The end user types a value into the input element, in this case the favorite color "Blue".
2. The form input element emits an "input" event with the latest value.
3. The control value accessor listening for events on the form input element immediately relays the new value to the [FormControl](#) instance.
4. The [FormControl](#) instance emits the new value through the valueChanges observable.
5. Any subscribers to the valueChanges observable receive the new value.

## REACTIVE FORMS - DATA FLOW (MODEL TO VIEW)

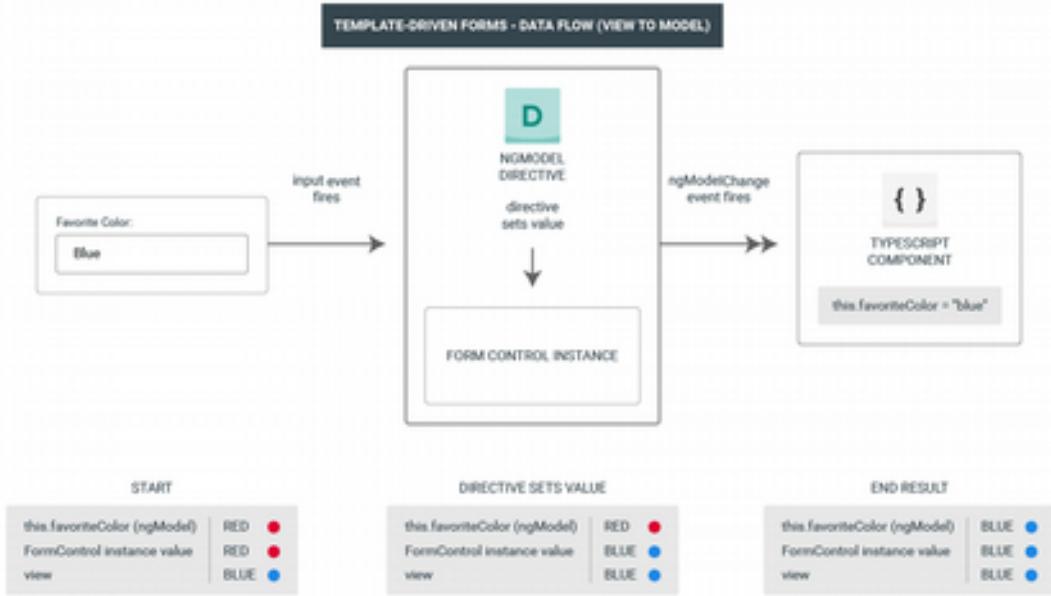


The steps below outline the model to view data flow.

1. The `favoriteColorControl.setValue()` method is called, which updates the [FormControl](#) value.
2. The [FormControl](#) instance emits the new value through the `valueChanges` observable.
3. Any subscribers to the `valueChanges` observable receive the new value.
4. The control value accessor on the form input element updates the element with the new value.

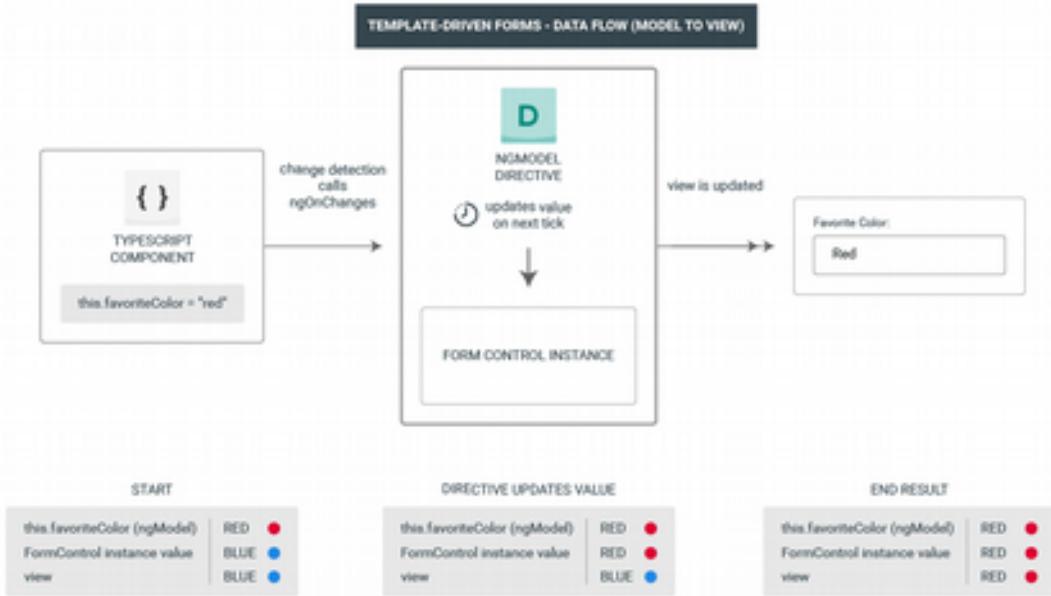
## Data flow in template-driven forms

In template-driven forms, each form element is linked to a directive that manages the form model internally. The diagrams below uses the same favorite color example to demonstrate how data flows when an input field's value is changed from the view and then from the model.



The steps below outline the view to model data flow.

1. The end user types "Blue" into the input element.
2. The input element emits an "input" event with the value "Blue".
3. The control value accessor attached to the input triggers the `setValue()` method on the [FormControl](#) instance.
4. The [FormControl](#) instance emits the new value through the `valueChanges` observable.
5. Any subscribers to the `valueChanges` observable receive the new value.
6. The control value accessor also calls the `NgModel.viewToModel()` method which emits an `ngModelChange` event.
7. Because the component template uses two-way data binding for the `favoriteColor`, the `favoriteColor` property in the component is updated to the value emitted by the `ngModelChange` event ("Blue").



The steps below outline the model to view data flow.

1. The favoriteColor value is updated in the component.
2. Change detection begins.
3. During change detection, the `ngOnChanges` lifecycle hook is called on the [NgModel](#) directive instance because the value of one of its inputs has changed.
4. The `ngOnChanges()` method queues an async task to set the value for the internal [FormControl](#) instance.
5. Change detection completes.
6. On the next tick, the task to set the [FormControl](#) instance value is executed.
7. The [FormControl](#) instance emits the latest value through the `valueChanges` observable.
8. Any subscribers to the `valueChanges` observable receive the new value.
9. The control value accessor updates the form input element in the view with the latest favoriteColor value.

## Form validation

Validation is an integral part of managing any set of forms. Whether you're checking for required fields or querying an external API for an existing username, Angular provides a set of built-in validators as well as the ability to create custom validators.

- Reactive forms define custom validators as functions that receive a control to validate.
- Template-driven forms are tied to template directives, and must provide custom validator directives that wrap validation functions.

For more on form validation, see the [Form Validation](#) guide.

# Testing

Testing also plays a large part in complex applications and an easier testing strategy is always welcomed. One difference in testing reactive forms and template-driven forms is their reliance on rendering the UI in order to perform assertions based on form control and form field changes. The following examples demonstrate the process of testing forms with reactive and template-driven forms.

## Testing reactive forms

Reactive forms provide a relatively easy testing strategy because they provide synchronous access to the form and data models, and they can be tested without rendering the UI. In these set of tests, controls and data are queried and manipulated through the control without interacting with the change detection cycle.

The following tests use the favorite color components mentioned earlier to verify the view to model and model to view data flows for a reactive form.

The following test verifies the view to model data flow:

Favorite color test - view to model

```
it('should update the value of the input field', () => {
  const input = fixture.nativeElement.querySelector('input');
  const event = createNewEvent('input');

  input.value = 'Red';
  input.dispatchEvent(event);

  expect(fixture.componentInstance.favoriteColorControl.value).toEqual('Red');
});
```

The steps performed in the view to model test.

1. Query the view for the form input element, and create a custom "input" event for the test.
2. Set the new value for the input is set to Red, and dispatch the "input" event on the form input element.
3. Assert that the favoriteColor [FormControl](#) instance value matches the value from the input.

The following test verifies the model to view data flow:

Favorite color test - model to view

```
it('should update the value in the control', () => {
  component.favoriteColorControl.setValue('Blue');

  const input = fixture.nativeElement.querySelector('input');
```

```
expect(input.value).toBe('Blue');
});
```

The steps performed in the model to view test.

1. Use the favoriteColor [FormControl](#) instance to set the new value.
2. Query the view for the form input element.
3. Assert that the new value set on the control matches the value in the input.

## Testing template-driven forms

Writing tests with template-driven forms requires more detailed knowledge of the change detection process and how directives run on each cycle to ensure elements are queried, tested, or changed at the correct time.

The following tests use the favorite color components mentioned earlier to verify the view to model and model to view data flows for a template-driven form.

The following test verifies the view to model data flow:

Favorite color test - view to model

```
it('should update the favorite color in the component', fakeAsync(() => {
  const input = fixture.nativeElement.querySelector('input');
  const event = createNewEvent('input');

  input.value = 'Red';
  input.dispatchEvent(event);

  fixture.detectChanges();

  expect(component.favoriteColor).toEqual('Red');
}));
```

The steps performed in the view to model test.

1. Query the view for the form input element, and create a custom "input" event for the test.
2. Set the new value for the input is set to Red, and dispatch the "input" event on the form input element.
3. Run change detection through the test fixture.
4. Assert that the component favoriteColor property value matches the value from the input.

The following test verifies the model to view data flow:

Favorite color test - model to view

```

it('should update the favorite color on the input field', fakeAsync(() => {
  component.favoriteColor = 'Blue';

  fixture.detectChanges();

  tick();

  const input = fixture.nativeElement.querySelector('input');

  expect(input.value).toBe('Blue');
});

```

The steps performed in the model to view test.

1. Use the component instance to set the value of favoriteColor property.
2. Run change detection through the test fixture.
3. Use the tick() method to simulate passage of time within the fakeAsync() task.
4. Query the view for the form input element.
5. Assert that the input value matches the favoriteColor value property in the component instance.

## Mutability

How changes are tracked plays a role in the efficiency of your application.

- Reactive forms keep the data model pure by providing it as an immutable data structure. Each time a change is triggered on the data model, the FormControl instance returns a new data model rather than updating the data model directly. This gives you the ability track unique changes to the data model through the control's observable. This allows change detection to be more efficient because it only needs to update on unique changes. It also follows reactive patterns that integrate with observable operators to transform data.
- Template-driven forms rely on mutability with two-way data binding to update the data model in the component as changes are made in the template. Because there are no unique changes to track on the data model when using two-way data binding, change detection is less efficient at determining when updates are required.

The difference is demonstrated in the examples above using the favorite color input element.

- With reactive forms, the FormControl instance always returns a new value when the control's value is updated.
- With template-driven forms, the favorite color property is always modified to its new value.

## Scalability

If forms are a central part of your application, scalability is very important. Being able to reuse form models across components is critical.

- Reactive forms make creating large scale forms easier by providing access to low-level APIs and synchronous access to the form model.
- Template-driven forms focus on simple scenarios, are not as reusable, abstract away the low-level APIs and access to the form model is provided asynchronously. The abstraction with template-driven forms surfaces in testing also, where testing reactive forms requires less setup and no dependence on the change detection cycle when updating and validating the form and data models during testing.

## Final Thoughts

Choosing a strategy begins with understanding the strengths and weaknesses of the options presented. Low-level API and form model access, predictability, mutability, straightforward validation and testing strategies, and scalability are all important consideration in choosing the infrastructure you use when building your forms in Angular. Template-driven forms are similar to patterns in AngularJS, but they have limitations given the criteria of many modern, large-scale Angular apps. Reactive forms integrate with reactive patterns already present in other areas of the Angular architecture, and complement those requirements well. Those limitations are alleviated with reactive forms.

## Next Steps

The following guides are the next steps in the learning process.

To learn more about reactive forms, see the following guides:

- [Reactive Forms](#)
- [Form Validation](#)
- [Dynamic forms](#)

To learn more about template-driven forms, see the following guides:

- [Template-driven Forms](#)
- [Form Validation](#)

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 simple form control, progress to using multiple controls in a group, validate form values, and implement more advanced forms.

Try the [Reactive Forms live-example](#) / [download example](#).

## Introduction to 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 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 in distinct ways. Reactive forms provide more predictability with synchronous access to the data model, immutability with observable operators, and change tracking through observable streams. If you prefer direct access to modify data in your template, template-driven forms are less explicit because they rely on directives embedded in the template, along with mutable data to track changes asynchronously. See the [Forms Overview](#)for detailed comparisons between the two paradigms.

## Getting started

This section describes 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.

### Step 1: Registering the reactive forms module

To use reactive forms, import [ReactiveFormsModule](#) from the @angular/forms package and add it to your NgModule's [imports](#)array.

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

```
import { ReactiveFormsModule } from '@angular/forms';
```

```
@NgModule({
  imports: [
    // other imports ...
    ReactiveFormsModule
  ],
})
export class AppModule {}
```

### Step 2: Generating and importing a new form control

Generate a component for the control.

ng generate component NameEditor

The [FormControl](#) class is the basic building block when using reactive forms. To register a single form control, import the [FormControl](#) class into your component and create a new instance of the form control to save as a class property.

src/app/name-editor/name-editor.component.ts

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

```
@Component({
  selector: 'app-name-editor',
  templateUrl: './name-editor.component.html',
  styleUrls: ['./name-editor.component.css']
})
export class NameEditorComponent {
  name = new FormControl("");
}
```

Use the constructor of [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.

### Step 3: Registering 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](#) included in [ReactiveFormsModule](#).

src/app/name-editor/name-editor.component.html

```
<label>
  Name:
  <input type="text" [formControl]="name">
</label>
```

Note: For a more detailed list of classes and directives provided by [ReactiveFormsModule](#), see the [Reactive forms API](#) section.

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.

### Displaying 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)

```
<app-name-editor></app-name-editor>
```

Name:

## Managing control values

Reactive forms give you access to the form control state and value at a point in time. You can manipulate the current state and value through the component class or the component template. The following examples display the value of the form control instance and change it.

### Displaying a form control value

You can display the value in these ways:

- Through the `valueChanges` observable where you can listen for changes in the form's value in the template using [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)
```

```
<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](#) class are used to control form state and determine when to display messages when handling validation. For more information, see [Simple form validation](#) later in this guide.

Read about other [FormControl](#) properties and methods in the [Reactive forms API](#) section.

### 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)
```

```
updateName() {  
  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)
```

```
<p>  
  <button (click)="updateName()">Update Name</button>  
</p>
```

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.

Name:  
Nancy

Value: Nancy

Update Name

Note: In this example, you're using a single control. When using the `setValue()` method with a form group or form array instance, the value needs to match the structure of the group or array.

## Grouping form controls

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](#) and [FormControl](#) classes from the `@angular/forms` package.

```
ng generate component ProfileEditor
```

```
src/app/profile-editor/profile-editor.component.ts (imports)
```

```
import { FormGroup, FormControl } from '@angular/forms';
```

## Step 1: Creating 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)
```

```
1. import { Component } from '@angular/core';
2. import { FormGroup, FormControl } from '@angular/forms';
3.
4. @Component({
5.   selector: 'app-profile-editor',
6.   templateUrl: './profile-editor.component.html',
7.   styleUrls: ['./profile-editor.component.css']
8. })
9. export class ProfileEditorComponent {
10.   profileForm = new FormGroup({
11.     firstName: new FormControl(),
12.     lastName: new FormControl(),
13.   });
14. }
```

The individual form controls are now collected within a group. A [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.

## Step 2: Associating 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)
```

```
<form [formGroup]="profileForm">
```

```
<label>
  First Name:
  <input type="text" formControlName="firstName">
</label>
```

```
<label>
  Last Name:
  <input type="text" formControlName="lastName">
</label>

</form>
```

Note that just as a form group contains a group of controls, the profile form [FormGroup](#) is bound to the form element with the [FormGroup](#) directive, creating a communication layer between the model and the form containing the inputs. The [formControlName](#) input provided by the [FormControlName](#) directive binds each individual input to the form control defined in [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.

## Saving 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](#) 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)
```

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

The onSubmit() method in the ProfileEditor component captures the current value of profileForm. Use [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)
```

```
onSubmit() {
  // TODO: Use 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)
```

```
<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. Simple form validation is covered in the [Simple form validation](#) section.

## Displaying the component

To display the ProfileEditor component that contains the form, add it to a component template.

```
src/app/app.component.html (profile editor)
```

```
<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.

The screenshot shows a user interface with two text input fields. The first field is labeled "First Name:" and the second is labeled "Last Name:". Both fields are empty. Below the inputs is a grey "Submit" button. The entire form is contained within a light gray box.

## Creating nested form groups

When building complex forms, managing the different areas of information is easier in smaller sections, and some groups of information naturally fall into the same group. Using a nested form group instance allows you to break large forms groups into smaller, more manageable ones.

### Step 1: Creating a nested group

An address is a good example of information that can be grouped together. Form groups can accept both form control and form group instances as children. This makes composing complex form models easier to maintain and logically group together. 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)
```

```
import { Component } from '@angular/core';
import { FormGroup, FormControl } from '@angular/forms';
```

```
@Component({
```

```

selector: 'app-profile-editor',
templateUrl: './profile-editor.component.html',
styleUrls: ['./profile-editor.component.css']
})
export class ProfileEditorComponent {
profileForm = new FormGroup({
  firstName: new FormControl(),
  lastName: new FormControl(),
  address: new FormGroup({
    street: new FormControl(),
    city: new FormControl(),
    state: new FormControl(),
    zip: new 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.

## Step 2: Grouping 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 `firstName` and `lastName` fields to the `ProfileEditor` template.

`src/app/profile-editor/profile-editor.component.html` (template nested form group)

```

<div formGroupName="address">
  <h3>Address</h3>

  <label>
    Street:
    <input type="text" formControlName="street">
  </label>

  <label>
    City:
    <input type="text" formControlName="city">
  </label>

```

```
<label>
  State:
  <input type="text" formControlName="state">
</label>

<label>
  Zip Code:
  <input type="text" formControlName="zip">
</label>
</div>
```

The ProfileEditor form is displayed as one group, but the model is broken down further to represent the logical grouping areas.

The screenshot shows a form titled "Address". It contains four groups of input fields:

- First Name:** An input field.
- Last Name:** An input field.
- Address** (Section Header):
  - Street:** An input field.
  - City:** An input field.
  - State:** An input field.
- Zip Code:** An input field.

A "Submit" button is located at the bottom left of the form area.

Note: Display the value for the form group instance in the component template using the `value` property and [JsonPipe](#).

# Partial model updates

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.

## Patching the model value

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)`

```
updateProfile() {
  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)`

```
<p>
  <button (click)="updateProfile()">Update Profile</button>
</p>
```

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.

# Generating form controls with FormBuilder

Creating form control instances manually can become repetitive when dealing with multiple forms. The [FormBuilder](#) service provides convenient methods for generating controls.

The following section refactors the ProfileEditor component to use the form builder service to create form control and form group instances.

## Step 1: Importing the FormBuilder class

Import the [FormBuilder](#) class from the `@angular/forms` package.

```
src/app/profile-editor/profile-editor.component.ts (import)
```

```
import { FormBuilder } from '@angular/forms';
```

## Step 2: Injecting the FormBuilder service

The [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)
```

```
constructor(private fb: FormBuilder) { }
```

## Step 3: Generating form controls

The [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)
```

```
1. import { Component } from '@angular/core';
2. import { FormBuilder } from '@angular/forms';
3.
4. @Component({
5.   selector: 'app-profile-editor',
6.   templateUrl: './profile-editor.component.html',
7.   styleUrls: ['./profile-editor.component.css']
8. })
9. export class ProfileEditorComponent {
10.   profileForm = this.fb.group({
11.     firstName: [],
12.     lastName: [],
13.     address: this.fb.group({
14.       street: [],
```

```
15.   city: [""],
16.   state: [""],
17.   zip: [""]
18. },
19. });
20.
21. constructor(private fb: FormBuilder) { }
22.}
```

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.

Note: 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)
```

```
profileForm = new FormGroup({
  firstName: new FormControl(""),
  lastName: new FormControl(""),
  address: new FormGroup({
    street: new FormControl(""),
    city: new FormControl(""),
    state: new FormControl(""),
    zip: new FormControl("")
  })
});
```

## Simple form validation

Form validation is used to validate user input to ensure it's 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](#) guide.

### Step 1: Importing 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](#) class from the @angular/forms package.

```
src/app/profile-editor/profile-editor.component.ts (import)
```

```
import { Validators } from '@angular/forms';
```

## Step 2: Making a field required

The most common validation is making a field required. This section describes how to add a required validation to the firstName control.

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)

```
profileForm = this.fb.group({
  firstName: ['', Validators.required],
  lastName: '',
  address: this.fb.group({
    street: '',
    city: '',
    state: '',
    zip: ''
  }),
});
```

HTML5 has a set of built-in attributes that you can use for native validation, including required, [minlength](#), and [maxlength](#). You can take advantage of these optional attributes on your form input elements. Add the required attribute to the firstName input element.

src/app/profile-editor/profile-editor.component.html (required attribute)

```
<input type="text" formControlName="firstName" required>
```

Caution: Use these HTML5 validation attributes in combination with the built-in validators provided by Angular's reactive forms. Using these in combination prevents errors when the expression is changed after the template has been checked.

## Displaying 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)

```
<p>
  Form Status: {{ profileForm.status }}
</p>
```

**First Name:**

**Last Name:**

## Address

**Street:**

**City:**

**State:**

**Zip Code:**

**Submit**

Form Value: { "firstName": "", "lastName": "", "address": { "street": "", "city": "", "state": "", "zip": "" } }

Form Status: INVALID

**Update Profile**

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](#) guide.

## Dynamic controls using form arrays

[FormArray](#) is an alternative to [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. The following example shows you how to manage an array of aliases in ProfileEditor.

## Step 1: Importing the FormArray class

Import the [FormArray](#) class from @angular/forms to use for type information. The [FormBuilder](#) service is ready to create a [FormArray](#) instance.

src/app/profile-editor/profile-editor.component.ts (import)

```
import { FormArray } from '@angular/forms';
```

## Step 2: Defining 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\(\)](#) method to define the array, and the [FormBuilder.control\(\)](#) method to populate the array with an initial control.

src/app/profile-editor/profile-editor.component.ts (aliases form array)

```
1. profileForm = this.fb.group({
2.   firstName: ['', Validators.required],
3.   lastName: [''],
4.   address: this.fb.group({
5.     street: [''],
6.     city: [''],
7.     state: [''],
8.     zip: ['']
9.   }),
10.  aliases: this.fb.array([
11.    this.fb.control('')
12.  ])
13.});
```

The `aliases` control in the form group instance is now populated with a single control until more controls are added dynamically.

## Step 3: Accessing 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)
```

```
get aliases() {  
  return this.profileForm.get('aliases') as FormArray;  
}
```

Note: Because the returned control is of the type [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\(\)](#) method inserts the control as a new item in the array.

```
src/app/profile-editor/profile-editor.component.ts (add alias)
```

```
addAlias() {  
  this.aliases.push(this.fb.control());  
}
```

In the template, each control is displayed as a separate input field.

## Step 4: Displaying 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](#) input provided by FormGroupNameDirective, [formArrayName](#) binds communication from the form array instance to the template with FormArrayNameDirective.

Add the template HTML below after the <div> closing the [formGroupName](#) element.

```
src/app/profile-editor/profile-editor.component.html (aliases form array template)
```

```
<div formArrayName="aliases">  
  <h3>Aliases</h3> <button (click)="addAlias()">Add Alias</button>  
  
<div *ngFor="let address of aliases.controls; let i=index">  
  <!-- The repeated alias template -->  
  <label>  
    Alias:  
    <input type="text" formControlName="i">  
  </label>  
</div>  
</div>
```

The [\\*ngFor](#) 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](#) input.

First Name:

Last Name:

## Address

Street:

City:

State:

Zip Code:

## Aliases

Add Alias

Alias:

Submit

Form Value: { "firstName": "", "lastName": "", "address": {  
"street": "", "city": "", "state": "", "zip": "" }, "aliases": [ [ "" ] ] }

Form Status: INVALID

Update Profile

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.

### Adding 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 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.

## Appendix

### Reactive forms API

Listed below are the base classes and services used to create and manage form controls.

#### Classes

Class	Description
<a href="#">AbstractControl</a>	The abstract base class for the concrete form control classes <a href="#">FormControl</a> , <a href="#">FormGroup</a> , and <a href="#">FormArray</a> . It provides their common behaviors and properties.
<a href="#">FormControl</a>	Manages the value and validity status of an individual form control. It corresponds to an HTML form control such as <input> or <select>.
<a href="#">FormGroup</a>	Manages the value and validity state of a group of <a href="#">AbstractControl</a> instances. The group's properties include its child controls. The top-level form in your component is <a href="#">FormGroup</a> .
<a href="#">FormArray</a>	Manages the value and validity state of a numerically indexed array of <a href="#">AbstractControl</a> instances.
<a href="#">FormBuilder</a>	An injectable service that provides factory methods for creating control instances.

#### Directives

Directive	Description
<a href="#">FormControlDirective</a>	Syncs a standalone <a href="#">FormControl</a> instance to a form control element.
<a href="#">FormControlName</a>	Syncs <a href="#">FormControl</a> in an existing <a href="#">FormGroup</a> instance to a form control element by name.
<a href="#">FormGroupDirective</a>	Syncs an existing <a href="#">FormGroup</a> instance to a DOM element.
<a href="#">FormGroupName</a>	Syncs a nested <a href="#">FormGroup</a> instance to a DOM element.
<a href="#">FormArrayName</a>	Syncs a nested <a href="#">FormArray</a> instance to a DOM element.

Forms are the mainstay of business applications. You use forms to log in, submit a help request, place an order, book a flight, schedule a meeting, and perform countless other data-entry tasks.

In developing a form, it's important to create a data-entry experience that guides the user efficiently and effectively through the workflow.

Developing forms requires design skills (which are out of scope for this page), as well as framework support for two-way data binding, change tracking, validation, and error handling, which you'll learn about on this page.

This page shows you how to build a simple form from scratch. Along the way you'll learn how to:

- Build an Angular form with a component and template.
- Use [ngModel](#) to create two-way data bindings for reading and writing input-control values.
- Track state changes and the validity of form controls.
- Provide visual feedback using special CSS classes that track the state of the controls.
- Display validation errors to users and enable/disable form controls.
- Share information across HTML elements using template reference variables.

You can run the [live example](#) / [download example](#) in Stackblitz and download the code from there.

## Template-driven forms

You can build forms by writing templates in the Angular [template syntax](#) with the form-specific directives and techniques described in this page.

You can also use a reactive (or model-driven) approach to build forms. However, this page focuses on template-driven forms.

You can build almost any form with an Angular template—login forms, contact forms, and pretty much any business form. You can lay out the controls creatively, bind them to data, specify validation rules and display validation errors, conditionally enable or disable specific controls, trigger built-in visual feedback, and much more.

Angular makes the process easy by handling many of the repetitive, boilerplate tasks you'd otherwise wrestle with yourself.

You'll learn to build a template-driven form that looks like this:

# Hero Form

Name

Dr IQ

Alter Ego

Chuck Overstreet

Hero Power

Really Smart

Submit

The Hero Employment Agency uses this form to maintain personal information about heroes. Every hero needs a job. It's the company mission to match the right hero with the right crisis.

Two of the three fields on this form are required. Required fields have a green bar on the left to make them easy to spot.

If you delete the hero name, the form displays a validation error in an attention-grabbing style:

# Hero Form

Name

Name is required

Alter Ego

Hero Power

▼

Submit

Note that the Submit button is disabled, and the "required" bar to the left of the input control changes from green to red.

You can customize the colors and location of the "required" bar with standard CSS.

You'll build this form in small steps:

1. Create the Hero model class.
2. Create the component that controls the form.
3. Create a template with the initial form layout.
4. Bind data properties to each form control using the [ngModel](#) two-way data-binding syntax.
5. Add a name attribute to each form-input control.
6. Add custom CSS to provide visual feedback.
7. Show and hide validation-error messages.
8. Handle form submission with ngSubmit.
9. Disable the form's Submit button until the form is valid.

## Setup

Create a new project named angular-forms:

```
ng new angular-forms
```

## Create the Hero model class

As users enter form data, you'll capture their changes and update an instance of a model. You can't lay out the form until you know what the model looks like.

A model can be as simple as a "property bag" that holds facts about a thing of application importance. That describes well the Hero class with its three required fields (id, name, power) and one optional field (alterEgo).

Using the Angular CLI, generate a new class named Hero:

```
ng generate class Hero
```

With this content:

```
src/app/hero.ts
```

```
export class Hero {
```

```
    constructor(
```

```
        public id: number,
```

```
        public name: string,
```

```
        public power: string,
```

```
        public alterEgo?: string
```

```
) { }
```

```
}
```

It's an anemic model with few requirements and no behavior. Perfect for the demo.

The TypeScript compiler generates a public field for each public constructor parameter and automatically assigns the parameter's value to that field when you create heroes.

The alterEgo is optional, so the constructor lets you omit it; note the question mark (?) in alterEgo?.

You can create a new hero like this:

```
let myHero = new Hero(42, 'SkyDog',
    'Fetch any object at any distance',
    'Leslie Rollover');
console.log('My hero is called ' + myHero.name); // "My hero is called SkyDog"
```

## Create a form component

An Angular form has two parts: an HTML-based template and a component class to handle data and user interactions programmatically. Begin with the class because it states, in brief, what the hero editor can do.

Using the Angular CLI, generate a new component named HeroForm:

ng generate component HeroForm

With this content:

src/app/hero-form/hero-form.component.ts (v1)

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

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

```
@Component({
```

```
  selector: 'app-hero-form',
```

```
  templateUrl: './hero-form.component.html',
```

```
  styleUrls: ['./hero-form.component.css']
```

```
})
```

```
export class HeroFormComponent {
```

```
  powers = ['Really Smart', 'Super Flexible',  
           'Super Hot', 'Weather Changer'];
```

```
  model = new Hero(18, 'Dr IQ', this.powers[0], 'Chuck Overstreet');
```

```
  submitted = false;
```

```
  onSubmit() { this.submitted = true; }
```

```
// TODO: Remove this when we're done
```

```
  get diagnostic() { return JSON.stringify(this.model); }
```

```
}
```

There's nothing special about this component, nothing form-specific, nothing to distinguish it from any component you've written before.

Understanding this component requires only the Angular concepts covered in previous pages.

- The code imports the Angular core library and the Hero model you just created.
- The `@Component` selector value of "app-hero-form" means you can drop this form in a parent template with a `<app-hero-form>` tag.
- The `templateUrl` property points to a separate file for the template HTML.
- You defined dummy data for `model` and `powers`, as befits a demo.

Down the road, you can inject a data service to get and save real data or perhaps expose these properties as inputs and outputs (see [Input and output properties](#) on the [Template Syntax](#) page) for binding to a parent component. This is not a concern now and these future changes won't affect the form.

- You added a diagnostic property to return a JSON representation of the model. It'll help you see what you're doing during development; you've left yourself a cleanup note to discard it later.

## Revise app.module.ts

app.module.ts defines the application's root module. In it you identify the external modules you'll use in the application and declare the components that belong to this module, such as the HeroFormComponent.

Because template-driven forms are in their own module, you need to add the [FormsModule](#) to the array of [imports](#) for the application module before you can use forms.

Update it with the following:

src/app/app.module.ts

```

1. import { NgModule }      from '@angular/core';
2. import { BrowserModule } from '@angular/platform-browser';
3. import { FormsModule }   from '@angular/forms';
4.
5. import { AppComponent }  from './app.component';
6. import { HeroFormComponent } from './hero-form/hero-form.component';
7.
8. @NgModule({
9.   imports: [
10.     BrowserModule,
11.     FormsModule
12.   ],
13.   declarations: [
14.     AppComponent,
15.     HeroFormComponent
16.   ],
17.   providers: [],
18.   bootstrap: [ AppComponent ]
19. })
20. export class AppModule { }
```

There are two changes:

1. You import [FormsModule](#).
2. You add the [FormsModule](#) to the list of [imports](#) defined in the [@NgModule](#) decorator.  
This gives the application access to all of the template-driven forms features, including [ngModel](#).

If a component, directive, or pipe belongs to a module in the [imports](#) array, don't re-declare it in the [declarations](#) array. If you wrote it and it should belong to this module, do declare it in the [declarations](#) array.

## Revise app.component.html

AppComponent is the application's root component. It will host the new HeroFormComponent.

Replace the contents of its template with the following:

```
src/app/app.component.html
```

```
<app-hero-form></app-hero-form>
```

There are only two changes. The [template](#) is simply the new element tag identified by the component's selector property. This displays the hero form when the application component is loaded. Don't forget to remove the namefield from the class body as well.

## Create an initial HTML form template

Update the template file with the following contents:

```
src/app/hero-form/hero-form.component.html
```

```
1. <div class="container">
2.   <h1>Hero Form</h1>
3.   <form>
4.     <div class="form-group">
5.       <label for="name">Name</label>
6.       <input type="text" class="form-control" id="name" required>
7.     </div>
8.
9.     <div class="form-group">
10.      <label for="alterEgo">Alter Ego</label>
11.      <input type="text" class="form-control" id="alterEgo">
12.    </div>
13.
14.    <button type="submit" class="btn btn-success">Submit</button>
15.
16.  </form>
17.</div>
```

The language is simply HTML5. You're presenting two of the Hero fields, name and alterEgo, and opening them up for user input in input boxes.

The Name <input> control has the HTML5 required attribute; the Alter Ego <input> control does not because alterEgo is optional.

You added a Submit button at the bottom with some classes on it for styling.

You're not using Angular yet. There are no bindings or extra directives, just layout.

In template driven forms, if you've imported [FormsModule](#), you don't have to do anything to the <form> tag in order to make use of [FormsModule](#). Continue on to see how this works.

The container, form-group, form-control, and btn classes come from [Twitter Bootstrap](#). These classes are purely cosmetic. Bootstrap gives the form a little style.

## ANGULAR FORMS DON'T REQUIRE A STYLE LIBRARY

Angular makes no use of the container, form-group, form-control, and btn classes or the styles of any external library. Angular apps can use any CSS library or none at all.

To add the stylesheet, open styles.css and add the following import line at the top:

```
src/styles.css
```

```
@import url('https://unpkg.com/bootstrap@3.3.7/dist/css/bootstrap.min.css');
```

## Add powers with \*ngFor

The hero must choose one superpower from a fixed list of agency-approved powers. You maintain that list internally (in HeroFormComponent).

You'll add a select to the form and bind the options to the powers list using [ngFor](#), a technique seen previously in the [Displaying Data](#) page.

Add the following HTML immediately below the Alter Ego group:

```
src/app/hero-form/hero-form.component.html (powers)
```

```
<div class="form-group">
  <label for="power">Hero Power</label>
  <select class="form-control" id="power" required>
    <option *ngFor="let pow of powers" [value]="pow">{{pow}}</option>
  </select>
</div>
```

This code repeats the <[option](#)> tag for each power in the list of powers. The pow template input variable is a different power in each iteration; you display its name using the interpolation syntax.

## Two-way data binding with ngModel

Running the app right now would be disappointing.

# Hero Form

Name

Alter Ego

Hero Power

Really Smart ▾

You don't see hero data because you're not binding to the Hero yet. You know how to do that from earlier pages. [Displaying Data](#) teaches property binding. [User Input](#) shows how to listen for DOM events with an event binding and how to update a component property with the displayed value.

Now you need to display, listen, and extract at the same time.

You could use the techniques you already know, but instead you'll use the new `[(ngModel)]` syntax, which makes binding the form to the model easy.

Find the `<input>` tag for Name and update it like this:

src/app/hero-form/hero-form.component.html (excerpt)

```
<input type="text" class="form-control" id="name"
    required
    [(ngModel)]="model.name" name="name">
```

TODO: remove this: `{{model.name}}`

You added a diagnostic interpolation after the input tag so you can see what you're doing. You left yourself a note to throw it away when you're done.

Focus on the binding syntax: `[(ngModel)]="..."`.

You need one more addition to display the data. Declare a template variable for the form.

Update the `<form>` tag with `#heroForm="ngForm"` as follows:

src/app/hero-form/hero-form.component.html (excerpt)

```
<form #heroForm="ngForm">
```

The variable heroForm is now a reference to the [NgForm](#) directive that governs the form as a whole.

## The NgForm directive

What [NgForm](#) directive? You didn't add an [NgForm](#) directive.

Angular did. Angular automatically creates and attaches an [NgForm](#) directive to the <form> tag.

The [NgForm](#) directive supplements the form element with additional features. It holds the controls you created for the elements with an [ngModel](#) directive and name attribute, and monitors their properties, including their validity. It also has its own valid property which is true only if every contained control is valid.

If you ran the app now and started typing in the Name input box, adding and deleting characters, you'd see them appear and disappear from the interpolated text. At some point it might look like this:

Dr IQ 3000

**TODO: remove this: Dr IQ 3000**

The diagnostic is evidence that values really are flowing from the input box to the model and back again.

That's two-way data binding. For more information, see [Two-way binding with NgModel](#) on the [Template Syntax](#) page.

Notice that you also added a name attribute to the <input> tag and set it to "name", which makes sense for the hero's name. Any unique value will do, but using a descriptive name is helpful. Defining a name attribute is a requirement when using [([ngModel](#))] in combination with a form.

Internally, Angular creates [FormControl](#) instances and registers them with an [NgForm](#) directive that Angular attached to the <form> tag. Each [FormControl](#) is registered under the name you assigned to the name attribute. Read more in the previous section, [The NgForm directive](#).

Add similar [([ngModel](#))] bindings and name attributes to Alter Ego and Hero Power. You'll ditch the input box binding message and add a new binding (at the top) to the component's diagnostic property. Then you can confirm that two-way data binding works for the entire hero model.

After revision, the core of the form should look like this:

src/app/hero-form/hero-form.component.html (excerpt)

```

{{diagnostic}}
<div class="form-group">
  <label for="name">Name</label>
  <input type="text" class="form-control" id="name"
    required
    [(ngModel)]="model.name" name="name">
</div>

<div class="form-group">
  <label for="alterEgo">Alter Ego</label>
  <input type="text" class="form-control" id="alterEgo"
    [(ngModel)]="model.alterEgo" name="alterEgo">
</div>

<div class="form-group">
  <label for="power">Hero Power</label>
  <select class="form-control" id="power"
    required
    [(ngModel)]="model.power" name="power">
    <option *ngFor="let pow of powers" [value]="pow">{{pow}}</option>
  </select>
</div>

```

- Each input element has an id property that is used by the label element's for attribute to match the label to its input control.
- Each input element has a name property that is required by Angular forms to register the control with the form.

If you run the app now and change every hero model property, the form might display like this:

# Hero Form

```
{"id":18,"name":"Dr IQ 3000","power":"Super Flexible","alterEgo":"Chuck OverUnderStreet"}
```

## Name

Dr IQ 3000

## Alter Ego

Chuck OverUnderStreet

## Hero Power

Super Flexible ▾

The diagnostic near the top of the form confirms that all of your changes are reflected in the model.

Delete the {{diagnostic}} binding at the top as it has served its purpose.

## Track control state and validity with ngModel

Using [ngModel](#) in a form gives you more than just two-way data binding. It also tells you if the user touched the control, if the value changed, or if the value became invalid.

The NgModel directive doesn't just track state; it updates the control with special Angular CSS classes that reflect the state. You can leverage those class names to change the appearance of the control.

State	Class if true	Class if false
-------	---------------	----------------

The control has been visited.	ng-touched	ng-untouched
-------------------------------	------------	--------------

The control's value has changed.	ng-dirty	ng-pristine
----------------------------------	----------	-------------

The control's value is valid.	ng-valid	ng-invalid
-------------------------------	----------	------------

Temporarily add a [template reference variable](#) named spy to the Name <input> tag and use it to display the input's CSS classes.

src/app/hero-form/hero-form.component.html (excerpt)

```
<input type="text" class="form-control" id="name"
required
[(ngModel)]="model.name" name="name"
#spy>
<br>TODO: remove this: {{spy.className}}
```

Now run the app and look at the Name input box. Follow these steps precisely:

1. Look but don't touch.
2. Click inside the name box, then click outside it.
3. Add slashes to the end of the name.
4. Erase the name.

The actions and effects are as follows:

```
Dr IQ
TODO: remove this: form-control ng-untouched ng-pristine ng-valid
```

You should see the following transitions and class names:

Dr IQ	Untouched
Dr IQ	Touched
Dr IQ///	Changed
Dr IQ////	Invalid

The ng-valid/ng-invalid pair is the most interesting, because you want to send a strong visual signal when the values are invalid. You also want to mark required fields. To create such visual feedback, add definitions for the ng-\* CSS classes.

Delete the #spy template reference variable and the TODO as they have served their purpose.

## Add custom CSS for visual feedback

You can mark required fields and invalid data at the same time with a colored bar on the left of the input box:

Dr IQ

Valid + Required

Chuck Overstreet

Valid + Optional

Invalid (required | optional)

You achieve this effect by adding these class definitions to a new forms.css file that you add to the project as a sibling to index.html:

src/assets/forms.css

```
.ng-valid[required], .ng-valid.required {  
border-left: 5px solid #42A948; /* green */  
}  
  
.ng-invalid:not(form) {  
border-left: 5px solid #a94442; /* red */  
}
```

Update the <head> of index.html to include this style sheet:

src/index.html (styles)

```
<link rel="stylesheet" href="assets/forms.css">
```

## Show and hide validation error messages

You can improve the form. The Name input box is required and clearing it turns the bar red. That says something is wrong but the user doesn't know what is wrong or what to do about it. Leverage the control's state to reveal a helpful message.

When the user deletes the name, the form should look like this:

Name



The image shows a simple web form element. At the top, there is an input field with the placeholder text "Name". Below the input field, a red rectangular box contains the text "Name is required". This visual cue serves as a validation message, indicating that the input field is mandatory.

To achieve this effect, extend the <input> tag with the following:

- A [template reference variable](#).
- The "is required" message in a nearby <div>, which you'll display only if the control is invalid.

Here's an example of an error message added to the name input box:

```
src/app/hero-form/hero-form.component.html (excerpt)

<label for="name">Name</label>
<input type="text" class="form-control" id="name"
    required
    [(ngModel)]="model.name" name="name"
    #name="ngModel">
<div [hidden]="name.valid || name.pristine"
    class="alert alert-danger">
  Name is required
</div>
```

You need a template reference variable to access the input box's Angular control from within the template. Here you created a variable called name and gave it the value "ngModel".

Why "ngModel"? A directive's [exportAs](#) property tells Angular how to link the reference variable to the directive. You set name to [ngModel](#) because the [ngModel](#) directive's [exportAs](#) property happens to be "ngModel".

You control visibility of the name error message by binding properties of the name control to the message <div> element's hidden property.

```
src/app/hero-form/hero-form.component.html (hidden-error-msg)
```

```
<div [hidden]="name.valid || name.pristine"
    class="alert alert-danger">
```

In this example, you hide the message when the control is valid or pristine; "pristine" means the user hasn't changed the value since it was displayed in this form.

This user experience is the developer's choice. Some developers want the message to display at all times. If you ignore the pristine state, you would hide the message only when the value is valid. If you arrive in this component with a new (blank) hero or an invalid hero, you'll see the error message immediately, before you've done anything.

Some developers want the message to display only when the user makes an invalid change. Hiding the message while the control is "pristine" achieves that goal. You'll see the significance of this choice when you add a new hero to the form.

The hero Alter Ego is optional so you can leave that be.

Hero Power selection is required. You can add the same kind of error handling to the <select> if you want, but it's not imperative because the selection box already constrains the power to valid values.

Now you'll add a new hero in this form. Place a New Hero button at the bottom of the form and bind its click event to a newHero component method.

```
src/app/hero-form/hero-form.component.html (New Hero button)
```

```
<button type="button" class="btn btn-default" (click)="newHero()">New Hero</button>
src/app/hero-form/hero-form.component.ts (New Hero method)

newHero() {
  this.model = new Hero(42, " ", " ");
}
```

Run the application again, click the New Hero button, and the form clears. The required bars to the left of the input box are red, indicating invalid name and power properties. That's understandable as these are required fields. The error messages are hidden because the form is pristine; you haven't changed anything yet.

Enter a name and click New Hero again. The app displays a Name is required error message. You don't want error messages when you create a new (empty) hero. Why are you getting one now?

Inspecting the element in the browser tools reveals that the name input box is no longer pristine. The form remembers that you entered a name before clicking New Hero. Replacing the hero object did not restore the pristine state of the form controls.

You have to clear all of the flags imperatively, which you can do by calling the form's reset() method after calling the newHero() method.

```
src/app/hero-form/hero-form.component.html (Reset the form)
```

```
<button type="button" class="btn btn-default" (click)="newHero(); heroForm.reset()">New
Hero</button>
```

Now clicking "New Hero" resets both the form and its control flags.

## Submit the form with ngSubmit

The user should be able to submit this form after filling it in. The Submit button at the bottom of the form does nothing on its own, but it will trigger a form submit because of its type (type="submit").

A "form submit" is useless at the moment. To make it useful, bind the form's ngSubmit event property to the hero form component's onSubmit() method:

```
src/app/hero-form/hero-form.component.html (ngSubmit)
```

```
<form (ngSubmit)="onSubmit()" #heroForm="ngForm">
```

You'd already defined a template reference variable, #heroForm, and initialized it with the value "ngForm". Now, use that variable to access the form with the Submit button.

You'll bind the form's overall validity via the heroForm variable to the button's disabled property using an event binding. Here's the code:

```
src/app/hero-form/hero-form.component.html (submit-button)
```

```
<button type="submit" class="btn btn-success" [disabled]="!heroForm.form.valid">Submit</button>
```

If you run the application now, you find that the button is enabled—although it doesn't do anything useful yet.

Now if you delete the Name, you violate the "required" rule, which is duly noted in the error message. The Submit button is also disabled.

Not impressed? Think about it for a moment. What would you have to do to wire the button's enable/disabled state to the form's validity without Angular's help?

For you, it was as simple as this:

1. Define a template reference variable on the (enhanced) form element.
2. Refer to that variable in a button many lines away.

## Toggle two form regions (extra credit)

Submitting the form isn't terribly dramatic at the moment.

An unsurprising observation for a demo. To be honest, jazzing it up won't teach you anything new about forms. But this is an opportunity to exercise some of your newly won binding skills. If you aren't interested, skip to this page's conclusion.

For a more strikingly visual effect, hide the data entry area and display something else.

Wrap the form in a `<div>` and bind its `hidden` property to the `HeroFormComponent.submitted` property.

`src/app/hero-form/hero-form.component.html (excerpt)`

```
<div [hidden]="submitted">
  <h1>Hero Form</h1>
  <form (ngSubmit)="onSubmit()" #heroForm="ngForm">

    <!-- ... all of the form ... -->

  </form>
</div>
```

The main form is visible from the start because the `submitted` property is false until you submit the form, as this fragment from the `HeroFormComponent` shows:

`src/app/hero-form/hero-form.component.ts (submitted)`

```
submitted = false;
```

```
onSubmit() { this.submitted = true; }
```

When you click the Submit button, the submitted flag becomes true and the form disappears as planned.

Now the app needs to show something else while the form is in the submitted state. Add the following HTML below the `<div>` wrapper you just wrote:

src/app/hero-form/hero-form.component.html (excerpt)

```
<div [hidden]="!submitted">
  <h2>You submitted the following:</h2>
  <div class="row">
    <div class="col-xs-3">Name</div>
    <div class="col-xs-9 pull-left">{{ model.name }}</div>
  </div>
  <div class="row">
    <div class="col-xs-3">Alter Ego</div>
    <div class="col-xs-9 pull-left">{{ model.alterEgo }}</div>
  </div>
  <div class="row">
    <div class="col-xs-3">Power</div>
    <div class="col-xs-9 pull-left">{{ model.power }}</div>
  </div>
  <br>
  <button class="btn btn-primary" (click)="submitted=false">Edit</button>
</div>
```

There's the hero again, displayed read-only with interpolation bindings. This `<div>` appears only while the component is in the submitted state.

The HTML includes an Edit button whose click event is bound to an expression that clears the submitted flag.

When you click the Edit button, this block disappears and the editable form reappears.

## Summary

The Angular form discussed in this page takes advantage of the following framework features to provide support for data modification, validation, and more:

- An Angular HTML form template.
- A form component class with a `@Component` decorator.
- Handling form submission by binding to the `NgForm.ngSubmit` event property.
- Template-reference variables such as `#heroForm` and `#name`.
- `[(ngModel)]` syntax for two-way data binding.
- The use of name attributes for validation and form-element change tracking.
- The reference variable's `valid` property on input controls to check if a control is valid and show/hide error messages.

- Controlling the Submit button's enabled state by binding to [NgForm](#) validity.
- Custom CSS classes that provide visual feedback to users about invalid controls.

Here's the code for the final version of the application:

hero-form/hero-form.component.ts

hero-form/hero-form.component.html

hero.ts

app.module.ts

app.component.html

app.component.ts

main.ts

forms.css

```

1. import { Component } from '@angular/core';
2.
3. import { Hero } from './hero';
4.
5. @Component({
6.   selector: 'app-hero-form',
7.   templateUrl: './hero-form.component.html',
8.   styleUrls: ['./hero-form.component.css']
9. })
10.export class HeroFormComponent {
11.
12. powers = ['Really Smart', 'Super Flexible',
13.           'Super Hot', 'Weather Changer'];
14.
15. model = new Hero(18, 'Dr IQ', this.powers[0], 'Chuck Overstreet');
16.
17. submitted = false;
18.
19. onSubmit() { this.submitted = true; }
20.
21. newHero() {
22.   this.model = new Hero(42, "", "");
23. }
24.}
```

Improve overall data quality by validating user input for accuracy and completeness.

This page shows how to validate user input in the UI and display useful validation messages using both reactive and template-driven forms. It assumes some basic knowledge of the two forms modules.

If you're new to forms, start by reviewing the [Forms](#) and [Reactive Forms](#) guides.

## Template-driven validation

To add validation to a template-driven form, you add the same validation attributes as you would with [native HTML form 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, which results in an INVALID status, or null, which results in a VALID status.

You can then inspect the control's state by exporting `ngModel` to a local template variable.

The following example exports `NgModel` into a variable called `name`:

template/hero-form-template.component.html (name)

```
<input id="name" name="name" class="form-control"
  required minlength="4" appForbiddenName="bob"
  [(ngModel)]="hero.name" #name="ngModel" >

<div *ngIf="name.invalid && (name.dirty || name.touched)"
  class="alert alert-danger">

  <div *ngIf="name.errors.required">
    Name is required.
  </div>
  <div *ngIf="name.errors.minlength">
    Name must be at least 4 characters long.
  </div>
  <div *ngIf="name.errors.forbiddenName">
    Name cannot be Bob.
  </div>

</div>
```

Note the following:

- The `<input>` element carries the HTML validation attributes: `required` and `minlength`. It also carries a custom validator directive, `appForbiddenName`. For more information, see [Custom validators](#) section.
- `#name="ngModel"` exports `NgModel` into a local variable called `name`. `NgModel` mirrors many of the properties of its underlying `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](#) API reference.

- The `*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](#), and [forbiddenName](#).

### Why check dirty and touched?

You may not want your application to display errors before the user has a chance to edit the form. The checks for dirty and touched prevent errors from showing until the user does one of two things: changes the value, turning the control dirty; or blurs the form control element, setting the control to touched.

## Reactive form validation

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

There are two types of validator functions: sync validators and async validators.

- Sync validators: 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](#).
- Async validators: 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](#).

Note: for performance reasons, Angular only runs async validators if all sync validators pass. Each must complete before errors are set.

### Built-in validators

You can choose to [write your own validator functions](#), 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](#), are all available to use as functions from the [Validators](#) class. For a full list of built-in validators, see the [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. See below:

```
reactive/hero-form-reactive.component.ts (validator functions)
```

```

ngOnInit(): void {
  this.heroForm = new FormGroup({
    'name': new FormControl(this.hero.name, [
      Validators.required,
      Validators.minLength(4),
      forbiddenNameValidator(/bob/i) // <-- Here's how you pass in the custom validator.
    ]),
    'alterEgo': new FormControl(this.hero.alterEgo),
    'power': new FormControl(this.hero.power, Validators.required)
  });
}

}

```

get name() { return this.heroForm.get('name'); }

get power() { return this.heroForm.get('power'); }

Note that:

- The name control sets up two built-in validators—Validators.required and Validators.minLength(4)—and one custom validator, forbiddenNameValidator. For more details see the [Custom validators](#) section in this guide.
- As these validators are all sync validators, you pass them in as the second argument.
- Support multiple validators by passing the functions in as an array.
- This example adds a few getter methods. In a reactive form, you can always access any form control through the getmethod on its parent group, but sometimes it's useful to define getters as shorthands 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)

```

<input id="name" class="form-control"
  formControlName="name" required >

<div *ngIf="name.invalid && (name.dirty || name.touched)"
  class="alert alert-danger">

  <div *ngIf="name.errors.required">
    Name is required.
  </div>
  <div *ngIf="name.errors.minlength">
    Name must be at least 4 characters long.
  </div>

```

```
<div *ngIf="name.errors.forbiddenName">
  Name cannot be Bob.
</div>
</div>
```

Key takeaways:

- The form no longer exports any directives, and instead uses the name getter defined in the component class.
- The required attribute is still present. While it's not necessary for validation purposes, you may want to keep it in your template for CSS styling or accessibility reasons.

## Custom validators

Since the built-in validators won't always match the exact use case of your application, sometimes you'll want to create a custom validator.

Consider the `forbiddenNameValidator` function from previous [examples](#) in this guide. Here's what the definition of that function looks like:

shared/forbidden-name.directive.ts (forbiddenNameValidator)

```
/** A hero's name can't match the given regular expression */
export function forbiddenNameValidator(nameRe: RegExp): ValidatorFn {
  return (control: AbstractControl): {[key: string]: any} | null => {
    const forbidden = nameRe.test(control.value);
    return forbidden ? {'forbiddenName': {value: control.value}} : null;
  };
}
```

The function is actually 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 to reactive forms

In reactive forms, custom validators are fairly simple to add. All you have to do is pass the function directly to the [FormControl](#).

reactive/hero-form-reactive.component.ts (validator functions)

```
this.heroForm = new FormGroup({
  'name': new FormControl(this.hero.name, [
    Validators.required,
    Validators.minLength(4),
    forbiddenNameValidator(/bob/i) // <-- Here's how you pass in the custom validator.
  ]),
  'alterEgo': new FormControl(this.hero.alterEgo),
  'power': new FormControl(this.hero.power, Validators.required)
});
```

## Adding to template-driven forms

In template-driven forms, you don't have direct access to the [FormControl](#) instance, so you can't pass the validator in like you can for reactive forms. Instead, you need to add a directive to the template.

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](#) provider, a provider with an extensible collection of validators.

shared/forbidden-name.directive.ts (providers)

```
providers: [{provide: NG_VALIDATORS, useExisting: ForbiddenValidatorDirective, multi: true}]
```

The directive class then implements the [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)

1. `@Directive({`
2.  `selector: '[appForbiddenName]',`
3.  `providers: [{provide: NG_VALIDATORS, useExisting: ForbiddenValidatorDirective, multi: true}]`
4. `})`
5. `export class ForbiddenValidatorDirective implements Validator {`
6.  `@Input('appForbiddenName') forbiddenName: string;`
7.
8.  `validate(control: AbstractControl): {[key: string]: any} | null {`

```
9.      return this.forbiddenName ? forbiddenNameValidator(new RegExp(this.forbiddenName, 'i'))(control)
10.     : null;
11. }
12.}
```

Once the `ForbiddenValidatorDirective` is ready, you can simply add its selector, `appForbiddenName`, to any input element to activate it. For example:

template/hero-form-template.component.html (forbidden-name-input)

```
<input id="name" name="name" class="form-control"
  required minlength="4" appForbiddenName="bob"
  [(ngModel)]="hero.name" #name="ngModel" >
```

You may have noticed 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

Like in AngularJS, 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`

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)

```
.ng-valid[required], .ng-valid.required {
  border-left: 5px solid #42A948; /* green */
}
```

```
.ng-invalid:not(form) {
```

```
border-left: 5px solid #a94442; /* red */  
}
```

## Cross field validation

This section shows how to perform cross field validation. It assumes some basic knowledge of creating custom validators.

If you haven't created custom validators before, start by reviewing the [custom validators section](#).

In the following section, we will make sure that our heroes do not reveal their true identities by filling out the Hero Form. We will do that by validating that the hero names and alter egos do not match.

### Adding to reactive forms

The form has the following structure:

```
const heroForm = new FormGroup({  
  'name': new FormControl(),  
  'alterEgo': new FormControl(),  
  'power': new FormControl()  
});
```

Notice that the name and alterEgo are sibling controls. To evaluate both controls in a single custom validator, we should perform the validation in a common ancestor control: the [FormGroup](#). That way, we can query the [FormGroup](#) for the child controls which will allow us to compare their values.

To add a validator to the [FormGroup](#), pass the new validator in as the second argument on creation.

```
const heroForm = new FormGroup({  
  'name': new FormControl(),  
  'alterEgo': new FormControl(),  
  'power': new FormControl()  
}, { validators: identityRevealedValidator });
```

The validator code is as follows:

shared/identity-revealed.directive.ts

```
/** A hero's name can't match the hero's alter ego */  
export const identityRevealedValidator: ValidatorFn = (control: FormGroup): 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](#) interface. It takes an Angular control object as an argument and returns either null if the form is valid, or [ValidationErrors](#) otherwise.

First we retrieve the child controls by calling the [FormGroup](#)'s [get](#) method. Then we simply compare the values of the name and alterEgo controls.

If the values do not match, the hero's identity remains secret, and we can safely return null. Otherwise, the hero's identity is revealed and we must mark the form as invalid by returning an error object.

Next, to provide better user experience, we show an appropriate error message when the form is invalid.

reactive/hero-form-template.component.html

```
content_copy<div *ngIf="heroForm.errors?.identityRevealed && (heroForm.touched ||  
heroForm.dirty)" class="cross-validation-error-message alert alert-danger">
```

Name cannot match alter ego.

```
</div>
```

Note that we check if:

- the [FormGroup](#) has the cross validation error returned by the [identityRevealed](#) validator,
- the user is yet to [interact](#) with the form.

## Adding to template driven forms

First we must create a directive that will wrap the validator function. We provide it as the validator using the [NG\\_VALIDATORS](#) token. If you are not sure why, or you do not fully understand the syntax, revisit the previous [section](#).

shared/identity-revealed.directive.ts

```
@Directive({  
  selector: '[appIdentityRevealed]',  
  providers: [{ provide: NG_VALIDATORS, useExisting: IdentityRevealedValidatorDirective, multi: true }]  
})  
export class IdentityRevealedValidatorDirective implements Validator {  
  validate(control: AbstractControl): ValidationErrors {  
    return identityRevealedValidator(control)  
  }  
}
```

Next, we have to add the directive to the html template. Since the validator must be registered at the highest level in the form, we put the directive on the form tag.

template/hero-form-template.component.html

```
content_copy<form #heroForm="ngForm" applIdentityRevealed>
```

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="heroForm.errors?.identityRevealed && (heroForm.touched || heroForm.dirty)" class="cross-validation-error-message alert alert-danger">
```

```
    Name cannot match alter ego.  
</div>
```

Note that we check if:

- the form has the cross validation error returned by the identityRevealed validator,
- the user is yet to [interact](#) with the form.

This completes the cross validation example. We managed to:

- validate the form based on the values of two sibling controls,
- show a descriptive error message after the user interacted with the form and the validation failed.

## Async Validation

This section shows how to create asynchronous validators. It assumes some basic knowledge of creating [custom validators](#).

### The Basics

Just like synchronous validators have the [ValidatorFn](#) and [Validator](#) interfaces, asynchronous validators have their own counterparts: [AsyncValidatorFn](#) and [AsyncValidator](#).

They are very similar with the only difference being:

- They 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.

It is important to note that the 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 more basic validation methods fail.

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.

A common UI pattern is to show a spinner while the async validation is being performed. The following example presents how to achieve this with template-driven forms:

```
<input [(ngModel)]="name" #model="ngModel" appSomeAsyncValidator>
<app-spinner *ngIf="model.pending"></app-spinner>
```

## Implementing Custom Async Validator

In the following section, validation is performed asynchronously to ensure that our heroes pick an alter ego that is not already taken. New heroes are constantly enlisting and old heroes are leaving the service. That means that we do not have the list of available alter egos ahead of time.

To validate the potential alter ego, we need to consult a central database of all currently enlisted heroes. The process is asynchronous, so we need a special validator for that.

Let's start by creating the validator class.

```
@Injectable({ providedIn: 'root' })
export class UniqueAlterEgoValidator implements AsyncValidator {
  constructor(private heroesService: HeroesService) {}

  validate(
    ctrl: AbstractControl
  ): Promise<ValidationErrors | null> | Observable<ValidationErrors | null> {
    return this.heroesService.isAlterEgoTaken(ctrl.value).pipe(
      map(isTaken => (isTaken ? { uniqueAlterEgo: true } : null)),
      catchError(() => null)
    );
  }
}
```

As you can see, the UniqueAlterEgoValidator class implements the [AsyncValidator](#) interface. In the constructor, we inject the HeroesService that has the following interface:

```
interface HeroesService {
  isAlterEgoTaken: (alterEgo: string) => Observable<boolean>;
}
```

In a real world application, the HeroesService is 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 we 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. We pipe the response through the map operator and transform it into a validation result. As always, we return null if the form is valid, and [ValidationErrors](#) if it is not. We make sure to handle any potential errors with the catchError operator.

Here we decided that isAlterEgoTaken() error is treated 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 async validation is done. The pending flag is set to false, and the form validity is updated.

## Note on performance

By default, all validators are run after every form value change. With synchronous validators, this will not likely have a noticeable impact on application performance. However, it's common for async validators to 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.

We can delay updating the form validity by changing the updateOn property from change (default) to submit or blur.

With template-driven forms:

```
<input [(ngModel)]="name" [ngModelOptions]={updateOn: 'blur'}>
```

With reactive forms:

```
new FormControl('', {updateOn: 'blur'});
```

You can run the [live example](#) / [download example](#) to see the complete reactive and template-driven example code.

Building handcrafted forms can be costly and time-consuming, especially if you need a great number of them, they're similar to each other, and they change frequently to meet rapidly changing business and regulatory requirements.

It may be more economical to create the forms dynamically, based on metadata that describes the business object model.

This cookbook shows you how to use formGroup to dynamically render a simple form with different control types and validation. It's a primitive start. It might evolve to support a much

richer variety of questions, more graceful rendering, and superior user experience. All such greatness has humble beginnings.

The example in this cookbook is a dynamic form to build an online application experience for heroes seeking employment. The agency is constantly tinkering with the application process. You can create the forms on the fly without changing the application code.

See the [live example](#) / [download example](#).

## Bootstrap

Start by creating an [NgModule](#) called AppModule.

This cookbook uses [reactive forms](#).

Reactive forms belongs to a different [NgModule](#) called [ReactiveFormsModule](#), so in order to access any reactive forms directives, you have to import [ReactiveFormsModule](#) from the @angular/forms library.

Bootstrap the AppModule in main.ts.

app.module.ts

main.ts

```
1. import { BrowserModule }      from '@angular/platform-browser';
2. import { ReactiveFormsModule } from '@angular/forms';
3. import { NgModule }           from '@angular/core';
4.
5. import { AppComponent }        from './app.component';
6. import { DynamicFormComponent } from './dynamic-form.component';
7. import { DynamicFormQuestionComponent } from './dynamic-form-
   question.component';
8.
9. @NgModule({
10.   imports: [ BrowserModule, ReactiveFormsModule ],
11.   declarations: [ AppComponent, DynamicFormComponent, DynamicFormQuestionCo
   mponent ],
12.   bootstrap: [ AppComponent ]
13. })
14. export class AppModule {
15.   constructor() {
16.   }
17. }
```

## Question model

The next step is to define an object model that can describe all scenarios needed by the form functionality. The hero application process involves a form with a lot of questions. The question is the most fundamental object in the model.

The following QuestionBase is a fundamental question class.

src/app/question-base.ts

```
1. export class QuestionBase<T> {  
2.   value: T;  
3.   key: string;  
4.   label: string;  
5.   required: boolean;  
6.   order: number;  
7.   controlType: string;  
8.  
9.   constructor(options: {  
10.    value?: T,  
11.    key?: string,  
12.    label?: string,  
13.    required?: boolean,  
14.    order?: number,  
15.    controlType?: string  
16.  } = {}) {  
17.   this.value = options.value;  
18.   this.key = options.key || " ";  
19.   this.label = options.label || " ";  
20.   this.required = !!options.required;  
21.   this.order = options.order === undefined ? 1 : options.order;  
22.   this.controlType = options.controlType || " ";  
23. }  
24.}
```

From this base you can derive two new classes in TextboxQuestion and DropdownQuestion that represent textbox and dropdown questions. The idea is that the form will be bound to specific question types and render the appropriate controls dynamically.

TextboxQuestion supports multiple HTML5 types such as text, email, and url via the type property.

src/app/question-textbox.ts

```
import { QuestionBase } from './question-base';  
  
export class TextboxQuestion extends QuestionBase<string> {
```

```
controlType = 'textbox';
type: string;

constructor(options: {} = {}) {
  super(options);
  this.type = options['type'] || '';
}
```

DropdownQuestion presents a list of choices in a select box.

src/app/question-dropdown.ts

```
import { QuestionBase } from './question-base';

export class DropdownQuestion extends QuestionBase<string> {
  controlType = 'dropdown';
  options: {key: string, value: string}[] = [];

  constructor(options: {} = {}) {
    super(options);
    this.options = options['options'] || [];
  }
}
```

Next is QuestionControlService, a simple service for transforming the questions to a [FormGroup](#). In a nutshell, the form group consumes the metadata from the question model and allows you to specify default values and validation rules.

src/app/question-control.service.ts

```
import { Injectable } from '@angular/core';
import { FormControl, FormGroup, Validators } from '@angular/forms';

import { QuestionBase } from './question-base';

@Injectable()
export class QuestionControlService {
  constructor() { }

  toFormGroup(questions: QuestionBase<any>[]): FormGroup {
    let group: any = {};

    questions.forEach(question => {
      group[question.key] = question.required ? new FormControl(question.value || '', Validators.re
```

```

    quired)
        : new FormControl(question.value || "");
    });
    return new FormGroup(group);
}
}

```

## Question form components

Now that you have defined the complete model you are ready to create components to represent the dynamic form.

DynamicFormComponent is the entry point and the main container for the form.

dynamic-form.component.html

dynamic-form.component.ts

1. <div>
2. <form (ngSubmit)="onSubmit()" [formGroup]="form">
- 3.
4. <div \*ngFor="let question of questions" class="form-row">
5. <app-question [question]="question" [form]="form"></app-question>
6. </div>
- 7.
8. <div class="form-row">
9. <button type="submit" [disabled]="!form.valid">Save</button>
10. </div>
11. </form>
- 12.
13. <div \*ngIf="payLoad" class="form-row">
14. <strong>Saved the following values</strong><br>{{payLoad}}
15. </div>
- 16.</div>

It presents a list of questions, each bound to a <app-question> component element. The <app-question> tag matches the DynamicFormQuestionComponent, the component responsible for rendering the details of each individual question based on values in the data-bound question object.

dynamic-form-question.component.html

dynamic-form-question.component.ts

1. <div [formGroup]="form">
2. <label [attr.for]="question.key">{{question.label}}</label>
- 3.

```

4. <div [ngSwitch]="question.controlType">
5.
6.   <input *ngSwitchCase="textbox" [formControlName]="question.key"
7.     [id]="question.key" [type]="question.type">
8.
9.
  <select [id]="question.key" *ngSwitchCase="dropdown" [formControlName]="question
  .key">
10.   <option *ngFor="let opt of
    question.options" [value]="opt.key">{{opt.value}}</option>
11.   </select>
12.
13. </div>
14.
15. <div class="errorMessage" *ngIf="!isValid">{{question.label}} is required</div>
16.</div>
```

Notice this component can present any type of question in your model. You only have two types of questions at this point but you can imagine many more. The `ngSwitch` determines which type of question to display.

In both components you're relying on Angular's `formGroup` to connect the template HTML to the underlying control objects, populated from the question model with display and validation rules.

`formControlName` and `formGroup` are directives defined in [ReactiveFormsModule](#). The templates can access these directives directly since you imported [ReactiveFormsModule](#) from `AppModule`.

## Questionnaire data

`DynamicFormComponent` expects the list of questions in the form of an array bound to `@Input()` `questions`.

The set of questions you've defined for the job application is returned from the `QuestionService`. In a real app you'd retrieve these questions from storage.

The key point is that you control the hero job application questions entirely through the objects returned from `QuestionService`. Questionnaire maintenance is a simple matter of adding, updating, and removing objects from the `questions` array.

`src/app/question.service.ts`

```

1. import { Injectable }      from '@angular/core';
2.
3. import { DropdownQuestion } from './question-dropdown';
4. import { QuestionBase }   from './question-base';
```

```
5. import { TextBoxQuestion } from './question-textbox';
6.
7. @Injectable()
8. export class QuestionService {
9.
10. // TODO: get from a remote source of question metadata
11. // TODO: make asynchronous
12. getQuestions() {
13.
14.   let questions: QuestionBase<any>[] = [
15.
16.   new DropdownQuestion({
17.     key: 'brave',
18.     label: 'Bravery Rating',
19.     options: [
20.       {key: 'solid', value: 'Solid'},
21.       {key: 'great', value: 'Great'},
22.       {key: 'good', value: 'Good'},
23.       {key: 'unproven', value: 'Unproven'}
24.     ],
25.     order: 3
26.   }),
27.
28.   new TextBoxQuestion({
29.     key: 'firstName',
30.     label: 'First name',
31.     value: 'Bombasto',
32.     required: true,
33.     order: 1
34.   }),
35.
36.   new TextBoxQuestion({
37.     key: 'emailAddress',
38.     label: 'Email',
39.     type: 'email',
40.     order: 2
41.   })
42. ];
43.
44.   return questions.sort((a, b) => a.order - b.order);
45. }
46.}
```

Finally, display an instance of the form in the AppComponent shell.

app.component.ts

```
1. import { Component }      from '@angular/core';
2.
3. import { QuestionService } from './question.service';
4.
5. @Component({
6.   selector: 'app-root',
7.   template: `
8.     <div>
9.       <h2>Job Application for Heroes</h2>
10.      <app-dynamic-form [questions]="questions"></app-dynamic-form>
11.    </div>
12. `,
13.   providers: [QuestionService]
14. })
15.export class AppComponent {
16.   questions: any[];
17.
18. constructor(service: QuestionService) {
19.   this.questions = service.getQuestions();
20. }
21.}
```

## Dynamic Template

Although in this example you're modelling a job application for heroes, there are no references to any specific hero question outside the objects returned by QuestionService.

This is very important since it allows you to repurpose the components for any type of survey as long as it's compatible with the question object model. The key is the dynamic data binding of metadata used to render the form without making any hardcoded assumptions about specific questions. In addition to control metadata, you are also adding validation dynamically.

The Save button is disabled until the form is in a valid state. When the form is valid, you can click Save and the app renders the current form values as JSON. This proves that any user input is bound back to the data model. Saving and retrieving the data is an exercise for another time.

The final form looks like this:

Observables provide support for passing messages between publishers and subscribers in your application. Observables offer significant benefits over other techniques for event handling, asynchronous programming, and handling multiple values.

Observables are declarative—that is, you define a function for publishing values, but it is not executed until a consumer subscribes to it. The subscribed consumer then receives notifications until the function completes, or until they unsubscribe.

An observable can deliver multiple values of any type—literals, messages, or events, depending on the context. The API for receiving values is the same whether the values are delivered synchronously or asynchronously. Because setup and teardown logic are both handled by the observable, your application code only needs to worry about subscribing to consume values, and when done, unsubscribing. Whether the stream was keystrokes, an HTTP response, or an interval timer, the interface for listening to values and stopping listening is the same.

Because of these advantages, observables are used extensively within Angular, and are recommended for app development as well.

## Basic usage and terms

As a publisher, you create an Observable instance that defines a subscriber function. This is the function that is executed when a consumer calls the subscribe() method. The subscriber function defines how to obtain or generate values or messages to be published.

To execute the observable you have created and begin receiving notifications, you call its subscribe() method, passing an observer. This is a JavaScript object that defines the handlers for the notifications you receive. The subscribe() call returns a Subscription object that has an [unsubscribe\(\)](#) method, which you call to stop receiving notifications.

Here's an example that demonstrates the basic usage model by showing how an observable could be used to provide geolocation updates.

### Observe geolocation updates

```
1. // Create an Observable that will start listening to geolocation updates
2. // when a consumer subscribes.
3. const locations = new Observable((observer) => {
4.   // Get the next and error callbacks. These will be passed in when
5.   // the consumer subscribes.
6.   const {next, error} = observer;
7.   let watchId;
8.
9.   // Simple geolocation API check provides values to publish
10. if ('geolocation' in navigator) {
11.   watchId = navigator.geolocation.watchPosition(next, error);
12. } else {
13.   error('Geolocation not available');
14. }
15.
```

```

16. // When the consumer unsubscribes, clean up data ready for next subscription.
17. return {unsubscribe() { navigator.geolocation.clearWatch(watchId); }};
18.});
19.
20.// Call subscribe() to start listening for updates.
21.const locationsSubscription = locations.subscribe({
22. next(position) { console.log('Current Position: ', position); },
23. error(msg) { console.log('Error Getting Location: ', msg); }
24.});
25.
26.// Stop listening for location after 10 seconds
27.setTimeout(() => { locationsSubscription.unsubscribe(); }, 10000);

```

## Defining observers

A handler for receiving observable notifications implements the Observer interface. It is an object that defines callback methods to handle the three types of notifications that an observable can send:

NOTIFICATION TYPE	DESCRIPTION
next	Required. A handler for each delivered value. Called zero or more times after execution starts.
error	Optional. A handler for an error notification. An error halts execution of the observable instance.
complete	Optional. A handler for the execution-complete notification. Delayed values can continue to be delivered to the next handler after execution is complete.

An observer object can define any combination of these handlers. If you don't supply a handler for a notification type, the observer ignores notifications of that type.

## Subscribing

An Observable instance begins publishing values only when someone subscribes to it. You subscribe by calling the subscribe() method of the instance, passing an observer object to receive the notifications.

In order to show how subscribing works, we need to create a new observable. There is a constructor that you use to create new instances, but for illustration, we can use some methods from the RxJS library that create simple observables of frequently used types:

- `of(...items)`—Returns an Observable instance that synchronously delivers the values provided as arguments.
- `from(iterable)`—Converts its argument to an Observable instance. This method is commonly used to convert an array to an observable.

Here's an example of creating and subscribing to a simple observable, with an observer that logs the received message to the console:

#### Subscribe using observer

```

1. // Create simple observable that emits three values
2. const myObservable = of(1, 2, 3);
3.
4. // Create observer object
5. const myObserver = {
6.   next: x => console.log('Observer got a next value: ' + x),
7.   error: err => console.error('Observer got an error: ' + err),
8.   complete: () => console.log('Observer got a complete notification'),
9. };
10.
11.// Execute with the observer object
12.myObservable.subscribe(myObserver);
13.// Logs:
14.// Observer got a next value: 1
15.// Observer got a next value: 2
16.// Observer got a next value: 3
17.// Observer got a complete notification

```

Alternatively, the `subscribe()` method can accept callback function definitions in line, for `next`, `error`, and `complete` handlers. For example, the following `subscribe()` call is the same as the one that specifies the predefined observer:

#### Subscribe with positional arguments

```

myObservable.subscribe(
  x => console.log('Observer got a next value: ' + x),
  err => console.error('Observer got an error: ' + err),
  () => console.log('Observer got a complete notification')
);

```

In either case, a `next` handler is required. The `error` and `complete` handlers are optional.

Note that a `next()` function could receive, for instance, message strings, or event objects, numeric values, or structures, depending on context. As a general term, we refer to data published by an observable as a stream. Any type of value can be represented with an observable, and the values are published as a stream.

# Creating observables

Use the Observable constructor to create an observable stream of any type. The constructor takes as its argument the subscriber function to run when the observable's subscribe() method executes. A subscriber function receives an Observerobject, and can publish values to the observer's next() method.

For example, to create an observable equivalent to the of(1, 2, 3) above, you could do something like this:

Create observable with constructor

```
1. // This function runs when subscribe() is called
2. function sequenceSubscriber(observer) {
3.   // synchronously deliver 1, 2, and 3, then complete
4.   observer.next(1);
5.   observer.next(2);
6.   observer.next(3);
7.   observer.complete();
8.
9.   // unsubscribe function doesn't need to do anything in this
10. // because values are delivered synchronously
11. return {unsubscribe() {}};
12.}
13.
14.// Create a new Observable that will deliver the above sequence
15.const sequence = new Observable(sequenceSubscriber);
16.
17.// execute the Observable and print the result of each notification
18.sequence.subscribe({
19.  next(num) { console.log(num); },
20.  complete() { console.log('Finished sequence'); }
21.});
22.
23.// Logs:
24.// 1
25.// 2
26.// 3
27.// Finished sequence
```

To take this example a little further, we can create an observable that publishes events. In this example, the subscriber function is defined inline.

Create with custom fromEvent function

```
1. function fromEvent(target, eventName) {
```

```

2. return new Observable((observer) => {
3.   const handler = (e) => observer.next(e);
4.
5.   // Add the event handler to the target
6.   target.addEventListener(eventName, handler);
7.
8.   return () => {
9.     // Detach the event handler from the target
10.    target.removeEventListener(eventName, handler);
11.  };
12.});
13.}

```

Now you can use this function to create an observable that publishes keydown events:

Use custom fromEvent function

```

const ESC_KEY = 27;
const nameInput = document.getElementById('name') as HTMLInputElement;

const subscription = fromEvent(nameInput, 'keydown')
  .subscribe((e: KeyboardEvent) => {
    if (e.keyCode === ESC_KEY) {
      nameInput.value = '';
    }
  });

```

## Multicasting

A typical observable creates a new, independent execution for each subscribed observer. When an observer subscribes, the observable wires up an event handler and delivers values to that observer. When a second observer subscribes, the observable then wires up a new event handler and delivers values to that second observer in a separate execution.

Sometimes, instead of starting an independent execution for each subscriber, you want each subscription to get the same values—even if values have already started emitting. This might be the case with something like an observable of clicks on the document object.

Multicasting is the practice of broadcasting to a list of multiple subscribers in a single execution. With a multicasting observable, you don't register multiple listeners on the document, but instead re-use the first listener and send values out to each subscriber.

When creating an observable you should determine how you want that observable to be used and whether or not you want to multicast its values.

Let's look at an example that counts from 1 to 3, with a one-second delay after each number emitted.

## Create a delayed sequence

```
1. function sequenceSubscriber(observer) {  
2.   const seq = [1, 2, 3];  
3.   let timeoutId;  
4.  
5.   // Will run through an array of numbers, emitting one value  
6.   // per second until it gets to the end of the array.  
7.   function doSequence(arr, idx) {  
8.     timeoutId = setTimeout(() => {  
9.       observer.next(arr[idx]);  
10.      if (idx === arr.length - 1) {  
11.        observer.complete();  
12.      } else {  
13.        doSequence(arr, ++idx);  
14.      }  
15.    }, 1000);  
16.  }  
17.  
18. doSequence(seq, 0);  
19.  
20. // Unsubscribe should clear the timeout to stop execution  
21. return {unsubscribe() {  
22.   clearTimeout(timeoutId);  
23. }};  
24.  
25.  
26.// Create a new Observable that will deliver the above sequence  
27. const sequence = new Observable(sequenceSubscriber);  
28.  
29. sequence.subscribe({  
30.   next(num) { console.log(num); },  
31.   complete() { console.log('Finished sequence'); }  
32.});  
33.  
34.// Logs:  
35.// (at 1 second): 1  
36.// (at 2 seconds): 2  
37.// (at 3 seconds): 3  
38.// (at 3 seconds): Finished sequence
```

Notice that if you subscribe twice, there will be two separate streams, each emitting values every second. It looks something like this:

## Two subscriptions

```
1. // Subscribe starts the clock, and will emit after 1 second
2. sequence.subscribe({
3.   next(num) { console.log('1st subscribe: ' + num); },
4.   complete() { console.log('1st sequence finished.'); }
5. });
6.
7. // After 1/2 second, subscribe again.
8. setTimeout(() => {
9.   sequence.subscribe({
10.   next(num) { console.log('2nd subscribe: ' + num); },
11.   complete() { console.log('2nd sequence finished.'); }
12. });
13.}, 500);
14.
15.// Logs:
16.// (at 1 second): 1st subscribe: 1
17.// (at 1.5 seconds): 2nd subscribe: 1
18.// (at 2 seconds): 1st subscribe: 2
19.// (at 2.5 seconds): 2nd subscribe: 2
20.// (at 3 seconds): 1st subscribe: 3
21.// (at 3 seconds): 1st sequence finished
22.// (at 3.5 seconds): 2nd subscribe: 3
23.// (at 3.5 seconds): 2nd sequence finished
```

Changing the observable to be multicasting could look something like this:

### Create a multicast subscriber

```
1. function multicastSequenceSubscriber() {
2.   const seq = [1, 2, 3];
3.   // Keep track of each observer (one for every active subscription)
4.   const observers = [];
5.   // Still a single timeoutId because there will only ever be one
6.   // set of values being generated, multicasted to each subscriber
7.   let timeoutId;
8.
9.   // Return the subscriber function (runs when subscribe())
10. // function is invoked)
11. return (observer) => {
12.   observers.push(observer);
13.   // When this is the first subscription, start the sequence
14.   if (observers.length === 1) {
15.     timeoutId = doSequence({
```

```

16.     next(val) {
17.         // Iterate through observers and notify all subscriptions
18.         observers.forEach(obs => obs.next(val));
19.     },
20.     complete() {
21.         // Notify all complete callbacks
22.         observers.slice(0).forEach(obs => obs.complete());
23.     }
24. }, seq, 0);
25. }
26.
27. return {
28.     unsubscribe() {
29.         // Remove from the observers array so it's no longer notified
30.         observers.splice(observers.indexOf(observer), 1);
31.         // If there's no more listeners, do cleanup
32.         if (observers.length === 0) {
33.             clearTimeout(timeoutId);
34.         }
35.     }
36. };
37. };
38.}
39.
40.// Run through an array of numbers, emitting one value
41.// per second until it gets to the end of the array.
42.function doSequence(observer, arr, idx) {
43. return setTimeout(() => {
44.     observer.next(arr[idx]);
45.     if (idx === arr.length - 1) {
46.         observer.complete();
47.     } else {
48.         doSequence(observer, arr, ++idx);
49.     }
50. }, 1000);
51.}
52.
53.// Create a new Observable that will deliver the above sequence
54.const multicastSequence = new Observable(multicastSequenceSubscriber());
55.
56.// Subscribe starts the clock, and begins to emit after 1 second
57.multicastSequence.subscribe({

```

```

58. next(num) { console.log('1st subscribe: ' + num); },
59. complete() { console.log('1st sequence finished.'); }
60.});
61.
62.// After 1 1/2 seconds, subscribe again (should "miss" the first value).
63.setTimeout(() => {
64. multicastSequence.subscribe({
65. next(num) { console.log('2nd subscribe: ' + num); },
66. complete() { console.log('2nd sequence finished.'); }
67.});
68.}, 1500);
69.
70.// Logs:
71.// (at 1 second): 1st subscribe: 1
72.// (at 2 seconds): 1st subscribe: 2
73.// (at 2 seconds): 2nd subscribe: 2
74.// (at 3 seconds): 1st subscribe: 3
75.// (at 3 seconds): 1st sequence finished
76.// (at 3 seconds): 2nd subscribe: 3
77.// (at 3 seconds): 2nd sequence finished

```

Multicasting observables take a bit more setup, but they can be useful for certain applications. Later we will look at tools that simplify the process of multicasting, allowing you to take any observable and make it multicasting.

## Error handling

Because observables produce values asynchronously, try/catch will not effectively catch errors. Instead, you handle errors by specifying an error callback on the observer. Producing an error also causes the observable to clean up subscriptions and stop producing values. An observable can either produce values (calling the next callback), or it can complete, calling either the complete or error callback.

```
myObservable.subscribe({
  next(num) { console.log('Next num: ' + num)},
  error(err) { console.log('Received an error: ' + err)}
});
```

Error handling (and specifically recovering from an error) is covered in more detail in a later section.

Reactive programming is an asynchronous programming paradigm concerned with data streams and the propagation of change ([Wikipedia](#)). RxJS (Reactive Extensions for JavaScript) is a library for reactive programming using observables that makes it easier to compose asynchronous or callback-based code ([RxJS Docs](#)).

RxJS provides an implementation of the Observable type, which is needed until the type becomes part of the language and until browsers support it. The library also provides utility functions for creating and working with observables. These utility functions can be used for:

- Converting existing code for async operations into observables
- Iterating through the values in a stream
- Mapping values to different types
- Filtering streams
- Composing multiple streams

## Observable creation functions

RxJS offers a number of functions that can be used to create new observables. These functions can simplify the process of creating observables from things such as events, timers, promises, and so on. For example:

Create an observable from a promise

```
import { fromPromise } from 'rxjs';

// Create an Observable out of a promise
const data = fromPromise(fetch('/api/endpoint'));
// Subscribe to begin listening for async result
data.subscribe({
  next(response) { console.log(response); },
  error(err) { console.error('Error: ' + err); },
  complete() { console.log('Completed'); }
});
```

Create an observable from a counter

```
import { interval } from 'rxjs';

// Create an Observable that will publish a value on an interval
const secondsCounter = interval(1000);
// Subscribe to begin publishing values
secondsCounter.subscribe(n =>
  console.log(`It's been ${n} seconds since subscribing!`));
```

Create an observable from an event

1. import { fromEvent } from 'rxjs';
- 2.
3. const el = document.getElementById('my-element');
- 4.
5. // Create an Observable that will publish mouse movements
6. const mouseMoves = fromEvent(el, 'mousemove');

```
7.  
8. // Subscribe to start listening for mouse-move events  
9. const subscription = mouseMoves.subscribe((evt: MouseEvent) => {  
10. // Log coords of mouse movements  
11. console.log(`Coords: ${evt.clientX} X ${evt.clientY}`);  
12.  
13. // When the mouse is over the upper-left of the screen,  
14. // unsubscribe to stop listening for mouse movements  
15. if (evt.clientX < 40 && evt.clientY < 40) {  
16.   subscription.unsubscribe();  
17. }  
18.});
```

Create an observable that creates an AJAX request

```
import { ajax } from 'rxjs/ajax';  
  
// Create an Observable that will create an AJAX request  
const apiData = ajax('/api/data');  
// Subscribe to create the request  
apiData.subscribe(res => console.log(res.status, res.response));
```

## Operators

Operators are functions that build on the observables foundation to enable sophisticated manipulation of collections. For example, RxJS defines operators such as [map\(\)](#), [filter\(\)](#), concat(), and flatMap().

Operators take configuration options, and they return a function that takes a source observable. When executing this returned function, the operator observes the source observable's emitted values, transforms them, and returns a new observable of those transformed values. Here is a simple example:

Map operator

```
1. import { map } from 'rxjs/operators';  
2.  
3. const nums = of(1, 2, 3);  
4.  
5. const squareValues = map((val: number) => val * val);  
6. const squaredNums = squareValues(nums);  
7.  
8. squaredNums.subscribe(x => console.log(x));  
9.  
10.// Logs
```

```
11.// 1  
12.// 4  
13.// 9
```

You can use pipes to link operators together. Pipes let you combine multiple functions into a single function. The pipe() function takes as its arguments the functions you want to combine, and returns a new function that, when executed, runs the composed functions in sequence.

A set of operators applied to an observable is a recipe—that is, a set of instructions for producing the values you’re interested in. By itself, the recipe doesn’t do anything. You need to call subscribe() to produce a result through the recipe.

Here’s an example:

### Standalone pipe function

```
1. import { filter, map } from 'rxjs/operators';  
2.  
3. const nums = of(1, 2, 3, 4, 5);  
4.  
5. // Create a function that accepts an Observable.  
6. const squareOddVals = pipe(  
7.   filter((n: number) => n % 2 !== 0),  
8.   map(n => n * n)  
9. );  
10.  
11.// Create an Observable that will run the filter and map functions  
12.const squareOdd = squareOddVals(nums);  
13.  
14.// Suscribe to run the combined functions  
15.squareOdd.subscribe(x => console.log(x));
```

The pipe() function is also a method on the RxJS Observable, so you use this shorter form to define the same operation:

### Observable.pipe function

```
import { filter, map } from 'rxjs/operators';  
  
const squareOdd = of(1, 2, 3, 4, 5)  
.pipe(  
  filter(n => n % 2 !== 0),  
  map(n => n * n)  
);  
  
// Suscribe to get values  
squareOdd.subscribe(x => console.log(x));
```

## Common operators

RxJS provides many operators, but only a handful are used frequently. For a list of operators and usage samples, visit the [RxJS API Documentation](#).

Note that, for Angular apps, we prefer combining operators with pipes, rather than chaining. Chaining is used in many RxJS examples.

AREA	OPERATORS
Creation	from , fromPromise , fromEvent , of
Combination	combineLatest , concat , merge , startWith , withLatestFrom , zip
Filtering	debounceTime , distinctUntilChanged , filter , take , takeUntil
Transformation	bufferTime , concatMap , map , mergeMap , scan , switchMap
Utility	tap
Multicasting	share

## Error handling

In addition to the [error\(\)](#) handler that you provide on subscription, RxJS provides the `catchError` operator that lets you handle known errors in the observable recipe.

For instance, suppose you have an observable that makes an API request and maps to the response from the server. If the server returns an error or the value doesn't exist, an error is produced. If you catch this error and supply a default value, your stream continues to process values rather than erroring out.

Here's an example of using the `catchError` operator to do this:

### catchError operator

```
1. import { ajax } from 'rxjs/ajax';
2. import { map, catchError } from 'rxjs/operators';
3. // Return "response" from the API. If an error happens,
4. // return an empty array.
5. const apiData = ajax('/api/data').pipe(
6.   map(res => {
7.     if (!res.response) {
8.       throw new Error('Value expected!');
9.     }
10.    return res.response;
11.  })
12. )
```

```
11. }),
12. catchError(err => of([]))
13.);
14.
15.apiData.subscribe({
16. next(x) { console.log('data: ', x); },
17. error(err) { console.log('errors already caught... will not run'); }
18.});
```

## Retry failed observable

Where the catchError operator provides a simple path of recovery, the retry operator lets you retry a failed request.

Use the retry operator before the catchError operator. It resubscribes to the original source observable, which can then re-run the full sequence of actions that resulted in the error. If this includes an HTTP request, it will retry that HTTP request.

The following converts the previous example to retry the request before catching the error:

### retry operator

```
1. import { ajax } from 'rxjs/ajax';
2. import { map, retry, catchError } from 'rxjs/operators';
3.
4. const apiData = ajax('/api/data').pipe(
5.   retry(3), // Retry up to 3 times before failing
6.   map(res => {
7.     if (!res.response) {
8.       throw new Error('Value expected!');
9.     }
10.    return res.response;
11. }),
12. catchError(err => of([]))
13.);
14.
15.apiData.subscribe({
16. next(x) { console.log('data: ', x); },
17. error(err) { console.log('errors already caught... will not run'); }
18.});
```

Do not retry authentication requests, since these should only be initiated by user action. We don't want to lock out user accounts with repeated login requests that the user has not initiated.

## Naming conventions for observables

Because Angular applications are mostly written in TypeScript, you will typically know when a variable is an observable. Although the Angular framework does not enforce a naming convention for observables, you will often see observables named with a trailing “\$” sign.

This can be useful when scanning through code and looking for observable values. Also, if you want a property to store the most recent value from an observable, it can be convenient to simply use the same name with or without the “\$”.

For example:

Naming observables

```
1. import { Component } from '@angular/core';
2. import { Observable } from 'rxjs';
3.
4. @Component({
5.   selector: 'app-stopwatch',
6.   templateUrl: './stopwatch.component.html'
7. })
8. export class StopwatchComponent {
9.
10. stopwatchValue: number;
11. stopwatchValue$: Observable<number>;
12.
13. start() {
14.   this.stopwatchValue$.subscribe(num =>
15.     this.stopwatchValue = num
16.   );
17. }
18.}
```

Angular makes use of observables as an interface to handle a variety of common asynchronous operations. For example:

- The [EventEmitter](#) class extends Observable.
- The HTTP module uses observables to handle AJAX requests and responses.
- The Router and Forms modules use observables to listen for and respond to user-input events.

## Event emitter

Angular provides an [EventEmitter](#) class that is used when publishing values from a component through the [@Output\(\)](#)decorator. [EventEmitter](#) extends Observable, adding an

`emit()` method so it can send arbitrary values. When you call `emit()`, it passes the emitted value to the next() method of any subscribed observer.

A good example of usage can be found on the [EventEmitter](#) documentation. Here is the example component that listens for open and close events:

```
<zippy (open)="onOpen($event)" (close)="onClose($event)"></zippy>
```

Here is the component definition:

### EventEmitter

```
1. @Component({
2.   selector: 'zippy',
3.   template: `
4.     <div class="zippy">
5.       <div (click)="toggle()">Toggle</div>
6.       <div [hidden]="!visible">
7.         <ng-content></ng-content>
8.       </div>
9.     </div>`)
10.
11.export class ZippyComponent {
12. visible = true;
13. @Output() open = new EventEmitter<any>();
14. @Output() close = new EventEmitter<any>();
15.
16. toggle() {
17.   this.visible = !this.visible;
18.   if (this.visible) {
19.     this.open.emit(null);
20.   } else {
21.     this.close.emit(null);
22.   }
23. }
24.}
```

## HTTP

Angular's [HttpClient](#) returns observables from HTTP method calls. For instance, `http.get('/api')` returns an observable. This provides several advantages over promise-based HTTP APIs:

- Observables do not mutate the server response (as can occur through chained `.then()` calls on promises). Instead, you can use a series of operators to transform values as needed.

- HTTP requests are cancellable through the [unsubscribe\(\)](#) method.
- Requests can be configured to get progress event updates.
- Failed requests can be retried easily.

## Async pipe

The [AsyncPipe](#) subscribes to an observable or promise and returns the latest value it has emitted. When a new value is emitted, the pipe marks the component to be checked for changes.

The following example binds the time observable to the component's view. The observable continuously updates the view with the current time.

Using async pipe

```
@Component({
  selector: 'async-observable-pipe',
  template: `<div><code>observable|async</code>:
    Time: {{ time | async }}</div>`
})
export class AsyncObservablePipeComponent {
  time = new Observable(observer =>
    setInterval(() => observer.next(new Date().toString()), 1000)
  );
}
```

## Router

[Router.events](#) provides events as observables. You can use the [filter\(\)](#) operator from RxJS to look for events of interest, and subscribe to them in order to make decisions based on the sequence of events in the navigation process. Here's an example:

Router events

1. import { [Router](#), [NavigationStart](#) } from '@angular/router';
2. import { filter } from 'rxjs/operators';
- 3.
4. [@Component](#)({
5. selector: 'app-routable',
6. templateUrl: './routable.component.html',
7. [styleUrls](#): ['./routable.component.css']
8. })
9. export class Routable1Component implements [OnInit](#) {
- 10.
11. navStart: Observable<[NavigationStart](#)>;
- 12.

```

13. constructor(private router: Router) {
14.   // Create a new Observable that publishes only the NavigationStart event
15.   this.navStart = router.events.pipe(
16.     filter(evt => evt instanceof NavigationStart)
17.   ) as Observable<NavigationStart>;
18. }
19.
20. ngOnInit() {
21.   this.navStart.subscribe(evt => console.log('Navigation Started!'));
22. }
23.

```

The [ActivatedRoute](#) is an injected router service that makes use of observables to get information about a route path and parameters. For example, `ActivateRoute.url` contains an observable that reports the route path or paths. Here's an example:

### ActivatedRoute

```

1. import { ActivatedRoute } from '@angular/router';
2.
3. @Component({
4.   selector: 'app-routable',
5.   templateUrl: './routable.component.html',
6.   styleUrls: ['./routable.component.css']
7. })
8. export class Routable2Component implements OnInit {
9.   constructor(private activatedRoute: ActivatedRoute) {}
10.
11. ngOnInit() {
12.   this.activatedRoute.url
13.     .subscribe(url => console.log('The URL changed to: ' + url));
14. }
15.

```

## Reactive forms

Reactive forms have properties that use observables to monitor form control values. The [FormControl](#) properties `valueChanges` and `statusChanges` contain observables that raise change events. Subscribing to an observable form-control property is a way of triggering application logic within the component class. For example:

### Reactive forms

```

1. import { FormGroup } from '@angular/forms';
2.

```

```

3. @Component({
4.   selector: 'my-component',
5.   template: 'MyComponent Template'
6. })
7. export class MyComponent implements OnInit {
8.   nameChangeLog: string[] = [];
9.   heroForm: FormGroup;
10.
11. ngOnInit() {
12.   this.logNameChange();
13. }
14. logNameChange() {
15.   const nameControl = this.heroForm.get('name');
16.   nameControl.valueChanges.forEach(
17.     (value: string) => this.nameChangeLog.push(value)
18. );
19. }
20.}

```

Here are some examples of domains in which observables are particularly useful.

## Type-ahead suggestions

Observables can simplify the implementation of type-ahead suggestions. Typically, a type-ahead has to do a series of separate tasks:

- Listen for data from an input.
- Trim the value (remove whitespace) and make sure it's a minimum length.
- Debounce (so as not to send off API requests for every keystroke, but instead wait for a break in keystrokes).
- Don't send a request if the value stays the same (rapidly hit a character, then backspace, for instance).
- Cancel ongoing AJAX requests if their results will be invalidated by the updated results.

Writing this in full JavaScript can be quite involved. With observables, you can use a simple series of RxJS operators:

### Typeahead

```

1. import { fromEvent } from 'rxjs';
2. import { ajax } from 'rxjs/ajax';
3. import { map, filter, debounceTime, distinctUntilChanged, switchMap } from
   'rxjs/operators';
4.
5. const searchBox = document.getElementById('search-box');

```

```
6.  
7. const typeahead = fromEvent(searchBox, 'input').pipe(  
8.   map((e: KeyboardEvent) => e.target.value),  
9.   filter(text => text.length > 2),  
10.  debounceTime(10),  
11.  distinctUntilChanged(),  
12.  switchMap(() => ajax('/api/endpoint'))  
13.);  
14.  
15.typeahead.subscribe(data => {  
16.// Handle the data from the API  
17.});
```

## Exponential backoff

Exponential backoff is a technique in which you retry an API after failure, making the time in between retries longer after each consecutive failure, with a maximum number of retries after which the request is considered to have failed. This can be quite complex to implement with promises and other methods of tracking AJAX calls. With observables, it is very easy:

### Exponential backoff

```
1. import { pipe, range, timer, zip } from 'rxjs';  
2. import { ajax } from 'rxjs/ajax';  
3. import { retryWhen, map, mergeMap } from 'rxjs/operators';  
4.  
5. function backoff(maxTries, ms) {  
6.   return pipe(  
7.     retryWhen(attempts => range(1, maxTries)  
8.       .pipe(  
9.         zip(attempts, (i) => i),  
10.        map(i => i * i),  
11.        mergeMap(i => timer(i * ms))  
12.      )  
13.    )  
14.);  
15.}  
16.  
17.ajax('/api/endpoint')  
18. .pipe(backoff(3, 250))  
19. .subscribe(data => handleData(data));  
20.  
21.function handleData(data) {  
22. // ...
```

23.}

You can often use observables instead of promises to deliver values asynchronously. Similarly, observables can take the place of event handlers. Finally, because observables deliver multiple values, you can use them where you might otherwise build and operate on arrays.

Observables behave somewhat differently from the alternative techniques in each of these situations, but offer some significant advantages. Here are detailed comparisons of the differences.

## Observables compared to promises

Observables are often compared to promises. Here are some key differences:

- Observables are declarative; computation does not start until subscription. Promises execute immediately on creation. This makes observables useful for defining recipes that can be run whenever you need the result.
- Observables provide many values. Promises provide one. This makes observables useful for getting multiple values over time.
- Observables differentiate between chaining and subscription. Promises only have `.then()` clauses. This makes observables useful for creating complex transformation recipes to be used by other part of the system, without causing the work to be executed.
- Observables `subscribe()` is responsible for handling errors. Promises push errors to the child promises. This makes observables useful for centralized and predictable error handling.

## Creation and subscription

- Observables are not executed until a consumer subscribes. The `subscribe()` executes the defined behavior once, and it can be called again. Each subscription has its own computation. Resubscription causes recomputation of values.

```
// declare a publishing operation
new Observable((observer) => { subscriber_fn });
// initiate execution
observable.subscribe(() => {
    // observer handles notifications
});
```

- Promises execute immediately, and just once. The computation of the result is initiated when the promise is created. There is no way to restart work. All `then` clauses (subscriptions) share the same computation.

```
// initiate execution
new Promise(\(resolve, reject\) => { executer_fn });
```

```
// handle return value
promise.then((value) => {
  // handle result here
});
```

## Chaining

- Observables differentiate between transformation function such as a map and subscription. Only subscription activates the subscriber function to start computing the values.

```
observable.map((v) => 2*v);
```

- Promises do not differentiate between the last .then clauses (equivalent to subscription) and intermediate .then clauses (equivalent to map).

```
promise.then((v) => 2*v);
```

## Cancellation

- Observable subscriptions are cancellable. Unsubscribing removes the listener from receiving further values, and notifies the subscriber function to cancel work.

```
const sub = obs.subscribe(...);
sub.unsubscribe();
```

- Promises are not cancellable.

## Error handling

- Observable execution errors are delivered to the subscriber's error handler, and the subscriber automatically unsubscribes from the observable.

```
obs.subscribe(() => {
  throw Error('my error');
});
```

- Promises push errors to the child promises.

```
promise.then(() => {
  throw Error('my error');
});
```

## Cheat sheet

The following code snippets illustrate how the same kind of operation is defined using observables and promises.

OPERATION OBSERVABLE

PROMISE

Creation	<pre>new Observable((observer) =&gt; { new Promise((resolve, reject) =&gt; {     observer.next(123);     resolve(123); });});</pre>
Transform	<pre>obs.map((value) =&gt; value * 2); promise.then((value) =&gt; value * 2);</pre>
Subscribe	<pre>sub = obs.subscribe((value) =&gt; { promise.then((value) =&gt; {     console.log(value) });});</pre>
Unsubscribe	<pre>sub.unsubscribe();</pre> Implied by promise resolution.

## Observables compared to events API

Observables are very similar to event handlers that use the events API. Both techniques define notification handlers, and use them to process multiple values delivered over time. Subscribing to an observable is equivalent to adding an event listener. One significant difference is that you can configure an observable to transform an event before passing the event to the handler.

Using observables to handle events and asynchronous operations can have the advantage of greater consistency in contexts such as HTTP requests.

Here are some code samples that illustrate how the same kind of operation is defined using observables and the events API.

	Observable	Events API
Creation & cancellation	<pre>// Setup let clicks\$ = fromEvent(buttonEl, 'click'); // Begin listening let subscription = clicks\$ .subscribe(e =&gt; console.log('Clicked', e)) // Stop listening subscription.unsubscribe();</pre>	<pre>function handler(e) {   console.log('Clicked', e); }  // Setup &amp; begin listening button.addEventListener('click', handler); // Stop listening button.removeEventListener('click', handler);</pre>
Subscription	<pre>observable.subscribe(() =&gt; {   // notification handlers here });</pre>	<pre>element.addEventListener(eventName, (event) =&gt; {   // notification handler here });</pre>

	Listen for keystrokes, but provide a stream representing the value in the input.	Does not support configuration.
Configuration	<pre>fromEvent(inputEl, 'keydown').pipe(   map(e =&gt; e.target.value) );</pre>	<pre>element.addEventListener(eventName, (event) =&gt; {   // Cannot change the passed Event into another   // value before it gets to the handler });</pre>

## Observables compared to arrays

An observable produces values over time. An array is created as a static set of values. In a sense, observables are asynchronous where arrays are synchronous. In the following examples, → implies asynchronous value delivery.

	Observable	Array
Given	obs: →1→2→3→5→7	arr: [1, 2, 3, 5, 7]
	obsB: →'a'→'b'→'c'	arrB: ['a', 'b', 'c']
concat()	obs.concat(obsB) →1→2→3→5→7→'a'→'b'→'c'	arr.concat(arrB) [1,2,3,5,7,'a','b','c']
filter()	obs.filter((v) => v>3) →5→7	arr.filter((v) => v>3) [5, 7]
find()	obs.find((v) => v>3) →5	arr.find((v) => v>3) 5
findIndex()	obs.findIndex((v) => v>3) →3	arr.findIndex((v) => v>3) 3

	obs.forEach((v) => { console.log(v); })	arr.forEach((v) => { console.log(v); })
forEach()	1 2 3 5 7	1 2 3 5 7
map()	obs.map((v) => -v) →-1→-2→-3→-5→-7	arr.map((v) => -v) [-1, -2, -3, -5, -7]
reduce()	obs.scan((s,v)=> s+v, 0) →1→3→6→11→18	arr.reduce((s,v) => s+v, 0) 18

## Prerequisites

A basic understanding of the following:

- [JavaScript Modules vs. NgModules.](#)
- 

An NgModule describes how the application parts fit together. Every application has at least one Angular module, the rootmodule that you bootstrap to launch the application. By convention, it is usually called AppModule.

If you use the CLI to generate an app, the default AppModule is as follows:

```

1. /* JavaScript imports */
2. import { BrowserModule } from '@angular/platform-browser';
3. import { NgModule } from '@angular/core';
4. import { FormsModule } from '@angular/forms';
5. import { HttpClientModule } from '@angular/common/http';
6.
7. import { AppComponent } from './app.component';
8.
9. /* the AppModule class with the @NgModule decorator */
10.@NgModule({
11. declarations: [
12. AppComponent
13. ],
14. imports: [
15. BrowserModule,
16. FormsModule,
```

```
17. HttpClientModule  
18. ],  
19. providers: [],  
20. bootstrap: [AppComponent]  
21.})  
22.export class AppModule { }
```

After the import statements is a class with the [@NgModule](#) decorator.

The [@NgModule](#) decorator identifies AppModule as an [NgModule](#) class. [@NgModule](#) takes a metadata object that tells Angular how to compile and launch the application.

- declarations—this application's lone component.
- imports—import [BrowserModule](#) to have browser specific services such as DOM rendering, sanitization, and location.
- providers—the service providers.
- bootstrap—the root component that Angular creates and inserts into the index.html host web page.

The default CLI application only has one component, AppComponent, so it is in both the [declarations](#) and the [bootstrap](#) arrays.

## The [declarations](#) array

The module's [declarations](#) array tells Angular which components belong to that module. As you create more components, add them to [declarations](#).

You must declare every component in exactly one [NgModule](#) class. If you use a component without declaring it, Angular returns an error message.

The [declarations](#) array only takes declarables. Declarables are components, [directives](#) and [pipes](#). All of a module's declarables must be in the [declarations](#) array. Declarables must belong to exactly one module. The compiler emits an error if you try to declare the same class in more than one module.

These declared classes are visible within the module but invisible to components in a different module unless they are exported from this module and the other module imports this one.

An example of what goes into a declarations array follows:

```
declarations: [  
  YourComponent,  
  YourPipe,  
  YourDirective  
,
```

A declarable can only belong to one module, so only declare it in one [@NgModule](#). When you need it elsewhere, import the module that has the declarable you need in it.

Only [@NgModule](#) references go in the [imports](#) array.

## Using directives with [@NgModule](#)

Use the [declarations](#) array for directives. To use a directive, component, or pipe in a module, you must do a few things:

1. Export it from the file where you wrote it.
2. Import it into the appropriate module.
3. Declare it in the [@NgModule declarations](#) array.

Those three steps look like the following. In the file where you create your directive, export it. The following example, named ItemDirective is the default directive structure that the CLI generates in its own file, item.directive.ts:

```
src/app/item.directive.ts
```

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

```
@Directive({
  selector: '[appItem]'
})
export class ItemDirective {
  // code goes here
  constructor() { }

}
```

The key point here is that you have to export it so you can import it elsewhere. Next, import it into the NgModule, in this example app.module.ts, with a JavaScript import statement:

```
src/app/app.module.ts
```

```
import { ItemDirective } from './item.directive';
```

And in the same file, add it to the [@NgModule declarations](#) array:

```
src/app/app.module.ts
```

```
declarations: [
  AppComponent,
  ItemDirective
],
```

Now you could use your ItemDirective in a component. This example uses AppModule, but you'd do it the same way for a feature module. For more about directives, see [Attribute Directives](#) and [Structural Directives](#). You'd also use the same technique for [pipes](#) and components.

Remember, components, directives, and pipes belong to one module only. You only need to declare them once in your app because you share them by importing the necessary modules. This saves you time and helps keep your app lean.

## The [imports](#) array

The module's [imports](#) array appears exclusively in the `@NgModule` metadata object. It tells Angular about other NgModules that this particular module needs to function properly.

This list of modules are those that export components, directives, or pipes that the component templates in this module reference. In this case, the component is `AppComponent`, which references components, directives, or pipes in [BrowserModule](#), [FormsModule](#), or [HttpClientModule](#). A component template can reference another component, directive, or pipe when the referenced class is declared in this module or the class was imported from another module.

## The [providers](#) array

The providers array is where you list the services the app needs. When you list services here, they are available app-wide. You can scope them when using feature modules and lazy loading. For more information, see [Providers](#).

## The [bootstrap](#) array

The application launches by bootstrapping the root `AppModule`, which is also referred to as an `entryComponent`. Among other things, the bootstrapping process creates the component(s) listed in the [bootstrap](#) array and inserts each one into the browser DOM.

Each bootstrapped component is the base of its own tree of components. Inserting a bootstrapped component usually triggers a cascade of component creations that fill out that tree.

While you can put more than one component tree on a host web page, most applications have only one component tree and bootstrap a single root component.

This one root component is usually called `AppComponent` and is in the root module's [bootstrap](#) array.

## More about Angular Modules

For more on NgModules you're likely to see frequently in apps, see [Frequently Used Modules](#).

### Prerequisites

A basic understanding of the following concepts:

- [Bootstrapping](#).
  - [JavaScript Modules vs. NgModules](#).
- 

NgModules configure the injector and the compiler and help organize related things together.

An NgModule is a class marked by the [@NgModule](#) decorator. [@NgModule](#) takes a metadata object that describes how to compile a component's template and how to create an injector at runtime. It identifies the module's own components, directives, and pipes, making some of them public, through the [exports](#) property, so that external components can use them. [@NgModule](#) can also add service providers to the application dependency injectors.

For an example app showcasing all the techniques that NgModules related pages cover, see the [live example](#) / [download example](#). For explanations on the individual techniques, visit the relevant NgModule pages under the NgModules section.

## Angular modularity

Modules are a great way to organize an application and extend it with capabilities from external libraries.

Angular libraries are NgModules, such as [FormsModule](#), [HttpClientModule](#), and [RouterModule](#). Many third-party libraries are available as NgModules such as [Material Design](#), [Ionic](#), and [AngularFire2](#).

NgModules consolidate components, directives, and pipes into cohesive blocks of functionality, each focused on a feature area, application business domain, workflow, or common collection of utilities.

Modules can also add services to the application. Such services might be internally developed, like something you'd develop yourself or come from outside sources, such as the Angular router and HTTP client.

Modules can be loaded eagerly when the application starts or lazy loaded asynchronously by the router.

NgModule metadata does the following:

- Declares which components, directives, and pipes belong to the module.
- Makes some of those components, directives, and pipes public so that other module's component templates can use them.
- Imports other modules with the components, directives, and pipes that components in the current module need.
- Provides services that the other application components can use.

Every Angular app has at least one module, the root module. You [bootstrap](#) that module to launch the application.

The root module is all you need in a simple application with a few components. As the app grows, you refactor the root module into [feature modules](#) that represent collections of related functionality. You then import these modules into the root module.

## The basic NgModule

The CLI generates the following basic app module when creating a new app.

src/app/app.module.ts

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

import { AppComponent } from './app.component';
import { ItemDirective } from './item.directive';

// @NgModule decorator with its metadata
@NgModule({
  declarations: [
    AppComponent,
    ItemDirective
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

At the top are the import statements. The next section is where you configure the [@NgModule](#) by stating what components and directives belong to it ([declarations](#)) as well as which other modules it uses ([imports](#)). This page builds on [Bootstrapping](#), which covers the structure of an NgModule in detail. If you need more information on the structure of an [@NgModule](#), be sure to read [Bootstrapping](#).

---

## More on NgModules

You may also be interested in the following:

- [Feature Modules.](#)
- [Entry Components.](#)
- [Providers.](#)
- [Types of NgModules.](#)

### Prerequisites

A basic understanding of [JavaScript/ECMAScript modules](#).

---

JavaScript and Angular use modules to organize code, and though they organize it differently, Angular apps rely on both.

## JavaScript modules

In JavaScript, modules are individual files with JavaScript code in them. To make what's in them available, you write an export statement, usually after the relevant code, like this:

```
export class AppComponent { ... }
```

Then, when you need that file's code in another file, you import it like this:

```
import { AppComponent } from './app.component';
```

JavaScript modules help you namespace, preventing accidental global variables.

## NgModules

NgModules are classes decorated with [@NgModule](#). The [@NgModule](#) decorator's [imports](#) array tells Angular what other NgModules the current module needs. The modules in the [imports](#) array are different than JavaScript modules because they are NgModules rather than regular JavaScript modules. Classes with an [@NgModule](#) decorator are by convention kept in their own files, but what makes them an [NgModule](#) isn't being in their own file, like JavaScript modules; it's the presence of [@NgModule](#) and its metadata.

The AppModule generated from the Angular CLI demonstrates both kinds of modules in action:

1. /\* These are JavaScript import statements. Angular doesn't know anything about these.  
\*/
2. import { [BrowserModule](#) } from '@angular/platform-browser';
3. import { [NgModule](#) } from '@angular/core';
- 4.
5. import { AppComponent } from './app.component';

```

6.
7. /* The @NgModule decorator lets Angular know that this is an NgModule. */
8. @NgModule({
9.   declarations: [
10.     AppComponent
11.   ],
12.   imports: [ /* These are NgModule imports. */
13.     BrowserModule
14.   ],
15.   providers: [],
16.   bootstrap: [AppComponent]
17. })
18.export class AppModule { }

```

The NgModule classes differ from JavaScript module in the following key ways:

- An NgModule bounds [declarable classes](#) only. Declarables are the only classes that matter to the [Angular compiler](#).
  - Instead of defining all member classes in one giant file as in a JavaScript module, you list the module's classes in the [@NgModule.declarations](#) list.
  - An NgModule can only export the [declarable classes](#) it owns or imports from other modules. It doesn't declare or export any other kind of class.
  - Unlike JavaScript modules, an NgModule can extend the entire application with services by adding providers to the [@NgModule.providers](#) list.
- 

## More on NgModules

For more information on NgModules, see:

- [Bootstrapping](#).
- [Frequently used modules](#).
- [Providers](#).

### Prerequisites

A basic understanding of [Bootstrapping](#).

---

An Angular app needs at least one module that serves as the root module. As you add features to your app, you can add them in modules. The following are frequently used Angular modules with examples of some of the things they contain:

NgModule	Import it from	Why you use it
----------	----------------	----------------

<a href="#">BrowserModule</a>	@angular/platform-browser	When you want to run your app in a browser
<a href="#">CommonModule</a>	@angular/common	When you want to use <a href="#">NgIf</a> , <a href="#">NgFor</a>
<a href="#">FormsModule</a>	@angular/forms	When you want to build template driven forms (includes <a href="#">NgModel</a> )
<a href="#">ReactiveFormsModule</a>	@angular/forms	When you want to build reactive forms
<a href="#">RouterModule</a>	@angular/router	When you want to use <a href="#">RouterLink</a> , <a href="#">.forRoot()</a> , and <a href="#">.forChild()</a>
<a href="#">HttpClientModule</a>	@angular/common/http	When you want to talk to a server

## Importing modules

When you use these Angular modules, import them in `AppModule`, or your feature module as appropriate, and list them in the `@NgModule imports` array. For example, in the basic app generated by the CLI, [BrowserModule](#) is the first import at the top of the `AppModule`, `app.module.ts`.

```

1. /* import modules so that AppModule can access them */
2. import { BrowserModule } from '@angular/platform-browser';
3. import { NgModule } from '@angular/core';
4.
5. import { AppComponent } from './app.component';
6.
7. @NgModule({
8.   declarations: [
9.     AppComponent
10.   ],
11.   imports: [ /* add modules here so Angular knows to use them */
12.     BrowserModule,
13.   ],
14.   providers: [],
15.   bootstrap: [AppComponent]
16. })
17.export class AppModule { }
```

The imports at the top of the array are JavaScript import statements while the `imports` array within `@NgModule` is Angular specific. For more information on the difference, see [JavaScript Modules vs. NgModules](#).

## [BrowserModule](#) and [CommonModule](#)

[BrowserModule](#) imports [CommonModule](#), which contributes many common directives such as `ngIf` and `ngFor`. Additionally, [BrowserModule](#) re-exports [CommonModule](#) making all of its directives available to any module that imports [BrowserModule](#).

For apps that run in the browser, import [BrowserModule](#) in the root AppModule because it provides services that are essential to launch and run a browser app. [BrowserModule](#)'s providers are for the whole app so it should only be in the root module, not in feature modules. Feature modules only need the common directives in [CommonModule](#); they don't need to re-install app-wide providers.

If you do import [BrowserModule](#) into a lazy loaded feature module, Angular returns an error telling you to use [CommonModule](#) instead.

```
EXCEPTION: Uncaught (in promise): Error: BrowserModule has already been loaded. If you need access to common directives such as NgIf and NgFor from a lazy loaded module, import CommonModule instead.  
Error: BrowserModule has already been loaded. If you need access to common directives such as NgIf and NgFor from a lazy loaded module, import CommonModule instead.
```

---

## More on NgModules

You may also be interested in the following:

- [Bootstrapping](#).
- [NgModules](#).
- [JavaScript Modules vs. NgModules](#).

### Prerequisites

A basic understanding of the following concepts:

- [Feature Modules](#).
- [JavaScript Modules vs. NgModules](#).
- [Frequently Used Modules](#).

---

There are five general categories of feature modules which tend to fall into the following groups:

- Domain feature modules.
- Routed feature modules.
- Routing modules.
- Service feature modules.
- Widget feature modules.

While the following guidelines describe the use of each type and their typical characteristics, in real world apps, you may see hybrids.

## Module

Domain feature modules deliver a user experience dedicated to a particular application domain like editing a customer or placing an order.

They typically have a top component that acts as the feature root and private, supporting sub-components descend from it.

Domain feature modules consist mostly of declarations. Only the top component is exported.

## Domain

Domain feature modules rarely have providers. When they do, the lifetime of the provided services should be the same as the lifetime of the module.

Domain feature modules are typically imported exactly once by a larger feature module.

They might be imported by the root AppModule of a small application that lacks routing.

Routed feature modules are domain feature modules whose top components are the targets of router navigation routes.

All lazy-loaded modules are routed feature modules by definition.

Routed feature modules don't export anything because their components never appear in the template of an external component.

## Routed

A lazy-loaded routed feature module should not be imported by any module. Doing so would trigger an eager load, defeating the purpose of lazy loading. That means you won't see them mentioned among the AppModule imports. An eager loaded routed feature module must be imported by another module so that the compiler learns about its components.

Routed feature modules rarely have providers for reasons explained in [Lazy Loading Feature Modules](#). When they do, the lifetime of the provided services should be the same as the lifetime of the module. Don't provide application-wide singleton services in a routed feature module or in a module that the routed module imports.

## Routing

A routing module provides routing configuration for another module and separates routing concerns from its companion module.

A routing module typically does the following:

- Defines routes.
- Adds router configuration to the module's imports.
- Adds guard and resolver service providers to the module's providers.

- The name of the routing module should parallel the name of its companion module, using the suffix "Routing". For example, FooModule in foo.module.ts has a routing module named FooRoutingModule in foo-routing.module.ts. If the companion module is the root AppModule, the AppRoutingModule adds router configuration to its imports with RouterModule.forRoot(routes). All other routing modules are children that import RouterModule.forChild(routes).
- A routing module re-exports the [RouterModule](#) as a convenience so that components of the companion module have access to router directives such as [RouterLink](#) and [RouterOutlet](#).
- A routing module does not have its own declarations. Components, directives, and pipes are the responsibility of the feature module, not the routing module.

A routing module should only be imported by its companion module.

**Service** Service modules provide utility services such as data access and messaging. Ideally, they consist entirely of providers and have no declarations. Angular's [HttpClientModule](#) is a good example of a service module.

The root AppModule is the only module that should import service modules.

A widget module makes components, directives, and pipes available to external modules. Many third-party UI component libraries are widget modules.

**Widget** A widget module should consist entirely of declarations, most of them exported.

A widget module should rarely have providers.

Import widget modules in any module whose component templates need the widgets.

The following table summarizes the key characteristics of each feature module group.

Feature Module	Declarations	Providers	Exports	Imported by
Domain	Yes	Rare	Top component	Feature, AppModule
Routed	Yes	Rare	No	None
Routing	No	Yes (Guards)	RouterModule	Feature (for routing)
Service	No	Yes	No	AppModule
Widget	Yes	Rare	Yes	Feature

## More on NgModules

You may also be interested in the following:

- [Lazy Loading Modules with the Angular Router](#).
- [Providers](#).

### Prerequisites:

A basic understanding of the following concepts:

- [Bootstrapping](#).
- 

An entry component is any component that Angular loads imperatively, (which means you're not referencing it in the template), by type. You specify an entry component by bootstrapping it in an NgModule, or including it in a routing definition.

To contrast the two types of components, there are components which are included in the template, which are declarative. Additionally, there are components which you load imperatively; that is, entry components.

There are two main kinds of entry components:

- The bootstrapped root component.
- A component you specify in a route definition.

## A bootstrapped entry component

The following is an example of specifying a bootstrapped component, AppComponent, in a basic app.module.ts:

```
1. @NgModule({
2.   declarations: [
3.     AppComponent
4.   ],
5.   imports: [
6.     BrowserModule,
7.     FormsModule,
8.     HttpClientModule,
9.     AppRoutingModule
10. ],
11. providers: [],
12. bootstrap: [AppComponent] // bootstrapped entry component
13.})
```

A bootstrapped component is an entry component that Angular loads into the DOM during the bootstrap process (application launch). Other entry components are loaded dynamically by other means, such as with the router.

Angular loads a root AppComponent dynamically because it's listed by type in [@NgModule.bootstrap](#).

A component can also be bootstrapped imperatively in the module's ngDoBootstrap() method. The [@NgModule.bootstrap](#) property tells the compiler that this is an entry component and it should generate code to bootstrap the application with this component.

A bootstrapped component is necessarily an entry component because bootstrapping is an imperative process, thus it needs to have an entry component.

## A routed entry component

The second kind of entry component occurs in a route definition like this:

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

A route definition refers to a component by its type with component: CustomerListComponent.

All router components must be entry components. Because this would require you to add the component in two places (router and entryComponents) the Compiler is smart enough to recognize that this is a router definition and automatically add the router component into entryComponents.

## The entryComponents array

Though the [@NgModule](#) decorator has an entryComponents array, most of the time you won't have to explicitly set any entry components because Angular adds components listed in [@NgModule.bootstrap](#) and those in route definitions to entry components automatically.

Though these two mechanisms account for most entry components, if your app happens to bootstrap or dynamically load a component by type imperatively, you must add it to entryComponents explicitly.

### entryComponents and the compiler

For production apps you want to load the smallest code possible. The code should contain only the classes that you actually need and exclude components that are never used. For this reason, the Angular compiler only generates code for components which are reachable from the entryComponents; This means that adding more references to

`@NgModule.declarations` does not imply that they will necessarily be included in the final bundle.

In fact, many libraries declare and export components you'll never use. For example, a material design library will export all components because it doesn't know which ones you will use. However, it is unlikely that you will use them all. For the ones you don't reference, the tree shaker drops these components from the final code package.

If a component isn't an entry component and isn't found in a template, the tree shaker will throw it away. So, it's best to add only the components that are truly entry components to help keep your app as trim as possible.

---

## More on Angular modules

You may also be interested in the following:

- [Types of NgModules](#)
- [Lazy Loading Modules with the Angular Router](#).
- [Providers](#).
- [NgModules FAQ](#).

Feature modules are NgModules for the purpose of organizing code.

### Prerequisites

A basic understanding of the following:

- [Bootstrapping](#).
- [JavaScript Modules vs. NgModules](#).
- [Frequently Used Modules](#).

For the final sample app with a feature module that this page describes, see the [live example](#) / [download example](#).

---

As your app grows, you can organize code relevant for a specific feature. This helps apply clear boundaries for features. With feature modules, you can keep code related to a specific functionality or feature separate from other code. Delineating areas of your app helps with collaboration between developers and teams, separating directives, and managing the size of the root module.

## Feature modules vs. root modules

A feature module is an organizational best practice, as opposed to a concept of the core Angular API. A feature module delivers a cohesive set of functionality focused on a specific application need such as a user workflow, routing, or forms. While you can do everything

within the root module, feature modules help you partition the app into focused areas. A feature module collaborates with the root module and with other modules through the services it provides and the components, directives, and pipes that it shares.

## How to make a feature module

Assuming you already have a CLI generated app, create a feature module using the CLI by entering the following command in the root project directory. Replace CustomerDashboard with the name of your module. You can omit the "Module" suffix from the name because the CLI appends it:

```
ng generate module CustomerDashboard
```

This causes the CLI to create a folder called customer-dashboard with a file inside called customer-dashboard.module.ts with the following contents:

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

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

The structure of an NgModule is the same whether it is a root module or a feature module. In the CLI generated feature module, there are two JavaScript import statements at the top of the file: the first imports [NgModule](#), which, like the root module, lets you use the [@NgModule](#) decorator; the second imports [CommonModule](#), which contributes many common directives such as [ngIf](#) and [ngFor](#). Feature modules import [CommonModule](#) instead of [BrowserModule](#), which is only imported once in the root module. [CommonModule](#) only contains information for common directives such as [ngIf](#) and [ngFor](#) which are needed in most templates, whereas [BrowserModule](#) configures the Angular app for the browser which needs to be done only once.

The [declarations](#) array is available for you to add declarables, which are components, directives, and pipes that belong exclusively to this particular module. To add a component, enter the following command at the command line where customer-dashboard is the directory where the CLI generated the feature module and CustomerDashboard is the name of the component:

```
ng generate component customer-dashboard/CustomerDashboard
```

This generates a folder for the new component within the customer-dashboard folder and updates the feature module with the CustomerDashboardComponent info:

```
src/app/customer-dashboard/customer-dashboard.module.ts

// import the new component
import { CustomerDashboardComponent } from './customer-dashboard/customer-
dashboard.component';
@NgModule({
  imports: [
    CommonModule
  ],
  declarations: [
    CustomerDashboardComponent
  ],
})
})
```

The CustomerDashboardComponent is now in the JavaScript import list at the top and added to the [declarations](#) array, which lets Angular know to associate this new component with this feature module.

## Importing a feature module

To incorporate the feature module into your app, you have to let the root module, app.module.ts, know about it. Notice the CustomerDashboardModule export at the bottom of customer-dashboard.module.ts. This exposes it so that other modules can get to it. To import it into the AppModule, add it to the imports in app.module.ts and to the [imports](#) array:

```
src/app/app.module.ts

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

import { AppComponent } from './app.component';
// import the feature module here so you can add it to the imports array below
import { CustomerDashboardModule } from './customer-dashboard/customer-
dashboard.module';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule,
    CustomerDashboardModule
  ]
})
```

```
CustomerDashboardModule // add the feature module here
],
providers: [],
bootstrap: [AppComponent]
})
export class AppModule { }
```

Now the AppModule knows about the feature module. If you were to add any service providers to the feature module, AppModule would know about those too, as would any other feature modules. However, NgModules don't expose their components.

## Rendering a feature module's component template

When the CLI generated the CustomerDashboardComponent for the feature module, it included a template, customer-dashboard.component.html, with the following markup:

```
src/app/customer-dashboard/customer-dashboard/customer-dashboard.component.html

<p>
  customer-dashboard works!
</p>
```

To see this HTML in the AppComponent, you first have to export the CustomerDashboardComponent in the CustomerDashboardModule. In customer-dashboard.module.ts, just beneath the [declarations](#) array, add an [exports](#) array containing CustomerDashboardModule:

```
src/app/customer-dashboard/customer-dashboard.module.ts
```

```
exports: [
  CustomerDashboardComponent
]
```

Next, in the AppComponent, app.component.html, add the tag <app-customer-dashboard>:

```
src/app/app.component.html
```

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

<!-- add the selector from the CustomerDashboardComponent -->
<app-customer-dashboard></app-customer-dashboard>
```

Now, in addition to the title that renders by default, the CustomerDashboardComponent template renders too:



# app works!

customer-dashboard works!

---

## More on NgModules

You may also be interested in the following:

- [Lazy Loading Modules with the Angular Router.](#)
- [Providers.](#)
- [Types of Feature Modules.](#)

### Prerequisites:

- A basic understanding of [Bootstrapping](#).
- Familiarity with [Frequently Used Modules](#).

For the final sample app using the provider that this page describes, see the [live example](#) / [download example](#).

---

A provider is an instruction to the DI system on how to obtain a value for a dependency. Most of the time, these dependencies are services that you create and provide.

## Providing a service

If you already have a CLI generated app, create a service using the following CLI command in the root project directory. Replace User with the name of your service.

```
ng generate service User
```

This command creates the following UserService skeleton:

```
src/app/user.service.0.ts

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

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

You can now inject UserService anywhere in your application.

The service itself is a class that the CLI generated and that's decorated with [@Injectable](#). By default, this decorator is configured with a [providedIn](#) property, which creates a provider for the service. In this case, [providedIn](#): 'root' specifies that the service should be provided in the root injector.

## Provider scope

When you add a service provider to the root application injector, it's available throughout the app. Additionally, these providers are also available to all the classes in the app as long they have the lookup token.

You should always provide your service in the root injector unless there is a case where you want the service to be available only if the consumer imports a particular [@NgModule](#).

## providedIn and NgModules

It's also possible to specify that a service should be provided in a particular [@NgModule](#). For example, if you don't want UserService to be available to applications unless they import a UserModule you've created, you can specify that the service should be provided in the module:

```
src/app/user.service.1.ts

import { Injectable } from '@angular/core';
import { UserModule } from './user.module';

@Injectable({
  providedIn: UserModule,
})
export class UserService {
```

The example above shows the preferred way to provide a service in a module. This method is preferred because it enables tree-shaking of the service if nothing injects it. If it's not possible to specify in the service which module should provide it, you can also declare a provider for the service within the module:

```
src/app/user.module.ts
```

```
import { NgModule } from '@angular/core';
import { UserService } from './user.service';

@NgModule({
  providers: [UserService],
})
export class UserModule { }
```

## **Limiting provider scope by lazy loading modules**

In the basic CLI generated app, modules are eagerly loaded which means that they are all loaded when the app launches. Angular uses an injector system to make things available between modules. In an eagerly loaded app, the root application injector makes all of the providers in all of the modules available throughout the app.

This behavior necessarily changes when you use lazy loading. Lazy loading is when you load modules only when you need them; for example, when routing. They aren't loaded right away like with eagerly loaded modules. This means that any services listed in their provider arrays aren't available because the root injector doesn't know about these modules.

When the Angular router lazy-loads a module, it creates a new injector. This injector is a child of the root application injector. Imagine a tree of injectors; there is a single root injector and then a child injector for each lazy loaded module. The router adds all of the providers from the root injector to the child injector. When the router creates a component within the lazy-loaded context, Angular prefers service instances created from these providers to the service instances of the application root injector.

Any component created within a lazy loaded module's context, such as by router navigation, gets the local instance of the service, not the instance in the root application injector.

Components in external modules continue to receive the instance created for the application root.

Though you can provide services by lazy loading modules, not all services can be lazy loaded. For instance, some modules only work in the root module, such as the Router. The Router works with the global location object in the browser.

## **Limiting provider scope with components**

Another way to limit provider scope is by adding the service you want to limit to the component's providers array. Component providers and NgModule providers are independent of each other. This method is helpful for when you want to eagerly load a module that needs a service all to itself. Providing a service in the component limits the service only to that component (other components in the same module can't access it.)

src/app/app.component.ts

```
@Component({
/* . . . */
providers: [UserService]
})
```

## **Providing services in modules vs. components**

Generally, provide services the whole app needs in the root module and scope services by providing them in lazy loaded modules.

The router works at the root level so if you put providers in a component, even AppComponent, lazy loaded modules, which rely on the router, can't see them.

Register a provider with a component when you must limit a service instance to a component and its component tree, that is, its child components. For example, a user editing component, UserEditorComponent, that needs a private copy of a caching UserService should register the UserService with the UserEditorComponent. Then each new instance of the UserEditorComponent gets its own cached service instance.

---

## **More on NgModules**

You may also be interested in:

- [Singleton Services](#), which elaborates on the concepts covered on this page.
- [Lazy Loading Modules](#).
- [Tree-shakable Providers](#).
- [NgModule FAQ](#).

### **Prerequisites:**

- A basic understanding of [Bootstrapping](#).
- Familiarity with [Providers](#).

For a sample app using the app-wide singleton service that this page describes, see the [live example](#) / [download example](#) showcasing all the documented features of NgModules.

---

# Providing a singleton service

There are two ways to make a service a singleton in Angular:

- Declare that the service should be provided in the application root.
- Include the service in the `AppModule` or in a module that is only imported by the `AppModule`.

Beginning with Angular 6.0, the preferred way to create a singleton services is to specify on the service that it should be provided in the application root. This is done by setting `providedIn` to `root` on the service's `@Injectable` decorator:

`src/app/user.service.0.ts`

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

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

For more detailed information on services, see the [Services](#) chapter of the [Tour of Heroes tutorial](#).

## forRoot()

If a module provides both providers and declarations (components, directives, pipes) then loading it in a child injector such as a route, would duplicate the provider instances. The duplication of providers would cause issues as they would shadow the root instances, which are probably meant to be singletons. For this reason Angular provides a way to separate providers out of the module so that same module can be imported into the root module with providers and child modules without providers.

1. Create a static method `forRoot()` (by convention) on the module.
2. Place the providers into the `forRoot` method as follows.

To make this more concrete, consider the `RouterModule` as an example. `RouterModule` needs to provide the `Router` service, as well as the `RouterOutlet` directive. `RouterModule` has to be imported by the root application module so that the application has a `Router` and the application has at least one `RouterOutlet`. It also must be imported by the individual route components so that they can place `RouterOutlet` directives into their template for sub-routes.

If the `RouterModule` didn't have `forRoot()` then each route component would instantiate a new `Router` instance, which would break the application as there can only be one `Router`. For this reason, the `RouterModule` has the `RouterOutlet` declaration so that it is available everywhere, but the `Router` provider is only in the `forRoot()`. The result is that the root application module

imports RouterModule.forRoot(...) and gets a [Router](#), whereas all route components import [RouterModule](#) which does not include the [Router](#).

If you have a module which provides both providers and declarations, use this pattern to separate them out.

A module that adds providers to the application can offer a facility for configuring those providers as well through the [forRoot\(\)](#) method.

[forRoot\(\)](#) takes a service configuration object and returns a [ModuleWithProviders](#), which is a simple object with the following properties:

- ngModule: in this example, the CoreModule class.
- providers: the configured providers.

In the [live example](#) / [download example](#) the root AppModule imports the CoreModule and adds the providers to the AppModule.providers. Specifically, Angular accumulates all imported providers before appending the items listed in [@NgModule.providers](#). This sequence ensures that whatever you add explicitly to the AppModule providers takes precedence over the providers of imported modules.

Import CoreModule and use its [forRoot\(\)](#) method one time, in AppModule, because it registers services and you only want to register those services one time in your app. If you were to register them more than once, you could end up with multiple instances of the service and a runtime error.

You can also add a [forRoot\(\)](#) method in the CoreModule that configures the core UserService.

In the following example, the optional, injected UserServiceConfig extends the core UserService. If a UserServiceConfig exists, the UserService sets the user name from that config.

src/app/core/user.service.ts (constructor)

```
constructor(@Optional() config: UserServiceConfig) {
  if (config) { this._userName = config.userName; }
}
```

Here's [forRoot\(\)](#) that takes a UserServiceConfig object:

src/app/core/core.module.ts (forRoot)

```
static forRoot(config: UserServiceConfig): ModuleWithProviders {
  return {
    ngModule: CoreModule,
    providers: [
      {provide: UserServiceConfig, useValue: config }
    ]
  };
}
```

Lastly, call it within the [imports](#) list of the AppModule.

```
src/app/app.module.ts (imports)
```

```
import { CoreModule } from './core/core.module';
/* . . . */
@NgModule({
  imports: [
    BrowserModule,
    ContactModule,
    CoreModule.forRoot({userName: 'Miss Marple'}),
    AppRoutingModule
  ],
/* . . . */
})
export class AppModule {}
```

The app displays "Miss Marple" as the user instead of the default "Sherlock Holmes".

Remember to import CoreModule as a Javascript import at the top of the file; don't add it to more than one [@NgModule imports](#) list.

## Prevent reimport of the CoreModule

Only the root AppModule should import the CoreModule. If a lazy-loaded module imports it too, the app can generate [multiple instances](#) of a service.

To guard against a lazy-loaded module re-importing CoreModule, add the following CoreModule constructor.

```
src/app/core/core.module.ts
```

```
constructor (@Optional() @SkipSelf() parentModule: CoreModule) {
  if (parentModule) {
    throw new Error(
      'CoreModule is already loaded. Import it in the AppModule only');
  }
}
```

The constructor tells Angular to inject the CoreModule into itself. The injection would be circular if Angular looked forCoreModule in the current injector. The [@SkipSelf](#) decorator means "look for CoreModule in an ancestor injector, above me in the injector hierarchy."

If the constructor executes as intended in the AppModule, there would be no ancestor injector that could provide an instance of CoreModule and the injector should give up.

By default, the injector throws an error when it can't find a requested provider. The [@Optional](#) decorator means not finding the service is OK. The injector returns null, the parentModule parameter is null, and the constructor concludes uneventfully.

It's a different story if you improperly import CoreModule into a lazy-loaded module such as CustomersModule.

Angular creates a lazy-loaded module with its own injector, a child of the root injector. [@SkipSelf](#) causes Angular to look for a CoreModule in the parent injector, which this time is the root injector. Of course it finds the instance imported by the root AppModule. Now parentModule exists and the constructor throws the error.

Here are the two files in their entirety for reference:

app.module.ts

core.module.ts

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

```
/* App Root */
```

```
import { AppComponent } from './app.component';
```

```
/* Feature Modules */
```

```
import { ContactModule } from './contact/contact.module';
```

```
import { CoreModule } from './core/core.module';
```

```
/* Routing Module */
```

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

```
@NgModule({
```

```
  imports: [
```

```
    BrowserModule,
```

```
    ContactModule,
```

```
    CoreModule.forRoot({userName: 'Miss Marple'}),
```

```
    AppRoutingModule
```

```
  ],
```

```
  providers: [],
```

```
  declarations: [
```

```
    AppComponent
```

```
  ],
```

```
  bootstrap: [AppComponent]
```

```
})
```

```
export class AppModule { }
```

---

## More on NgModules

You may also be interested in:

- [Sharing Modules](#), which elaborates on the concepts covered on this page.
- [Lazy Loading Modules](#).
- [NgModule FAQ](#).

### Prerequisites

A basic understanding of the following:

- [Feature Modules](#).
- [JavaScript Modules vs. NgModules](#).
- [Frequently Used Modules](#).
- [Types of Feature Modules](#).
- [Routing and Navigation](#).

For the final sample app with two lazy loaded modules that this page describes, see the [live example](#) / [download example](#).

---

## High level view

There are three main steps to setting up a lazy loaded feature module:

1. Create the feature module.
2. Create the feature module's routing module.
3. 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 do already have an app, skip to [Configure the routes](#). Enter the following command where customer-app is the name of your app:

```
ng new customer-app --routing
```

This creates an app 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.

## Create a feature module with routing

Next, you'll need a feature module to route to. To make one, enter the following command at the terminal window prompt where customers is the name of the module:

```
ng generate module customers --routing
```

This creates a customers folder with two files inside; CustomersModule and CustomersRoutingModule. CustomersModule will act as the gatekeeper for anything that concerns customers. CustomersRoutingModule will handle any customer-related routing. This keeps the app's structure organized as the app grows and allows you to reuse this module while easily keeping its routing intact.

The CLI imports the CustomersRoutingModule into the CustomersModule by adding a JavaScript import statement at the top of the file and adding CustomersRoutingModule to the [@NgModule imports](#) array.

## Add a component to the feature module

In order to see the module being lazy loaded in the browser, create a component to render some HTML when the app loads CustomersModule. At the command line, enter the following:

```
ng generate component customers/customer-list
```

This creates a folder inside of customers called customer-list with the four files that make up the component.

Just like with the routing module, the CLI imports the CustomerListComponent into the CustomersModule.

## Add another feature module

For another place to route to, create a second feature module with routing:

```
ng generate module orders --routing
```

This makes a new folder called orders containing an OrdersModule and an OrdersRoutingModule.

Now, just like with the CustomersModule, give it some content:

```
ng generate component orders/order-list
```

## Set up the UI

Though you can type the URL into the address bar, a nav 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
```

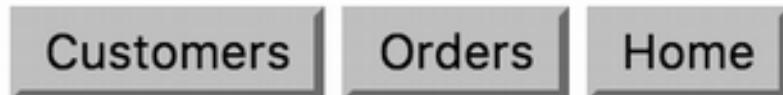
```
<h1>  
{{title}}  
</h1>  
  
<button routerLink="/customers">Customers</button>  
<button routerLink="/orders">Orders</button>  
<button routerLink="">Home</button>  
  
<router-outlet></router-outlet>
```

To see your app in the browser so far, enter the following command in the terminal window:  
ng serve

Then go to localhost:4200 where you should see “app works!” and three buttons.



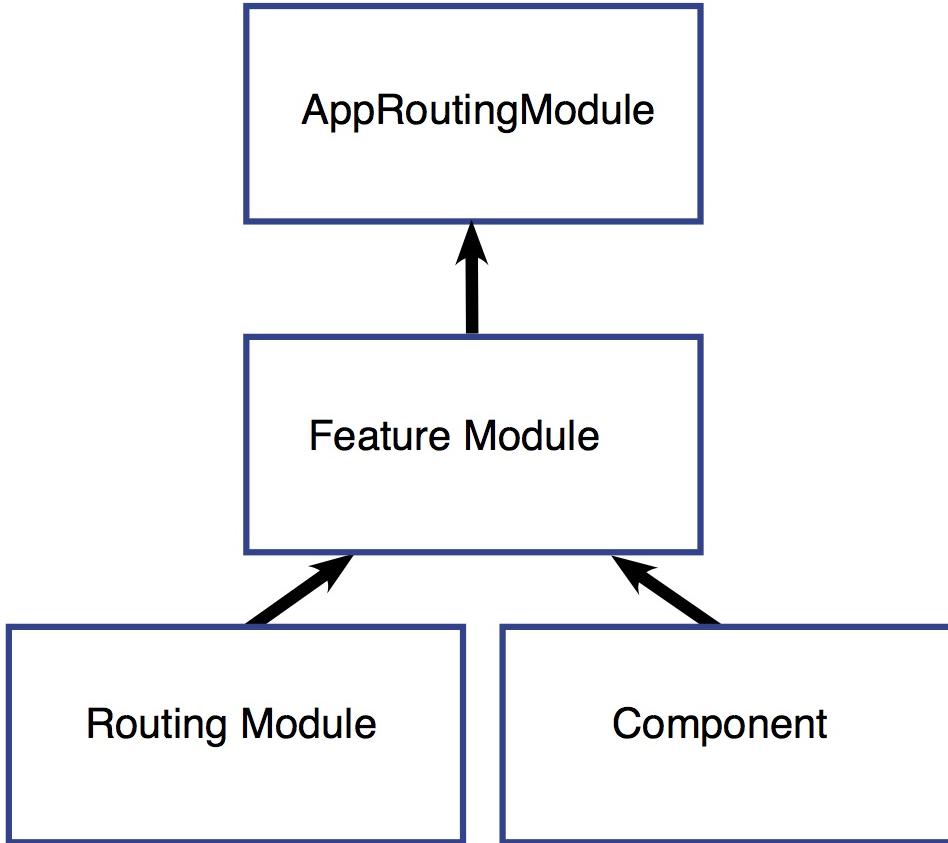
# app works!



To make the buttons work, you need to configure the routing modules.

## Configure the routes

The two feature modules, OrdersModule and CustomersModule, have to be wired up to the AppRoutingModule so the router knows about them. The structure is as follows:



Each feature module acts as a doorway via the router. In the AppRoutingModule, you configure the routes to the feature modules, in this case OrdersModule and CustomersModule. This way, the router knows to go to the feature module. The feature module then connects the AppRoutingModule to the CustomersRoutingModule or the OrdersRoutingModule. Those routing modules tell the router where to go to load relevant components.

## Routes at the app level

In AppRoutingModule, update the routes array with the following:

src/app/app-routing.module.ts

```

const routes: Routes = [
  {
    path: 'customers',
    loadChildren: 'app/customers/customers.module#CustomersModule'
  },
  {
    path: 'orders',
    loadChildren: 'app/orders/orders.module#OrdersModule'
  },
]
  
```

```
{  
  path: "",  
  redirectTo: "",  
  pathMatch: "full"  
}  
];
```

The import statements stay the same. The first two paths are the routes to the CustomersModule and the OrdersModule respectively. Notice that the lazy loading syntax uses [loadChildren](#) followed by a string that is the path to the module, a hash mark or #, and the module's class name.

## Inside the feature module

Next, take a look at customers.module.ts. If you're using the CLI and following the steps outlined in this page, you don't have to do anything here. The feature module is like a connector between the AppRoutingModule and the feature routing module. The AppRoutingModule imports the feature module, CustomersModule, and CustomersModule in turn imports the CustomersRoutingModule.

src/app/customers/customers.module.ts

```
import { NgModule } from '@angular/core';  
import { CommonModule } from '@angular/common';  
import { CustomersRoutingModule } from './customers-routing.module';  
import { CustomerListComponent } from './customer-list/customer-list.component';  
  
@NgModule({  
  imports: [  
    CommonModule,  
    CustomersRoutingModule  
,  
  declarations: [CustomerListComponent]  
})  
export class CustomersModule {}
```

The customers.module.ts file imports the CustomersRoutingModule and CustomerListComponent so the CustomersModule class can have access to them. CustomersRoutingModule is then listed in the [@NgModule imports](#) array giving CustomersModule access to its own routing module, and CustomerListComponent is in the [declarations](#) array, which means CustomerListComponent belongs to the CustomersModule.

## Configure the feature module's routes

The next step is in customers-routing.module.ts. First, import the component at the top of the file with the other JavaScript import statements. Then, add the route to CustomerListComponent.

```
src/app/customers/customers-routing.module.ts

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

import { CustomerListComponent } from './customer-list/customer-list.component';

const routes: Routes = [
{
  path: '',
  component: CustomerListComponent
};

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

Notice that the path is set to an empty string. This is 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.

Repeat this last step of importing the OrdersListComponent and configuring the Routes array for the orders-routing.module.ts:

```
src/app/orders/orders-routing.module.ts (excerpt)

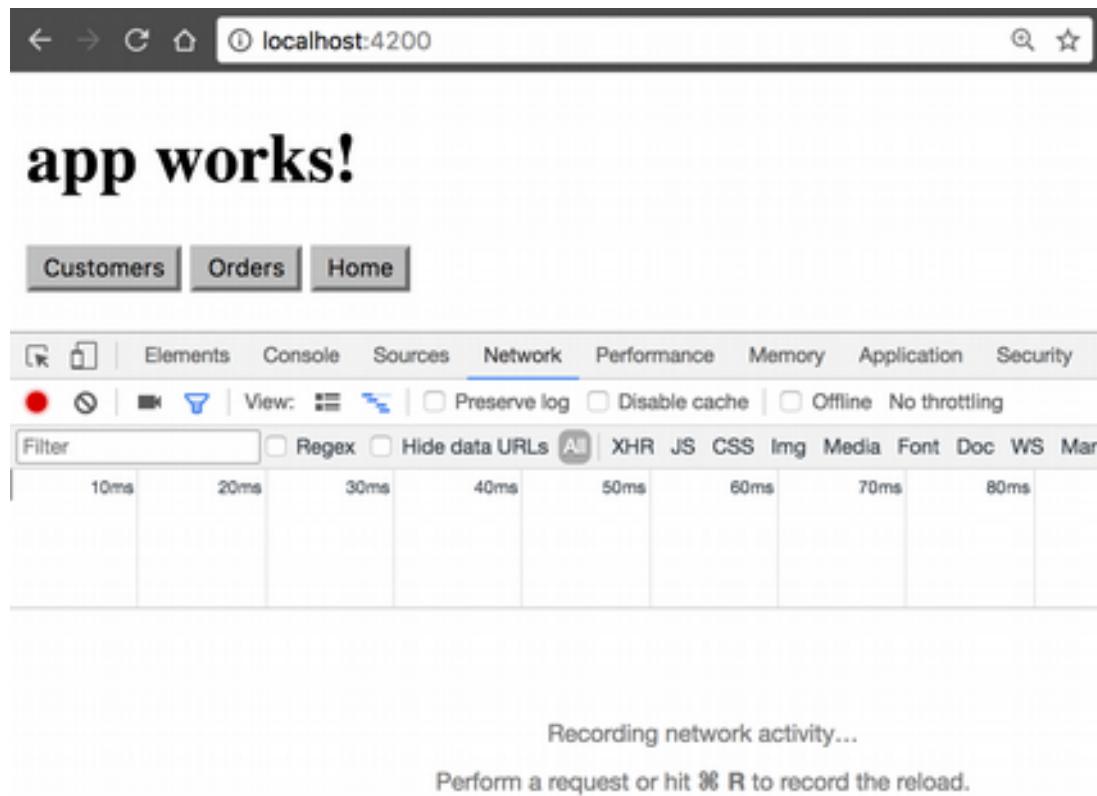
import { OrderListComponent } from './order-list/order-list.component';

const routes: Routes = [
{
  path: '',
  component: OrderListComponent
};
```

Now, if you view the app in the browser, the three buttons take you to each module.

## Confirm it's working

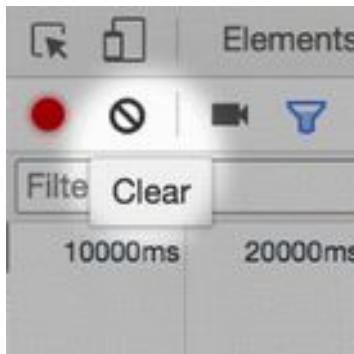
You can check to see that a module is indeed being lazy loaded with the Chrome developer tools. In Chrome, open the dev tools by pressing Cmd+Option+i on a Mac or Ctrl+Alt+i on a PC and go to the Network Tab.



Click on the Orders or Customers button. If you see a chunk appear, you've wired everything 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.

The screenshot shows a browser window with the URL `localhost:4200/orders`. The page content says "app works!" and has three navigation buttons: "Customers", "Orders", and "Home". Below the content is the Chrome DevTools Network tab. The table in the Network tab has columns for Name, Status, Type, Initiator, Size, Time, and Waterfall. One row in the table is highlighted with a blue oval around the "Name" column, and a blue arrow points from this oval to a callout box containing the text "Chunk shows up here.".

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 app-`routing.module.ts` [imports](#) array. This lets Angular know that this module, `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 `RouterService`. Use `forRoot()` in the `AppRoutingModule`—that is, one time in the app at the root level.

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.

[forRoot\(\)](#) contains injector configuration which is global; such as configuring the Router.  
[forChild\(\)](#) has no injector configuration, only directives such as [RouterOutlet](#) and [RouterLink](#).

---

## More on NgModules and routing

You may also be interested in the following:

- [Routing and Navigation.](#)
- [Providers.](#)
- [Types of Feature Modules.](#)

### Prerequisites

A basic understanding of the following:

- [Feature Modules.](#)
  - [JavaScript Modules vs. NgModules.](#)
  - [Frequently Used Modules.](#)
  - [Routing and Navigation.](#)
  - [Lazy loading modules.](#)
- 

Creating shared modules allows you to organize and streamline your code. You can put commonly used directives, pipes, and components into one module and then import just that module wherever you need it in other parts of your app.

Consider the following module from an imaginary app:

```
1. import { CommonModule } from '@angular/common';
2. import { NgModule } from '@angular/core';
3. import { FormsModule } from '@angular/forms';
4. import { CustomerComponent } from './customer.component';
5. import { NewItemDirective } from './new-item.directive';
6. import { OrdersPipe } from './orders.pipe';
7.
8. @NgModule({
9.   imports: [ CommonModule ],
10. declarations: [ CustomerComponent, NewItemDirective, OrdersPipe ],
11. exports: [ CustomerComponent, NewItemDirective, OrdersPipe,
12.           CommonModule, FormsModule ]
13. })
14. export class SharedModule { }
```

Note the following:

- It imports the [CommonModule](#) because the module's component needs common directives.
- It declares and exports the utility pipe, directive, and component classes.
- It re-exports the [CommonModule](#) and [FormsModule](#).

By re-exporting [CommonModule](#) and [FormsModule](#), any other module that imports this SharedModule, gets access to directives like [NgIf](#) and [NgFor](#) from [CommonModule](#) and can bind to component properties with `[(ngModel)]`, a directive in the [FormsModule](#).

Even though the components declared by SharedModule might not bind with `[(ngModel)]` and there may be no need for SharedModule to import [FormsModule](#), SharedModule can still export [FormsModule](#) without listing it among its [imports](#). This way, you can give other modules access to [FormsModule](#) without having to import it directly into the [@NgModule](#) decorator.

## Using components vs services from other modules.

There is an important distinction between using another module's component and using a service from another module. Import modules when you want to use directives, pipes, and components. Importing a module with services means that you will have a new instance of that service, which typically is not what you need (typically one wants to reuse an existing service). Use module imports to control service instantiation.

The most common way to get a hold of shared services is through Angular [dependency injection](#), rather than through the module system (importing a module will result in a new service instance, which is not a typical usage).

To read about sharing services, see [Providers](#).

---

## More on NgModules

You may also be interested in the following:

- [Providers](#).
- [Types of Feature Modules](#).

## Prerequisites

A basic understanding of the following concepts:

- [Bootstrapping](#).
  - [JavaScript Modules vs. NgModules](#).
-

## Purpose of `@NgModule`

At a high level, NgModules are a way to organize Angular apps and they accomplish this through the metadata in the `@NgModule` decorator. The metadata falls into three categories:

- Static: Compiler configuration which tells the compiler about directive selectors and where in templates the directives should be applied through selector matching. This is configured via the `declarations` array.
- Runtime: Injector configuration via the providers array.
- Composability/Grouping: Bringing NgModules together and making them available via the `imports` and `exports` arrays.

```
1. @NgModule({
2. // Static, that is compiler configuration
3.  declarations: [], // Configure the selectors
4.  entryComponents: [], // Generate the host factory
5.
6. // Runtime, or injector configuration
7.  providers: [], // Runtime injector configuration
8.
9. // Composability / Grouping
10.  imports: [], // composing NgModules together
11.  exports: [] // making NgModules available to other parts of the app
12. })
```

## `@NgModule` metadata

The following table summarizes the `@NgModule` metadata properties.

Property	Description
<code>declarations</code>	A list of <code>declarable</code> classes, (components, directives, and pipes) that belong to this module. <ol style="list-style-type: none"><li>1. When compiling a template, you need to determine a set of selectors which should be used for triggering their corresponding directives.</li><li>2. The template is compiled within the context of an NgModule—the NgModule within which the template's component is declared—which determines the set of selectors using the following rules:<ul style="list-style-type: none"><li>• All selectors of directives listed in `declarations`.</li><li>• All selectors of directives exported from imported NgModules.</li></ul></li></ol>

Components, directives, and pipes must belong to exactly one module. The compiler emits an error if you try to declare the same class in more

## providers

than one module. Be careful not to re-declare a class that is imported directly or indirectly from another module.

A list of dependency-injection providers.

Angular registers these providers with the NgModule's injector. If it is the NgModule used for bootstrapping then it is the root injector.

These services become available for injection into any component, directive, pipe or service which is a child of this injector.

A lazy-loaded module has its own injector which is typically a child of the application root injector.

Lazy-loaded services are scoped to the lazy module's injector. If a lazy-loaded module also provides the UserService, any component created within that module's context (such as by router navigation) gets the local instance of the service, not the instance in the root application injector.

Components in external modules continue to receive the instance provided by their injectors.

For more information on injector hierarchy and scoping, see [Providers](#) and the [DI Guide](#).

A list of modules which should be folded into this module. Folded means it is as if all the imported NgModule's exported properties were declared here.

Specifically, it is as if the list of modules whose exported components, directives, or pipes are referenced by the component templates were declared in this module.

## imports

A component template can [reference](#) another component, directive, or pipe when the reference is declared in this module or if the imported module has exported it. For example, a component can use the [NgIf](#) and NgFor directives only if the module has imported the Angular [CommonModule](#)(perhaps indirectly by importing [BrowserModule](#)).

You can import many standard directives from the [CommonModule](#) but some familiar directives belong to other modules. For example, you can use `[(ngModel)]` only after importing the Angular [FormsModule](#).

## exports

A list of declarations—component, directive, and pipe classes—that an importing module can use.

Exported declarations are the module's public API. A component in another module can [use](#) thismodule's UserComponent if it imports this

module and this module exports `UserComponent`.

Declarations are private by default. If this module does not export `UserComponent`, then only the components within this module can use `UserComponent`.

Importing a module does not automatically re-export the imported module's imports. Module 'B' can't use `nglf` just because it imported module 'A' which imported `CommonModule`. Module 'B' must import `CommonModule` itself.

A module can list another module among its `exports`, in which case all of that module's public components, directives, and pipes are exported.

`Re-export` makes module transitivity explicit. If Module 'A' re-exports `CommonModule` and Module 'B' imports Module 'A', Module 'B' components can use `nglf` even though 'B' itself didn't import `CommonModule`.

A list of components that are automatically bootstrapped.

Usually there's only one component in this list, the root component of the application.

## bootstrap

Angular can launch with multiple bootstrap components, each with its own location in the host web page.

A bootstrap component is automatically added to `entryComponents`.

## entryComponents A list of components that can be dynamically loaded into the view.

By default, an Angular app always has at least one entry component, the root component, `AppComponent`. Its purpose is to serve as a point of entry into the app, that is, you bootstrap it to launch the app.

Routed components are also entry components because they need to be loaded dynamically. The router creates them and drops them into the DOM near a `<router-outlet>`.

While the bootstrapped and routed components are entry components, you don't have to add them to a module's `entryComponents` list, as they are added implicitly.

Angular automatically adds components in the module's `bootstrap` and route definitions into the `entryComponents` list.

That leaves only components bootstrapped using one of the imperative techniques, such as `ViewComponentRef.createComponent()` as

undiscoverable.

Dynamic component loading is not common in most apps beyond the router. If you need to dynamically load components, you must add these components to the entryComponents list yourself.

For more information, see [Entry Components](#).

---

## More on NgModules

You may also be interested in the following:

- [Feature Modules](#).
- [Entry Components](#).
- [Providers](#).
- [Types of Feature Modules](#).

### Prerequisites:

A basic understanding of the following concepts:

- [NgModules](#).

---

NgModules help organize an application into cohesive blocks of functionality.

This page answers the questions many developers ask about NgModule design and implementation.

## What classes should I add to the [declarations](#) array?

Add [declarable](#) classes—components, directives, and pipes—to a [declarations](#) list.

Declare these classes in exactly one module of the application. Declare them in a module if they belong to that particular module.

---

## What is a declarable?

Declarables are the class types—components, directives, and pipes—that you can add to a module's [declarations](#) list. They're the only classes that you can add to [declarations](#).

---

## What classes should I not add to [declarations](#)?

Add only [declarable](#) classes to an NgModule's [declarations](#) list.

Do not declare the following:

- A class that's already declared in another module, whether an app module, @NgModule, or third-party module.
  - An array of directives imported from another module. For example, don't declare FORMS\_DIRECTIVES from @angular/forms because the [FormsModule](#) already declares it.
  - Module classes.
  - Service classes.
  - Non-Angular classes and objects, such as strings, numbers, functions, entity models, configurations, business logic, and helper classes.
- 

## Why list the same component in multiple [NgModule](#) properties?

AppComponent is often listed in both [declarations](#) and [bootstrap](#). You might see the same component listed in [declarations](#), [exports](#), and entryComponents.

While that seems redundant, these properties have different functions. Membership in one list doesn't imply membership in another list.

- AppComponent could be declared in this module but not bootstrapped.
  - AppComponent could be bootstrapped in this module but declared in a different feature module.
  - A component could be imported from another app module (so you can't declare it) and re-exported by this module.
  - A component could be exported for inclusion in an external component's template as well as dynamically loaded in a pop-up dialog.
- 

## What does "Can't bind to 'x' since it isn't a known property of 'y'" mean?

This error often means that you haven't declared the directive "x" or haven't imported the NgModule to which "x" belongs.

Perhaps you declared "x" in an application sub-module but forgot to export it. The "x" class isn't visible to other modules until you add it to the [exports](#) list.

---

## What should I import?

Import NgModules whose public (exported) [declarable classes](#) you need to reference in this module's component templates.

This always means importing [CommonModule](#) from @angular/common for access to the Angular directives such as [NgIf](#) and NgFor. You can import it directly or from another NgModule that [re-exports](#) it.

Import [FormsModule](#) from @angular/forms if your components have [(ngModel)] two-way binding expressions.

Import shared and feature modules when this module's components incorporate their components, directives, and pipes.

Import only [BrowserModule](#) in the root AppModule.

---

## Should I import [BrowserModule](#) or [CommonModule](#)?

The root application module, AppModule, of almost every browser application should import [BrowserModule](#) from @angular/platform-browser.

[BrowserModule](#) provides services that are essential to launch and run a browser app.

[BrowserModule](#) also re-exports [CommonModule](#) from @angular/common, which means that components in the AppModule module also have access to the Angular directives every app needs, such as [NgIf](#) and NgFor.

Do not import [BrowserModule](#) in any other module. Feature modules and lazy-loaded modules should import [CommonModule](#) instead. They need the common directives. They don't need to re-install the app-wide providers.

Importing [CommonModule](#) also frees feature modules for use on any target platform, not just browsers.

---

## What if I import the same module twice?

That's not a problem. When three modules all import Module 'A', Angular evaluates Module 'A' once, the first time it encounters it, and doesn't do so again.

That's true at whatever level A appears in a hierarchy of imported NgModules. When Module 'B' imports Module 'A', Module 'C' imports 'B', and Module 'D' imports [C, B, A], then 'D' triggers the evaluation of 'C', which triggers the evaluation of 'B', which evaluates 'A'. When Angular gets to the 'B' and 'A' in 'D', they're already cached and ready to go.

Angular doesn't like NgModules with circular references, so don't let Module 'A' import Module 'B', which imports Module 'A'.

---

## What should I export?

Export [declarable](#) classes that components in other NgModules are able to reference in their templates. These are your public classes. If you don't export a declarable class, it stays private, visible only to other components declared in this NgModule.

You can export any declarable class—components, directives, and pipes—whether it's declared in this NgModule or in an imported NgModule.

You can re-export entire imported NgModules, which effectively re-exports all of their exported classes. An NgModule can even export a module that it doesn't import.

---

## What should I not export?

Don't export the following:

- Private components, directives, and pipes that you need only within components declared in this NgModule. If you don't want another NgModule to see it, don't export it.
  - Non-declarable objects such as services, functions, configurations, and entity models.
  - Components that are only loaded dynamically by the router or by bootstrapping. Such [entry components](#) can never be selected in another component's template. While there's no harm in exporting them, there's also no benefit.
  - Pure service modules that don't have public (exported) declarations. For example, there's no point in re-exporting [HttpClientModule](#) because it doesn't export anything. Its only purpose is to add http service providers to the application as a whole.
- 

## Can I re-export classes and modules?

Absolutely.

NgModules are a great way to selectively aggregate classes from other NgModules and re-export them in a consolidated, convenience module.

An NgModule can re-export entire NgModules, which effectively re-exports all of their exported classes. Angular's own [BrowserModule](#) exports a couple of NgModules like this:

`exports: [CommonModule, ApplicationModule]`

An NgModule can export a combination of its own declarations, selected imported classes, and imported NgModules.

Don't bother re-exporting pure service modules. Pure service modules don't export [declarable](#) classes that another NgModule could use. For example, there's no point in re-exporting [HttpClientModule](#) because it doesn't export anything. Its only purpose is to add http service providers to the application as a whole.

---

## What is the [forRoot\(\)](#) method?

The [forRoot\(\)](#) static method is a convention that makes it easy for developers to configure services and providers that are intended to be singletons. A good example of [forRoot\(\)](#) is the [RouterModule.forRoot\(\)](#) method.

Apps pass a [Routes](#) object to [RouterModule.forRoot\(\)](#) in order to configure the app-wide [Router](#) service with routes. [RouterModule.forRoot\(\)](#) returns a [ModuleWithProviders](#). You add that result to the [imports](#) list of the root AppModule.

Only call and import a .forRoot() result in the root application module, AppModule. Importing it in any other module, particularly in a lazy-loaded module, is contrary to the intent and will likely produce a runtime error. For more information, see [Singleton Services](#).

For a service, instead of using [forRoot\(\)](#), specify [providedIn](#): 'root' on the service's [@Injectable\(\)](#) decorator, which makes the service automatically available to the whole application and thus singleton by default.

[RouterModule](#) also offers a [forChild](#) static method for configuring the routes of lazy-loaded modules.

[forRoot\(\)](#) and [forChild\(\)](#) are conventional names for methods that configure services in root and feature modules respectively.

Angular doesn't recognize these names but Angular developers do. Follow this convention when you write similar modules with configurable service providers.

---

## Why is a service provided in a feature module visible everywhere?

Providers listed in the [@NgModule.providers](#) of a bootstrapped module have application scope. Adding a service provider to [@NgModule.providers](#) effectively publishes the service to the entire application.

When you import an NgModule, Angular adds the module's service providers (the contents of its providers list) to the application root injector.

This makes the provider visible to every class in the application that knows the provider's lookup token, or name.

Extensibility through NgModule imports is a primary goal of the NgModule system. Merging NgModule providers into the application injector makes it easy for a module library to enrich the entire application with new services. By adding the [HttpClientModule](#) once, every application component can make HTTP requests.

However, this might feel like an unwelcome surprise if you expect the module's services to be visible only to the components declared by that feature module. If the HeroModule provides the HeroService and the root AppModule imports HeroModule, any class that knows the HeroService type can inject that service, not just the classes declared in the HeroModule.

To limit access to a service, consider lazy loading the NgModule that provides that service. See [How do I restrict service scope to a module?](#) for more information.

---

## Why is a service provided in a lazy-loaded module visible only to that module?

Unlike providers of the modules loaded at launch, providers of lazy-loaded modules are module-scoped.

When the Angular router lazy-loads a module, it creates a new execution context. That [context has its own injector](#), which is a direct child of the application injector.

The router adds the lazy module's providers and the providers of its imported NgModules to this child injector.

These providers are insulated from changes to application providers with the same lookup token. When the router creates a component within the lazy-loaded context, Angular prefers service instances created from these providers to the service instances of the application root injector.

---

## What if two modules provide the same service?

When two imported modules, loaded at the same time, list a provider with the same token, the second module's provider "wins". That's because both providers are added to the same injector.

When Angular looks to inject a service for that token, it creates and delivers the instance created by the second provider.

Every class that injects this service gets the instance created by the second provider. Even classes declared within the first module get the instance created by the second provider.

If NgModule A provides a service for token 'X' and imports an NgModule B that also provides a service for token 'X', then NgModule A's service definition "wins".

The service provided by the root AppModule takes precedence over services provided by imported NgModules. The AppModule always wins.

---

## How do I restrict service scope to a module?

When a module is loaded at application launch, its [@NgModule.providers](#) have application-wide scope; that is, they are available for injection throughout the application.

Imported providers are easily replaced by providers from another imported NgModule. Such replacement might be by design. It could be unintentional and have adverse consequences.

As a general rule, import modules with providers exactly once, preferably in the application's root module. That's also usually the best place to configure, wrap, and override them.

Suppose a module requires a customized [HttpBackend](#) that adds a special header for all Http requests. If another module elsewhere in the application also customizes [HttpBackend](#) or merely imports the [HttpClientModule](#), it could override this module's [HttpBackend](#) provider, losing the special header. The server will reject http requests from this module.

To avoid this problem, import the [HttpClientModule](#) only in the AppModule, the application root module.

If you must guard against this kind of "provider corruption", don't rely on a launch-time module's providers.

Load the module lazily if you can. Angular gives a [lazy-loaded module](#) its own child injector. The module's providers are visible only within the component tree created with this injector.

If you must load the module eagerly, when the application starts, provide the service in a component instead.

Continuing with the same example, suppose the components of a module truly require a private, custom [HttpBackend](#).

Create a "top component" that acts as the root for all of the module's components. Add the custom [HttpBackend](#) provider to the top component's providers list rather than the module's providers. Recall that Angular creates a child injector for each component instance and populates the injector with the component's own providers.

When a child of this component asks for the [HttpBackend](#) service, Angular provides the local [HttpBackend](#) service, not the version provided in the application root injector. Child components make proper HTTP requests no matter what other modules do to [HttpBackend](#).

Be sure to create module components as children of this module's top component.

You can embed the child components in the top component's template. Alternatively, make the top component a routing host by giving it a <[router-outlet](#)>. Define child routes and let the router load module components into that outlet.

Though you can limit access to a service by providing it in a lazy loaded module or providing it in a component, providing services in a component can lead to multiple instances of those services. Thus, the lazy loading is preferable.

---

## Should I add application-wide providers to the root AppModule or the root AppComponent?

Define application-wide providers by specifying [providedIn](#): 'root' on its [@Injectable\(\)](#) decorator (in the case of services) or at [InjectionToken](#) construction (in the case where tokens are provided). Providers that are created this way automatically are made available to the entire application and don't need to be listed in any module.

If a provider cannot be configured in this way (perhaps because it has no sensible default value), then register application-wide providers in the root AppModule, not in the AppComponent.

Lazy-loaded modules and their components can inject AppModule services; they can't inject AppComponent services.

Register a service in AppComponent providers only if the service must be hidden from components outside the AppComponent tree. This is a rare use case.

More generally, [prefer registering providers in NgModules](#) to registering in components.

## Discussion

Angular registers all startup module providers with the application root injector. The services that root injector providers create have application scope, which means they are available to the entire application.

Certain services, such as the [Router](#), only work when you register them in the application root injector.

By contrast, Angular registers AppComponent providers with the AppComponent's own injector. AppComponent services are available only to that component and its component tree. They have component scope.

The AppComponent's injector is a child of the root injector, one down in the injector hierarchy. For applications that don't use the router, that's almost the entire application. But in routed applications, routing operates at the root level where AppComponent services don't exist. This means that lazy-loaded modules can't reach them.

---

## Should I add other providers to a module or a component?

Providers should be configured using `@Injectable` syntax. If possible, they should be provided in the application root (`providedIn: 'root'`). Services that are configured this way are lazily loaded if they are only used from a lazily loaded context.

If it's the consumer's decision whether a provider is available application-wide or not, then register providers in modules (`@NgModule.providers`) instead of registering in components (`@Component.providers`).

Register a provider with a component when you must limit the scope of a service instance to that component and its component tree. Apply the same reasoning to registering a provider with a directive.

For example, an editing component that needs a private copy of a caching service should register the service with the component. Then each new instance of the component gets its own cached service instance. The changes that editor makes in its service don't touch the instances elsewhere in the application.

[Always register application-wide services with the root AppModule](#), not the root AppComponent.

---

## Why is it bad if a shared module provides a service to a lazy-loaded module?

### The eagerly loaded scenario

When an eagerly loaded module provides a service, for example a `UserService`, that service is available application-wide. If the root module provides `UserService` and imports another module that provides the same `UserService`, Angular registers one of them in the root app injector (see [What if I import the same module twice?](#)).

Then, when some component injects `UserService`, Angular finds it in the app root injector, and delivers the app-wide singleton service. No problem.

### The lazy loaded scenario

Now consider a lazy loaded module that also provides a service called `UserService`.

When the router lazy loads a module, it creates a child injector and registers the `UserService` provider with that child injector. The child injector is not the root injector.

When Angular creates a lazy component for that module and injects `UserService`, it finds a `UserService` provider in the lazy module's child injector and creates a new instance of the `UserService`. This is an entirely different `UserService` instance than the app-wide singleton version that Angular injected in one of the eagerly loaded components.

This scenario causes your app to create a new instance every time, instead of using the singleton.

---

## Why does lazy loading create a child injector?

Angular adds `@NgModule.providers` to the application root injector, unless the NgModule is lazy-loaded. For a lazy-loaded NgModule, Angular creates a child injector and adds the module's providers to the child injector.

This means that an NgModule behaves differently depending on whether it's loaded during application start or lazy-loaded later. Neglecting that difference can lead to [adverse consequences](#).

Why doesn't Angular add lazy-loaded providers to the app root injector as it does for eagerly loaded NgModules?

The answer is grounded in a fundamental characteristic of the Angular dependency-injection system. An injector can add providers until it's first used. Once an injector starts creating and delivering services, its provider list is frozen; no new providers are allowed.

When an application starts, Angular first configures the root injector with the providers of all eagerly loaded NgModules before creating its first component and injecting any of the provided services. Once the application begins, the app root injector is closed to new providers.

Time passes and application logic triggers lazy loading of an NgModule. Angular must add the lazy-loaded module's providers to an injector somewhere. It can't add them to the app root injector because that injector is closed to new providers. So Angular creates a new child injector for the lazy-loaded module context.

---

## How can I tell if an NgModule or service was previously loaded?

Some NgModules and their services should be loaded only once by the root AppModule. Importing the module a second time by lazy loading a module could [produce errant behavior](#) that may be difficult to detect and diagnose.

To prevent this issue, write a constructor that attempts to inject the module or service from the root app injector. If the injection succeeds, the class has been loaded a second time. You can throw an error or take other remedial action.

Certain NgModules, such as `BrowserModule`, implement such a guard. Here is a custom constructor for an NgModule called CoreModule.

`src/app/core/core.module.ts` (Constructor)

```
constructor (@Optional() @SkipSelf() parentModule: CoreModule) {  
  if (parentModule) {  
    throw new Error(  
      'CoreModule is already loaded. Import it in the AppModule only');  
  }  
}
```

---

## What is an entry component?

An entry component is any component that Angular loads imperatively by type.

A component loaded declaratively via its selector is not an entry component.

Angular loads a component declaratively when using the component's selector to locate the element in the template. Angular then creates the HTML representation of the component and inserts it into the DOM at the selected element. These aren't entry components.

The bootstrapped root AppComponent is an entry component. True, its selector matches an element tag in index.html. But index.html isn't a component template and the AppComponent selector doesn't match an element in any component template.

Components in route definitions are also entry components. A route definition refers to a component by its type. The router ignores a routed component's selector, if it even has one, and loads the component dynamically into a [RouterOutlet](#).

For more information, see [Entry Components](#).

---

## What's the difference between a bootstrap component and an entry component?

A bootstrapped component is an [entry component](#) that Angular loads into the DOM during the bootstrap process (application launch). Other entry components are loaded dynamically by other means, such as with the router.

The [@NgModule.bootstrap](#) property tells the compiler that this is an entry component and it should generate code to bootstrap the application with this component.

There's no need to list a component in both the [bootstrap](#) and entryComponents lists, although doing so is harmless.

For more information, see [Entry Components](#).

---

## When do I add components to entryComponents?

Most application developers won't need to add components to the entryComponents.

Angular adds certain components to entry components automatically. Components listed in `@NgModule.bootstrap` are added automatically. Components referenced in router configuration are added automatically. These two mechanisms account for almost all entry components.

If your app happens to bootstrap or dynamically load a component by type in some other manner, you must add it to entryComponents explicitly.

Although it's harmless to add components to this list, it's best to add only the components that are truly entry components. Don't include components that [are referenced](#) in the templates of other components.

For more information, see [Entry Components](#).

---

## Why does Angular need entryComponents?

The reason is tree shaking. For production apps you want to load the smallest, fastest code possible. The code should contain only the classes that you actually need. It should exclude a component that's never used, whether or not that component is declared.

In fact, many libraries declare and export components you'll never use. If you don't reference them, the tree shaker drops these components from the final code package.

If the [Angular compiler](#) generated code for every declared component, it would defeat the purpose of the tree shaker.

Instead, the compiler adopts a recursive strategy that generates code only for the components you use.

The compiler starts with the entry components, then it generates code for the declared components it [finds](#) in an entry component's template, then for the declared components it discovers in the templates of previously compiled components, and so on. At the end of the process, the compiler has generated code for every entry component and every component reachable from an entry component.

If a component isn't an entry component or wasn't found in a template, the compiler omits it.

---

# What kinds of modules should I have and how should I use them?

Every app is different. Developers have various levels of experience and comfort with the available choices. Some suggestions and guidelines appear to have wide appeal.

## SharedModule

SharedModule is a conventional name for an [NgModule](#) with the components, directives, and pipes that you use everywhere in your app. This module should consist entirely of [declarations](#), most of them exported.

The SharedModule may re-export other widget modules, such as [CommonModule](#), [FormsModule](#), and NgModules with the UI controls that you use most widely.

The SharedModule should not have providers for reasons [explained previously](#). Nor should any of its imported or re-exported modules have providers.

Import the SharedModule in your feature modules, both those loaded when the app starts and those you lazy load later.

## CoreModule

CoreModule is a conventional name for an [NgModule](#) with providers for the singleton services you load when the application starts.

Import CoreModule in the root AppModule only. Never import CoreModule in any other module.

Consider making CoreModule a pure services module with no [declarations](#).

For more information, see [Sharing NgModules](#) and [Singleton Services](#).

## Feature Modules

Feature modules are modules you create around specific application business domains, user workflows, and utility collections. They support your app by containing a particular feature, such as routes, services, widgets, etc. To conceptualize what a feature module might be in your app, consider that if you would put the files related to a certain functionality, like a search, in one folder, that the contents of that folder would be a feature module that you might call your SearchModule. It would contain all of the components, routing, and templates that would make up the search functionality.

For more information, see [Feature Modules](#) and [Module Types](#)

## What's the difference between NgModules and JavaScript Modules?

In an Angular app, NgModules and JavaScript modules work together.

In modern JavaScript, every file is a module (see the [Modules](#) page of the Exploring ES6 website). Within each file you write an export statement to make parts of the module public.

An Angular NgModule is a class with the `@NgModule` decorator—JavaScript modules don't have to have the `@NgModule` decorator. Angular's `NgModule` has [imports](#) and [exports](#) and they serve a similar purpose.

You import other NgModules so you can use their exported classes in component templates. You export this NgModule's classes so they can be imported and used by components of other NgModules.

For more information, see [JavaScript Modules vs. NgModules](#).

---

## How does Angular find components, directives, and pipes in a template?

### What is a template reference?

The [Angular compiler](#) looks inside component templates for other components, directives, and pipes. When it finds one, that's a template reference.

The Angular compiler finds a component or directive in a template when it can match the selector of that component or directive to some HTML in that template.

The compiler finds a pipe if the pipe's name appears within the pipe syntax of the template HTML.

Angular only matches selectors and pipe names for classes that are declared by this module or exported by a module that this module imports.

---

### What is the Angular compiler?

The Angular compiler converts the application code you write into highly performant JavaScript code. The `@NgModule` metadata plays an important role in guiding the compilation process.

The code you write isn't immediately executable. For example, components have templates that contain custom elements, attribute directives, Angular binding declarations, and some peculiar syntax that clearly isn't native HTML.

The Angular compiler reads the template markup, combines it with the corresponding component class code, and emits component factories.

A component factory creates a pure, 100% JavaScript representation of the component that incorporates everything described in its [@Component](#) metadata: the HTML, the binding instructions, the attached styles.

Because directives and pipes appear in component templates, the Angular compiler incorporates them into compiled component code too.

[@NgModule](#) metadata tells the Angular compiler what components to compile for this module and how to link this module with other modules.

Dependency injection (DI), is an important application design pattern. Angular has its own DI framework, which is typically used in the design of Angular applications to increase their efficiency and modularity.

Dependencies are services or objects that a class needs to perform its function. DI is a coding pattern in which a class asks for dependencies from external sources rather than creating them itself.

In Angular, the DI framework provides declared dependencies to a class when that class is instantiated. This guide explains how DI works in Angular, and how you use it to make your apps flexible, efficient, and robust, as well as testable and maintainable.

You can run the [live example](#) / [download example](#) of the sample app that accompanies this guide.

Start by reviewing this simplified version of the heroes feature from the [The Tour of Heroes](#). This simple version doesn't use DI; we'll walk through converting it to do so.

src/app/heroes/heroes.component.ts

src/app/heroes/hero-list.component.ts

src/app/heroes/hero.ts

src/app/heroes/mock-heroes.ts

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

```
@Component({
  selector: 'app-heroes',
  template: `
    <h2>Heroes</h2>
    <app-hero-list></app-hero-list>
  `
})
export class HeroesComponent {}
```

HeroesComponent is the top-level heroes component. Its only purpose is to display HeroListComponent, which displays a list of hero names.

This version of the HeroListComponent gets heroes from the HEROES array, an in-memory collection defined in a separate mock-heroes file.

```
src/app/heroes/hero-list.component.ts (class)
```

```
export class HeroListComponent {  
  heroes = HEROES;  
}
```

This approach works for prototyping, but is not robust or maintainable. As soon as you try to test this component or get heroes from a remote server, you have to change the implementation of HeroesListComponent and replace every use of the HEROES mock data.

## Create and register an injectable service

The DI framework lets you supply data to a component from an injectable service class, defined in its own file. To demonstrate, we'll create an injectable service class that provides a list of heroes, and register that class as a provider of that service.

Having multiple classes in the same file can be confusing. We generally recommend 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, you get a run-time null reference error.

It is possible to define the component first with the help of the [forwardRef\(\)](#) method as explained in this [blog post](#).

You can also use forward references to break circular dependencies. See an example in the [DI Cookbook](#).

### Create an injectable service class

The [Angular CLI](#) can generate a new HeroService class in the src/app/heroes folder with this command.

```
ng generate service heroes/hero
```

The command creates the following HeroService skeleton.

```
src/app/heroes/hero.service.ts (CLI-generated)
```

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

```
@Injectable({  
  providedIn: 'root',
```

```
})
export class HeroService {
  constructor() { }
}
```

The `@Injectable()` is an essential ingredient in every Angular service definition. The rest of the class has been written to expose a `getHeroes` method that returns the same mock data as before. (A real app would probably get its data asynchronously from a remote server, but we'll ignore that to focus on the mechanics of injecting the service.)

src/app/heroes/hero.service.ts

```
import { Injectable } from '@angular/core';
import { HEROES } from './mock-heroes';
```

```
@Injectable({
  // we declare that this service should be created
  // by the root application injector.
  providedIn: 'root',
})
export class HeroService {
  getHeroes() { return HEROES; }
}
```

## Configure an injector with a service provider

The class we have created provides a service. The `@Injectable()` decorator marks it as a service that can be injected, but Angular can't actually inject it anywhere until you configure an Angular [dependency injector](#) with a [provider](#) of that service.

The injector is responsible for creating service instances and injecting them into classes like `HeroListComponent`.

You rarely create an Angular injector yourself. Angular creates injectors for you as it executes the app, starting with the root injector that it creates during the [bootstrap process](#).

A provider tells an injector how to create the service. You must configure an injector with a provider before that injector can create a service (or provide any other kind of dependency).

A provider can be the service class itself, so that the injector can use new to create an instance. You might also define more than one class to provide the same service in different ways, and configure different injectors with different providers.

Injectors are inherited, which means that if a given injector can't resolve a dependency, it asks the parent injector to resolve it.

A component can get services from its own injector, from the injectors of its component ancestors, from the injector of its parent NgModule, or from the root injector.

- Learn more about the [different kinds of providers](#).
- Learn more about how the [injector hierarchy](#) works.

You can configure injectors with providers at different levels of your app, by setting a metadata value in one of three places:

- In the [@Injectable\(\)](#) decorator for the service itself.
- In the [@NgModule\(\)](#) decorator for an NgModule.
- In the [@Component\(\)](#) decorator for a component.

The [@Injectable\(\)](#) decorator has the [providedIn](#) metadata option, where you can specify the provider of the decorated service class with the root injector, or with the injector for a specific NgModule.

The [@NgModule\(\)](#) and [@Component\(\)](#) decorators have the providers metadata option, where you can configure providers for NgModule-level or component-level injectors.

Components are directives, and the providers option is inherited from [@Directive\(\)](#). You can also configure providers for directives and pipes at the same level as the component.

Learn more about [where to configure providers](#).

## Injecting services

In order for HeroListComponent to get heroes from HeroService, it needs to ask for HeroService to be injected, rather than creating its own HeroService instance with new.

You can tell Angular to inject a dependency in a component's constructor by specifying a constructor parameter with the dependency type. Here's the HeroListComponent constructor, asking for the HeroService to be injected.

src/app/heroes/hero-list.component (constructor signature)

```
constructor(heroService: HeroService)
```

Of course, HeroListComponent should do something with the injected HeroService. Here's the revised component, making use of the injected service, side-by-side with the previous version for comparison.

hero-list.component (with DI)

hero-list.component (without DI)

1. import { [Component](#) } from '@angular/core';
2. import { Hero } from './hero';
3. import { HeroService } from './hero.service';
- 4.
5. [@Component\({](#)
6. selector: 'app-hero-list',
7. [template: `](#)

```
8. <div *ngFor="let hero of heroes">
9.   {{hero.id}} - {{hero.name}}
10. </div>
11. `
12.})
13.export class HeroListComponent {
14. heroes: Hero[];
15.
16. constructor(heroService: HeroService) {
17.   this.heroes = heroService.getHeroes();
18. }
19.}
```

HeroService must be provided in some parent injector. The code in HeroListComponent doesn't depend on where HeroService comes from. If you decided to provide HeroService in AppModule, HeroListComponent wouldn't change.

## Injector hierarchy and service instances

Services are singletons within the scope of an injector. That is, there is at most one instance of a service in a given injector.

There is only one root injector for an app. Providing UserService at the root or AppModule level means it is registered with the root injector. There is just one UserService instance in the entire app and every class that injects UserService gets this service instance unless you configure another provider with a child injector.

Angular DI has a [hierarchical injection system](#), which means that nested injectors can create their own service instances. Angular regularly creates nested injectors. Whenever Angular creates a new instance of a component that has providers specified in `@Component()`, it also creates a new child injector for that instance. Similarly, when a new NgModule is lazy-loaded at run time, Angular can create an injector for it with its own providers.

Child modules and component injectors are independent of each other, and create their own separate instances of the provided services. When Angular destroys an NgModule or component instance, it also destroys that injector and that injector's service instances.

Thanks to [injector inheritance](#), you can still inject application-wide services into these components. A component's injector is a child of its parent component's injector, and a descendent of its parent's parent's injector, and so on all the way back to the application's root injector. Angular can inject a service provided by any injector in that lineage.

For example, Angular can inject HeroListComponent with both the HeroService provided in HeroComponent and the UserService provided in AppModule.

## Testing components with dependencies

Designing a class with dependency injection makes the class easier to test. Listing dependencies as constructor parameters may be all you need to test application parts effectively.

For example, you can create a new HeroListComponent with a mock service that you can manipulate under test.

src/app/test.component.ts

```
const expectedHeroes = [{name: 'A'}, {name: 'B'}]
const mockService = <HeroService> {getHeroes: () => expectedHeroes }
```

```
it('should have heroes when HeroListComponent created', () => {
  // Pass the mock to the constructor as the Angular injector would
  const component = new HeroListComponent(mockService);
  expect(component.heroes.length).toEqual(expectedHeroes.length);
});
```

Learn more in the [Testing](#) guide.

## Services that need other services

Service can have their own dependencies. HeroService is very simple and doesn't have any dependencies of its own. Suppose, however, that you want it to report its activities through a logging service. You can apply the same constructor injection pattern, adding a constructor that takes a Logger parameter.

Here is the revised HeroService that injects Logger, side by side with the previous service for comparison.

src/app/heroes/hero.service (v2)

src/app/heroes/hero.service (v1)

src/app/logger.service

1. import {[Injectable](#)} from '@angular/core';
2. import { HEROES } from './mock-heroes';
3. import { Logger } from '../logger.service';
- 4.
5. [@Injectable\({](#)
6. [providedIn:](#) 'root',
7. [}\)](#)
8. export class HeroService {
- 9.
10. constructor(private logger: Logger) { }

```
11.  
12. getHeroes() {  
13.   this.logger.log('Getting heroes ...');  
14.   return HEROES;  
15. }  
16.}
```

The constructor asks for an injected instance of Logger and stores it in a private field called logger. The getHeroes() method logs a message when asked to fetch heroes.

Notice that the Logger service also has the [@Injectable\(\)](#) decorator, even though it might not need its own dependencies. In fact, the [@Injectable\(\)](#) decorator is required for all services.

When Angular creates a class whose constructor has parameters, it looks for type and injection metadata about those parameters so that it can inject the correct service. If Angular can't find that parameter information, it throws an error. Angular can only find the parameter information if the class has a decorator of some kind. The [@Injectable\(\)](#) decorator is the standard decorator for service classes.

The decorator requirement is imposed by TypeScript. TypeScript normally discards parameter type information when it [transpiles](#) the code to JavaScript. TypeScript preserves this information if the class has a decorator and the emitDecoratorMetadata compiler option is set true in TypeScript's tsconfig.json configuration file. The CLI configures tsconfig.json with emitDecoratorMetadata: true.

This means you're responsible for putting [@Injectable\(\)](#) on your service classes.

## Dependency injection tokens

When you configure an injector with a provider, you associate that provider with a [DI token](#). The injector maintains an internal token-provider map that it references when asked for a dependency. The token is the key to the map.

In simple examples, the dependency value is an instance, and the class type serves as its own lookup key. Here you get a HeroService directly from the injector by supplying the HeroService type as the token:

```
src/app/injector.component.ts
```

```
heroService: HeroService;
```

The behavior is similar when you write a constructor that requires an injected class-based dependency. 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
```

```
constructor(heroService: HeroService)
```

Many dependency values are provided by classes, but not all. The expanded provide object lets you associate different kinds of providers with a DI token.

- Learn more about [different kinds of providers](#).

## Optional dependencies

HeroService requires a logger, but what if it could get by without one?

When a component or service declares a dependency, the class constructor takes that dependency as a parameter. You can tell Angular that the dependency is optional by annotating the constructor parameter with `@Optional()`.

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

constructor(@Optional() private logger: Logger) {
  if (this.logger) {
    this.logger.log(some_message);
  }
}
```

When using `@Optional()`, your code must be prepared for a null value. If you don't register a logger provider anywhere, the injector sets the value of logger to null.

`@Inject()` and `@Optional()` are parameter decorators. They alter the way the DI framework provides a dependency, by annotating the dependency parameter on the constructor of the class that requires the dependency.

Learn more about parameter decorators in [Hierarchical Dependency Injectors](#).

## Summary

You learned the basics of Angular dependency injection in this page. You can register various kinds of providers, and you know how to ask for an injected object (such as a service) by adding a parameter to a constructor.

Dive deeper into the capabilities and advanced feature of the Angular DI system in the following pages:

- Learn more about nested injectors in [Hierarchical Dependency Injection](#).
- Learn more about [DI tokens and providers](#).
- [Dependency Injection in Action](#) is a cookbook for some of the interesting things you can do with DI.

The Angular dependency injection system is hierarchical. There is a tree of injectors that parallel an app's component tree. You can reconfigure the injectors at any level of that component tree.

This guide explores this system and how to use it to your advantage. It uses examples based on this [live example / download example](#).

## Where to configure providers

You can configure providers for different injectors in the injector hierarchy. An internal platform-level injector is shared by all running apps. The AppModule injector is the root of an app-wide injector hierarchy, and within an NgModule, directive-level injectors follow the structure of the component hierarchy.

The choices you make about where to configure providers lead to differences in the final bundle size, service scope, and service lifetime.

When you specify providers in the [@Injectable\(\)](#) decorator of the service itself (typically at the app root level), optimization tools such as those used by the CLI's production builds can perform tree shaking, which removes services that aren't used by your app. Tree shaking results in smaller bundle sizes.

- Learn more about [tree-shakable providers](#).

You're likely to inject UserService in many places throughout the app and will want to inject the same service instance every time. Providing UserService through the root injector is a good choice, and is the default that the CLI uses when you generate a service for your app.

### Platform injector

When you use [providedIn:'root'](#), you are configuring the root injector for the app, which is the injector for AppModule. The actual root of the entire injector hierarchy is a platform injector that is the parent of app-root injectors. This allows multiple apps to share a platform configuration. For example, a browser has only one URL bar, no matter how many apps you have running.

The platform injector is used internally during bootstrap, to configure platform-specific dependencies. You can configure additional platform-specific providers at the platform level by supplying extraProviders using the [platformBrowser\(\)](#) function.

Learn more about dependency resolution through the injector hierarchy: [What you always wanted to know about Angular Dependency Injection tree](#)

NgModule-level providers can be specified with [@NgModule\(\)](#) providers metadata option, or in the [@Injectable\(\)](#) [providedIn](#) option (with some module other than the root AppModule).

Use the [@NgModule\(\)](#) [provides](#) option if a module is [lazy loaded](#). The module's own injector is configured with the provider when that module is loaded, and Angular can inject the corresponding services in any class it creates in that module. If you use the [@Injectable\(\)](#) option [providedIn](#): MyLazyloadModule, the provider could be shaken out at compile time, if it is not used anywhere else in the app.

- Learn more about [tree-shakable providers](#).

For both root-level and module-level injectors, a service instance lives for the life of the app or module, and Angular injects this one service instance in every class that needs it.

Component-level providers configure each component instance's own injector. Angular can only inject the corresponding services in that component instance or one of its descendant component instances. Angular can't inject the same service instance anywhere else.

A component-provided service may have a limited lifetime. Each new instance of the component gets its own instance of the service. When the component instance is destroyed, so is that service instance.

In our sample app, HeroComponent is created when the application starts and is never destroyed, so the HeroService instance created for HeroComponent lives for the life of the app. If you want to restrict HeroService access to HeroComponent and its nested HeroListComponent, provide HeroService at the component level, in HeroComponent metadata.

- See more [examples of component-level injection](#) below.

## @Injectable-level configuration

The `@Injectable()` decorator identifies every service class. The `providedIn` metadata option for a service class configures a specific injector (typically root) to use the decorated class as a provider of the service. When an injectable class provides its own service to the root injector, the service is available anywhere the class is imported.

The following example configures a provider for HeroService using the `@Injectable()` decorator on the class.

```
src/app/heroes/heroes.service.ts
```

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

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

This configuration tells Angular that the app's root injector is responsible for creating an instance of HeroService by invoking its constructor, and for making that instance available across the application.

Providing a service with the app's root injector is a typical case, and the CLI sets up this kind of a provider automatically for you when generating a new service. However, you might not always want to provide your service at the root level. You might, for instance, want users to explicitly opt-in to using the service.

Instead of specifying the root injector, you can set `providedIn` to a specific NgModule.

For example, in the following excerpt, the `@Injectable()` decorator configures a provider that is available in any injector that includes the `HeroModule`.

src/app/heroes/hero.service.ts

```
import { Injectable } from '@angular/core';
import { HeroModule } from './hero.module';
import { HEROES } from './mock-heroes';

@Injectable({
  // we declare that this service should be created
  // by any injector that includes HeroModule.
  providedIn: HeroModule,
})
export class HeroService {
  getHeroes() { return HEROES; }
}
```

This is generally no different from configuring the injector of the `NgModule` itself, except that the service is tree-shakable if the `NgModule` doesn't use it. It can be useful for a library that offers a particular service that some components might want to inject optionally, and leave it up to the app whether to provide the service.

- Learn more about [tree-shakable providers](#).

## **@NgModule-level injectors**

You can configure a provider at the module level using the `providedIn` metadata option for a non-root `NgModule`, in order to limit the scope of the provider to that module. This is the equivalent of specifying the non-root module in the `@Injectable()` metadata, except that the service provided this way is not tree-shakable.

You generally don't need to specify `AppModule` with `providedIn`, because the app's root injector is the `AppModule` injector. However, if you configure a app-wide provider in the `@NgModule()` metadata for `AppModule`, it overrides one configured for root in the `@Injectable()` metadata. You can do this to configure a non-default provider of a service that is shared with multiple apps.

Here is an example of the case where the component router configuration includes a non-default [location strategy](#) by listing its provider in the providers list of the `AppModule`.

src/app/app.module.ts (providers)

```
providers: [
  { provide: LocationStrategy, useClass: HashLocationStrategy }
]
```

## @Component-level injectors

Individual components within an NgModule have their own injectors. You can limit the scope of a provider to a component and its children by configuring the provider at the component level using the [@Component](#) metadata.

The following example is a revised HeroesComponent that specifies HeroService in its providers array. HeroService can provide heroes to instances of this component, or to any child component instances.

```
src/app/heroes/heroes.component.ts

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

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

@Component({
  selector: 'app-heroes',
  providers: [ HeroService ],
  template: `
    <h2>Heroes</h2>
    <app-hero-list></app-hero-list>
  `
})
export class HeroesComponent { }
```

## Element injectors

An injector does not actually belong to a component, but rather to the component instance's anchor element in the DOM. A different component instance on a different DOM element uses a different injector.

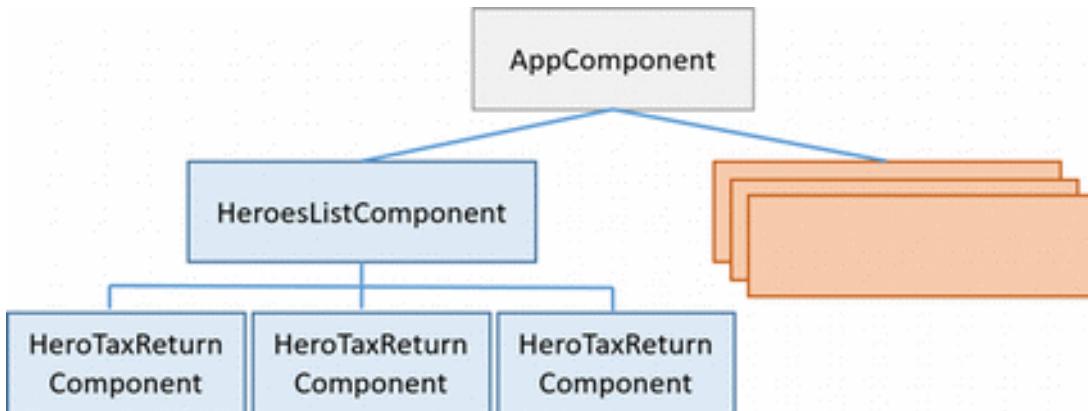
Components are a special type of directive, and the providers property of [@Component\(\)](#) is inherited from [@Directive\(\)](#). Directives can also have dependencies, and you can configure providers in their [@Directive\(\)](#) metadata. When you configure a provider for a component or directive using the providers property, that provider belongs to the injector for the anchor DOM element. Components and directives on the same element share an injector.

- Learn more about [Element Injectors in Angular](#).

## Injector bubbling

Consider this guide's variation on the Tour of Heroes application. At the top is the AppComponent which has some subcomponents, such as the HeroesListComponent. The HeroesListComponent holds and manages multiple instances of the HeroTaxReturnComponent. The following diagram represents the state of this three-level

component tree when there are three instances of HeroTaxReturnComponent open simultaneously.



When a component requests a dependency, Angular tries to satisfy that dependency with a provider registered in that component's own injector. If the component's injector lacks the provider, it passes the request up to its parent component's injector. If that injector can't satisfy the request, it passes the request along to the next parent injector up the tree. The requests keep bubbling up until Angular finds an injector that can handle the request or runs out of ancestor injectors. If it runs out of ancestors, Angular throws an error.

If you have registered a provider for the same DI token at different levels, the first one Angular encounters is the one it uses to provide the dependency. If, for example, a provider is registered locally in the component that needs a service, Angular doesn't look for another provider of the same service.

You can cap the bubbling by adding the `@Host()` parameter decorator on the dependant-service parameter in a component's constructor. The hunt for providers stops at the injector for the host element of the component.

- See an [example](#) of using `@Host` together with `@Optional`, another parameter decorator that lets you handle the null case if no provider is found.
- Learn more about the [@Host decorator and Element Injectors](#).

If you only register providers with the root injector at the top level (typically the root `AppModule`), the tree of injectors appears to be flat. All requests bubble up to the root injector, whether you configured it with the `bootstrapModule` method, or registered all providers with root in their own services.

## Component injectors

The ability to configure one or more providers at different levels opens up interesting and useful possibilities. The guide sample offers some scenarios where you might want to do so.

## Scenario: service isolation

Architectural reasons may lead you to restrict access to a service to the application domain where it belongs. For example, the guide sample includes a VillainsListComponent that displays a list of villains. It gets those villains from a VillainsService.

If you provide VillainsService in the root AppModule (where you registered the HeroesService), that would make the VillainsService available everywhere in the application, including the Hero workflows. If you later modified the VillainsService, you could break something in a hero component somewhere. Providing the service in the root AppModule creates that risk.

Instead, you can provide the VillainsService in the providers metadata of the VillainsListComponent like this:

```
src/app/villains-list.component.ts (metadata)
```

```
@Component({
  selector: 'app-villains-list',
  templateUrl: './villains-list.component.html',
  providers: [ VillainsService ]
})
```

By providing VillainsService in the VillainsListComponent metadata and nowhere else, the service becomes available only in the VillainsListComponent and its sub-component tree. It's still a singleton, but it's a singleton that exists solely in the villain domain.

Now you know that a hero component can't access it. You've reduced your exposure to error.

## Scenario: multiple edit sessions

Many applications allow users to work on several open tasks at the same time. For example, in a tax preparation application, the preparer could be working on several tax returns, switching from one to the other throughout the day.

This guide demonstrates that scenario with an example in the Tour of Heroes theme. Imagine an outer HeroListComponent that displays a list of super heroes.

To open a hero's tax return, the preparer clicks on a hero name, which opens a component for editing that return. Each selected hero tax return opens in its own component and multiple returns can be open at the same time.

Each tax return component has the following characteristics:

- Is its own tax return editing session.
- Can change a tax return without affecting a return in another component.
- Has the ability to save the changes to its tax return or cancel them.

# Hero Tax Returns

- RubberMan
- Tornado

Suppose that the HeroTaxReturnComponent has logic to manage and restore changes. That would be a pretty easy task for a simple hero tax return. In the real world, with a rich tax return data model, the change management would be tricky. You could delegate that management to a helper service, as this example does.

Here is the HeroTaxReturnService. It caches a single HeroTaxReturn, tracks changes to that return, and can save or restore it. It also delegates to the application-wide singleton HeroService, which it gets by injection.

src/app/hero-tax-return.service.ts

```
1. import { Injectable } from '@angular/core';
2. import { HeroTaxReturn } from './hero';
3. import { HeroesService } from './heroes.service';
4.
5. @Injectable()
6. export class HeroTaxReturnService {
7.   private currentTaxReturn: HeroTaxReturn;
8.   private originalTaxReturn: HeroTaxReturn;
9.
10. constructor(private heroService: HeroesService) { }
11.
12. set taxReturn (htr: HeroTaxReturn) {
13.   this.originalTaxReturn = htr;
```

```
14. this.currentTaxReturn = htr.clone();
15. }
16.
17. get taxReturn (): HeroTaxReturn {
18.   return this.currentTaxReturn;
19. }
20.
21. restoreTaxReturn() {
22.   this.taxReturn = this.originalTaxReturn;
23. }
24.
25. saveTaxReturn() {
26.   this.taxReturn = this.currentTaxReturn;
27.   this.heroService.saveTaxReturn(this.currentTaxReturn).subscribe();
28. }
29.}
```

Here is the HeroTaxReturnComponent that makes use of it.

src/app/hero-tax-return.component.ts

```
1. import { Component, EventEmitter, Input, Output } from '@angular/core';
2. import { HeroTaxReturn }      from './hero';
3. import { HeroTaxReturnService } from './hero-tax-return.service';
4.
5. @Component({
6.   selector: 'app-hero-tax-return',
7.   templateUrl: './hero-tax-return.component.html',
8.   styleUrls: [ './hero-tax-return.component.css' ],
9.   providers: [ HeroTaxReturnService ]
10.})
11.export class HeroTaxReturnComponent {
12.   message = "";
13.
14.   @Output() close = new EventEmitter<void>();
15.
16.   get taxReturn(): HeroTaxReturn {
17.     return this.heroTaxReturnService.taxReturn;
18.   }
19.
20.   @Input()
21.   set taxReturn (htr: HeroTaxReturn) {
22.     this.heroTaxReturnService.taxReturn = htr;
23.   }
24.
```

```
24.  
25. constructor(private heroTaxReturnService: HeroTaxReturnService) { }  
26.  
27. onCanceled() {  
28.   this.flashMessage('Canceled');  
29.   this.heroTaxReturnService.restoreTaxReturn();  
30. };  
31.  
32. onClose() { this.close.emit(); };  
33.  
34. onSaved() {  
35.   this.flashMessage('Saved');  
36.   this.heroTaxReturnService.saveTaxReturn();  
37. }  
38.  
39. flashMessage(msg: string) {  
40.   this.message = msg;  
41.   setTimeout(() => this.message = "", 500);  
42. }  
43.}
```

The tax-return-to-edit arrives via the input property which is implemented with getters and setters. The setter initializes the component's own instance of the HeroTaxReturnService with the incoming return. The getter always returns what that service says is the current state of the hero. The component also asks the service to save and restore this tax return.

This won't work if the service is an application-wide singleton. Every component would share the same service instance, and each component would overwrite the tax return that belonged to another hero.

To prevent this, we configure the component-level injector of HeroTaxReturnComponent to provide the service, using the providers property in the component metadata.

```
src/app/hero-tax-return.component.ts (providers)
```

```
providers: [ HeroTaxReturnService ]
```

The HeroTaxReturnComponent has its own provider of the HeroTaxReturnService. Recall that every component instance has its own injector. Providing the service at the component level ensures that every instance of the component gets its own, private instance of the service, and no tax return gets overwritten.

The rest of the scenario code relies on other Angular features and techniques that you can learn about elsewhere in the documentation. You can review it and download it from the [live example](#) / [download example](#).

## Scenario: specialized providers

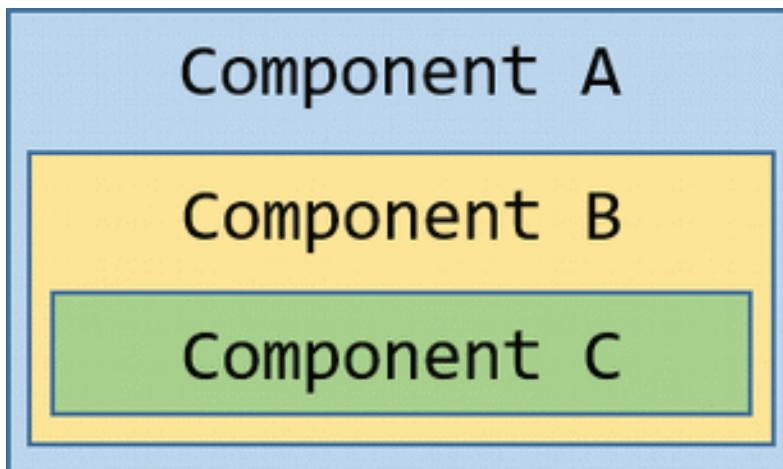
Another reason to re-provide a service at another level is to substitute a more specialized implementation of that service, deeper in the component tree.

Consider a Car component that depends on several services. Suppose you configured the root injector (marked as A) with generic providers for CarService, EngineService and TiresService.

You create a car component (A) that displays a car constructed from these three generic services.

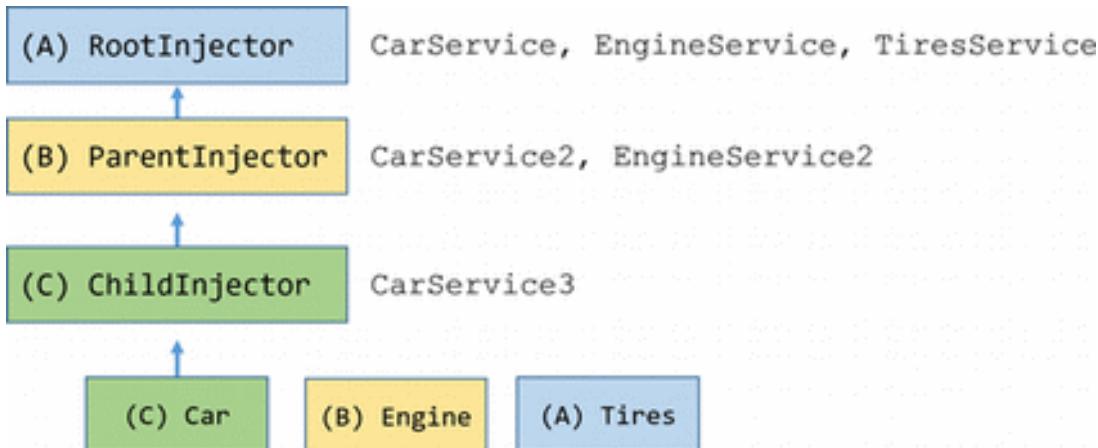
Then you create a child component (B) that defines its own, specialized providers for CarService and EngineService that have special capabilities suitable for whatever is going on in component (B).

Component (B) is the parent of another component (C) that defines its own, even more specialized provider for CarService.



Behind the scenes, each component sets up its own injector with zero, one, or more providers defined for that component itself.

When you resolve an instance of Car at the deepest component (C), its injector produces an instance of Car resolved by injector (C) with an Engine resolved by injector (B) and Tires resolved by the root injector (A).



The code for this cars scenario is in the `car.components.ts` and `car.services.ts` files of the sample which you can review and download from the [live example](#) / [download example](#).

A dependency [provider](#) configures an injector with a [DI token](#), which that injector uses to provide the concrete, runtime version of a dependency value. The injector relies on the provider configuration to create instances of the dependencies that it injects into components, directives, pipes, and other services.

You must configure an injector with a provider, or it won't know how to create the dependency.

The most obvious way for an injector to create an instance of a service class is with the class itself. If you specify the service class itself as the provider token, the default behavior is for the injector to instantiate that class with `new`.

In the following typical example, the `Logger` class itself provides a `Logger` instance.

`providers: [Logger]`

You can, however, configure an injector with an alternative provider, in order to deliver some other object that provides the needed logging functionality. For instance:

- You can provide a substitute class.
- You can provide a logger-like object.
- Your provider can call a logger factory function.

## The [Provider](#) object literal

The class-provider syntax is a shorthand expression that expands into a provider configuration, defined by the [Providerinterface](#). The following code snippets shows how a class that is given as the providers value is expanded into a full provider object.

`providers: [Logger]`

`[{ provide: Logger, useClass: Logger }]`

The expanded provider configuration is an object literal with two properties.

- The provide property holds the [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.

## Alternative class providers

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.

```
[{ provide: Logger, useClass: BetterLogger }]
```

## Class providers with dependencies

Another class, EvenBetterLogger, might display the user name in the log message. This logger gets the user from an injected UserService instance.

```
@Injectable()
export class EvenBetterLogger extends Logger {
  constructor(private userService: UserService) { super(); }

  log(message: string) {
    let 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. Configure this alternative logger with the useClass provider-definition key, like BetterLogger. The following array specifies both providers in the providers metadata option of the parent module or component.

```
[ UserService,
  { provide: Logger, useClass: EvenBetterLogger }]
```

## Aliased class providers

Suppose an old component depends upon the OldLogger class. OldLogger has the same interface as NewLogger, but for some reason you can't update the old component to use it.

When the old component logs a message with OldLogger, you want the singleton instance of NewLogger to handle it instead. In this case, the dependency injector should inject that singleton instance when a component asks for either the new or the old logger. OldLogger should be an alias for NewLogger.

If you try to alias OldLogger to NewLogger with useClass, you end up with two different NewLogger instances in your app.

```
[ NewLogger,  
  // Not aliased! Creates two instances of `NewLogger`  
  { provide: OldLogger, useClass: NewLogger}]
```

To make sure there is only one instance of NewLogger, alias OldLogger with the useExisting option.

```
[ NewLogger,  
  // Alias OldLogger w/ reference to NewLogger  
  { provide: OldLogger, useExisting: NewLogger}]
```

## Value providers

Sometimes it's easier to provide a ready-made object rather than ask the injector to create it from a class. To inject an object you have already created, configure the injector with the useValue option

The following code defines a variable that creates such an object to play the logger role.

```
// An object in the shape of the logger service  
export function SilentLoggerFn() {}  
  
const silentLogger = {  
  logs: ['Silent logger says "Shhhhh!". Provided via "useValue"'],  
  log: SilentLoggerFn  
};
```

The following provider object uses the useValue key to associate the variable with the Logger token.

```
[{ provide: Logger, useValue: silentLogger }]
```

## Non-class dependencies

Not all dependencies are classes. Sometimes you want to inject a string, function, or object.

Apps often define configuration objects with lots of small facts, like the title of the application or the address of a web API endpoint. These configuration objects aren't always instances of a class. They can be object literals, as shown in the following example.

```
src/app/app.config.ts (excerpt)  
  
export const HERO_DI_CONFIG: AppConfig = {  
  apiEndpoint: 'api.heroes.com',  
  title: 'Dependency Injection'  
};
```

TypeScript interfaces are not valid tokens

The HERO\_DI\_CONFIG constant conforms to the AppConfig interface. Unfortunately, you cannot use a TypeScript interface as a token. In TypeScript, an interface is a design-time artifact, and doesn't have a runtime representation (token) that the DI framework can use.

```
// FAIL! Can't use interface as provider token
[{ provide: AppConfig, useValue: HERO_DI_CONFIG }])

// FAIL! Can't inject using the interface as the parameter type
constructor(private config: AppConfig){ }
```

This might seem strange if you're used to dependency injection in strongly typed languages where an interface is the preferred dependency lookup key. However, JavaScript, doesn't have interfaces, so when TypeScript is transpiled to JavaScript, the interface disappears. There is no interface type information left for Angular to find at runtime.

One alternative is to provide and inject the configuration object in an NgModule like AppModule.

src/app/app.module.ts (providers)

```
providers: [
  UserService,
  { provide: APP_CONFIG, useValue: HERO_DI_CONFIG }
],
```

Another solution to choosing a provider token for non-class dependencies is to define and use an [InjectionToken](#) object. The following example shows how to define such a token.

src/app/app.config.ts

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

export const APP_CONFIG = new InjectionToken<AppConfig>('app.config');
```

The type parameter, while optional, conveys the dependency's type to developers and tooling. The token description is another developer aid.

Register the dependency provider using the [InjectionToken](#) object:

```
providers: [{ provide: APP_CONFIG, useValue: HERO_DI_CONFIG }]
```

Now you can inject the configuration object into any constructor that needs it, with the help of an [@Inject\(\)](#) parameter decorator.

src/app/app.component.ts

```
constructor(@Inject(APP_CONFIG) config: AppConfig) {
  this.title = config.title;
}
```

Although the AppConfig interface plays no role in dependency injection, it supports typing of the configuration object within the class.

## Factory providers

Sometimes you need to create a dependent value dynamically, based on information you won't have until run time. For example, you might need information that changes repeatedly in the course of the browser session. Also, your injectable service might not have independent access to the source of the information.

In cases like this you can use a factory provider. Factory providers can also be useful when creating an instance of a dependency from a third-party library that wasn't designed to work with DI.

For example, suppose HeroService must hide secret heroes from normal users. Only authorized users should see secret heroes.

Like EvenBetterLogger, HeroService needs to know if the user is authorized to see secret heroes. That authorization can change during the course of a single application session, as when you log in a different user.

Let's say you don't want to inject UserService directly into HeroService, because you don't want to complicate that service with security-sensitive information. HeroService won't have direct access to the user information to decide who is authorized and who isn't.

To resolve this, we give the HeroService constructor a boolean flag to control display of secret heroes.

```
src/app/heroes/hero.service.ts (excerpt)

constructor(
  private logger: Logger,
  private isAuthorized: boolean) { }

getHeroes() {
  let auth = this.isAuthorized ? 'authorized' : 'unauthorized';
  this.logger.log(`Getting heroes for ${auth} user.`);
  return HEROES.filter(hero => this.isAuthorized || !hero.isSecret);
}
```

You can inject Logger, but you can't inject the isAuthorized flag. Instead, you can use a factory provider to create a new logger instance for HeroService.

A factory provider needs a factory function.

```
src/app/heroes/hero.service.provider.ts (excerpt)
```

```
let heroServiceFactory = (logger: Logger, userService: UserService) => {
  return new HeroService(logger, userService.user.isAuthorized);
};
```

Although HeroService has no access to UserService, the factory function does. You inject both Logger and UserService into the factory provider and let the injector pass them along to the factory function.

src/app/heroes/hero.service.provider.ts (excerpt)

```
export let heroServiceProvider =
{ provide: HeroService,
  useFactory: heroServiceFactory,
  deps: [Logger, UserService]
};
```

- The useFactory field tells Angular that the provider is a factory function whose implementation is heroServiceFactory.
- The deps property is an array of provider tokens. 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 factory function parameters.

Notice that you captured the factory provider in an exported variable, heroServiceProvider. This extra step makes the factory provider reusable. You can configure a provider of HeroService with this variable wherever you need it. In this sample, you need it only in HeroesComponent, where heroServiceProvider replaces HeroService in the metadata providers array.

The following shows the new and the old implementations side-by-side.

src/app/heroes/heroes.component (v3)

src/app/heroes/heroes.component (v2)

```
1. import { Component }      from '@angular/core';
2. import { heroServiceProvider } from './hero.service.provider';
3.
4. @Component({
5.   selector: 'app-heroes',
6.   providers: [ heroServiceProvider ],
7.   template: `
8.     <h2>Heroes</h2>
9.     <app-hero-list></app-hero-list>
10.    `
11.  })
12. export class HeroesComponent { }
```

## Predefined tokens and multiple providers

Angular provides a number of built-in injection-token constants that you can use to customize the behavior of various systems.

For example, you can use the following built-in tokens as hooks into the framework's bootstrapping and initialization process.

A provider object can associate any of these injection tokens with one or more callback functions that take app-specific initialization actions.

- [PLATFORM\\_INITIALIZER](#): Callback is invoked when a platform is initialized.
- [APP\\_BOOTSTRAP\\_LISTENER](#): Callback is invoked for each component that is bootstrapped. The handler function receives the ComponentRef instance of the bootstrapped component.
- [APP\\_INITIALIZER](#): Callback is invoked before an app is initialized. All registered initializers can optionally return a Promise. All initializer functions that return Promises must be resolved before the application is bootstrapped. If one of the initializers fails to resolve, the application is not bootstrapped.

The provider object can have a third option, multi: true, which you can use with [APP\\_INITIALIZER](#) to register multiple handlers for the provide event.

For example, when bootstrapping an application, you can register many initializers using the same token.

```
export const APP_TOKENS = [
  { provide: PLATFORM\_INITIALIZER, useFactory: platformInitialized, multi: true },
  { provide: APP\_INITIALIZER, useFactory: delayBootstrapping, multi: true },
  { provide: APP\_BOOTSTRAP\_LISTENER, useFactory: appBootstrapped, multi: true },
];
```

Multiple providers can be associated with a single token in other areas as well. For example, you can register a custom form validator using the built-in [NG\\_VALIDATORS](#) token, and provide multiple instances of a given validator provider by using the multi: true property in the provider object. Angular adds your custom validators to the existing collection.

The Router also makes use of multiple providers associated with a single token. When you provide multiple sets of routes using [RouterModule.forRoot](#) and [RouterModule.forChild](#) in a single module, the [ROUTES](#) token combines all the different provided sets of routes into a single value.

Constants in API documentation to find more built-in tokens.

## Tree-shakable providers

Tree shaking refers to a compiler option that removes code from the final bundle if that code is not referenced in an application. When providers are tree-shakable, the Angular compiler

removes the associated services from the final output when it determines that they are not used in your application. This significantly reduces the size of your bundles.

Ideally, if an application isn't injecting a service, it shouldn't be included in the final output. However, Angular has to be able to identify at build time whether the service will be required or not. Because it's always possible to inject a service directly using injector.get(Service), Angular can't identify all of the places in your code where this injection could happen, so it has no choice but to include the service in the injector. Thus, services provided at the NgModule or component level are not tree-shakable.

The following example of non-tree-shakable providers in Angular configures a service provider for the injector of an NgModule.

src/app/tree-shaking/service-and-modules.ts

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

```
@Injectable()
export class Service {
  doSomething(): void {
  }
}
```

```
@NgModule({
  providers: [Service],
})
export class ServiceModule { }
```

This module can then be imported into your application module to make the service available for injection in your app, as shown in the following example.

src/app/tree-shaking/app.modules.ts

```
@NgModule({
  imports: [
    BrowserModule,
    RouterModule.forRoot([]),
    ServiceModule,
  ],
})
export class AppModule { }
```

When ngc runs, it compiles AppModule into a module factory, which contains definitions for all the providers declared in all the modules it includes. At runtime, this factory becomes an injector that instantiates these services.

Tree-shaking doesn't work here because Angular can't decide to exclude one chunk of code (the provider definition for the service within the module factory) based on whether another chunk of code (the service class) is used. To make services tree-shakable, the information about how to construct an instance of the service (the provider definition) needs to be a part of the service class itself.

## Creating tree-shakable providers

You can make a provider tree-shakable by specifying it in the [@Injectable\(\)](#) decorator on the service itself, rather than in the metadata for the NgModule or component that depends on the service.

The following example shows the tree-shakable equivalent to the ServiceModule example above.

```
src/app/tree-shaking/service.ts
```

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

```
}
```

The service can be instantiated by configuring a factory function, as in the following example.

```
src/app/tree-shaking/service.0.ts
```

```
@Injectable({
  providedIn: 'root',
  useFactory: () => new Service('dependency'),
})
export class Service {
  constructor(private dep: string) {
  }
}
```

To override a tree-shakable provider, configure the injector of a specific NgModule or component with another provider, using the providers: [] array syntax of the [@NgModule\(\)](#) or [@Component\(\)](#) decorator.

This section explores many of the features of dependency injection (DI) in Angular.

See the [live example](#) / [download example](#) of the code in this cookbook.

## Nested service dependencies

The consumer of an injected service doesn't need to know how to create that service. It's the job of the DI framework to create and cache dependencies. The consumer just needs to let the DI framework know which dependencies it needs.

Sometimes a service depends on other services, which may depend on yet other services. The dependency injection framework resolves these nested dependencies in the correct order. At each step, the consumer of dependencies declares what it requires in its constructor, and lets the framework provide them.

The following example shows that AppComponent declares its dependence on LoggerService and UserContext.

src/app/app.component.ts

```
constructor(logger: LoggerService, public userContext: UserContextService) {  
  userContext.loadUser(this.userId);  
  logger.logInfo('AppComponent initialized');  
}
```

UserContext in turn depends on both LoggerService and UserService, another service that gathers information about a particular user.

user-context.service.ts (injection)

```
@Injectable({  
  providedIn: 'root'  
})  
export class UserContextService {  
  constructor(private userService: UserService, private loggerService: LoggerService) {  
  }  
}
```

When Angular creates AppComponent, the DI framework creates an instance of LoggerService and starts to create UserContextService. UserContextService also needs LoggerService, which the framework already has, so the framework can provide the same instance. UserContextService also needs UserService, which the framework has yet to create. UserService has no further dependencies, so the framework can simply use new to instantiate the class and provide the instance to the UserContextService constructor.

The parent AppComponent doesn't need to know about the dependencies of dependencies. Declare what's needed in the constructor (in this case LoggerService and UserContextService) and the framework resolves the nested dependencies.

When all dependencies are in place, AppComponent displays the user information.

## Logged in user

Name: Bombasto  
Role: Admin

## Limit service scope to a component subtree

An Angular application has multiple injectors, arranged in a tree hierarchy that parallels the component tree. Each injector creates a singleton instance of a dependency. That same instance is injected wherever that injector provides that service. A particular service can be provided and created at any level of the injector hierarchy, which means that there can be multiple instances of a service if it is provided by multiple injectors.

Dependencies provided by the root injector can be injected into any component anywhere in the application. In some cases, you might want to restrict service availability to a particular region of the application. For instance, you might want to let users explicitly opt in to use a service, rather than letting the root injector provide it automatically.

You can limit the scope of an injected service to a branch of the application hierarchy by providing that service at the sub-root component for that branch. This example shows how to make a different instance of HeroService available to HeroesBaseComponent by adding it to the providers array of the [@Component\(\)](#) decorator of the sub-component.

src/app/sorted-heroes.component.ts (HeroesBaseComponent excerpt)

```
@Component({
  selector: 'app-unsorted-heroes',
  template: `<div *ngFor="let hero of heroes">{{hero.name}}</div>`,
  providers: [HeroService]
})
export class HeroesBaseComponent implements OnInit {
  constructor(private heroService: HeroService) { }
}
```

When Angular creates HeroesBaseComponent, it also creates a new instance of HeroService that is visible only to that component and its children, if any.

You could also provide HeroService to a different component elsewhere in the application. That would result in a different instance of the service, living in a different injector.

Examples of such scoped HeroService singletons appear throughout the accompanying sample code, including HeroBiosComponent, HeroOfTheMonthComponent, and HeroesBaseComponent. Each of these components has its own HeroService instance managing its own independent collection of heroes.

## Multiple service instances (sandboxing)

Sometimes you want multiple instances of a service at the same level of the component hierarchy.

A good example is a service that holds state for its companion component instance. You need a separate instance of the service for each component. Each service has its own work-state, isolated from the service-and-state of a different component. This is called sandboxing because each service and component instance has its own sandbox to play in.

In this example, HeroBiosComponent presents three instances of HeroBioComponent.

ap/hero-bios.component.ts

```
@Component({
  selector: 'app-hero-bios',
  template: `
    <app-hero-bio [heroid]="1"></app-hero-bio>
    <app-hero-bio [heroid]="2"></app-hero-bio>
    <app-hero-bio [heroid]="3"></app-hero-bio>`,
  providers: [HeroService]
})
export class HeroBiosComponent {
```

Each HeroBioComponent can edit a single hero's biography. HeroBioComponent relies on HeroCacheService to fetch, cache, and perform other persistence operations on that hero.

src/app/hero-cache.service.ts

1. `@Injectable()`
2. `export class HeroCacheService {`
3.  `hero: Hero;`
4.  `constructor(private heroService: HeroService) {}`
5.
6.  `fetchCachedHero(id: number) {`
7.  `if (!this.hero) {`
8.  `this.hero = this.heroService.getHeroById(id);`
9.  `}`
10.  `return this.hero;`
11. `}`
12. `}`

Three instances of HeroBioComponent can't share the same instance of HeroCacheService, as they'd be competing with each other to determine which hero to cache.

Instead, each HeroBioComponent gets its own HeroCacheService instance by listing HeroCacheService in its metadata providersarray.

src/app/hero-bio.component.ts

```
1. @Component({
2.   selector: 'app-hero-bio',
3.   template: `
4.     <h4>{{hero.name}}</h4>
5.     <ng-content></ng-content>
6.     <textarea cols="25" [(ngModel)]="hero.description"></textarea>`,
7.   providers: [HeroCacheService]
8. })
9.
10.export class HeroBioComponent implements OnInit {
11. @Input() heroid: number;
12.
13. constructor(private heroCache: HeroCacheService) { }
14.
15. ngOnInit() { this.heroCache.fetchCachedHero(this.heroid); }
16.
17. get hero() { return this.heroCache.hero; }
18.}
```

The parent HeroBiosComponent binds a value to heroid. ngOnInit passes that ID to the service, which fetches and caches the hero. The getter for the hero property pulls the cached hero from the service. The template displays this data-bound property.

Find this example in [live code](#) / [download example](#) and confirm that the three HeroBioComponent instances have their own cached hero data.

Hero Bios

RubberMan

Hero of many talents

Magma

Hero of all trades

Mr. Nice

The name says it all

## Qualify dependency lookup with parameter decorators

When a class requires a dependency, that dependency is added to the constructor as a parameter. When Angular needs to instantiate the class, it calls upon the DI framework to supply the dependency. By default, the DI framework searches for a provider in the injector hierarchy, starting at the component's local injector of the component, and if necessary bubbling up through the injector tree until it reaches the root injector.

- The first injector configured with a provider supplies the dependency (a service instance or value) to the constructor.
- If no provider is found in the root injector, the DI framework returns null to the constructor.

There are a number of options for modifying the default search behavior, using parameter decorators on the service-valued parameters of a class constructor.

### Make a dependency [@Optional](#) and limit search with [@Host](#)

Dependencies can be registered at any level in the component hierarchy. When a component requests a dependency, Angular starts with that component's injector and walks up the injector tree until it finds the first suitable provider.

Angular throws an error if it can't find the dependency during that walk.

In some cases, you need to limit the search or accommodate a missing dependency. You can modify Angular's search behavior with the [@Host](#) and [@Optional](#) qualifying decorators on a service-valued parameter of the component's constructor.

- The `@Optional` property decorator tells Angular to return null when it can't find the dependency.
- The `@Host` property decorator stops the upward search at the host component. The host component is typically the component requesting the dependency. However, when this component is projected into a parent component, that parent component becomes the host. The following example covers this second case.

These decorators can be used individually or together, as shown in the example. This `HeroBiosAndContactsComponent` is a revision of `HeroBiosComponent` which you looked at [above](#).

`src/app/hero-bios.component.ts (HeroBiosAndContactsComponent)`

```

1. @Component({
2.   selector: 'app-hero-bios-and-contacts',
3.   template: `
4.     <app-hero-bio [heroid]="1"> <app-hero-contact></app-hero-contact> </app-hero-
    bio>
5.     <app-hero-bio [heroid]="2"> <app-hero-contact></app-hero-contact> </app-hero-
    bio>
6.     <app-hero-bio [heroid]="3"> <app-hero-contact></app-hero-contact> </app-hero-
    bio>`,
7.   providers: [HeroService]
8. })
9. export class HeroBiosAndContactsComponent {
10.   constructor(logger: LoggerService) {
11.     logger.logInfo('Creating HeroBiosAndContactsComponent');
12.   }
13.}

```

Focus on the template:

`dependency-injection-in-action/src/app/hero-bios.component.ts`

`template: ``  
`<app-hero-bio [heroid]="1"> <app-hero-contact></app-hero-contact> </app-hero-bio>`  
`<app-hero-bio [heroid]="2"> <app-hero-contact></app-hero-contact> </app-hero-bio>`  
`<app-hero-bio [heroid]="3"> <app-hero-contact></app-hero-contact> </app-hero-bio>`,`

Now there's a new `<hero-contact>` element between the `<hero-bio>` tags. Angular projects, or transcludes, the corresponding `HeroContactComponent` into the `HeroBioComponent` view, placing it in the `<ng-content>` slot of the `HeroBioComponent` template.

`src/app/hero-bio.component.ts (template)`

`template: ``  
`<h4>{{hero.name}}</h4>`

```
<ng-content></ng-content>
<textarea cols="25" [(ngModel)]="hero.description"></textarea>,
```

The result is shown below, with the hero's telephone number from HeroContactComponent projected above the hero description.

RubberMan

Phone #: 123-456-7899

Hero of many talents

Here's HeroContactComponent, which demonstrates the qualifying decorators.

src/app/hero-contact.component.ts

```
1. @Component({
2.   selector: 'app-hero-contact',
3.   template: `
4.     <div>Phone #: {{phoneNumber}}
5.     <span *ngIf="hasLogger">!!!</span></div>
6.   })
7. export class HeroContactComponent {
8.
9.   hasLogger = false;
10.
11. constructor(
12.   @Host() // limit to the host component's instance of the HeroCacheService
13.   private heroCache: HeroCacheService,
14.
15.   @Host() // limit search for logger; hides the application-wide logger
16.   @Optional() // ok if the logger doesn't exist
17.   private loggerService: LoggerService
18. ) {
19.   if (loggerService) {
20.     this.hasLogger = true;
21.     loggerService.logInfo('HeroContactComponent can log!');
22.   }
23. }
24.
25. get phoneNumber() { return this.heroCache.hero.phone; }
26.
27.}
```

Focus on the constructor parameters.

src/app/hero-contact.component.ts

```
@Host() // limit to the host component's instance of the HeroCacheService  
private heroCache: HeroCacheService,
```

```
@Host() // limit search for logger; hides the application-wide logger  
@Optional() // ok if the logger doesn't exist  
private loggerService: LoggerService
```

The `@Host()` function decorating the `heroCache` constructor property ensures that you get a reference to the cache service from the parent `HeroBioComponent`. Angular throws an error if the parent lacks that service, even if a component higher in the component tree includes it.

A second `@Host()` function decorates the `loggerService` constructor property. The only `LoggerService` instance in the app is provided at the `AppComponent` level. The host `HeroBioComponent` doesn't have its own `LoggerService` provider.

Angular throws an error if you haven't also decorated the property with `@Optional()`. When the property is marked as optional, Angular sets `loggerService` to null and the rest of the component adapts.

Here's `HeroBiosAndContactsComponent` in action.

## Hero Bios and Contacts

### RubberMan

Phone #: 123-456-7899

Hero of many talents

### Magma

Phone #: 555-555-5555

Hero of all trades

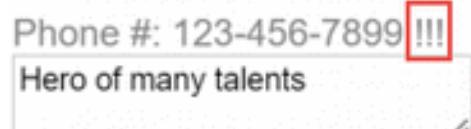
### Mr. Nice

Phone #: 111-222-3333

The name says it all

If you comment out the `@Host()` decorator, Angular walks up the injector ancestor tree until it finds the logger at the AppComponent level. The logger logic kicks in and the hero display updates with the "!!!" marker to indicate that the logger was found.

## RubberMan



If you restore the `@Host()` decorator and comment out `@Optional`, the app throws an exception when it cannot find the required logger at the host component level.

EXCEPTION: No provider for LoggerService! (HeroContactComponent -> LoggerService)

## Supply a custom provider with `@Inject`

Using a custom provider allows you to provide a concrete implementation for implicit dependencies, such as built-in browser APIs. The following example uses an `InjectionToken` to provide the `localStorage` browser API as a dependency in the `BrowserStorageService`.

src/app/storage.service.ts

```
1. import {Inject, Injectable, InjectionToken} from '@angular/core';
2.
3. export const BROWSER_STORAGE = new InjectionToken<Storage>('Browser
   Storage', {
4.   providedIn: 'root',
5.   factory: () => localStorage
6. });
7.
8. @Injectable({
9.   providedIn: 'root'
10.})
11.export class BrowserStorageService {
12. constructor(@Inject(BROWSER_STORAGE) public storage: Storage) {}
13.
14. get(key: string) {
15.   this.storage.getItem(key);
16. }
17.
18. set(key: string, value: string) {
19.   this.storage.setItem(key, value);
20. }
21.
```

```

22. remove(key: string) {
23.   this.storage.removeItem(key);
24. }
25.
26. clear() {
27.   this.storage.clear();
28. }
29.}

```

The [factory](#) function returns the localStorage property that is attached to the browser window object. The [Inject](#) decorator is a constructor parameter used to specify a custom provider of a dependency. This custom provider can now be overridden during testing with a mock API of localStorage instead of interactive with real browser APIs.

## Modify the provider search with [@Self](#) and [@SkipSelf](#)

Providers can also be scoped by injector through constructor parameter decorators. The following example overrides the BROWSER\_STORAGE token in the [Component](#) class providers with the sessionStorage browser API. The same BrowserStorageService is injected twice in the constructor, decorated with [@Self](#) and [@SkipSelf](#) to define which injector handles the provider dependency.

src/app/storage.component.ts

```

1. import { Component, OnInit, Self, SkipSelf } from '@angular/core';
2. import { BROWSER_STORAGE, BrowserStorageService } from './storage.service';
3.
4. @Component({
5.   selector: 'app-storage',
6.   template: `
7.     Open the inspector to see the local/session storage keys:
8.
9.     <h3>Session Storage</h3>
10.    <button (click)="setSession()">Set Session Storage</button>
11.
12.    <h3>Local Storage</h3>
13.    <button (click)="setLocal()">Set Local Storage</button>
14.  `,
15.   providers: [
16.     BrowserStorageService,
17.     { provide: BROWSER_STORAGE, useFactory: () => sessionStorage }
18.   ]
19. })
20. export class StorageComponent implements OnInit {
21.

```

```

22. constructor(
23.   @Self\(\) private sessionStorageService: BrowserStorageService,
24.   @SkipSelf\(\) private localStorageService: BrowserStorageService,
25. ) { }
26.
27. ngOnInit() {
28. }
29.
30. setSession() {
31.   this.sessionStorageService.set('hero', 'Mr. Nice - Session');
32. }
33.
34. setLocal() {
35.   this.localStorageService.set('hero', 'Mr. Nice - Local');
36. }
37.}

```

Using the [@Self](#) decorator, the injector only looks at the component's injector for its providers. The [@SkipSelf](#) decorator allows you to skip the local injector and look up in the hierarchy to find a provider that satisfies this dependency. The sessionStorageService instance interacts with the BrowserStorageService using the sessionStorage browser API, while the localStorageService skips the local injector and uses the root BrowserStorageService that uses the localStorage browser API.

## Inject the component's DOM element

Although developers strive to avoid it, many visual effects and third-party tools, such as jQuery, require DOM access. As a result, you might need to access a component's DOM element.

To illustrate, here's a simplified version of HighlightDirective from the [Attribute Directives](#) page.

src/app/highlight.directive.ts

```

1. import { Directive, ElementRef, HostListener, Input } from '@angular/core';
2.
3. @Directive({
4.   selector: '[appHighlight]'
5. })
6. export class HighlightDirective {
7.
8.   @Input('appHighlight') highlightColor: string;
9.
10. private el: HTMLElement;

```

```
11.  
12. constructor(el: ElementRef) {  
13.   this.el = el.nativeElement;  
14. }  
15.  
16. @HostListener('mouseenter') onMouseEnter() {  
17.   this.highlight(this.highlightColor || 'cyan');  
18. }  
19.  
20. @HostListener('mouseleave') onMouseLeave() {  
21.   this.highlight(null);  
22. }  
23.  
24. private highlight(color: string) {  
25.   this.el.style.backgroundColor = color;  
26. }  
27.}
```

The directive sets the background to a highlight color when the user mouses over the DOM element to which the directive is applied.

Angular sets the constructor's el parameter to the injected [ElementRef](#). (An [ElementRef](#) is a wrapper around a DOM element, whose nativeElement property exposes the DOM element for the directive to manipulate.)

The sample code applies the directive's myHighlight attribute to two <div> tags, first without a value (yielding the default color) and then with an assigned color value.

src/app/app.component.html (highlight)

```
<div id="highlight" class="di-component" appHighlight>  
  <h3>Hero Bios and Contacts</h3>  
  <div appHighlight="yellow">  
    <app-hero-bios-and-contacts></app-hero-bios-and-contacts>  
  </div>  
</div>
```

The following image shows the effect of mousing over the <hero-bios-and-contacts> tag.

## Hero Bios and Contacts

RubberMan

Phone #: 123-456-7899

Hero of many talents

## Define dependencies with providers

This section demonstrates how to write providers that deliver dependent services.

In order to get a service from a dependency injector, you have to give it a [token](#). Angular usually handles this transaction by specifying a constructor parameter and its type. The parameter type serves as the injector lookup token. Angular passes this token to the injector and assigns the result to the parameter.

The following is a typical example.

```
src/app/hero-bios.component.ts (component constructor injection)
```

```
constructor(logger: LoggerService) {  
  logger.logInfo('Creating HeroBiosComponent');  
}
```

Angular asks the injector for the service associated with LoggerService and assigns the returned value to the loggerparameter.

If the injector has already cached an instance of the service associated with the token, it provides that instance. If it doesn't, it needs to make one using the provider associated with the token.

If the injector doesn't have a provider for a requested token, it delegates the request to its parent injector, where the process repeats until there are no more injectors. If the search fails, the injector throws an error—unless the request was [optional](#).

A new injector has no providers. Angular initializes the injectors it creates with a set of preferred providers. You have to configure providers for your own app-specific dependencies.

### Defining providers

A dependency can't always be created by the default method of instantiating a class. You learned about some other methods in [Dependency Providers](#). The following

HeroOfTheMonthComponent example demonstrates many of the alternatives and why you need them. It's visually simple: a few properties and the logs produced by a logger.

## Hero of the Month

Winner: **Magma**

Reason for award: **Had a great month!**

Runners-up: **RubberMan, Mr. Nice**

Logs:

INFO: starting up at Fri Apr 01 2016  
23:31:10 GMT-0700 (Pacific Daylight Time)

The code behind it customizes how and where the DI framework provides dependencies. The use cases illustrate different ways to use the [provide object literal](#) to associate a definition object with a DI token.

hero-of-the-month.component.ts

```
1. import { Component, Inject } from '@angular/core';
2.
3. import { DateLoggerService } from './date-logger.service';
4. import { Hero }      from './hero';
5. import { HeroService }   from './hero.service';
6. import { LoggerService }  from './logger.service';
7. import { MinimalLogger }   from './minimal-logger.service';
8. import { RUNNERS_UP,
9.         runnersUpFactory } from './runners-up';
10.
11.@Component{
12. selector: 'app-hero-of-the-month',
13. templateUrl: './hero-of-the-month.component.html',
14. providers: [
15.   { provide: Hero,      useValue: someHero },
16.   { provide: TITLE,     useValue: 'Hero of the Month' },
17.   { provide: HeroService, useClass: HeroService },
18.   { provide: LoggerService, useClass: DateLoggerService },
19.   { provide: MinimalLogger, useExisting: LoggerService },
20.   { provide: RUNNERS_UP,  useFactory:
21.     runnersUpFactory(2), deps: [Hero, HeroService] }
22. ]
23.})
24.export class HeroOfTheMonthComponent {
25. logs: string[] = [];
```

```

26. constructor(
27.   logger: MinimalLogger,
28.   public heroOfTheMonth: Hero,
29.   @Inject(RUNNERS_UP) public runnersUp: string,
30.   @Inject(TITLE) public title: string)
31. {
32.   this.logs = logger.logs;
33.   logger.logInfo('starting up');
34. }
35.)

```

The providers array shows how you might use the different provider-definition keys; `useValue`, `useClass`, `useExisting`, or `useFactory`.

### Value providers: `useValue`

The `useValue` key lets you associate a fixed value with a DI token. Use this technique to provide runtime configuration constants such as website base addresses and feature flags. You can also use a value provider in a unit test to provide mock data in place of a production data service.

The `HeroOfTheMonthComponent` example has two value providers.

`dependency-injection-in-action/src/app/hero-of-the-month.component.ts`

```
{
  provide: Hero,      useValue: someHero },
  { provide: TITLE,    useValue: 'Hero of the Month' },
```

- The first provides an existing instance of the `Hero` class to use for the `Hero` token, rather than requiring the injector to create a new instance with `new` or use its own cached instance. Here, the token is the class itself.
- The second specifies a literal string resource to use for the `TITLE` token. The `TITLE` provider token is not a class, but is instead a special kind of provider lookup key called an [injection token](#), represented by an [InjectionToken](#) instance.

You can use an injection token for any kind of provider but it's particularly helpful when the dependency is a simple value like a string, a number, or a function.

The value of a value provider must be defined before you specify it here. The title string literal is immediately available. The `someHero` variable in this example was set earlier in the file as shown below. You can't use a variable whose value will be defined later.

`dependency-injection-in-action/src/app/hero-of-the-month.component.ts`

```
const someHero = new Hero(42, 'Magma', 'Had a great month!', '555-555-5555');
```

Other types of providers can create their values lazily; that is, when they're needed for injection.

## Class providers: useClass

The useClass provider key lets you create and return a new instance of the specified class.

You can use this type of provider to substitute an alternative implementation for a common or default class. The alternative implementation could, for example, implement a different strategy, extend the default class, or emulate the behavior of the real class in a test case.

The following code shows two examples in HeroOfTheMonthComponent.

dependency-injection-in-action/src/app/hero-of-the-month.component.ts

```
{ provide: HeroService, useClass: HeroService },  
{ provide: LoggerService, useClass: DateLoggerService },
```

The first provider is the de-sugared, expanded form of the most typical case in which the class to be created (HeroService) is also the provider's dependency injection token. The short form is generally preferred; this long form makes the details explicit.

The second provider substitutes DateLoggerService for LoggerService. LoggerService is already registered at the AppComponent level. When this child component requests LoggerService, it receives a DateLoggerService instance instead.

This component and its tree of child components receive DateLoggerService instance. Components outside the tree continue to receive the original LoggerService instance.

DateLoggerService inherits from LoggerService; it appends the current date/time to each message:

src/app/date-logger.service.ts

```
@Injectable({  
  providedIn: 'root'  
})  
export class DateLoggerService extends LoggerService  
{  
  logInfo(msg: any) { super.logInfo(stamp(msg)); }  
  logDebug(msg: any) { super.logInfo(stamp(msg)); }  
  logError(msg: any) { super.logError(stamp(msg)); }  
}  
  
function stamp(msg: any) { return msg + ' at ' + new Date(); }
```

## Alias providers: useExisting

The useExisting provider key lets you map one token to another. In effect, the first token is an alias for the service associated with the second token, creating two ways to access the same service object.

dependency-injection-in-action/src/app/hero-of-the-month.component.ts

```
{ provide: MinimalLogger, useExisting: LoggerService },
```

You can use this technique to narrow an API through an aliasing interface. The following example shows an alias introduced for that purpose.

Imagine that LoggerService had a large API, much larger than the actual three methods and a property. You might want to shrink that API surface to just the members you actually need. In this example, the MinimalLogger [class-interface](#) reduces the API to two members:

src/app/minimal-logger.service.ts

```
// Class used as a "narrowing" interface that exposes a minimal logger
// Other members of the actual implementation are invisible
export abstract class MinimalLogger {
  logs: string[];
  logInfo: (msg: string) => void;
}
```

The following example puts MinimalLogger to use in a simplified version of HeroOfTheMonthComponent.

src/app/hero-of-the-month.component.ts (minimal version)

```
@Component({
  selector: 'app-hero-of-the-month',
  templateUrl: './hero-of-the-month.component.html',
  // TODO: move this aliasing, `useExisting` provider to the AppModule
  providers: [{ provide: MinimalLogger, useExisting: LoggerService }]
})
export class HeroOfTheMonthComponent {
  logs: string[] = [];
  constructor(logger: MinimalLogger) {
    logger.logInfo('starting up');
  }
}
```

The HeroOfTheMonthComponent constructor's logger parameter is typed as MinimalLogger, so only the logs and logInfomembers are visible in a TypeScript-aware editor.

```
this.logs = logger.logs;
logger.logInfo('sta logInfo (method) MinimalLogger.logInfo(msg: string): void
logs
```

Behind the scenes, Angular sets the logger parameter to the full service registered under the LoggingService token, which happens to be the DateLoggerService instance that was [provided above](#).

This is illustrated in the following image, which displays the logging date.

```
INFO: starting up at Fri Apr 01 2016  
23:31:10 GMT-0700 (Pacific Daylight Time)
```

### Factory providers: `useFactory`

The `useFactory` provider key lets you create a dependency object by calling a factory function, as in the following example.

`dependency-injection-in-action/src/app/hero-of-the-month.component.ts`

```
{ provide: RUNNERS_UP, useFactory: runnersUpFactory(2), deps: [Hero, HeroService] }
```

The injector provides the dependency value by invoking a factory function, that you provide as the value of the `useFactory` key. Notice that this form of provider has a third key, `deps`, which specifies dependencies for the `useFactory` function.

Use this technique to create a dependency object with a factory function whose inputs are a combination of injected services and local state.

The dependency object (returned by the factory function) is typically a class instance, but can be other things as well. In this example, the dependency object is a string of the names of the runners up to the "Hero of the Month" contest.

In the example, the local state is the number 2, the number of runners up that the component should show. The state value is passed as an argument to `runnersUpFactory()`. The `runnersUpFactory()` returns the provider factory function, which can use both the passed-in state value and the injected services `Hero` and `HeroService`.

`runners-up.ts` (excerpt)

```
export function runnersUpFactory(take: number) {  
  return (winner: Hero, heroService: HeroService): string => {  
    /* ... */  
  };  
};
```

The provider factory function (returned by `runnersUpFactory()`) returns the actual dependency object, the string of names.

- The function takes a winning `Hero` and a `HeroService` as arguments. Angular supplies these arguments from injected values identified by the two tokens in the `deps` array.
- The function returns the string of names, which Angular then injects into the `runnersUp` parameter of `HeroOfTheMonthComponent`.

The function retrieves candidate heroes from the `HeroService`, takes 2 of them to be the runners-up, and returns their concatenated names. Look at the [live example](#) / [download example](#) for the full source code.

# Provider token alternatives: class interface and 'InjectionToken'

Angular dependency injection is easiest when the provider token is a class that is also the type of the returned dependency object, or service.

However, a token doesn't have to be a class and even when it is a class, it doesn't have to be the same type as the returned object. That's the subject of the next section.

## Class interface

The previous Hero of the Month example used the MinimalLogger class as the token for a provider of LoggerService.

dependency-injection-in-action/src/app/hero-of-the-month.component.ts

```
{ provide: MinimalLogger, useExisting: LoggerService },
```

MinimalLogger is an abstract class.

dependency-injection-in-action/src/app/minimal-logger.service.ts

```
// Class used as a "narrowing" interface that exposes a minimal logger
// Other members of the actual implementation are invisible
export abstract class MinimalLogger {
  logs: string[];
  logInfo: (msg: string) => void;
}
```

An abstract class is usually a base class that you can extend. In this app, however there is no class that inherits from MinimalLogger. The LoggerService and the DateLoggerService could have inherited from MinimalLogger, or they could have implemented it instead, in the manner of an interface. But they did neither. MinimalLogger is used only as a dependency injection token.

When you use a class this way, it's called a class interface.

As mentioned in [DI Providers](#), an interface is not a valid DI token because it is a TypeScript artifact that doesn't exist at run time. Use this abstract class interface to get the strong typing of an interface, and also use it as a provider token in the way you would a normal class.

A class interface should define only the members that its consumers are allowed to call. Such a narrowing interface helps decouple the concrete class from its consumers.

Using a class as an interface gives you the characteristics of an interface in a real JavaScript object. To minimize memory cost, however, the class should have no implementation. The MinimalLogger transpiles to this unoptimized, pre-minified JavaScript for a constructor function.

dependency-injection-in-action/src/app/minimal-logger.service.ts

```
var MinimalLogger = (function () {
  function MinimalLogger() {}
  return MinimalLogger;
}());
exports("MinimalLogger", MinimalLogger);
```

Notice that it doesn't have any members. It never grows no matter how many members you add to the class, as long as those members are typed but not implemented.

Look again at the TypeScript MinimalLogger class to confirm that it has no implementation.

## '[InjectionToken](#)' objects

Dependency objects can be simple values like dates, numbers and strings, or shapeless objects like arrays and functions.

Such objects don't have application interfaces and therefore aren't well represented by a class. They're better represented by a token that is both unique and symbolic, a JavaScript object that has a friendly name but won't conflict with another token that happens to have the same name.

[InjectionToken](#) has these characteristics. You encountered them twice in the Hero of the Month example, in the title value provider and in the runnersUp factory provider.

dependency-injection-in-action/src/app/hero-of-the-month.component.ts

```
{ provide: TITLE,      useValue: 'Hero of the Month' },
{ provide: RUNNERS_UP, useFactory: runnersUpFactory(2), deps: [Hero, HeroService] }
```

You created the TITLE token like this:

dependency-injection-in-action/src/app/hero-of-the-month.component.ts

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

```
export const TITLE = new InjectionToken<string>('title');
```

The type parameter, while optional, conveys the dependency's type to developers and tooling. The token description is another developer aid.

## Inject into a derived class

Take care when writing a component that inherits from another component. If the base component has injected dependencies, you must re-provide and re-inject them in the derived class and then pass them down to the base class through the constructor.

In this contrived example, SortedHeroesComponent inherits from HeroesBaseComponent to display a sorted list of heroes.

## Sorted Heroes

Magma  
Mr. Nice  
RubberMan

The HeroesBaseComponent can stand on its own. It demands its own instance of HeroService to get heroes and displays them in the order they arrive from the database.

src/app/sorted-heroes.component.ts (HeroesBaseComponent)

```
1. @Component({
2.   selector: 'app-unsorted-heroes',
3.   template: `<div *ngFor="let hero of heroes">{{hero.name}}</div>`,
4.   providers: [HeroService]
5. })
6. export class HeroesBaseComponent implements OnInit {
7.   constructor(private heroService: HeroService) { }
8.
9.   heroes: Array<Hero>;
10.
11. ngOnInit() {
12.   this.heroes = this.heroService.getAllHeroes();
13.   this.afterGetHeroes();
14. }
15.
16. // Post-process heroes in derived class override.
17. protected afterGetHeroes() {}
18.
19.}
```

## Keep constructors simple

Constructors should do little more than initialize variables. This rule makes the component safe to construct under test without fear that it will do something dramatic like talk to the server. That's why you call the HeroService from within the ngOnInit rather than the constructor.

Users want to see the heroes in alphabetical order. Rather than modify the original component, sub-class it and create a SortedHeroesComponent that sorts the heroes before presenting them. The SortedHeroesComponent lets the base class fetch the heroes.

Unfortunately, Angular cannot inject the HeroService directly into the base class. You must provide the HeroService again for this component, then pass it down to the base class inside the constructor.

src/app/sorted-heroes.component.ts (SortedHeroesComponent)

```
1. @Component({
2.   selector: 'app-sorted-heroes',
3.   template: `<div *ngFor="let hero of heroes">{{hero.name}}</div>`,
4.   providers: [HeroService]
5. })
6. export class SortedHeroesComponent extends HeroesBaseComponent {
7.   constructor(heroService: HeroService) {
8.     super(heroService);
9.   }
10.
11. protected afterGetHeroes() {
12.   this.heroes = this.heroes.sort((h1, h2) => {
13.     return h1.name < h2.name ? -1 :
14.       (h1.name > h2.name ? 1 : 0);
15.   });
16. }
17.}
```

Now take note of the `afterGetHeroes()` method. Your first instinct might have been to create an `ngOnInit` method in `SortedHeroesComponent` and do the sorting there. But Angular calls the derived class's `ngOnInit` before calling the base class's `ngOnInit` so you'd be sorting the `heroes` array before they arrived. That produces a nasty error.

Overriding the base class's `afterGetHeroes()` method solves the problem.

These complications argue for avoiding component inheritance.

## Break circularities with a forward class reference (`forwardRef`)

The order of class declaration matters in TypeScript. You can't refer directly to a class until it's been defined.

This isn't usually a problem, especially if you adhere to the recommended one class per file rule. But sometimes circular references are unavoidable. You're in a bind when class 'A' refers to class 'B' and 'B' refers to 'A'. One of them has to be defined first.

The Angular `forwardRef()` function creates an indirect reference that Angular can resolve later.

The Parent Finder sample is full of circular class references that are impossible to break.

You face this dilemma when a class makes a reference to itself as does `AlexComponent` in its providers array. The `providersarray` is a property of the `@Component()` decorator function which must appear above the class definition.

Break the circularity with `forwardRef`.

```
parent-finder.component.ts (AlexComponent providers)
```

```
providers: [{ provide: Parent, useExisting: forwardRef(() => AlexComponent) }],
```

Application components often need to share information. You can often use loosely coupled techniques for sharing information, such as data binding and service sharing, but sometimes it makes sense for one component to have a direct reference to another component. You need a direct reference, for instance, to access values or call methods on that component.

Obtaining a component reference is a bit tricky in Angular. Angular components themselves do not have a tree that you can inspect or navigate programmatically. The parent-child relationship is indirect, established through the components' [view objects](#).

Each component has a host view, and can have additional embedded views. An embedded view in component A is the host view of component B, which can in turn have embedded view. This means that there is a [view hierarchy](#) for each component, of which that component's host view is the root.

There is an API for navigating down the view hierarchy. Check out [Query](#), [QueryList](#), [ViewChildren](#), and [ContentChildren](#) in the [API Reference](#).

There is no public API for acquiring a parent reference. However, because every component instance is added to an injector's container, you can use Angular dependency injection to reach a parent component.

This section describes some techniques for doing that.

## Find a parent component of known type

You use standard class injection to acquire a parent component whose type you know.

In the following example, the parent AlexComponent has several children including a CathyComponent:

```
parent-finder.component.ts (AlexComponent v.1)
```

```
@Component({
  selector: 'alex',
  template: `
    <div class="a">
      <h3>{{name}}</h3>
      <cathy></cathy>
      <craig></craig>
      <carol></carol>
    </div>`,
})
export class AlexComponent extends Base
{
```

```
name = 'Alex';
}
```

Cathy reports whether or not she has access to Alex after injecting an AlexComponent into her constructor:

parent-finder.component.ts (CathyComponent)

```
@Component({
  selector: 'cathy',
  template: `
    <div class="c">
      <h3>Cathy</h3>
      {{alex ? 'Found' : 'Did not find'}} Alex via the component class.<br>
    </div>`
})
export class CathyComponent {
  constructor( @Optional() public alex: AlexComponent ) { }
}
```

Notice that even though the [@Optional](#) qualifier is there for safety, the [live example](#) / [download example](#) confirms that the alex parameter is set.

## Unable to find a parent by its base class

What if you don't know the concrete parent component class?

A re-usable component might be a child of multiple components. Imagine a component for rendering breaking news about a financial instrument. For business reasons, this news component makes frequent calls directly into its parent instrument as changing market data streams by.

The app probably defines more than a dozen financial instrument components. If you're lucky, they all implement the same base class whose API your NewsComponent understands.

Looking for components that implement an interface would be better. That's not possible because TypeScript interfaces disappear from the transpiled JavaScript, which doesn't support interfaces. There's no artifact to look for.

This isn't necessarily good design. This example is examining whether a component can inject its parent via the parent's base class.

The sample's CraigComponent explores this question. [Looking back](#), you see that the Alex component extends (inherits) from a class named Base.

parent-finder.component.ts (Alex class signature)

```
export class AlexComponent extends Base
```

The CraigComponent tries to inject Base into its alex constructor parameter and reports if it succeeded.

parent-finder.component.ts (CraigComponent)

```
@Component({
  selector: 'craig',
  template: `
    <div class="c">
      <h3>Craig</h3>
      {{alex ? 'Found' : 'Did not find'}} Alex via the base class.
    </div>`
})
export class CraigComponent {
  constructor( @Optional() public alex: Base ) { }
}
```

Unfortunately, this doesn't work. The [live example](#) / [download example](#) confirms that the alex parameter is null. You cannot inject a parent by its base class.

## Find a parent by its class interface

You can find a parent component with a [class interface](#).

The parent must cooperate by providing an alias to itself in the name of a class interface token.

Recall that Angular always adds a component instance to its own injector; that's why you could inject Alex into Cathy [earlier](#).

Write an [alias provider](#)—a provide object literal with a useExisting definition—that creates an alternative way to inject the same component instance and add that provider to the providers array of the [@Component\(\)](#) metadata for the AlexComponent.

parent-finder.component.ts (AlexComponent providers)

```
providers: [{ provide: Parent, useExisting: forwardRef(() => AlexComponent) }],
```

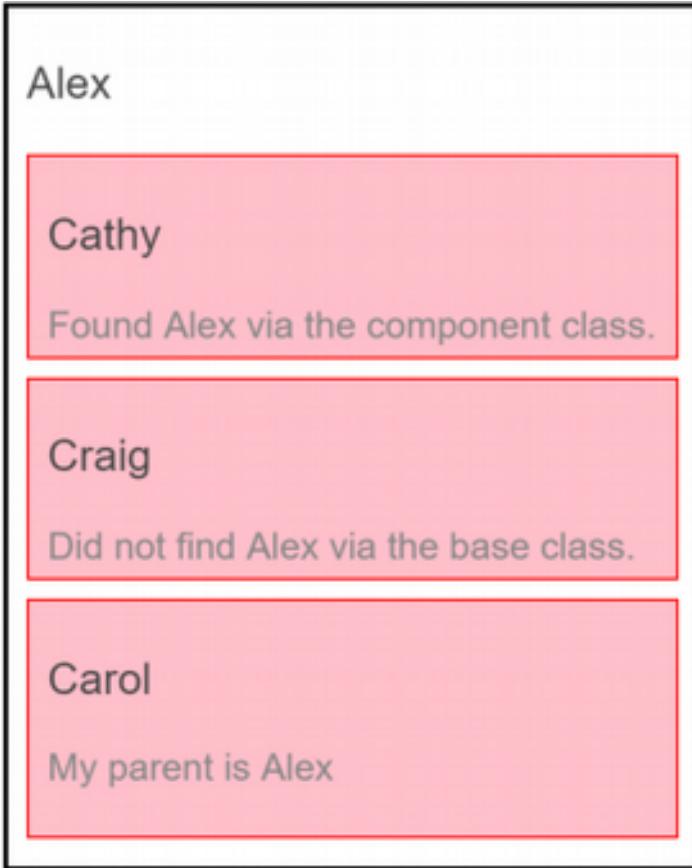
[Parent](#) is the provider's class interface token. The [forwardRef](#) breaks the circular reference you just created by having the AlexComponent refer to itself.

Carol, the third of Alex's child components, injects the parent into its parent parameter, the same way you've done it before.

parent-finder.component.ts (CarolComponent class)

```
export class CarolComponent {
  name = 'Carol';
  constructor( @Optional() public parent: Parent ) { }
}
```

Here's Alex and family in action.



## Find a parent in a tree with `@SkipSelf()`

Imagine one branch of a component hierarchy: Alice -> Barry -> Carol. Both Alice and Barry implement the `Parent` class interface.

Barry is the problem. He needs to reach his parent, Alice, and also be a parent to Carol. That means he must both inject the Parent class interface to get Alice and provide a Parent to satisfy Carol.

Here's Barry.

parent-finder.component.ts (BarryComponent)

```
const templateB = `<div class="b"><div><h3>{{name}}</h3><p>My parent is {{parent?.name}}</p></div><carol></carol><chris></chris></div>`;
```

```
@Component({
  selector: 'barry',
  template: templateB,
  providers: [{ provide: Parent, useExisting: forwardRef(() => BarryComponent) }]
})
export class BarryComponent implements Parent {
  name = 'Barry';
  constructor( @SkipSelf() @Optional() public parent: Parent ) { }
}
```

Barry's providers array looks just like [Alex's](#). If you're going to keep writing [alias providers](#) like this you should create a [helper function](#).

For now, focus on Barry's constructor.

Barry's constructor

Carol's constructor

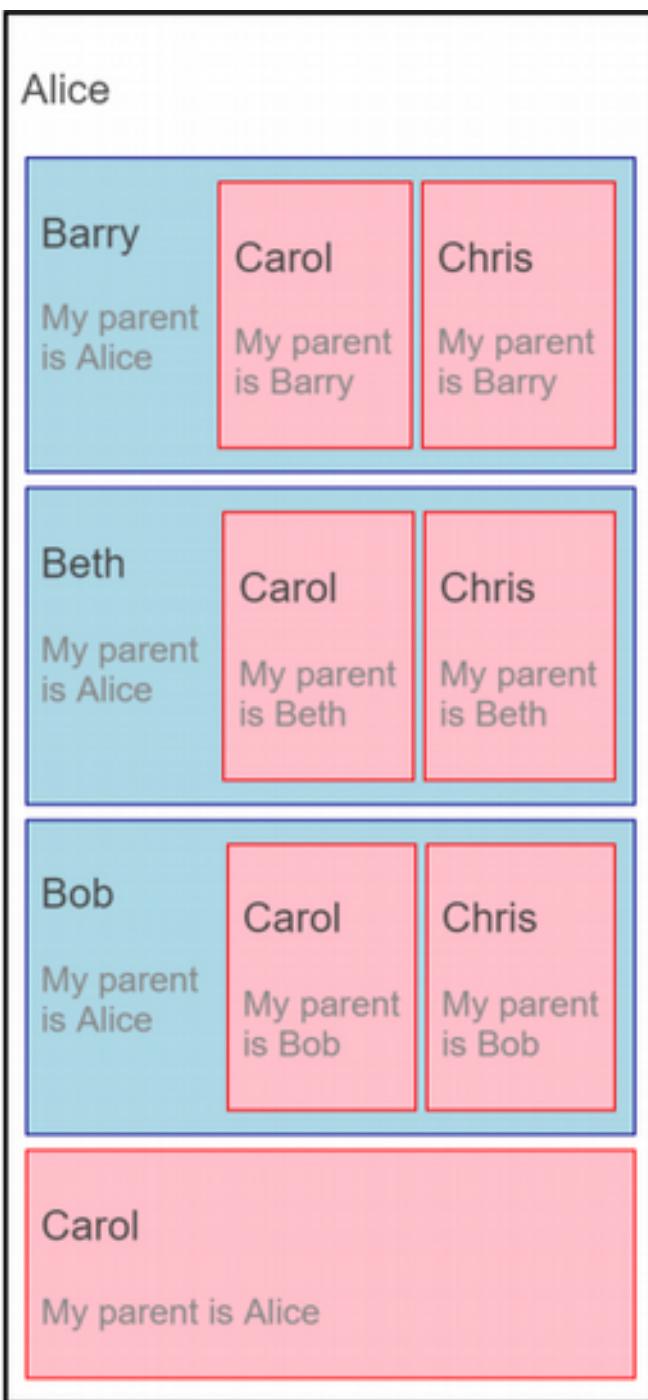
```
constructor( @SkipSelf() @Optional() public parent: Parent ) { }
```

It's identical to Carol's constructor except for the additional [@SkipSelf](#) decorator.

[@SkipSelf](#) is essential for two reasons:

1. It tells the injector to start its search for a Parent dependency in a component above itself, which is what parent means.
2. Angular throws a cyclic dependency error if you omit the [@SkipSelf](#) decorator.
3. Cannot instantiate cyclic dependency! (BethComponent -> Parent -> BethComponent)

Here's Alice, Barry, and family in action.



## Parent class interface

You [learned earlier](#) that a class interface is an abstract class used as an interface rather than as a base class.

The example defines a Parent class interface.

`parent-finder.component.ts (Parent class-interface)`

```
export abstract class Parent { name: string; }
```

The Parent class interface defines a name property with a type declaration but no implementation. The name property is the only member of a parent component that a child component can call. Such a narrow interface helps decouple the child component class from its parent components.

A component that could serve as a parent should implement the class interface as the AliceComponent does.

parent-finder.component.ts (AliceComponent class signature)

```
export class AliceComponent implements Parent
```

Doing so adds clarity to the code. But it's not technically necessary. Although AlexComponent has a name property, as required by its Base class, its class signature doesn't mention Parent.

parent-finder.component.ts (AlexComponent class signature)

```
export class AlexComponent extends Base
```

AlexComponent should implement Parent as a matter of proper style. It doesn't in this example only to demonstrate that the code will compile and run without the interface.

## provideParent() helper function

Writing variations of the same parent alias provider gets old quickly, especially this awful mouthful with a [forwardRef](#).

dependency-injection-in-action/src/app/parent-finder.component.ts

```
providers: [{ provide: Parent, useExisting: forwardRef(() => AlexComponent) }],
```

You can extract that logic into a helper function like the following.

dependency-injection-in-action/src/app/parent-finder.component.ts

```
// Helper method to provide the current component instance in the name of a `parentType`.
const provideParent =
(component: any) => {
  return { provide: Parent, useExisting: forwardRef(() => component) };
};
```

Now you can add a simpler, more meaningful parent provider to your components.

dependency-injection-in-action/src/app/parent-finder.component.ts

```
providers: [ provideParent(AliceComponent) ]
```

You can do better. The current version of the helper function can only alias the Parent class interface. The application might have a variety of parent types, each with its own class interface token.

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

```
// Helper method to provide the current component instance in the name of a `parentType`.  
// The `parentType` defaults to `Parent` when omitting the second parameter.  
const provideParent =  
(component: any, parentType?: any) => {  
  return { provide: parentType || Parent, useExisting: forwardRef\(\) => component };  
};
```

And here's how you could use it with a different parent type.

dependency-injection-in-action/src/app/parent-finder.component.ts

```
providers: [ provideParent(BethComponent, DifferentParent) ]
```

Most front-end applications communicate with backend services over the HTTP protocol.

Modern browsers support two different APIs for making HTTP requests: the

XMLHttpRequest interface and the `fetch()` API.

The [HttpClient](#) in `@angular/common/http` offers a simplified client HTTP API for Angular applications that rests on the XMLHttpRequest interface exposed by browsers. Additional benefits of [HttpClient](#) include testability features, typed request and response objects, request and response interception, Observable apis, and streamlined error handling.

You can run the [live example](#) / [download example](#) that accompanies this guide.

The sample app does not require a data server. It relies on the [Angular in-memory-web-api](#), which replaces the `HttpClient` module's [HttpBackend](#). The replacement service simulates the behavior of a REST-like backend.

Look at the `AppModule` imports to see how it is configured.

## Setup

Before you can use the [HttpClient](#), you need to import the Angular [HttpClientModule](#). Most apps do so in the root `AppModule`.

app/app.module.ts (excerpt)

```
import { NgModule }      from '@angular/core';  
import { BrowserModule } from '@angular/platform-browser';  
import { HttpClientModule } from '@angular/common/http';
```

```
@NgModule({  
  imports: [  
    BrowserModule,
```

```
// import HttpClientModule after BrowserModule.  
 HttpClientModule,  
 ],  
 declarations: [  
   AppComponent,  
 ],  
 bootstrap: [ AppComponent ]  
}  
export class AppModule {}
```

Having imported [HttpClientModule](#) into the AppModule, you can inject the [HttpClient](#) into an application class as shown in the following ConfigService example.

app/config/config.service.ts (excerpt)

```
import { Injectable } from '@angular/core';  
import { HttpClient } from '@angular/common/http';  
  
@Injectable()  
export class ConfigService {  
  constructor(private http: HttpClient) {}  
}
```

## Getting JSON data

Applications often request JSON data from the server. For example, the app might need a configuration file on the server, config.json, that specifies resource URLs.

assets/config.json

```
{  
  "heroesUrl": "api/heroes",  
  "textfile": "assets/textfile.txt"  
}
```

The ConfigService fetches this file with a get() method on [HttpClient](#).

app/config/config.service.ts (getConfig v.1)

```
configUrl = 'assets/config.json';
```

```
getConfig() {  
  return this.http.get(this.configUrl);  
}
```

A component, such as ConfigComponent, injects the ConfigService and calls the getConfig service method.

```
app/config/config.component.ts (showConfig v.1)

showConfig() {
  this.configService.getConfig()
    .subscribe((data: Config) => this.config = {
      heroesUrl: data['heroesUrl'],
      textfile: data['textfile']
    });
}
```

Because the service method returns an Observable of configuration data, the component subscribes to the method's return value. The subscription callback copies the data fields into the component's [config](#) object, which is data-bound in the component template for display.

## Why write a service

This example is so simple that it is tempting to write the [Http.get\(\)](#) inside the component itself and skip the service.

However, data access rarely stays this simple. You typically post-process the data, add error handling, and maybe some retry logic to cope with intermittent connectivity.

The component quickly becomes cluttered with data access minutia. The component becomes harder to understand, harder to test, and the data access logic can't be re-used or standardized.

That's why it is a best practice to separate presentation of data from data access by encapsulating data access in a separate service and delegating to that service in the component, even in simple cases like this one.

## Type-checking the response

The subscribe callback above requires bracket notation to extract the data values.

```
.subscribe((data: Config) => this.config = {
  heroesUrl: data['heroesUrl'],
  textfile: data['textfile']
});
```

You can't write `data.heroesUrl` because TypeScript correctly complains that the data object from the service does not have a `heroesUrl` property.

The [HttpClient.get\(\)](#) method parsed the JSON server response into the anonymous Object type. It doesn't know what the shape of that object is.

You can tell [HttpClient](#) the type of the response to make consuming the output easier and more obvious.

First, define an interface with the correct shape:

```
export interface Config {  
  heroesUrl: string;  
  textfile: string;  
}
```

Then, specify that interface as the [HttpClient.get\(\)](#) call's type parameter in the service:

app/config/config.service.ts (getConfig v.2)

```
getConfig() {  
  // now returns an Observable of Config  
  return this.http.get<Config>(this.configUrl);  
}
```

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)

```
config: Config;  
  
showConfig() {  
  this.configService.getConfig()  
    // clone the data object, using its known Config shape  
    .subscribe((data: Config) => this.config = { ...data });  
}
```

## Reading the full response

The response body doesn't return all the data you may need. Sometimes servers return special headers or status codes to indicate certain conditions that are important to the application workflow.

Tell [HttpClient](#) that you want the full response with the observe option:

```
getConfigResponse(): Observable<HttpResponse<Config>> {  
  return this.http.get<Config>(  
    this.configUrl, { observe: 'response' });  
}
```

Now [HttpClient.get\(\)](#) returns an Observable of typed [HttpResponse](#) rather than just the JSON data.

The component's showConfigResponse() method displays the response headers as well as the configuration:

app/config/config.component.ts (showConfigResponse)

```
showConfigResponse() {  
  this.configService.getConfigResponse()
```

```

// resp is of type `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.

## Error handling

What happens if the request fails on the server, or if a poor network connection prevents it from even reaching the server? [HttpClient](#) will return an error object instead of a successful response.

You could handle in the component by adding a second callback to the .subscribe():

app/config/config.component.ts (showConfig v.3 with error handling)

```

showConfig() {
  this.configService.getConfig()
    .subscribe(
      (data: Config) => this.config = { ...data }, // success path
      error => this.error = error // error path
    );
}

```

It's certainly a good idea to give the user some kind of feedback when data access fails. But displaying the raw error object returned by [HttpClient](#) is far from the best way to do it.

## Getting error details

Detecting that an error occurred is one thing. Interpreting that error and composing a user-friendly response is a bit more involved.

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.

Or 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 produce JavaScript ErrorEvent objects.

The [HttpClient](#) captures both kinds of errors in its [HttpErrorResponse](#) and you can inspect that response to figure out what really happened.

Error inspection, interpretation, and resolution is something you want to do in the service, not in the component.

You might first devise an error handler like this one:

```
app/config/config.service.ts (handleError)

private handleError(error: HttpErrorResponse) {
  if (error.error instanceof ErrorEvent) {
    // A client-side or network error occurred. Handle it accordingly.
    console.error('An error occurred:', error.error.message);
  } 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.');
};
```

Notice that this handler returns an RxJS [ErrorObservable](#) with a user-friendly error message. Consumers of the service expect service methods to return an Observable of some kind, even a "bad" one.

Now you take the Observables returned by the [HttpClient](#) methods and pipe them through to the error handler.

```
app/config/config.service.ts (getConfig v.3 with error handler)

getConfig() {
  return this.http.get<Config>(this.configUrl)
    .pipe(
      catchError(this.handleError)
    );
}
```

## retry()

Sometimes the error is transient and will go away automatically if you try again. For example, network interruptions are common in mobile scenarios, and trying again may produce a successful result.

The [RxJS library](#) offers several retry operators that are worth exploring. The simplest is called `retry()` and it automatically re-subscribes to a failed Observable a specified number of times. Re-subscribing to the result of an [HttpClient](#) method call has the effect of reissuing the HTTP request.

Pipe it onto the [HttpClient](#) method result just before the error handler.

```
app/config/config.service.ts (getConfig with retry)
```

```
getConfig() {
  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
    );
}
```

## Observables and operators

The previous sections of this guide referred to RxJS Observables and operators such as `catchError` and `retry`. You will encounter more RxJS artifacts as you continue below.

[RxJS](#) is a library for composing asynchronous and callback-based code in a functional, reactive style. Many Angular APIs, including [HttpClient](#), produce and consume RxJS Observables.

RxJS itself is out-of-scope for this guide. You will find many learning resources on the web. While you can get by with a minimum of RxJS knowledge, you'll want to grow your RxJS skills over time in order to use [HttpClient](#) effectively.

If you're following along with these code snippets, note that you must import the RxJS observable and operator symbols that appear in those snippets. These ConfigService imports are typical.

```
app/config/config.service.ts (RxJS imports)
```

```
import { Observable, throwError } from 'rxjs';
import { catchError, retry } from 'rxjs/operators';
```

## 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)
```

```
getTextFile(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\(\)](#) returns a string rather than the default JSON because of the responseType option.

The RxJS tap operator (as in "wiretap") lets the code inspect good 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)  
  
download() {  
  this.downloaderService.getTextFile('assets/textfile.txt')  
  .subscribe(results => this.contents = results);  
}
```

## Sending data to the server

In addition to fetching data from the server, [HttpClient](#) supports mutating requests, that is, sending data to the server with other HTTP methods such as PUT, POST, and DELETE.

The sample app for this guide includes a simplified version of the "Tour of Heroes" example that fetches heroes and enables users to add, delete, and update them.

The following sections excerpt methods of the sample's HeroesService.

### Adding headers

Many servers require extra headers for save operations. For example, they may require a "Content-Type" header to explicitly declare the MIME type of the request body. Or perhaps the server requires an authorization token.

The HeroesService defines such headers in an httpOptions object that will be passed to every [HttpClient](#) save method.

```
app/heroes/heroes.service.ts (httpOptions)
```

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

const httpOptions = {
  headers: new HttpHeaders({
    'Content-Type': 'application/json',
    'Authorization': 'my-auth-token'
  })
};
```

## Making a POST request

Apps often POST data to a server. They POST when submitting a form. In the following example, the HeroesService posts when adding a hero to the database.

```
app/heroes/heroes.service.ts (addHero)

/** 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\(\)](#) method is similar to get() in that it has a type parameter (you're expecting the server to return the new hero) and it takes a resource URL.

It takes two more parameters:

1. hero - the data to POST in the body of the request.
2. httpOptions - the method options which, in this case, [specify required headers](#).

Of course it catches errors in much the same manner [described above](#).

The HeroesComponent initiates the actual POST operation by subscribing to the Observable returned by this service method.

```
app/heroes/heroes.component.ts (addHero)
```

```
this.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 heroeslist.

## 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)

/** DELETE: delete the hero from the server */
deleteHero (id: number): Observable<{}> {
  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)

this.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.

```
// oops ... subscribe() is missing so nothing happens
this.heroesService.deleteHero(hero.id);
```

Always subscribe!

An [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](#) methods.

The [AsyncPipe](#) subscribes (and unsubscribes) for you automatically.

All observables returned from [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](#) 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.

```
const 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 will send a PUT request to completely replace a resource with updated data. The following HeroesService example is just like the POST example.

app/heroes/heroes.service.ts (updateHero)

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

For the reasons [explained above](#), the caller (HeroesComponent.update() in this case) must subscribe() to the observable returned from the [HttpClient.put\(\)](#) in order to initiate the request.

## Advanced usage

We have discussed the basic HTTP functionality in @angular/common/[http](#), but sometimes you need to do more than make simple requests and get data back.

### Configuring the request

Other aspects of an outgoing request can be configured via the options object passed as the last argument to the [HttpClient](#) method.

You [saw earlier](#) that the HeroesService sets the default headers by passing an options object (httpOptions) to its save methods. You can do more.

### Update headers

You can't directly modify the existing headers within the previous options object because instances of the [HttpHeaders](#) class are immutable.

Use the set() method instead. It returns a clone of the current instance with the new changes applied.

Here's how you might update the authorization header (after the old token expired) before making the next request.

```
httpOptions.headers =
  httpOptions.headers.set('Authorization', 'my-new-auth-token');
```

## URL Parameters

Adding URL search parameters works a similar way. Here is a `searchHeroes` method that queries for heroes whose names contain the search term.

```
/* 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().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 were "foo", the GET request URL would be `api/heroes/?name=foo`.

The `HttpParams` are immutable so you'll have to use the `set()` method to update the options.

## Debouncing requests

The sample includes an npm package search feature.

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.

Here's a pertinent excerpt from the template:

```
app/package-search/package-search.component.html (search)

<input (keyup)="search($event.target.value)" id="name" placeholder="Search"/>

<ul>
  <li *ngFor="let package of packages$ | async">
    <b>{{package.name}} v.{{package.version}}</b> -
    <i>{{package.description}}</i>
  </li>
</ul>
```

The `(keyup)` event binding sends every keystroke to the component's `search()` method.

Sending a request for every keystroke could be expensive. It's better to wait until the user stops typing and then send a request. That's easy to implement with RxJS operators, as shown in this excerpt.

app/package-search/package-search.component.ts (excerpt)

```
1. withRefresh = false;
2. packages$: Observable<NpmPackageInfo[]>;
3. private searchText$ = new Subject<string>();
4.
5. search\(packageName: string\) {
6.   this.searchText$.next(packageName);
7. }
8.
9. ngOnInit() {
10.   this.packages$ = this.searchText$.pipe(
11.     debounceTime(500),
12.     distinctUntilChanged(),
13.     switchMap(packageName =>
14.       this.searchService.search(packageName, this.withRefresh))
15. );
16. }
17.
18. 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 produce 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:

1. debounceTime(500) - wait for the user to stop typing (1/2 second in this case).
2. distinctUntilChanged() - wait until the search text changes.
3. 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](#) and displays search results as they arrive.

A search value reaches the service only if it's a new value and the user has stopped typing.

The withRefresh option is explained [below](#).

### **switchMap()**

The switchMap() operator has three important characteristics.

1. It takes a function argument that returns an Observable.  
PackageSearchService.search returns an Observable, as other data service methods do.
2. If a previous search request is still in-flight (as when the connection is poor), it cancels that request and sends a new one.
3. It 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.

## Intercepting requests and responses

HTTP Interception is a major feature of @angular/common/[http](#). With interception, you declare interceptors that inspect and transform HTTP requests from your application to the server. The same interceptors may also inspect and transform the 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](#) method call.

### Write an interceptor

To implement an interceptor, declare a class that implements the intercept() method of the [HttpInterceptor](#) interface.

Here is a do-nothing noop interceptor that simply passes the request through without touching it:

```
app/http-interceptors/noop-interceptor.ts

content_copyimport { Injectable } from '@angular/core';
import {
  HttpEvent, HttpInterceptor, HttpHandler, HttpRequest
} from '@angular/common/http';

import { Observable } from 'rxjs';

/** Pass untouched request through to the next request handler. */
@Injectable()
export class NoopInterceptor implements HttpInterceptor {

  intercept(req: HttpRequest<any>, next: HttpHandler):
```

```
Observable<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](#) interface.

```
export abstract class HttpHandler {
  abstract handle(req: HttpRequest<any>): Observable<HttpEvent<any>>;
}
```

Like intercept(), the handle() method transforms an HTTP request into an Observable of [HttpEvents](#) 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 simply 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](#) 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](#) 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 \(DI\)](#) system. Like other services, you must provide the interceptor class before the app can use it.

Because interceptors are (optional) dependencies of the [HttpClient](#) service, you must provide them in the same injector (or a parent of the injector) that provides [HttpClient](#). Interceptors provided after DI creates the [HttpClient](#) are ignored.

This app provides [HttpClient](#) in the app's root injector, as a side-effect of importing the [HttpClientModule](#) in AppModule. You should provide interceptors in AppModule as well.

After importing the [HTTP\\_INTERCEPTORS](#) injection token from `@angular/common/http`, write the NoopInterceptor provider like this:

```
{ provide: HTTP\_INTERCEPTORS, useClass: NoopInterceptor, multi: true },
```

Note the multi: true option. This required setting tells Angular that [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](#) 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

```
/* "Barrel" of Http Interceptors */
import { HTTP\_INTERCEPTORS } from '@angular/common/http';

import { NoopInterceptor } from './noop-interceptor';

/** Http interceptor providers in outside-in order */
export const httpInterceptorProviders = [
  { provide: HTTP\_INTERCEPTORS, useClass: NoopInterceptor, multi: true },
];
```

Then import and add it to the AppModule providers array like this:

app/app.module.ts (interceptor providers)

```
providers: [
  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. If you provide interceptors A, then B, then C, requests will flow in A->B->C and responses will flow out C->B->A.

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.

## HttpEvents

You may have expected the intercept() and handle() methods to return observables of [HttpResponse<any>](#) as most [HttpClient](#) methods do.

Instead they return observables of [HttpEvent<any>](#).

That's because interceptors work at a lower level than those [HttpClient](#) methods. A single HTTP request can generate multiple events, including upload and download progress events. The [HttpResponse](#) class itself is actually an event, whose type is `HttpEventType.HttpResponseEvent`.

Many interceptors are only concerned with the outgoing request and simply return the event stream from `next.handle()` without modifying it.

But interceptors that examine and modify the response from `next.handle()` will see all of these events. Your interceptor should return every event untouched unless it has a compelling reason to do otherwise.

## Immutability

Although interceptors are capable of mutating requests and responses, the [HttpRequest](#) and [HttpResponse](#) instance properties are readonly, rendering them largely immutable.

They are immutable for a good reason: the app may retry a request several times before it succeeds, which means that the interceptor chain may 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.

TypeScript will prevent you from setting [HttpRequest](#) readonly properties.

```
// Typescript disallows the following assignment because req.url is readonly
req.url = req.url.replace('http://', 'https://');
```

To alter the 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 in this example.

app/http-interceptors/ensure-https-interceptor.ts (excerpt)

```
// clone request and replace 'http://' with 'https://' at the same time
const secureReq = req.clone({
  url: req.url.replace('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.

## The 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.

```
req.body.name = req.body.name.trim(); // bad idea!
```

If you must mutate the request body, copy it first, change the copy, clone() the request, and set the clone's body with the new body, as in the following example.

app/http-interceptors/trim-name-interceptor.ts (excerpt)

```
// 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

Sometimes you need to clear the request body rather than replace it. If you set the cloned request body to undefined, Angular assumes you intend to leave the body as is. That is not what you want. If you set the cloned request body to null, Angular knows you intend to clear the request body.

```
newReq = req.clone({ ... }); // body not mentioned => preserve original body
newReq = req.clone({ body: undefined }); // preserve original body
newReq = req.clone({ body: null }); // clear the body
```

### Set 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

1. import { AuthService } from './auth.service';
- 2.
3. `@Injectable()`
4. export class AuthInterceptor implements [HttpInterceptor](#) {
- 5.
6. constructor(private auth: AuthService) {}
- 7.
8. intercept(req: [HttpRequest](#)<any>, next: [HttpHandler](#)) {
9. // [Get](#) the auth token from the service.
10. const authToken = this.auth.getAuthorizationToken();
- 11.
12. // Clone the request and replace the original headers with
13. // cloned headers, updated with the authorization.
14. const authReq = req.clone({
15. headers: req.headers.set('Authorization', authToken)

```
16. });
17.
18. // send cloned request with header to the next handler.
19. return next.handle(authReq);
20. }
21.}
```

The practice of cloning a request to set new headers is so common that there's a `setHeaders` shortcut for it:

```
// 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

Because interceptors can process the request and response together, they can do things like time and log 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)

```
1. import { finalize, tap } from 'rxjs/operators';
2. import { MessageService } from './message.service';
3.
4. @Injectable()
5. export class LoggingInterceptor implements HttpInterceptor {
6.   constructor(private messenger: MessageService) {}
7.
8.   intercept(req: HttpRequest<any>, next: HttpHandler) {
9.     const started = Date.now();
10.    let ok: string;
11.
12.    // extend server response observable with logging
13.    return next.handle(req)
14.      .pipe(
15.        tap(
16.          // Succeeds when there is a response; ignore other events
17.          event => ok = event instanceof HttpResponse ? 'succeeded' : '',
18.          // Operation failed; error is an HttpErrorResponse
```

```

19.     error => ok = 'failed'
20.   ),
21.   // Log when response observable either completes or errors
22.   finalize(() => {
23.     const elapsed = Date.now() - started;
24.     const msg = `${req.method} ${req.urlWithParams}`
25.       `${ok} in ${elapsed} ms.`;
26.     this.messenger.add(msg);
27.   })
28. );
29. }
30.}

```

The RxJS tap operator captures whether the request succeed 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.

## Caching

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 demonstrates this approach.

app/http-interceptors/caching-interceptor.ts)

```

@Injectable()
export class CachingInterceptor implements HttpInterceptor {
  constructor(private cache: RequestCache) {}

  intercept(req: HttpRequest<any>, next: HttpHandler) {
    // continue if not cachable.
    if (!isCachable(req)) { return next.handle(req); }

    const cachedResponse = this.cache.get(req);
    return cachedResponse ?
      of(cachedResponse) : sendRequest(req, next, this.cache);
  }
}

```

The isCachable() function determines if the request is cachable. In this sample, only GET requests to the npm package search api are cachable.

If the request is not cachable, the interceptor simply forwards the request to the next handler in the chain.

If a cachable 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 cachable request is not in cache, the code calls sendRequest.

```
1. /**
2. * Get server response observable by sending request to `next()` .
3. * Will add the response to the cache on the way out.
4. */
5. function sendRequest(
6.   req: HttpRequest<any>,
7.   next: HttpHandler,
8.   cache: RequestCache): Observable<HttpEvent<any>> {
9.
10. // No headers allowed in npm search request
11. const noHeaderReq = req.clone({ headers: new HttpHeaders\(\) });
12.
13. return next.handle(noHeaderReq).pipe(
14.   tap(event => {
15.     // There may be other events besides the response.
16.     if (event instanceof HttpResponse) {
17.       cache.put(req, event); // Update the cache.
18.     }
19.   })
20. );
21.}
```

The sendRequest function creates a [request clone](#) without headers because the npm api forbids them.

It forwards that request to next.handle() which ultimately calls the server and returns the server's response.

Note how sendRequest intercepts the response on its way back to the application. It 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](#) requests actually return cached responses.

## Return a multi-valued Observable

The [HttpClient.get\(\)](#) method normally returns an observable that either emits the data or an error. Some folks describe it as a "one and done" observable.

But an interceptor can change this to an observable that emits more than once.

A revised version of the CachingInterceptor optionally returns an observable that immediately emits the cached response, sends the request to the NPM web API anyway, and emits again later with the updated search results.

```
// 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\(\)](#).

The revised CachingInterceptor sets up a server request whether there's a cached value or not, using the same sendRequest() method described [above](#). The results\$ observable will make 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.

## Listening to progress events

Sometimes applications transfer large amounts of data and those transfers can take a long time. File uploads are a typical example. 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](#) with the [reportProgress](#) option set true to enable tracking of progress events.

app/uploaders/uploaders.service.ts (upload request)

```
const req = new HttpRequest('POST', '/upload/file', file, {  
  reportProgress: true  
});
```

Every progress event triggers change detection, so only turn them on if you truly intend to report progress in the UI.

Next, pass this request object to the [HttpClient.request\(\)](#) method, which returns an Observable of HttpEvents, the same events processed by interceptors:

```
app/uploader/uploader.service.ts (upload body)  
  
// 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](#) in the event stream.

```
app/uploader/uploader.service.ts (getEventMessage)
```

```
/** Return distinct message for sent, upload progress, & response events */  
private getEventMessage(event: HttpEvent<any>, file: File) {  
  switch (event.type) {  
    case HttpEventType.Sent:  
      return `Uploading file "${file.name}" of size ${file.size}.`;  
  
    case HttpEventType.UploadProgress:  
      // Compute and show the % done:  
      const percentDone = Math.round(100 * event.loaded / event.total);  
      return `File "${file.name}" is ${percentDone}% uploaded.`;  
  
    case 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.

## Security: XSRF Protection

[Cross-Site Request Forgery \(XSRF\)](#) is an attack technique by which the attacker can trick an authenticated user into unknowingly executing actions on your website. [HttpClient](#) supports a [common mechanism](#) 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 cookie on all mutating requests (POST, etc.) 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.

Note that [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. If not, Angular's default protection will be ineffective.

### Configuring custom cookie/header names

If your backend service uses different names for the XSRF token cookie or header, use [HttpClientXsrfModule.withOptions\(\)](#) to override the defaults.

```
imports: [
  HttpClientModule,
  HttpClientXsrfModule.withOptions({
    cookieName: 'My-Xsrf-Cookie',
    headerName: 'My-Xsrf-Header',
  }),
],
```

## Testing HTTP requests

Like any external dependency, the HTTP backend needs to be mocked so your tests can simulate interaction with a remote server. The `@angular/common/http` testing library makes setting up such mocking straightforward.

## Mocking philosophy

Angular's HTTP testing library is designed for a pattern of testing wherein the app executes code and makes requests first.

Then a test expects that certain requests have or have not been made, performs assertions against those requests, and finally provide responses by "flushing" each expected request.

At the end, tests may verify that the app has made no unexpected requests.

You can run [these sample tests](#) / [download example](#) 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](#) in `src/app/heroes/heroes.service.spec.ts`.

## Setup

To begin testing calls to [HttpClient](#), import the [HttpClientTestingModule](#) and the mocking controller, [HttpTestingController](#), along with the other symbols your tests require.

```
app/testing/http-client.spec.ts (imports)
```

```
// Http testing module and mocking controller
```

```
import { HttpClientTestingModule, HttpTestingController } from '@angular/common/http/testing';
```

```
// Other imports
```

```
import { TestBed } from '@angular/core/testing';
```

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

Then add the [HttpClientTestingModule](#) to the [TestBed](#) and continue with the setup of the service-under-test.

```
app/testing/http-client.spec.ts(setup)
```

```
describe('HttpClient testing', () => {
```

```
let httpClient: HttpClient;
```

```
let httpTestingController: HttpTestingController;
```

```
beforeEach(() => {
```

```
TestBed.configureTestingModule({  
  imports: [ HttpClientTestingModule ]  
});
```

```
});
```

```
// Inject the http service and test controller for each test
```

```
httpClient = TestBed.get(HttpClient);
```

```
httpTestingController = TestBed.get(HttpTestingController);
```

```
});
```

```
/// Tests begin ///
});
```

Now requests made in the course of your tests will hit the testing backend instead of the normal backend.

This setup also calls TestBed.get() to inject the [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)

it('can test HttpClient.get', () => {
  const testData: Data = {name: 'Test Data'};

  // Make an HTTP GET request
  httpClient.get<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 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:

```
afterEach(() => {
  // 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:

```
// Expect one request with an authorization header
const req = httpTestingController.expectOne(
  req => req.headers.has('Authorization')
);
```

As with the previous [expectOne\(\)](#), the test will fail 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.

```
// 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.

```
it('can test for 404 error', () => {
  const emsg = 'deliberate 404 error';

  httpClient.get<Data[]>(testUrl).subscribe(
    data => fail('should have failed with the 404 error'),
    (error: 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`.

```
it('can test for network error', () => {
  const emsg = 'simulated network error';

  httpClient.get<Data[]>(testUrl).subscribe(
    data => fail('should have failed with the network error'),
    (error: 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 throwError = new ErrorEvent('Network error', {
  message: emsg,
});
```

```
// Respond with mock error
req.error(throwError);
});
```

The Angular [Router](#) enables navigation from one [view](#) to the next as users perform application tasks.

This guide covers the router's primary features, illustrating them through the evolution of a small application that you can [run live in the browser](#) / [download example](#).

## Overview

The browser is a familiar model of application navigation:

- Enter a URL in the address bar and the browser navigates to a corresponding page.
- Click links on the page and the browser navigates to a new page.
- Click the browser's back and forward buttons and the browser navigates backward and forward through the history of pages you've seen.

The Angular [Router](#) ("the router") borrows from this model. It can interpret a browser URL as an instruction to navigate to a client-generated view. It can pass optional parameters along to the supporting view component that help it decide what specific content to present. You can bind the router to links on a page and it will navigate to the appropriate application view when the user clicks a link. You can navigate imperatively when the user clicks a button, selects from a drop box, or in response to some other stimulus from any source. And the router logs activity in the browser's history journal so the back and forward buttons work as well.

## The Basics

This guide proceeds in phases, marked by milestones, starting from a simple two-pager and building toward a modular, multi-view design with child routes.

An introduction to a few core router concepts will help orient you to the details that follow.

### <base href>

Most routing applications should add a `<base>` element to the `index.html` as the first child in the `<head>` tag to tell the router how to compose navigation URLs.

If the app folder is the application root, as it is for the sample application, set the `href` value exactly as shown here.

```
src/index.html (base-href)
```

```
<base href="/">
```

### 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. It 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)
```

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

You'll learn about more options in the [details below](#).

### Configuration

A routed Angular application has one singleton instance of the [Router](#) service. When the browser's URL changes, that router looks for a corresponding [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` method, and adds the result to the `AppModule`'s [imports](#) array.

```
src/app/app.module.ts (excerpt)
```

```
const appRoutes: 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({
  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` method in the module [importsto](#) configure the router.

Each [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, allowing 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` will use that value to find and present the hero whose id is 42. You'll learn more about route parameters later in this guide.

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, staticdata. You'll use the [resolve guard](#) to retrieve dynamic data later in the guide.

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, will display the HeroesListComponent.

The \*\* path in the last route is a wildcard. The router will select this route if the requested URL doesn't match any paths for routes defined earlier in the configuration. This is useful for displaying a "404 - Not Found" page or redirecting to another route.

The order of the routes in the configuration matters and this is by design. The router uses a first-match wins strategy when matching routes, so more specific routes should be placed above less specific routes. In the configuration above, routes with a static path are listed first, followed by an empty path route, that matches the default route. The wildcard route comes last because it matches every URL and should be selected only if no other routes are matched first.

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. This should only be used for debugging purposes. You set the `enableTracing: true` option in the object passed as the second argument to the `RouterModule.forRoot()` method.

## Router outlet

The `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.

```
<router-outlet></router-outlet>
<!-- 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` that you've placed in the host component's template.

## Router links

Now you have routes configured and a place to render them, but how do you navigate? The URL could arrive directly from the browser address bar. But most of the time you navigate as a result of some user action such as the click of an anchor tag.

Consider the following template:

```
src/app/app.component.html
<h1>Angular Router</h1>
<nav>
<a routerLink="/crisis-center" routerLinkActive="active">Crisis Center</a>
```

```
<a routerLink="/heroes" routerLinkActive="active">Heroes</a>
</nav>
<router-outlet></router-outlet>
```

The [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](#) (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 (the link parameters array). The router resolves that array into a complete URL.

## Active router links

The [RouterLinkActive](#) directive toggles css classes for active [RouterLink](#) bindings based on the current [RouterState](#).

On each anchor tag, you see a [property binding](#) to the [RouterLinkActive](#) directive that look like `routerLinkActive="..."`.

The template expression to the right of the equals (=) contains a space-delimited string of CSS classes that the Router will add when this link is active (and remove when the link is inactive). You set the [RouterLinkActive](#) directive to a string of classes such as `[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](#) 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](#) objects that make up the current state of the router. You can access the current [RouterState](#) from anywhere in the application using the [Router](#) service and the [routerState](#) property.

Each [ActivatedRoute](#) in the [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](#). 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 <a href="#">resolve guard</a> .
paramMap	An Observable that contains a <a href="#">map</a> of the required and <a href="#">optional parameters</a> specific to the route. The map supports retrieving single and multiple values from the same parameter.
queryParamMap	An Observable that contains a <a href="#">map</a> of the <a href="#">query parameters</a> available to all routes. The map supports retrieving single and multiple values from the query parameter.
fragment	An Observable of the URL <a href="#">fragment</a> available to all routes.
outlet	The name of the <a href="#">RouterOutlet</a> 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 <a href="#">ActivatedRoute</a> when this route is a <a href="#">child route</a> .
firstChild	Contains the first <a href="#">ActivatedRoute</a> in the list of this route's child routes.
children	Contains all the <a href="#">child routes</a> activated under the current route.

Two older properties are still available. They are less capable than their replacements, discouraged, and may be deprecated in a future Angular version.

params—An Observable that contains the required and [optional parameters](#) specific to the route. Use paramMap instead.

queryParams—An Observable that contains the [query parameters](#) available to all routes. Use queryParamMap instead.

## Router events

During each navigation, the [Router](#) emits navigation events through the [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
<a href="#">NavigationStart</a>	An <a href="#">event</a> triggered when navigation starts.
<a href="#">RouteConfigLoadStart</a>	An <a href="#">event</a> triggered before the <a href="#">Router</a> <a href="#">lazy loads</a> a route configuration.
<a href="#">RouteConfigLoadEnd</a>	An <a href="#">event</a> triggered after a route has been lazy loaded.

<a href="#">RoutesRecognized</a>	An <a href="#">event</a> triggered when the Router parses the URL and the routes are recognized.
<a href="#">GuardsCheckStart</a>	An <a href="#">event</a> triggered when the Router begins the Guards phase of routing.
<a href="#">ChildActivationStart</a>	An <a href="#">event</a> triggered when the Router begins activating a route's children.
<a href="#">ActivationStart</a>	An <a href="#">event</a> triggered when the Router begins activating a route.
<a href="#">GuardsCheckEnd</a>	An <a href="#">event</a> triggered when the Router finishes the Guards phase of routing successfully.
<a href="#">ResolveStart</a>	An <a href="#">event</a> triggered when the Router begins the Resolve phase of routing.
<a href="#">ResolveEnd</a>	An <a href="#">event</a> triggered when the Router finishes the Resolve phase of routing successfully.
<a href="#">ChildActivationEnd</a>	An <a href="#">event</a> triggered when the Router finishes activating a route's children.
<a href="#">ActivationEnd</a>	An <a href="#">event</a> triggered when the Router finishes activating a route.
<a href="#">NavigationEnd</a>	An <a href="#">event</a> triggered when navigation ends successfully.
<a href="#">NavigationCancel</a>	An <a href="#">event</a> triggered when navigation is canceled. This is due to a <a href="#">Route Guard</a> returning false during navigation.
<a href="#">NavigationError</a>	An <a href="#">event</a> triggered when navigation fails due to an unexpected error.
<a href="#">Scroll</a>	An <a href="#">event</a> that represents a scrolling event.

These events are logged to the console when the [enableTracing](#) option is enabled also. For an example of filtering router navigation events, visit the [router section](#) of the [Observables in Angular](#) guide.

## Summary

The application has a configured router. The shell component has a [RouterOutlet](#) where it can display views produced by the router. It has [RouterLinks](#) that users can click to navigate via the router.

Here are the key [Router](#) terms and their meanings:

Router Part	Meaning
<a href="#">Router</a>	Displays the application component for the active URL. Manages navigation from one component to the next.
<a href="#">RouterModule</a>	A separate NgModule that provides the necessary service providers and directives for navigating through application views.
<a href="#">Routes</a>	Defines an array of Routes, each mapping a URL path to a component.
<a href="#">Route</a>	Defines how the router should navigate to a component based on a URL pattern. Most routes consist of a path and a component type.
<a href="#">RouterOutlet</a>	The directive ( <code>&lt;router-outlet&gt;</code> ) that marks where the router displays a view.
<a href="#">RouterLink</a>	The directive for binding a clickable HTML element to a route. Clicking an element with a <a href="#">routerLink</a> directive that is bound to a string or a link parameters array triggers a navigation.
<a href="#">RouterLinkActive</a>	The directive for adding/removing classes from an HTML element when an associated <a href="#">routerLink</a> contained on or inside the element becomes active/inactive.
<a href="#">ActivatedRoute</a>	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.
<a href="#">RouterState</a>	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 <a href="#">RouterLink</a> or pass the array as an argument to the Router.navigate method.
Routing component	An Angular component with a <a href="#">RouterOutlet</a> that displays views based on router navigations.

## The sample application

This guide describes development of a multi-page routed sample application. Along the way, it highlights design decisions and describes 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](#) guard (checking route access).
- The [CanActivateChild](#) guard (checking child route access).
- The [CanDeactivate](#) guard (ask permission to discard unsaved changes).
- The [Resolve](#) guard (pre-fetching route data).
- Lazy loading feature modules.
- The [CanLoad](#) guard (check before loading feature module assets).

The guide proceeds as a sequence of milestones as if you were building the app step-by-step. But, it is not a tutorial and it glosses over details of Angular application construction that are more thoroughly covered elsewhere in the documentation.

The full source for the final version of the app can be seen and downloaded from the [live example](#) / [download example](#).

## The sample application in action

Imagine an application that helps the Hero Employment Agency run its business. Heroes need work and the agency finds crises for them 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](#) / [download example](#).

Once the app warms up, you'll see a row of navigation buttons and the Heroes view with its list of heroes.

	HEROES
11	Mr. Nice
12	Narco
13	Bombasto
14	Celeritas
15	Magneta
16	RubberMan

Select one hero and the app takes you to a hero editing screen.

## HEROES

"Magneta"

Id: 15

Name:

[Back](#)

Alter the name. Click the "Back" button and the app 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 "Back" button, the app would have returned you to the heroes list as well. Angular app navigation updates the browser history as normal web navigation does.

Now click the Crisis Center link for a list of ongoing crises.

## CRISIS CENTER

- 1 Dragon Burning Cities
- 2 Sky Rains Great White Sharks
- 3 Giant Asteroid Heading For Earth
- 4 Procrastinators Meeting Delayed Again

Welcome to the Crisis Center

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.

## CRISIS CENTER

- 1 Dragon Burning Cities
- 2 Sky Rains Great White Sharks
- 3 Giant Asteroid Heading For Earth
- 4 Procrastinators Meeting Delayed Again

"GIGANTIC Asteroid Heading For Earth"

Id: 3  
Name:

[Save](#) [Cancel](#)

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.

Do not click either button yet. Click the browser back button or the "Heroes" link instead.

Up pops a dialog box.



You can say "OK" and lose your changes or click "Cancel" and continue editing.

Behind this behavior is the router's [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 to be covered later in the guide. This short introduction will do for now.

Proceed to the first application milestone.

## Milestone 1: Getting started

Begin with a simple version of the app that navigates between two empty views.

# Component Router

Crisis Center   Heroes

Generate a sample application to follow the walkthrough.

```
ng new angular-router-sample
```

## Define Routes

A router must be configured with a list of route definitions.

Each definition translates to a [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.

In simpler terms, you might say this of the first route:

- 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 simple paths leading to the CrisisListComponent and HeroListComponent. Generate the CrisisList and HeroList components.

```
ng generate component crisis-list
```

```
ng 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
```

```
<h2>CRISIS CENTER</h2>
```

```
<p>Get your crisis here</p>
```

## Register Router and Routes

In order to use the Router, you must first register the [RouterModule](#) from the `@angular/router` package. Define an array of routes, `appRoutes`, and pass them to the [RouterModule.forRoot\(\)](#) method. It returns a module, containing the configured [Router](#) service provider, plus other providers that the routing library requires. Once the application is bootstrapped, the [Router](#) performs the initial navigation based on the current browser URL.

Note: The `RouterModule.forRoot` method is a pattern used to register application-wide providers. Read more about application-wide providers in the [Singleton services](#) guide.

`src/app/app.module.ts (first-config)`

```
import { NgModule }           from '@angular/core';
import { BrowserModule }     from '@angular/platform-browser';
import { FormsModule }        from '@angular/forms';
import { RouterModule, 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 = [
  { path: 'crisis-center', component: CrisisListComponent },
  { path: 'heroes', component: HeroListComponent },
];

@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    RouterModule.forRoot(
      appRoutes,
      { enableTracing: true } // <-- debugging purposes only
    )
  ],
  declarations: [
    AppComponent,
    HeroListComponent,
    CrisisListComponent,
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

Adding the configured [RouterModule](#) to the [AppModule](#) is sufficient for simple route configurations. As the application grows, you'll want to [refactor the routing configuration](#) into a separate file and create a [Routing Module](#), a special type of Service Module dedicated to the purpose of routing in feature modules.

Registering the [RouterModule.forRoot\(\)](#) in the [AppModule](#) imports makes the [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 swaps components on and off the page. Here's what you get:



The router outlet serves as a placeholder when the routed components will be rendered below it.

The corresponding component template looks like this:

```
src/app/app.component.html
```

```
<h1>Angular Router</h1>
<nav>
  <a routerLink="/crisis-center" routerLinkActive="active">Crisis Center</a>
  <a routerLink="/heroes" routerLinkActive="active">Heroes</a>
</nav>
<router-outlet></router-outlet>
```

## Define a Wildcard route

You've created two routes in the app 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. The router will select this 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](#) to an existing route.

The router selects the route with a [first match wins](#) strategy. Wildcard routes are the least specific routes in the route configuration. Be sure it is the last route in the configuration.

To test this feature, add a button with a [RouterLink](#) to the [HeroListComponent](#) template and set the link to `"/sidekicks"`.

```
src/app/hero-list/hero-list.component.html (excerpt)
```

```
<h2>HEROES</h2>
<p>Get your heroes here</p>

<button routerLink="/sidekicks">Go to sidekicks</button>
```

The application will fail 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 instead and have it navigate to a simple PageNotFoundComponent.

```
src/app/app.module.ts (wildcard)
```

```
{ path: '**', component: PageNotFoundComponent }
```

Create the PageNotFoundComponent to display when users visit invalid URLs.

```
ng generate component page-not-found
```

```
src/app/page-not-found.component.html (404 component)
```

```
<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 something like:

```
localhost:4200
```

That doesn't match any of the concrete configured 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 app is the list of heroes. The app should navigate there as if the user clicked the "Heroes" link or pasted localhost:4200/heroes into the address bar.

The preferred solution is to add a redirect route that translates the initial relative URL ("") to the desired default path (/heroes). The browser address bar shows .../heroes as if you'd navigated there directly.

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)
```

```
const appRoutes: Routes = [
  { path: 'crisis-center', component: CrisisListComponent },
  { path: 'heroes', component: HeroListComponent },
```

```
{ path: "", redirectTo: '/heroes', pathMatch: 'full' },
{ path: '**', component: PageNotFoundComponent }
];
```

A redirect route requires a [pathMatch](#) property to tell the router how to match a URL to the path of a route. The router throws an error if you don't. 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'](#).

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.

Don't do that here. If the [pathMatch](#) value were 'prefix', every URL would match ".

Try setting it to 'prefix' then click the Go to sidekicks button. Remember that's a bad URL and 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 will be 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](#).

## Basics wrap up

You've got a very basic navigating app, one that can switch between two views when the user clicks a link.

You've learned how to do the following:

- Load the router library.
- Add a nav bar to the shell template with anchor tags, [routerLink](#) and [routerLinkActive](#) directives.
- Add a [router-outlet](#) to the shell template where views will be displayed.
- Configure the router module with `RouterModule.forRoot`.
- Set the router to compose HTML5 browser URLs.
- handle invalid routes with a wildcard route.
- navigate to the default route when the app launches with an empty path.

The starter app'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 discussed 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
```

```
<h1>Angular Router</h1>
<nav>
  <a routerLink="/crisis-center" routerLinkActive="active">Crisis Center</a>
  <a routerLink="/heroes" routerLinkActive="active">Heroes</a>
</nav>
<router-outlet></router-outlet>
```

## Milestone 2: Routing module

In the initial route configuration, you provided a simple setup with two routes used to configure the application for routing. This is perfectly fine for simple routing. As the application grows and you make use of more [Router](#) features, such as guards, resolvers, and child routing, you'll naturally want to refactor the routing configuration into its own file. We recommend moving the routing information into a special-purpose module called a Routing Module.

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 including 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 CLI to create a project that will use routing, set the `--routing` option for the project or app, and for each `NgModule`. When you create or initialize a new project (using the CLI `new` command) or a new app (using the `generate app` 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 app.

For example, the following command generates an `NgModule` that can use routing.

```
ng 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](#) object that you can fill with routes to different components and `NgModules`.

### Refactor the routing configuration into a routing module

Create an `AppRoutingModule` module in the `/app` folder to contain the routing configuration.

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

Import the `CrisisListComponent`, `HeroListComponent`, and `PageNotFoundCompononent` symbols just like you did in the `app.module.ts`. Then move the

[Router](#) imports and routing configuration, including `RouterModule.forRoot`, into this routing module.

Re-export the Angular [RouterModule](#) by adding it to the module [exports](#) array. By re-exporting the [RouterModule](#) here the components declared in `AppModule` will have access to router directives such as [RouterLink](#) and [RouterOutlet](#).

After these steps, the file should look like this.

`src/app/app-routing.module.ts`

```
1. import { NgModule }           from '@angular/core';
2. import { RouterModule, Routes } from '@angular/router';
3.
4. import { CrisisListComponent } from './crisis-list/crisis-list.component';
5. import { HeroListComponent }  from './hero-list/hero-list.component';
6. import { PageNotFoundComponent } from './page-not-found/page-not-
    found.component';
7.
8. const appRoutes: Routes = [
9.   { path: 'crisis-center', component: CrisisListComponent },
10.  { path: 'heroes',       component: HeroListComponent },
11.  { path: '/',           redirectTo: '/heroes', pathMatch: 'full' },
12.  { path: '**',          component: PageNotFoundComponent }
13.];
14.
15.@NgModule({
16.  imports: [
17.    RouterModule.forRoot(
18.      appRoutes,
19.      { enableTracing: true } // <-- debugging purposes only
20.    )
21.  ],
22.  exports: [
23.    RouterModule
24.  ]
25.})
26.export class AppRoutingModule {}
```

Next, update the `app.module.ts` file, removing `RouterModule.forRoot` in the [imports](#) array.

`src/app/app.module.ts`

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

```
4.
5. import { AppComponent } from './app.component';
6. import { AppRoutingModule } from './app-routing.module';
7.
8. import { CrisisListComponent } from './crisis-list/crisis-list.component';
9. import { HeroListComponent } from './hero-list/hero-list.component';
10.import { PageNotFoundComponent } from './page-not-found/page-not-
    found.component';
11.
12.@NgModule({
13. imports: [
14.   BrowserModule,
15.   FormsModule,
16.   AppRoutingModule
17. ],
18. declarations: [
19.   AppComponent,
20.   HeroListComponent,
21.   CrisisListComponent,
22.   PageNotFoundComponent
23. ],
24. bootstrap: [ AppComponent ]
25.})
26.export class AppModule { }
```

Later in this guide you will create [multiple routing modules](#) and discover that you must import those routing modules [in the correct order](#).

The application continues to work just the same, and you can use AppRoutingModule as the central place to maintain future routing configuration.

## Do you need a Routing Module?

The Routing Module replaces the routing configuration in the root or feature module. Either configure routes in the Routing Module or within the module itself but not in both.

The Routing Module is a design choice whose value is most obvious when the configuration is complex and includes specialized guard and resolver services. It can seem like overkill when the actual configuration is dead simple.

Some developers skip the Routing Module (for example, AppRoutingModule) when the configuration is simple and merge the routing configuration directly into the companion module (for example, AppModule).

Choose one pattern or the other and follow that pattern consistently.

Most developers should always implement a Routing Module for the sake of 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

You've seen how to navigate using the [RouterLink](#) directive. Now you'll learn the following:

- Organize the app and routes into feature areas using modules.
- Navigate imperatively from one component to another.
- Pass required and optional information in route parameters.

This example recreates the heroes feature in the "Services" episode of the [Tour of Heroes tutorial](#), and you'll be copying much of the code from the [Tour of Heroes: Services example code / download example](#).

Here's how the user will experience this version of the app:



A typical application has multiple feature areas, each dedicated to a particular business purpose.

While you could continue to add files to the `src/app/` folder, that is unrealistic and ultimately not maintainable. Most developers prefer to put each feature area in its own folder.

You are about to break up the app into different feature modules, each with its own concerns. Then you'll import into the main module and navigate among them.

## Add heroes functionality

Follow these steps:

- Create a `HeroesModule` with routing in the `heroes` folder and register it with the root `AppModule`. This is where you'll be implementing the hero management.

ng generate module heroes/heroes --module app --flat --routing

- Move the placeholder `hero-list` folder that's in the `app` into the `heroes` folder.
- Copy the contents of the `heroes/heroes.component.html` from the ["Services" tutorial / download example](#) into the `hero-list.component.html` template.
- Relabel 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 due to the components are dynamically inserted when the page is rendered, but 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, you'll update the `HeroesModule` metadata.

- Import and add the `HeroDetailComponent` and `HeroListComponent` to the [declarations](#) array in the `HeroesModule`.

`src/app/heroes/heroes.module.ts`

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

```
3. import { FormsModule } from '@angular/forms';
4.
5. import { HeroListComponent } from './hero-list/hero-list.component';
6. import { HeroDetailComponent } from './hero-detail/hero-detail.component';
7.
8. import { HeroesRoutingModule } from './heroes-routing.module';
9.
10. @NgModule({
11.   imports: [
12.     CommonModule,
13.     FormsModule,
14.     HeroesRoutingModule
15.   ],
16.   declarations: [
17.     HeroListComponent,
18.     HeroDetailComponent
19.   ]
20. })
21. export class HeroesModule {}
```

When you're done, you'll have these hero management files:

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. The list view is self-sufficient; you navigate to it, it gets a list of heroes and displays them.

The detail view is different. It displays a particular hero. It can't know which hero to show on its own. That information must come from outside.

When the user selects a hero from the list, the app should navigate to the detail view and show that 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, define the two hero routes.

Now that you have routes for the Heroes module, register them with the [Router](#) via the [RouterModule](#) almost as you did in the AppRoutingModule.

There is a small but critical difference. In the AppRoutingModule, you used the static `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` 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` method to register additional routes.

The updated HeroesRoutingModule looks like this:

`src/app/heroes/heroes-routing.module.ts`

```
1. import { NgModule }      from '@angular/core';
2. import { RouterModule, Routes } from '@angular/router';
3.
4. import { HeroListComponent } from './hero-list/hero-list.component';
5. import { HeroDetailComponent } from './hero-detail/hero-detail.component';
6.
7. const heroesRoutes: Routes = [
8.   { path: 'heroes', component: HeroListComponent },
9.   { path: 'hero/:id', component: HeroDetailComponent }
10.];
11.
12.@NgModule({
13.  imports: [
14.    RouterModule.forChild(heroesRoutes)
15.  ],
16.  exports: [
17.    RouterModule
18.  ]
}
```

```
19.})
20.export class HeroesRoutingModule { }
```

Consider giving each feature module its own route configuration file. It may seem like overkill early when the feature routes are simple. But routes have a tendency to grow more complex and consistency in patterns pays off over time.

### 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! These are concerns at the top level of the application itself.

src/app/app-routing.module.ts (v2)

```
import { NgModule }           from '@angular/core';
import { RouterModule, 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 = [
  { path: 'crisis-center', component: CrisisListComponent },
  // { path: 'heroes',      component: HeroListComponent }, // <-- delete this line
  { path: '',   redirectTo: '/heroes', pathMatch: 'full' },
  { path: '**', component: PageNotFoundComponent }
];
```

```
@NgModule({
  imports: [
    RouterModule.forRoot(
      appRoutes,
      { enableTracing: true } // <-- debugging purposes only
    )
  ],
  exports: [
    RouterModule
  ]
})
```

```
]  
}  
export class AppRoutingModule {}
```

## Remove heroes declarations

Remove the HeroListComponent from the AppModule's [declarations](#) because it's now provided by the HeroesModule. You can evolve the hero feature with more components and different routes. That's a key benefit of creating a separate module for each feature area.

After these steps, the AppModule should look like this:

src/app/app.module.ts

```
1. import { NgModule }      from '@angular/core';  
2. import { BrowserModule } from '@angular/platform-browser';  
3. import { FormsModule }   from '@angular/forms';  
4.  
5. import { AppComponent }   from './app.component';  
6. import { AppRoutingModule } from './app-routing.module';  
7. import { HeroesModule }   from './heroes/heroes.module';  
8.  
9. import { CrisisListComponent } from './crisis-list/crisis-list.component';  
10.import { PageNotFoundComponent } from './page-not-found/page-not-  
    found.component';  
11.  
12.@NgModule({  
13.  imports: [  
14.    BrowserModule,  
15.    FormsModule,  
16.    HeroesModule,  
17.    AppRoutingModule  
18.  ],  
19.  declarations: [  
20.    AppComponent,  
21.    CrisisListComponent,  
22.    PageNotFoundComponent  
23.  ],  
24.  bootstrap: [ AppComponent ]  
25.})  
26.export class AppModule { }
```

## Module import order matters

Look at the module [imports](#) array. Notice that the AppRoutingModule is last. Most importantly, it comes after the HeroesModule.

```
src/app/app.module.ts (module-imports)
```

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

The order of route configuration matters. 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](#) 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".

The routes are no longer in one file. They are distributed across two modules, AppRoutingModule and HeroesRoutingModule.

Each routing module augments the route configuration in the order of import. If you list AppRoutingModule first, the wildcard route will be registered before the hero routes. The wildcard route—which matches every URL—will intercept the attempt to navigate to a hero route.

Reverse the routing modules and see for yourself that a click of the heroes link results in "Page not found". Learn about inspecting the runtime router configuration [below](#).

## Route Parameters

### Route definition with a parameter

Return to the HeroesRoutingModule and look at the route definitions again. The route to HeroDetailComponent has a twist.

```
src/app/heroes/heroes-routing.module.ts (excerpt)
```

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

Notice the :id token in the path. That creates a slot in the path for a Route Parameter. In this case, the router will 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:

```
localhost: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.ts (link-parameters-array)
```

```
['/hero', hero.id] // { 15 }
```

The router composes the destination URL from the array like this: localhost:4200/hero/15.

How does the target HeroDetailComponent learn about that id? Don't analyze the URL. Let the router do it.

The router extracts the route parameter (id:15) from the URL and supplies it to the HeroDetailComponent via the [ActivatedRoute](#) service.

### Activated Route in action

Import the [Router](#), [ActivatedRoute](#), and [ParamMap](#) tokens from the router package.

```
src/app/heroes/hero-detail/hero-detail.component.ts (activated route)
```

```
import { Router, ActivatedRoute, 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)
```

```
import { switchMap } from 'rxjs/operators';
```

As usual, you write a constructor that asks Angular to inject services that the component requires and reference them as private variables.

```
src/app/heroes/hero-detail/hero-detail.component.ts (constructor)
```

```
constructor(  
  private route: ActivatedRoute,  
  private router: Router,  
  private service: HeroService  
) {}
```

Later, in the ngOnInit method, you use the [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)

ngOnInit() {
  this.hero$ = this.route.paramMap.pipe(
    switchMap((params: ParamMap) =>
      this.service.getHero(params.get('id')))
  );
}

```

The paramMap processing is a bit tricky. When the map changes, you get() 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.

You might think to use the RxJS map operator. But the HeroService returns an Observable<Hero>. So you flatten the Observable with the switchMap operator instead.

The switchMap operator also cancels previous in-flight 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.

The observable Subscription will be handled by the [AsyncPipe](#) and the component's hero property will be (re)set with the retrieved hero.

## ParamMap API

The [ParamMap](#) API is inspired by the [URLSearchParams interface](#). It provides methods to handle parameter access for both route parameters (paramMap) and query parameters (queryParamMap).

Member	Description
<code>has(name)</code>	Returns true if the parameter name is in the map of parameters.
<code>get(name)</code>	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.
<code>getAll(name)</code>	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.
<code>keys</code>	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.

They might. 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 don't want the router to remove the current HeroDetailComponent instance from the DOM only to re-create it for the next id. That could be visibly jarring. Better to simply re-use the same component instance and update the parameter.

Unfortunately, `ngOnInit` is only called once per component instantiation. You need a way to detect when the route parameters change from within the same instance. The observable `paramMap` property handles that beautifully.

When subscribing to an observable in a component, you almost always arrange to unsubscribe when the component is destroyed.

There are a few exceptional observables where this is not necessary. The [ActivatedRoute](#) observables are among the exceptions.

The [ActivatedRoute](#) and its observables are insulated from the [Router](#) itself. The [Router](#) destroys a routed component when it is no longer needed and the injected [ActivatedRoute](#) dies with it.

Feel free to unsubscribe anyway. It is harmless and never a bad practice.

## 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, never, ever be re-used, you can simplify the code with the snapshot.

The `route.snapshot` provides the initial value of the route parameter map. You can access the parameters directly without subscribing or adding observable operators. It's much simpler to write and read:

```
src/app/heroes/hero-detail/hero-detail.component.ts (ngOnInit snapshot)
```

```
ngOnInit() {  
  let id = this.route.snapshot.paramMap.get('id');
```

```
this.hero$ = this.service.getHero(id);  
}
```

Remember: you only get the initial value of the parameter map with this technique. Stick with the observable paramMap approach if there's even a chance that the router could re-use the component. This sample stays with the observable paramMap strategy just in case.

## Navigating back to the list component

The HeroDetailComponent has a "Back" button wired to its 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 [\[routerLink\]](#) directive. It holds the path to the HeroListComponent:

src/app/heroes/hero-detail/hero-detail.component.ts (excerpt)

```
gotoHeroes() {  
  this.router.navigate(['/heroes']);  
}
```

## Route Parameters: Required or optional?

Use [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:

localhost: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 was preselected in the list.

14	Celeritas
15	Magneta
16	RubberMan

You'll implement this feature in a moment by including the viewed hero's id in the URL as an optional parameter when returning from the HeroDetailComponent.

Optional information takes other forms. Search criteria are often loosely structured, e.g., name='wind\*'. Multiple values are common—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'.

These kinds of parameters don't fit easily in a URL path. Even if you could define a suitable URL token scheme, doing so greatly complicates the pattern matching required to translate an incoming URL to a named route.

Optional parameters are the ideal vehicle 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, prefer a required route parameter when the value is mandatory (for example, if necessary to distinguish one route path from another); prefer 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](#).

```
src/app/heroes/hero-list/hero-list.component.ts (link-parameters-array)
```

```
['/hero', hero.id] // { 15 }
```

The router embedded the id value in the navigation URL because you had defined it as a route parameter with an :idplaceholder token in the route path:

```
src/app/heroes/heroes-routing.module.ts (hero-detail-route)
```

```
{ 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)
```

```
gotoHeroes() {
  this.router.navigate(['/heroes']);
}
```

This array lacks a route parameter because you had no reason to send information to the HeroListComponent.

Now you have a reason. You'd like to send the id of the current hero with the navigation request so that theHeroListComponent can highlight that hero in its list. This is a nice-to-have feature; the list will display perfectly well without it.

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)
```

```
gotoHeroes(hero: Hero) {
  let herold = 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: herold, 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:

```
localhost: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—something you may not have seen before.

Matrix URL notation is an idea first introduced in a [1996 proposal](#) 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. The Router is such a system and provides support for the matrix notation across browsers.

The syntax may seem strange to you but users are unlikely to notice or care as long as the URL can be emailed and pasted into a browser address bar as this one can.

## Route parameters in the ActivatedRoute service

The list of heroes is unchanged. No hero row is highlighted.

The [live example](#) / [download example](#) does highlight the selected row because it demonstrates the final state of the application which includes the steps you're about to cover. At the moment this guide is describing the state of affairs prior to those steps.

The HeroListComponent isn't expecting any parameters at all and wouldn't know what to do with them. You can change that.

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](#) 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 you extend the router import statement to include the [ActivatedRoute](#) service symbol:

```
src/app/heroes/hero-list/hero-list.component.ts (import)
```

```
import { 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)
```

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

```
import { switchMap } from 'rxjs/operators';
```

Then you inject the [ActivatedRoute](#) in the HeroListComponent constructor.

```
src/app/heroes/hero-list/hero-list.component.ts (constructor and ngOnInit)
```

```
export class HeroListComponent implements OnInit {
```

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

```
  selectedId: number;
```

```
  constructor(
```

```
    private service: HeroService,
```

```
    private route: ActivatedRoute
```

```
  ) {}
```

```
  ngOnInit() {
```

```
    this.heroes$ = this.route.paramMap.pipe(
```

```
      switchMap(params => {
```

```
        // (+) before `params.get()` turns the string into a number
```

```
        this.selectedId = +params.get('id');
```

```
        return this.service.getHeroes();
```

```
      })
```

```
    );
```

```
  }
```

```
}
```

The [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](#). 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

```
<h2>HEROES</h2>
<ul class="heroes">
<li *ngFor="let hero of heroes$ | async"
    [class.selected]="hero.id === selectedId">
    <a [routerLink]=["/hero", hero.id]>
        <span class="badge">{{ hero.id }}</span>{{ hero.name }}
    </a>
</li>
</ul>
```

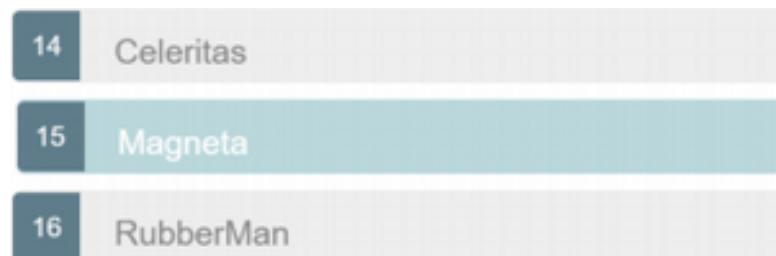
```
<button routerLink="/sidekicks">Go to sidekicks</button>
```

Add some styles to apply when the list item is selected.

src/app/heroes/hero-list/hero-list.component.css

```
.heroes li.selected {
background-color: #CFD8DC;
color: white;
}
.heroes li.selected:hover {
background-color: #BBD8DC;
}
```

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 continues to be ignored.

## Adding routable animations

### Adding animations to the routed component

The heroes feature module is almost complete, but what is a feature without some smooth transitions?

This section shows you how to add some [animations](#) to the HeroDetailComponent.

First import the [BrowserAnimationsModule](#) and add it to the [imports](#) array:

src/app/app.module.ts (animations-module)

```
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
```

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

Next, add a data object to the routes for HeroListComponent and HeroDetailComponent.

Transitions are based on states and you'll use the [animation](#) data from the route to provide a named animation [state](#) for the transitions.

src/app/heroes/heroes-routing.module.ts (animation data)

```
1. import { NgModule }           from '@angular/core';
2. import { RouterModule, Routes } from '@angular/router';
3.
4. import { HeroListComponent }   from './hero-list/hero-list.component';
5. import { HeroDetailComponent } from './hero-detail/hero-detail.component';
6.
7. const heroesRoutes: Routes = [
8.   { path: 'heroes', component: HeroListComponent, data: { animation: 'heroes' } },
9.   { path: 'hero/:id', component: HeroDetailComponent, data: { animation: 'hero' } }
10.];
11.
12. @NgModule({
13.   imports: [
14.     RouterModule.forChild(heroesRoutes)
15.   ],
16.   exports: [
17.     RouterModule
18.   ]
19.})
20. export class HeroesRoutingModule { }
```

Create an animations.ts file in the root src/app/ folder. The contents look like this:

src/app/animations.ts (excerpt)

```
import {
  trigger, animateChild, group,
```

## [transition](#), [animate](#), [style](#), [query](#)

} from '@angular/animations';

```
// Routable animations
export const slideInAnimation =
  trigger('routeAnimation', [
    transition('heroes <=> hero', [
      style({ position: 'relative' }),
      query(':enter, :leave', [
        style({
          position: 'absolute',
          top: 0,
          left: 0,
          width: '100%'
        })
      ]),
      query(':enter', [
        style({ left: '-100%' })
      ]),
      query(':leave', animateChild()),
      group([
        query(':leave', [
          animate('300ms ease-out', style({ left: '100%' }))
        ]),
        query(':enter', [
          animate('300ms ease-out', style({ left: '0%' }))
        ])
      ]),
      query(':enter', 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).

You could also create more transitions for other routes. This trigger is sufficient for the current milestone.

Back in the AppComponent, import the [RouterOutlet](#) token from the @angular/router package and the slideInAnimation from ./animations.ts.

Add an animations array to the [@Component](#) metadata's that contains the slideInAnimation.

```
src/app/app.component.ts (animations)
```

```
import { RouterOutlet } from '@angular/router';
import { slideInAnimation } from './animations';
```

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

In order to use the routable animations, you'll need to wrap the [RouterOutlet](#) inside an element. You'll use the [@routeAnimation](#) trigger and bind it to the element.

For the [@routeAnimation](#) transitions to key off states, you'll need to provide it with the data from the [ActivatedRoute](#). The [RouterOutlet](#) is exposed as an outlet template variable, so you bind a reference to the router outlet. A variable of routerOutlet is an ideal choice.

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

```
<h1>Angular Router</h1>
<nav>
  <a routerLink="/crisis-center" routerLinkActive="active">Crisis Center</a>
  <a routerLink="/heroes" routerLinkActive="active">Heroes</a>
</nav>
<div [@routeAnimation]="getAnimationData(routerOutlet)">
  <router-outlet #routerOutlet="outlet"></router-outlet>
</div>
```

The [@routeAnimation](#) property is bound to the [getAnimationData](#) with the provided routerOutlet reference, so you'll need to define that function in the AppComponent. The [getAnimationData](#) function returns the animation property from the data provided through the [ActivatedRoute](#). The [animation](#) property matches the [transition](#) names you used in the slideDownAnimation defined in [animations.ts](#).

```
src/app/app.component.ts (router outlet)
```

```
export class AppComponent {
  getAnimationData(outlet: RouterOutlet) {
```

```
    return outlet && outlet.activatedRouteData && outlet.activatedRouteData['animation'];
}
}
```

When switching between the two routes, the HeroDetailComponent and HeroListComponent will ease in from the left when routed to and will slide to the right when navigating away.

## Milestone 3 wrap up

You've learned how to do the following:

- Organize the app into feature areas.
- Navigate imperatively from one component to another.
- Pass information along in route parameters and subscribe to them in the component.
- Import the feature area NgModule into the AppModule.
- Applying routable animations based on the page.

After these changes, the folder structure looks like this:

```
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

```
1. import {  
2.   trigger, animateChild, group,  
3.   transition, animate, style, query  
4. } from '@angular/animations';  
5.  
6.  
7. // Routable animations  
8. export const slideInAnimation =  
9.   trigger('routeAnimation', [  
10.    transition('heroes <=> hero', [  
11.      style({ position: 'relative' }),  
12.      query(':enter, :leave', [  
13.        style({  
14.          position: 'absolute',  
15.          top: 0,  
16.          left: 0,  
17.          width: '100%'  
18.        })  
19.      ]),  
20.      query(':enter', [  
21.        style({ left: '-100%' })  
22.      ]),  
23.      query(':leave', animateChild()),  
24.      group([  
25.        query(':leave', [  
26.          animate('300ms ease-out', style({ left: '100%' }))  
27.        ]),  
28.        query(':enter', [  
29.          animate('300ms ease-out', style({ left: '0%' }))  
30.        ])  
31.      ]),  
32.      query(':enter', animateChild()),  
33.    ])  
34. ]);
```

## Milestone 4: Crisis center feature

It's time to add real features to the app's current placeholder crisis center.

Begin by imitating 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.

You'll use mock crises instead of mock heroes:

src/app/crisis-center/mock-crises.ts

```
import { 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 and decide later if the differences are worthwhile.

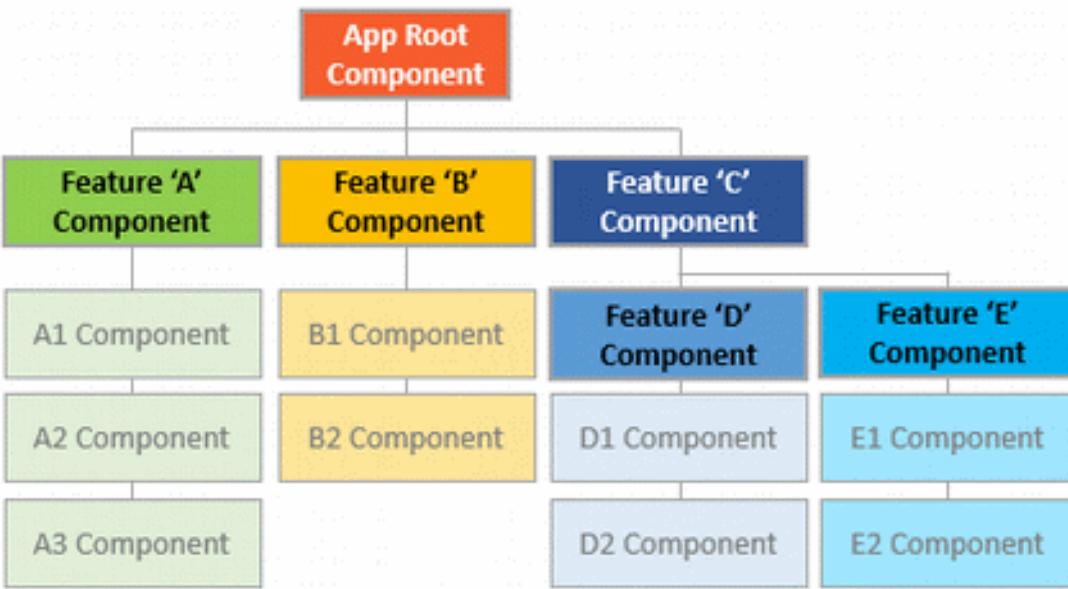
In keeping with the [Separation of Concerns principle](#), changes to the Crisis Center won't affect the AppModule or any other feature's component.

### 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 app had many feature areas, the app component trees might look like this:



## Child routing component

Generate a CrisisCenter component in the crisis-center folder:

```
ng generate component crisis-center/crisis-center
```

Update the component template to look like this:

```
src/app/crisis-center/crisis-center/crisis-center.component.html
```

```
<h2>CRISIS CENTER</h2>
<router-outlet></router-outlet>
```

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 very simple, simpler even than AppComponent: it has no business logic, and its template has no links, just a title and `<router-outlet>` 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.

```
ng 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
```

```
<p>Welcome to the Crisis Center</p>
```

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)

```
const crisisCenterRoutes: Routes = [
  {
    path: 'crisis-center',
    component: CrisisCenterComponent,
    children: [
      {
        path: '',
        component: CrisisListComponent,
        children: [
          {
            path: ':id',
            component: CrisisDetailComponent
          },
          {
            path: '',
            component: CrisisCenter HomeComponent
          }
        ]
      }
    ]
  ];
];
```

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 these child routes.

The router displays the components of these routes in the [RouterOutlet](#) of the CrisisCenterComponent, not in the [RouterOutlet](#) of the AppComponent shell.

The CrisisListComponent contains the crisis list and a [RouterOutlet](#) to display the Crisis Center Home and Crisis Detailroute components.

The Crisis Detail route is a child of the Crisis [List](#). The router [reuses components](#) by default, so the Crisis Detailcomponent will be re-used as you select different crises. In contrast, back

in the Hero Detail route, the component was recreated each time you selected a different hero.

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 CrisisCenter HomeComponent, the full URL is /crisis-center (/crisis-center + " + ").
- To navigate to the CrisisDetail Component 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

localhost: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)

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

import { CrisisCenter HomeComponent } 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 = [
{
  path: 'crisis-center',
  component: CrisisCenterComponent,
  children: [
    {
      path: '',
      component: CrisisListComponent,
      children: [
        {
          path: ':id',
          component: CrisisDetailComponent
        },
        {
          path: '',
          component: CrisisCenter HomeComponent
        }
      ]
    }
  ]
}
```

```

        }
    ]
}
];
}

@NgModule({
  imports: [
    RouterModule.forChild(crisisCenterRoutes)
  ],
  exports: [
    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)

1. import { [NgModule](#) } from '@angular/core';
2. import { [FormsModule](#) } from '@angular/forms';
3. import { [CommonModule](#) } from '@angular/common';
- 4.
5. import { CrisisCenter HomeComponent } from './crisis-center-home/crisis-center-home.component';
6. import { CrisisListComponent } from './crisis-list/crisis-list.component';
7. import { CrisisCenterComponent } from './crisis-center/crisis-center.component';
8. import { CrisisDetailComponent } from './crisis-detail/crisis-detail.component';
- 9.
10. import { CrisisCenterRoutingModule } from './crisis-center-routing.module';
- 11.
12. [`@NgModule\({`](#)
13. [`imports: \[`](#)
14.  [`CommonModule,`](#)
15.  [`FormsModule,`](#)
16.  [`CrisisCenterRoutingModule`](#)
17. [`\],`](#)
18. [`declarations: \[`](#)

```
19. CrisisCenterComponent,  
20. CrisisListComponent,  
21. CrisisCenter HomeComponent,  
22. CrisisDetailComponent  
23.]  
24.})  
25.export class CrisisCenterModule {}
```

Remove the initial crisis center route from the app-routing.module.ts. The feature routes are now provided by the HeroesModule and the CrisisCenter modules.

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)

```
import { NgModule }           from '@angular/core';  
import { RouterModule, Routes } from '@angular/router';  
  
import { PageNotFoundComponent } from './page-not-found/page-not-found.component';  
  
const appRoutes: Routes = [  
  { path: '', redirectTo: '/heroes', pathMatch: 'full' },  
  { path: '**', component: PageNotFoundComponent }  
];  
  
@NgModule({  
  imports: [  
    RouterModule.forRoot(  
      appRoutes,  
      { enableTracing: true } // <-- debugging purposes only  
    )  
  ],  
  exports: [  
    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.

Here's an example:

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](#) 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](#). 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](#) that you need to compose the relative navigation path.

When using a [RouterLink](#) to navigate instead of the [Router](#) service, you'd use the same link parameters array, but you wouldn't provide the object with the [relativeTo](#) property. The [ActivatedRoute](#) is implicit in a [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)
```

```
// 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 be displaying 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)

```
<div [@routeAnimation]="getAnimationData(routerOutlet)">
  <router-outlet #routerOutlet="outlet"></router-outletrouter-outlet name="popup"></router-outlet
```

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.

ng generate component compose-message

It displays a simple form with a header, an input box for the message, and two buttons, "Send" and "Cancel".

## Contact Crisis Center

Message:

```
I want a crisis of my own
```

[Send](#) [Cancel](#)

Here's the component, its template and styles:

```
src/app/compose-message/compose-message.component.css
```

```
src/app/compose-message/compose-message.component.html
```

```
src/app/compose-message/compose-message.component.ts
```

```
:host {  
  position: relative; bottom: 10%;  
}
```

It looks about the same as any other component you've seen in this guide. There are two noteworthy 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. That's a peculiarity covered [below](#).

### Add a secondary route

Open the `AppRoutingModule` and add a new compose route to the `appRoutes`.

```
src/app/app-routing.module.ts (compose route)
```

```
{  
  path: 'compose',  
  component: ComposeMessageComponent,  
  outlet: 'popup'  
},
```

The path and component properties should be familiar. There's a new property, outlet, set to 'popup'. This route now targets the popup outlet and the ComposeMessageComponent will display there.

The user needs a way to open the popup. Open the AppComponent and add a "Contact" link.

src/app/app.component.html (contact-link)

```
<a [routerLink]="[{ outlets: { popup: ['compose'] } }]>Contact</a>
```

Although the compose route is pinned to the "popup" outlet, that's not sufficient for wiring the route to a [RouterLink](#) directive. You have to specify the named outlet in a link parameters array and bind it to the [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.

You are in effect saying, when the user clicks this link, display the component associated with the compose route in the popupoutlet.

This outlets object within an outer object was completely unnecessary when there was only one route and one unnamed outlet to think about.

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 previously hidden router truth: you can target multiple outlets with multiple routes in the same [RouterLink](#) directive.

You're not actually doing that here. But to target a named outlet, you must use the richer, more verbose syntax.

### 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.

[http://.../crisis-center\(popup:compose\)](http://.../crisis-center(popup:compose))

The interesting 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.

[http://.../heroes\(popup:compose\)](http://.../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. The router will generate the URL to go with it.

You can tell the router to navigate an entire tree at once by filling out the outlets object mentioned above. Then pass that object inside a link parameters array to the router.navigate method.

Experiment with these possibilities at your leisure.

### **Clearing secondary routes**

As you've learned, a component in an outlet persists until you navigate away to a new component. Secondary outlets are no different in this regard.

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.

Clicking the "send" or "cancel" buttons does clear the popup view. To see how, look at the closePopup() method again:

```
src/app/compose-message/compose-message.component.ts (closePopup)  
  
closePopup() {  
  // Providing a `null` value to the named outlet  
  // clears the contents of the named outlet  
  this.router.navigate([{ outlets: { popup: null }}]);  
}
```

It navigates imperatively with the [Router.navigate\(\)](#) method, passing in a [link parameters array](#).

Like the array bound to the Contact [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](#) 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 anytime. That's not always the right thing to do.

- 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.

Note: The guard can also tell the router to navigate elsewhere, effectively canceling the current navigation.

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 must also complete. If the observable does not complete, the navigation will not continue.

The router supports multiple guard interfaces:

- [CanActivate](#) to mediate navigation to a route.
- [CanActivateChild](#) to mediate navigation to a child route.
- [CanDeactivate](#) to mediate navigation away from the current route.
- [Resolve](#) to perform route data retrieval before route activation.
- [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](#) and [CanActivateChild](#) guards first, from the deepest child route to the top.

Then it checks the [CanActivate](#) guards from the top down to the deepest child route. If the feature module is loaded asynchronously, the [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: 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](#) guard is the tool to manage these navigation business rules.

## Add an admin feature module

In this next section, you'll extend the crisis center with some new administrative features. Those features aren't defined yet. But you can start by adding a new feature module named AdminModule.

Generate an admin folder with a feature module file and a routing configuration file.

```
ng generate module admin --routing
```

Next, generate the supporting components.

```
ng generate component admin/admin-dashboard
```

```
ng generate component admin/admin
```

```
ng generate component admin/manage-crises
```

```
ng 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

```
<h3>ADMIN</h3>
```

```
<nav>
```

```
  <a routerLink="./" routerLinkActive="active"
    [routerLinkActiveOptions]="{ exact: true }">Dashboard</a>
  <a routerLink=".//crises" routerLinkActive="active">Manage Crises</a>
  <a routerLink=".//heroes" routerLinkActive="active">Manage Heroes</a>
```

```
</nav>
```

```
<router-outlet></router-outlet>
```

Although the admin dashboard [RouterLink](#) only contains a relative slash without an additional URL segment, it is considered 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)

```
const adminRoutes: Routes = [
  {
    path: 'admin',
    component: AdminComponent,
    children: [
      {
        path: '',
        children: [
          { path: 'crises', component: ManageCrisesComponent },
          { path: 'heroes', component: ManageHeroesComponent },
          { path: '', component: AdminDashboardComponent }
        ]
      ]
    ]
}
```

```

        }
    ]
}
];
;

@NgModule({
  imports: [
    RouterModule.forChild(adminRoutes)
  ],
  exports: [
    RouterModule
  ]
})
export class AdminRoutingModule {}

```

Looking at the child route under the AdminComponent, there is a path and a children property but it's not using a component. You haven't made a mistake in the configuration. You've defined a component-less route.

The goal is to group the Crisis Center management routes under the admin path. You don't need a component to do it. A component-less route makes it easier to [guard child routes](#).

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)

```

import { NgModule }      from '@angular/core';
import { CommonModule } from '@angular/common';
import { 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({
  imports: [
    CommonModule,

```

```

    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)

```

<h1 class="title">Angular Router</h1>
<nav>
  <a routerLink="/crisis-center" routerLinkActive="active">Crisis Center</a>
  <a routerLink="/heroes" routerLinkActive="active">Heroes</a>
  <a routerLink="/admin" routerLinkActive="active">Admin</a>
  <a [routerLink]="[{ outlets: { popup: ['compose'] } }]">Contact</a>
</nav>
<div [@routeAnimation]="getAnimationData(routerOutlet)">
  <router-outlet #routerOutlet="outlet"></router-outlet>
</div>
<router-outlet name="popup"></router-outlet>

```

### 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.

You could hide the link until the user logs in. But that's tricky and difficult to maintain.

Instead you'll write a [canActivate\(\)](#) guard method to redirect anonymous users to the login page when they try to enter the admin area.

This is a general purpose guard—you can imagine other features that require authenticated users—so you generate anAuthGuard in the auth folder.

ng generate guard auth/auth

At the moment you're interested in seeing how guards work so the first version does nothing useful. It simply logs to console and returns true immediately, allowing navigation to proceed:

src/app/auth/auth.guard.ts (excerpt)

```
import { Injectable } from '@angular/core';
import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot } from '@angular/router';

@Injectable({
  providedIn: 'root',
})
export class AuthGuard implements CanActivate {
  canActivate(
    next: ActivatedRouteSnapshot,
    state: 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)

```
import { AuthGuard } from './auth/auth.guard';

const adminRoutes: Routes = [
{
  path: 'admin',
  component: AdminComponent,
  canActivate: [AuthGuard],
  children: [
    {
      path: '',
      children: [
        { path: 'crises', component: ManageCrisesComponent },
        { path: 'heroes', component: ManageHeroesComponent },
        { path: '', component: AdminDashboardComponent }
      ],
    }
  ]
};

@NgModule({
  imports: [
```

```
    RouterModule.forChild(adminRoutes)
  ],
  exports: [
    RouterModule
  ]
})
export class AdminRoutingModule {}
```

The admin feature is now protected by the guard, albeit protected poorly.

### Teach AuthGuard to authenticate

Make the AuthGuard at least pretend to authenticate.

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 admin folder:

```
ng generate service admin/auth
```

Update the AuthService to log in the user:

```
src/app/auth/auth.service.ts (excerpt)
```

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

```
import { Observable, of } from 'rxjs';
import { tap, delay } from 'rxjs/operators';
```

```
@Injectable({
  providedIn: 'root',
})
export class AuthService {
  isLoggedIn = false;
```

```
// store the URL so we can redirect after logging in
redirectUrl: string;
```

```
login(): Observable<boolean> {
  return of(true).pipe(
    delay(1000),
    tap(val => this.isLoggedIn = true)
  );
}
```

```
logout(): void {
  this.isLoggedIn = false;
```

```
}
```

Although it doesn't actually log in, it has what you need for this discussion. 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 will store the attempted URL so you can navigate to it after authenticating.

Revise the AuthGuard to call it.

src/app/auth/auth.guard.ts (v2)

```
import { Injectable } from '@angular/core';
import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot, Router } from '@angular/router';

import { AuthService } from './auth.service';

@Injectable({
  providedIn: 'root',
})
export class AuthGuard implements CanActivate {
  constructor(private authService: AuthService, private router: Router) {}

  canActivate(
    next: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): boolean {
    let url: string = state.url;

    return this.checkLogin(url);
  }

  checkLogin(url: string): boolean {
    if (this.authService.isLoggedIn) { return true; }

    // Store the attempted URL for redirecting
    this.authService.redirectUrl = url;

    // Navigate to the login page with extras
    this.router.navigate(['/login']);
    return false;
  }
}
```

Notice that you inject the AuthService and the [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](#) contains the future route that will be activated and the [RouterStateSnapshot](#) contains the future [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](#) and tell the router to navigate to a login page—a page you haven't created yet. This secondary navigation automatically cancels the current navigation; checkLogin() returns false just to be clear about that.

## Add the LoginComponent

You need a LoginComponent for the user to log in to the app. 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 wire it into the router configuration.

ng 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

```
1. import { NgModule }      from '@angular/core';
2. import { BrowserModule } from '@angular/platform-browser';
3. import { FormsModule }   from '@angular/forms';
4. import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
5.
6. import { AppComponent }      from './app.component';
7. import { PageNotFoundComponent } from './page-not-found/page-not-
found.component';
8. import { ComposeMessageComponent } from './compose-message/compose-
message.component';
9.
10.import { AppRoutingModule }   from './app-routing.module';
11.import { HeroesModule }      from './heroes/heroes.module';
12.import { AuthModule }        from './auth/auth.module';
```

```
13.  
14. @NgModule({  
15.   imports: [  
16.     BrowserModule,  
17.     BrowserAnimationsModule,  
18.     FormsModule,  
19.     HeroesModule,  
20.     AuthModule,  
21.     AppRoutingModule,  
22.   ],  
23.   declarations: [  
24.     AppComponent,  
25.     ComposeMessageComponent,  
26.     PageNotFoundComponent  
27.   ],  
28.   bootstrap: [ AppComponent ]  
29. })  
30. export class AppModule {  
31. }
```

## CanActivateChild: guarding child routes

You can also protect child routes with the [CanActivateChild](#) guard. The [CanActivateChild](#) guard is similar to the [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](#) 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](#) and [RouterStateSnapshot](#). The [canActivateChild\(\)](#) method can return an `Observable<boolean>` or `Promise<boolean>` for async checks and a boolean for sync checks. This one returns a boolean:

src/app/auth/auth.guard.ts (excerpt)

```
import { Injectable }      from '@angular/core';  
import {  
  CanActivate, Router,  
  ActivatedRouteSnapshot,  
  RouterStateSnapshot,  
  CanActivateChild
```

```

}           from '@angular/router';
import { AuthService } from './auth.service';

@Injectable({
  providedIn: 'root',
})
export class AuthGuard implements CanActivate, CanActivateChild {
  constructor(private authService: AuthService, private router: Router) {}

  canActivate(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): boolean {
    let url: string = state.url;

    return this.checkLogin(url);
  }

  canActivateChild(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): boolean {
    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)

```

const adminRoutes: Routes = [
{
  path: 'admin',
  component: AdminComponent,
  canActivate: [AuthGuard],
  children: [
    {
      path: '',
      canActivateChild: [AuthGuard],
      children: [
        { path: 'crises', component: ManageCrisesComponent },
        { path: 'heroes', component: ManageHeroesComponent },
        { path: '', component: AdminDashboardComponent }
      ]
    }
  ]
}

```

```
        ]
    }
]
}
];
}

@NgModule({
  imports: [
    RouterModule.forChild(adminRoutes)
  ],
  exports: [
    RouterModule
  ]
})
export class AdminRoutingModule {}
```

## CanDeactivate: handling unsaved changes

Back in the "Heroes" workflow, the app accepts every change to a hero immediately without hesitation or validation.

In the real world, you might have to accumulate the user's changes. You might have to validate across fields. You might have to validate on the server. You might have to hold changes in a pending state until the user confirms them as a group or cancels and reverts all changes.

What do you do about unapproved, unsaved changes when the user navigates away? You can't just leave and risk losing the user's changes; that would be a terrible experience.

It's better to pause and let the user decide what to do. If the user cancels, you'll stay put and allow more changes. If the user approves, the app can save.

You still might delay navigation until the save succeeds. If you let the user move to the next screen immediately and the save were to fail (perhaps the data are ruled invalid), you would lose the context of the error.

You can't block while waiting for the server—that's not possible in a browser. You need to stop the navigation while you wait, asynchronously, for the server to return with its answer.

You need the [CanDeactivate](#) guard.

## Cancel and save

The sample application doesn't talk to a server. Fortunately, you have another way to demonstrate an asynchronous router hook.

Users update crisis information in the CrisisDetailComponent. Unlike the HeroDetailComponent, the user changes do not update the crisis entity immediately. Instead,

the app 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)

```
cancel() {  
  this.gotoCrises();  
}  
  
save() {  
  this.crisis.name = this.editName;  
  this.gotoCrises();  
}
```

What if the user tries to navigate away without saving or canceling? The user could push the browser back button or click the heroes link. Both actions trigger a navigation. Should the app save or cancel automatically?

This demo does neither. Instead, it asks the user to make that choice explicitly in a confirmation dialog box that waits asynchronously for the user's answer.

You could wait for the user's answer with synchronous, blocking code. The app will be more responsive—and can do other work—by waiting for the user's answer asynchronously.

Waiting for the user asynchronously is like waiting for the server asynchronously.

Generate a Dialog service to handle user confirmation.

ng 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

```
1. import { Injectable } from '@angular/core';  
2. import { Observable, of } from 'rxjs';  
3.  
4. /**  
5. * Async modal dialog service  
6. * DialogService makes this app easier to test by faking this service.  
7. * TODO: better modal implementation that doesn't use window.confirm  
8. */  
9. @Injectable({  
10. providedIn: 'root',  
11.})  
12. export class DialogService {  
13. /**
```

```

14. * Ask user to confirm an action. `message` explains the action and choices.
15. * Returns observable resolving to `true`=confirm or `false`=cancel
16. */
17. confirm(message?: string): Observable<boolean> {
18.   const confirmation = window.confirm(message || 'Is it OK?');
19.
20.   return of(confirmation);
21. };
22.

```

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.

ng generate guard can-deactivate

The CrisisDetailComponent will have this method. But the guard doesn't have to know that. The guard shouldn't know the details of any component's deactivation method. It need only detect that the component has a [canDeactivate\(\)](#) method and call it. This approach makes the guard reusable.

src/app/can-deactivate.guard.ts

```

1. import { Injectable } from '@angular/core';
2. import { CanDeactivate } from '@angular/router';
3. import { Observable } from 'rxjs';
4.
5. export interface CanComponentDeactivate {
6.   canDeactivate: () => Observable<boolean> | Promise<boolean> | boolean;
7. }
8.
9. @Injectable({
10.   providedIn: 'root',
11. })
12.export class CanDeactivateGuard implements CanDeactivate<CanComponentDeactiv
ate> {
13.   canDeactivate(component: CanComponentDeactivate) {
14.     return component.canDeactivate ? component.canDeactivate() : true;
15.   }
16.

```

Alternatively, you could make a component-specific [CanDeactivate](#) guard for the CrisisDetailComponent. The [canDeactivate\(\)](#) method provides you with the current instance

of the component, the current [ActivatedRoute](#), and [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)

```
import { Injectable }      from '@angular/core';
import { Observable }     from 'rxjs';
import { CanDeactivate,
        ActivatedRouteSnapshot,
        RouterStateSnapshot } from '@angular/router';

import { CrisisDetailComponent } from './crisis-center/crisis-detail/crisis-detail.component';

@Injectable({
  providedIn: 'root',
})
export class CanDeactivateGuard implements CanDeactivate<CrisisDetailComponent> {

  canDeactivate(
    component: CrisisDetailComponent,
    route: ActivatedRouteSnapshot,
    state: 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)

```

canDeactivate(): 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 put).

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)`

```

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

import { CrisisCenter HomeComponent } 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 = [
{
  path: 'crisis-center',
  component: CrisisCenterComponent,
  children: [
    {
      path: '',
      component: CrisisListComponent,
      children: [
        {
          path: ':id',
          component: CrisisDetailComponent,
          canDeactivate: [CanDeactivateGuard]
        },
        {

```

```
        path: '',
        component: CrisisCenter HomeComponent
    }
]
}
]
}
];

```

```
@NgModule({
imports: [
    RouterModule.forChild(crisisCenterRoutes)
],
exports: [
    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 app waited until the route was activated to fetch the respective hero or crisis.

This worked well, but there's a better way. 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.

It's preferable to pre-fetch data from the server 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 have been fetched.

You need a resolver.

## Fetch data before navigating

At the moment, the CrisisDetailComponent retrieves the selected crisis. If the crisis is not found, it 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 does not exist before activating the route and creating the CrisisDetailComponent.

Generate a CrisisDetailResolver service file within the Crisis Center feature area.

```
ng generate service crisis-center/crisis-detail-resolver
```

```
src/app/crisis-center/crisis-detail-resolver.service.ts (generated)
```

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

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

  constructor() { }

}
```

Take the relevant parts of the crisis retrieval logic in CrisisDetailComponent.ngOnInit and move them into the CrisisDetailResolverService. Import the Crisis model, CrisisService, and the [Router](#) so you can navigate elsewhere if you can't fetch the crisis.

Be explicit. Implement the [Resolve](#) interface with a type of Crisis.

Inject the CrisisService and [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](#) guards require an observable to complete, meaning it has emitted all of its values. You use the takeoperator 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, return an empty Observable, canceling the previous in-flight navigation to the CrisisDetailComponent and navigate the user back to the CrisisListComponent. The update resolver service looks like this:

```
src/app/crisis-center/crisis-detail-resolver.service.ts
```

1. import { Injectable } from '@angular/core';
2. import {
3. [Router](#), [Resolve](#),
4. [RouterStateSnapshot](#),
5. [ActivatedRouteSnapshot](#)
6. } from '@angular/router';
7. import { Observable, of, EMPTY } from 'rxjs';

```

8. import { mergeMap, take }      from 'rxjs/operators';
9.
10.import { CrisisService } from './crisis.service';
11.import { Crisis } from './crisis';
12.
13.@Injectable({
14.  providedIn: 'root',
15.})
16.export class CrisisDetailResolverService implements Resolve<Crisis> {
17.  constructor(private cs: CrisisService, private router: Router) {}
18.
19.  resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<Crisis> | Observable<never> {
20.    let id = route.paramMap.get('id');
21.
22.    return this.cs.getCrisis(id).pipe(
23.      take(1),
24.      mergeMap(crisis => {
25.        if (crisis) {
26.          return of(crisis);
27.        } else { // id not found
28.          this.router.navigate(['/crisis-center']);
29.          return EMPTY;
30.        }
31.      })
32.    );
33.  }
34.}

```

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)

```

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

import { CrisisCenter HomeComponent } 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 = [
{
  path: 'crisis-center',
  component: CrisisCenterComponent,
  children: [
    {
      path: '',
      component: CrisisListComponent,
      children: [
        {
          path: ':id',
          component: CrisisDetailComponent,
          canDeactivate: [CanDeactivateGuard],
          resolve: {
            crisis: CrisisDetailResolverService
          }
        },
        {
          path: '',
          component: CrisisCenterHomeComponent
        }
      ]
    }
  ]
};

@NgModule({
  imports: [
    RouterModule.forChild(crisisCenterRoutes)
  ],
  exports: [
    RouterModule
  ]
})
export class CrisisCenterRoutingModule { }

```

The CrisisDetailComponent should no longer fetch the crisis. Update the CrisisDetailComponent to get the crisis from theActivatedRoute.data.crisis property instead; that's where you said it should be when you re-configured the route. It will be there when the CrisisDetailComponent ask for it.

```
src/app/crisis-center/crisis-detail/crisis-detail.component.ts (ngOnInit v2)
```

```
ngOnInit() {
  this.route.data
    .subscribe((data: { crisis: Crisis }) => {
      this.editName = data.crisis.name;
      this.crisis = data.crisis;
    });
}
```

Two critical points

1. The router's [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. Rely on the router to call the resolver. Don't worry about all the ways that the user could navigate away. That's the router's job. Write this class and let the router take it from there.

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.html

crisis-detail-resolver.service.ts

crisis.service.ts

dialog.service.ts

1. <h1 class="title">Angular [Router](#)</h1>
2. <nav>
3. <a routerLink="/crisis-center" routerLinkActive="active">Crisis Center</a>
4. <a routerLink="/superheroes" routerLinkActive="active">Heroes</a>
5. <a routerLink="/admin" routerLinkActive="active">Admin</a>
6. <a routerLink="/login" routerLinkActive="active">Login</a>
7. <a [routerLink]="{{ outlets: { popup: ['compose'] } }}>Contact</a>
8. </nav>
9. <div [@routeAnimation]="getAnimationData(routerOutlet)">

```
10. <router-outlet #routerOutlet="outlet"></router-outlet>
11.</div>
12.<router-outlet name="popup"></router-outlet>
```

## Guards

auth.guard.ts

can-deactivate.guard.ts

```
1. import { Injectable }      from '@angular/core';
2. import {
3.   CanActivate, Router,
4.   ActivatedRouteSnapshot,
5.   RouterStateSnapshot,
6.   CanActivateChild
7. }              from '@angular/router';
8. import { AuthService }    from './auth.service';
9.
10.@Injectable({
11.  providedIn: 'root',
12.})
13.export class AuthGuard implements CanActivate, CanActivateChild {
14.  constructor(private authService: AuthService, private router: Router) {}
15.
16.  canActivate(
17.    route: ActivatedRouteSnapshot,
18.    state: RouterStateSnapshot): boolean {
19.    let url: string = state.url;
20.
21.    return this.checkLogin(url);
22.  }
23.
24.  canActivateChild(
25.    route: ActivatedRouteSnapshot,
26.    state: RouterStateSnapshot): boolean {
27.    return this.canActivate(route, state);
28.  }
29.
30.  checkLogin(url: string): boolean {
31.    if (this.authService.isLoggedIn) { return true; }
32.
33.    // Store the attempted URL for redirecting
34.    this.authService.redirectUrl = url;
35.
```

```
36. // Navigate to the login page
37. this.router.navigate(['/login']);
38. return false;
39. }
40.}
```

## Query parameters and fragments

In the [route parameters](#) example, you only dealt with parameters specific to the route, but what if you wanted optional parameters available to all routes? This is where query parameters come into play.

[Fragments](#) 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](#) object to the router.navigate method that navigates you to the /login route.

src/app/auth/auth.guard.ts (v3)

```
import { Injectable }      from '@angular/core';
import {
  CanActivate, Router,
  ActivatedRouteSnapshot,
  RouterStateSnapshot,
  CanActivateChild,
  NavigationExtras
}                      from '@angular/router';
import { AuthService }   from './auth.service';

@Injectable({
  providedIn: 'root',
})
export class AuthGuard implements CanActivate, CanActivateChild {
  constructor(private authService: AuthService, private router: Router) {}

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean {
    let url: string = state.url;

    return this.checkLogin(url);
  }

  canActivateChild(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean {
```

```

    return this.canActivate(route, state);
}

checkLogin(url: string): boolean {
  if (this.authService.isLoggedIn) { return true; }

  // Store the attempted URL for redirecting
  this.authService.redirectUrl = url;

  // Create a dummy session id
  let sessionId = 123456789;

  // Set our navigation extras object
  // that contains our global query params and fragment
  let navigationExtras: NavigationExtras = {
    queryParams: { 'session_id': sessionId },
    fragment: 'anchor'
  };

  // Navigate to the login page with extras
  this.router.navigate(['/login'], navigationExtras);
  return false;
}
}

```

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.navigate 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)

// Set our navigation extras object
// that passes on our global query params and fragment
let navigationExtras: NavigationExtras = {
  queryParamsHandling: 'preserve',
  preserveFragment: true
};

// Redirect the user
this.router.navigate([redirect], navigationExtras);

```

The queryParamsHandling feature also provides a merge option, which will preserve and combine the current query parameters with any provided query parameters when navigating.

As you'll be navigating to the Admin Dashboard route after logging in, you'll update it to handle the query parameters and fragment.

src/app/admin/admin-dashboard/admin-dashboard.component.ts (v2)

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from '@angular/router';
import { Observable } from 'rxjs';
import { map } from 'rxjs/operators';

@Component({
  selector: 'app-admin-dashboard',
  templateUrl: './admin-dashboard.component.html',
  styleUrls: ['./admin-dashboard.component.css']
})
export class AdminDashboardComponent implements OnInit {
  sessionId: Observable<string>;
  token: Observable<string>;

  constructor(private route: 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](#) service. Just 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](#).

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](#) params and fragment can also be preserved using a [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. As you continue to build out feature areas, the overall application size will continue to grow. At some point you'll reach a tipping point where the application takes a long time to load.

How do you combat this problem? With 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.

The [Router](#) supports empty path routes; use them 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, set to the address of the `AdminModule`. The address is the `AdminModule` file location (relative to the app root), followed by a # separator, followed by the name of the exported module class, `AdminModule`.

`app-routing.module.ts (load children)`

```
{  
  path: 'admin',  
  loadChildren: './admin/admin.module#AdminModule',  
},
```

Note: When using absolute paths, the [NgModule](#) file location must begin with `src/app` in order to resolve correctly. For custom [path mapping with absolute paths](#), the `baseUrl` and `paths` properties in the project `tsconfig.json` must be configured.

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 Guard: guarding unauthorized loading of feature modules

You're already protecting the `AdminModule` with a [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](#) 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](#) guard.

Open `auth.guard.ts`. Import the [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)`

```
canLoad(route: Route): boolean {
```

```
  let 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)

```
{  
  path: 'admin',  
  loadChildren: './admin/admin.module#AdminModule',  
  canLoad: [AuthGuard]  
},
```

## Preloading: background loading of feature areas

You've learned how to load modules on-demand. You can also load modules asynchronously with preloading.

This may seem like what the app has been doing all along. Not quite. The AppModule is loaded when the application starts; that's eager loading. Now the AdminModule loads only when the user clicks on a link; that's lazy loading.

Preloading is something in between. 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 app 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 to go.

That's preloading.

### 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](#) offers two preloading strategies out of the box:

- No preloading at all which is the default. Lazy loaded feature areas are still loaded on demand.
- Preloading of all lazy loaded feature areas.

Out of the box, the router either never preloads, or preloads every lazy load module. The [Router](#) also supports [custom preloading strategies](#) for fine control over which modules to preload and when.

In this next section, you'll update the CrisisCenterModule to load lazily by default and use the [PreloadAllModules](#) strategy to load it (and all other lazy loaded modules) as soon as possible.

## Lazy load the crisis center

Update the route configuration to lazy load the CrisisCenterModule. Take the same steps you used to configure AdminModulefor lazy load.

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

```
1. import { NgModule }      from '@angular/core';
2. import { BrowserModule } from '@angular/platform-browser';
3. import { FormsModule }   from '@angular/forms';
4. import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
5.
6. import { Router } from '@angular/router';
7.
8. import { AppComponent }      from './app.component';
9. import { PageNotFoundComponent } from './page-not-found/page-not-
found.component';
10.import { ComposeMessageComponent } from './compose-message/compose-
message.component';
11.
12.import { AppRoutingModule }      from './app-routing.module';
13.import { HeroesModule }        from './heroes/heroes.module';
14.import { AuthModule }         from './auth/auth.module';
15.
16.@NgModule({
17.  imports: [
18.    BrowserModule,
19.    BrowserAnimationsModule,
20.    FormsModule,
21.    HeroesModule,
22.    AuthModule,
23.    AppRoutingModule,
24.  ],
25.  declarations: [
26.    AppComponent,
27.    ComposeMessageComponent,
```

```
28. PageNotFoundComponent
29. ],
30. bootstrap: [ AppComponent ]
31.})
32.export class AppModule {
33.}
```

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](#) token from the Angular router package.

The second argument in the RouterModule.forRoot method takes an object for additional configuration options. The [preloadingStrategy](#) is one of those options. Add the [PreloadAllModules](#) token to the forRoot call:

```
src/app/app-routing.module.ts (preload all)
```

```
RouterModule.forRoot(
  appRoutes,
  {
    enableTracing: true, // <-- debugging purposes only
    preloadingStrategy: PreloadAllModules
  }
)
```

This tells the [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.

Surprisingly, the AdminModule does not preload. Something is blocking it.

### CanLoad blocks preload

The [PreloadAllModules](#) strategy does not load feature areas protected by a [CanLoad](#) guard. This is by design.

You added a [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](#) guard takes precedence over the preload strategy.

If you want to preload a module and guard against unauthorized access, drop the [canLoad\(\)](#) guard method and rely on the [canActivate\(\)](#) guard alone.

## Custom Preloading Strategy

Preloading every lazy loaded modules works well in many situations, but it isn't always the right choice, especially on mobile devices and over low bandwidth connections. You may choose to preload only certain feature modules, based on user metrics and other business and technical factors.

You can control what and how the router preloads with a custom preloading strategy.

In this section, you'll add 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)

```
{  
  path: 'crisis-center',  
  loadChildren: './crisis-center/crisis-center.module#CrisisCenterModule',  
  data: { preload: true }  
},
```

Generate a new SelectivePreloadingStrategy service.

ng generate service selective-preloading-strategy

src/app/selective-preloading-strategy.service.ts (excerpt)

```
import { Injectable } from '@angular/core';  
import { PreloadingStrategy, Route } from '@angular/router';  
import { Observable, of } from 'rxjs';  
  
@Injectable(  
  providedIn: 'root',  
)  
export class SelectivePreloadingStrategyService implements PreloadingStrategy {  
  preloadedModules: string[] = [];  
  
  preload(route: Route, load: () => Observable<any>): Observable<any> {  
    if (route.data && route.data['preload']) {  
      // 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](#), 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 should preload, it returns the observable returned by calling the loader function. If the route should 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.

It also has 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](#) strategy in the call to forRoot with this SelectivePreloadingStrategyService.
3. Add the SelectivePreloadingStrategyService strategy to the AppRoutingModule providers array so it can be injected elsewhere in the app.

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.

When you're done it looks like this.

src/app/admin/admin-dashboard/admin-dashboard.component.ts (preloaded modules)

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from '@angular/router';
import { Observable } from 'rxjs';
import { map } from 'rxjs/operators';

import { SelectivePreloadingStrategyService } from '../../selective-preloading-strategy.service';
```

```

@Component({
  selector: 'app-admin-dashboard',
  templateUrl: './admin-dashboard.component.html',
  styleUrls: ['./admin-dashboard.component.css']
})
export class AdminDashboardComponent implements OnInit {
  sessionId: Observable<string>;
  token: Observable<string>;
  modules: string[];
}

constructor(
  private route: 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's also logged to the browser's console.

## Migrating URLs with Redirects

You've setup the routes for navigating around your application. You've used navigation imperatively and declaratively to many different routes. 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 was a requirement that links to heroes become superheroes, you still want the previous URLs to navigate correctly. You also don't want to go and update every link in your application, so redirects makes refactoring routes trivial.

## Changing /heroes to /superheroes

Let's take the Hero routes and migrate them to new URLs. The [Router](#) checks for redirects in your configuration before navigating, so each redirect is triggered when needed. To support this change, you'll add redirects from the old routes to the new routes in the heroes-routing.module.

src/app/heroes/heroes-routing.module.ts (heroes redirects)

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

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

const heroesRoutes: Routes = [
  { path: 'heroes', redirectTo: '/superheroes' },
  { path: 'hero/:id', redirectTo: '/superhero/:id' },
  { path: 'superheroes', component: HeroListComponent, data: { animation: 'heroes' } },
  { path: 'superhero/:id', component: HeroDetailComponent, data: { animation: 'hero' } }
];

@NgModule({
  imports: [
    RouterModule.forChild(heroesRoutes)
  ],
  exports: [
    RouterModule
  ]
})
export class HeroesRoutingModule {}
```

You'll notice two different types of redirects. The first change is from /heroes to /superheroes without any parameters. This is a straightforward redirect, unlike the change from /hero/:id to /superhero/:id, which includes the :id route parameter. Router redirects also use powerful pattern matching, so the [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](#) also supports [query parameters](#) and the [fragment](#) when using redirects.

- When using absolute redirects, the [Router](#) will use the query parameters and the fragment from the redirectTo in the route config.
- When using relative redirects, the [Router](#) use the query params and the fragment from the source URL.

Before updating the app-routing.module.ts, you'll need to consider an important rule. Currently, our empty path route redirects to /heroes, which redirects to /superheroes. This won't work and is by design as the [Router](#) handles redirects once at each level of routing configuration. This prevents chaining of redirects, which can lead to endless redirect loops.

So instead, you'll update the empty path route in app-routing.module.ts to redirect to /superheroes.

src/app/app-routing.module.ts (superheroes redirect)

```
import { NgModule }           from '@angular/core';
import { RouterModule, 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 = [
{
  path: 'compose',
  component: ComposeMessageComponent,
  outlet: 'popup'
},
{
  path: 'admin',
  loadChildren: './admin/admin.module#AdminModule',
  canLoad: [AuthGuard]
},
{
  path: 'crisis-center',
  loadChildren: './crisis-center/crisis-center.module#CrisisCenterModule',
  data: { preload: true }
},
{ path: "", redirectTo: '/superheroes', pathMatch: 'full' },
{ path: '**', component: PageNotFoundComponent }
];

@NgModule({
  imports: [
    RouterModule.forRoot(
      appRoutes,
      { useHash: true }
    )
  ],
  providers: [
    SelectivePreloadingStrategyService
  ]
})
```

```

{
  enableTracing: false, // <-- debugging purposes only
  preloadingStrategy: SelectivePreloadingStrategyService,
}
),
],
exports: [
  RouterModule
]
})
)
export class AppRoutingModule { }

```

RouterLinks aren't tied to route configuration, so you'll need to update the associated router links so they remain active when the new route is active. You'll update the app.component.ts template for the /heroes routerLink.

src/app/app.component.html (superheroes active routerLink)

```

<h1 class="title">Angular Router</h1>
<nav>
  <a routerLink="/crisis-center" routerLinkActive="active">Crisis Center</a>
  <a routerLink="/superheroes" routerLinkActive="active">Heroes</a>
  <a routerLink="/admin" routerLinkActive="active">Admin</a>
  <a routerLink="/login" routerLinkActive="active">Login</a>
  <a [routerLink]="{{ outlets: { popup: ['compose'] } }}>Contact</a>
</nav>
<div [@routeAnimation]="getAnimationData(routerOutlet)">
  <router-outlet #routerOutlet="outlet"></router-outlet>
</div>
<router-outlet name="popup"></router-outlet>

```

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)

```

gotoHeroes(hero: Hero) {
  let 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

You put a lot of effort into configuring the router in several routing module files and were careful to list them [in the proper order](#). Are routes actually evaluated as you planned? How is the router really configured?

You can inspect the router's current configuration any time by injecting it and examining 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)

```
export class AppModule {  
  // Diagnostic only: inspect router configuration  
  constructor(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: ', JSON.stringify(router.config, replacer, 2));  
  }  
}
```

## Wrap up and final app

You've covered a lot of ground in this guide and the application is too big to reprint here.

Please visit the [Router Sample in Stackblitz](#) / [download example](#) where you can download the final source code.

## Appendices

The balance of this guide is a set of appendices that elaborate some of the points you covered quickly above.

The appendix material isn't essential. Continued reading is for the curious.

### Appendix: 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](#) directive to such an array like this:

src/app/app.component.ts (h-anchor)

```
<a [routerLink]=["/heroes"]>Heroes</a>
```

You've written a two element array when specifying a route parameter like this:

```
src/app/heroes/hero-list/hero-list.component.html (nav-to-detail)
```

```
<a [routerLink]="/hero", hero.id]>  
  <span class="badge">{{ hero.id }}</span>{{ hero.name }}  
</a>
```

You can provide optional route parameters in an object like this:

```
src/app/app.component.ts (cc-query-params)
```

```
<a [routerLink]="/crisis-center", { foo: 'foo' }]>Crisis Center</a>
```

These three examples cover the need for an app with one level routing. The moment you add a child router, such as the crisis center, you create new link array possibilities.

Recall that you specified a default child route for the crisis center so this simple [RouterLink](#) is fine.

```
src/app/app.component.ts (cc-anchor-w-default)
```

```
<a [routerLink]="/crisis-center"]>Crisis Center</a>
```

Parse it out.

- The first item in the array identifies the parent route (/crisis-center).
- There are no parameters for this parent route so you're done with it.
- 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.
- Voilà! ['/crisis-center'].

Take it a step further. 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)
```

```
<a [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 so you're done with it.
- 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.

If you wanted to, you could redefine the AppComponent template with Crisis Center routes exclusively:

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

```
template: `<h1 class="title">Angular Router</h1>
```

```
<nav>
  <a [routerLink]="/crisis-center">Crisis Center</a>
  <a [routerLink]="/crisis-center/1", { foo: 'foo' }>Dragon Crisis</a>
  <a [routerLink]="/crisis-center/2">Shark Crisis</a>
</nav>
<router-outlet></router-outlet>
```

In sum, 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.

## Appendix: 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. This is a strictly local URL. The browser shouldn't send this URL to the server and should not reload the page.

Modern HTML5 browsers support [history.pushState](#), 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:

localhost: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.

localhost:3002/src/#/crisis-center/

The router supports both styles with two [LocationStrategy](#) providers:

1. [PathLocationStrategy](#)—the default "HTML5 pushState" style.
2. [HashLocationStrategy](#)—the "hash URL" style.

The `RouterModule.forRoot` function sets the [LocationStrategy](#) to the [PathLocationStrategy](#), making it the default strategy. You can switch to the [HashLocationStrategy](#) with an override during the bootstrapping process if you prefer it.

Learn about providers and the bootstrap process in the [Dependency Injection guide](#).

### Which strategy is best?

You must choose a strategy and you need to make the right call early in the project. It won't be easy to change later once the application is in production and there are lots of application URL references in the wild.

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 later.

Rendering critical pages on the server is a technique that can greatly improve perceived responsiveness when the app first loads. An app 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.

Stick with the default unless you have a compelling reason to resort to hash routes.

### The <base href>

The router uses the browser's [history.pushState](#) for navigation. Thanks to pushState, you can make in-app URL paths look the way you want them to look, e.g. localhost:4200/crisis-center. The in-app 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](#) Appendix, learn why HTML5 style is preferred, how to adjust its behavior, and how to switch to the older hash (#) style, if necessary.

You must add a [<base href> element](#) to the app'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 exactly as shown here.

src/index.html (base-href)

```
<base href="/">
```

### HTML5 URLs and the <base href>

While the router uses the [HTML5 pushState](#) 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](#) tag in the <head> of the index.html.

src/index.html (base-href)

```
<base href="/">
```

Without that tag, the browser may not be able to load resources (images, CSS, scripts) when "deep linking" into the app. Bad things could happen when someone pastes an application link into the browser's address bar or clicks such a link in an email.

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 two remedial steps:

1. Provide the router with an appropriate [APP\\_BASE\\_HREF](#) value.
2. Use root URLs for all web resources: CSS, images, scripts, and template HTML files.

### HashLocationStrategy

You can go old-school with the [HashLocationStrategy](#) by providing the [useHash](#): true in an object as the second argument of the RouterModule.forRoot in the AppModule.

src/app/app.module.ts (hash URL strategy)

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

import { AppComponent }       from './app.component';
import { PageNotFoundComponent } from './page-not-found/page-not-found.component';

const routes: Routes = [
];

@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    RouterModule.forRoot(routes, {useHash: true}) // ...#/crisis-center/
  ],
  declarations: [
    AppComponent,
    PageNotFoundComponent
  ],
  providers: [
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule {}
```

Animation provides the illusion of motion: HTML elements change styling over time. Well-designed animations can make your application more fun and easier to use, but they aren't just cosmetic. Animations can improve your app and user experience in a number of ways:

- Without animations, web page transitions can seem abrupt and jarring.
- Motion greatly enhances the user experience, so animations give users a chance to detect the application's response to their actions.
- Good animations intuitively call the user's attention to where it is needed.

Typically, animations involve multiple style transformations over time. An HTML element can move, change color, grow or shrink, fade, or slide off the page. These changes can occur simultaneously or sequentially. You can control the timing of each transformation.

Angular's animation system is built on CSS functionality, which means 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](#) page.

## About this guide

This guide covers the basic Angular animation features to get you started on adding Angular animations to your project.

The features described in this guide — and the more advanced features described in the related Angular animations guides — are demonstrated in an example app available as a [live example](#) / [download example](#).

### Prerequisites

The guide assumes that you're familiar with building basic Angular apps, as described in the following sections:

- [Tutorial](#)
- [Architecture Overview](#)

## Getting started

The main Angular modules for animations are `@angular/animations` and `@angular/platform-browser`. When you create a new project using the CLI, these dependencies are automatically added to your project.

To get started with adding Angular animations to your project, import the animation-specific modules along with standard Angular functionality.

### Step 1: Enabling the animations module

Import [BrowserAnimationsModule](#), which introduces the animation capabilities into your Angular root application module.

```
src/app/app.module.ts
```

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

@NgModule({
  imports: [
    BrowserModule,
    BrowserAnimationsModule
  ],
  declarations: [ ],
  bootstrap: [ ]
})
export class AppModule {}
```

Note: When you use the CLI to create your app, the root application module app.module.ts is placed in the src/app folder.

## Step 2: Importing animation functions into component files

If you plan to use specific animation functions in component files, import those functions from @angular/animations.

```
src/app/app.component.ts
```

```
import { Component, HostBinding } from '@angular/core';
import {
  trigger,
  state,
  style,
  animate,
  transition,
  ...
} from '@angular/animations';
```

Note: See a [summary of available animation functions](#) at the end of this guide.

## Step 3: Adding the animation metadata property

In the component file, add a metadata property called animations: within the [@Component\(\)](#) decorator. You put the trigger that defines an animation within the animations metadata property.

```
src/app/app.component.ts
```

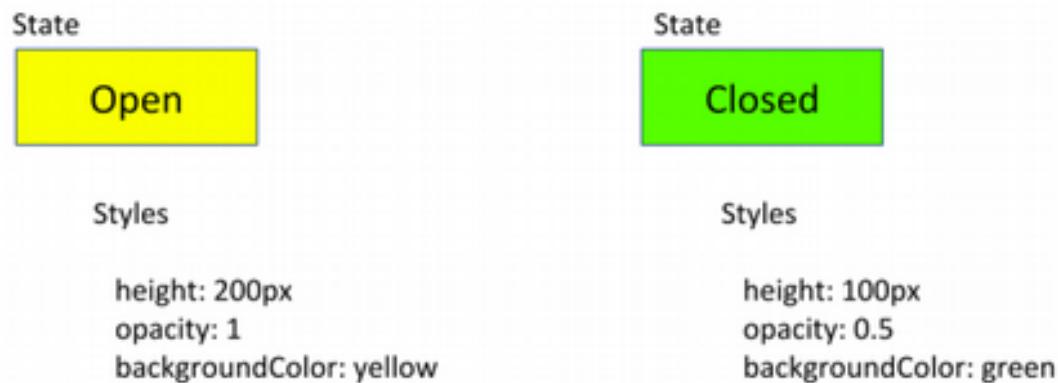
```
@Component({
  selector: 'app-root',
```

```
templateUrl: 'app.component.html',
styleUrls: ['app.component.css'],
animations: [
  // animation triggers go here
]
})
```

## Animating a simple transition

Let's animate a simple transition that changes a single HTML element from one state to another. For example, you can specify that a button displays either Open or Closed based on the user's last action. When the button is in the open state, it's visible and yellow. When it's the closed state, it's transparent and green.

In HTML, these attributes are set using ordinary CSS styles such as color and opacity. In Angular, use the [style\(\)](#) function to specify a set of CSS styles for use with animations. You can collect a set of styles in an animation state, and give the state a name, such as open or closed.



### Animation state and styles

Use Angular's [state\(\)](#) function to define different states to call at the end of each transition. This function takes two arguments: a unique name like open or closed and a [style\(\)](#) function.

Use the [style\(\)](#) function to define a set of styles to associate with a given state name. Note that the style attributes must be in [camelCase](#).

Let's see how Angular's [state\(\)](#) function works with the [style\(\)](#) function to set CSS style attributes. In this code snippet, multiple style attributes are set at the same time for the state. In the open state, the button has a height of 200 pixels, an opacity of 1, and a background color of yellow.

src/app/open-close.component.ts

```
// ...
state('open', style({
height: '200px',
opacity: 1,
backgroundColor: 'yellow'
})),
```

In the closed state, shown below, the button has a height of 100 pixels, an opacity of 0.5, and a background color of green.

src/app/open-close.component.ts

```
state('closed', style({
height: '100px',
opacity: 0.5,
backgroundColor: 'green'
})),
```

## Transitions and timing

In Angular, you can set multiple styles without any animation. However, without further refinement, the button instantly transforms with no fade, no shrinkage, or other visible indicator that a change is occurring.

To make the change less abrupt, we need to define an animation transition to specify the changes that occur between one state and another over a period of time. The [transition\(\)](#) function accepts two arguments: the first argument accepts an expression that defines the direction between two transition states, and the second argument accepts an [animate\(\)](#) function.

Use the [animate\(\)](#) function to define the length, delay, and easing of a transition, and to designate the style function for defining styles while transitions are taking place. You can also use the [animate\(\)](#) function to define the [keyframes\(\)](#) function for multi-step animations. These definitions are placed in the second argument of the [animate\(\)](#) function.

### Animation metadata: duration, delay, and easing

The [animate\(\)](#) function (second argument of the transition function) accepts the timings and styles input parameters.

The timings parameter takes a string defined in three parts.

[animate \('duration delay easing'\)](#)

The first part, duration, is required. The duration can be expressed in milliseconds as a simple number without quotes, or in seconds with quotes and a time specifier. For example, a duration of a tenth of a second can be expressed as follows:

- As a plain number, in milliseconds: 100

- In a string, as milliseconds: '100ms'
- In a string, as seconds: '0.1s'

The second argument, `delay`, has the same syntax as duration. For example:

- Wait for 100ms and then run for 200ms: '0.2s 100ms'

The third argument, `easing`, controls how the animation [accelerates and decelerates](#) during its runtime. For example, `ease-in` causes the animation to begin slowly, and to pick up speed as it progresses.

- Wait for 100ms, run for 200ms. Use a deceleration curve to start out fast and slowly decelerate to a resting point: '0.2s 100ms ease-out'
- Run for 200ms, with no delay. Use a standard curve to start slow, accelerate in the middle, and then decelerate slowly at the end: '0.2s ease-in-out'
- Start immediately, run for 200ms. Use a acceleration curve to start slow and end at full velocity: '0.2s ease-in'

Note: See the Angular Material Design website's topic on [Natural easing curves](#) for general information on easing curves.

This example provides a state transition from open to closed with a one second transition between states.

`src/app/open-close.component.ts`

```
transition('open => closed', [
  animate('1s')
]),
```

In the code snippet above, the `=>` operator indicates unidirectional transitions, and `<=>` is bidirectional. Within the transition, `animate()` specifies how long the transition takes. In this case, the state change from open to closed takes one second, expressed here as `1s`.

This example adds a state transition from the closed state to the open state with a 0.5 second transition animation arc.

`src/app/open-close.component.ts`

```
transition('closed => open', [
  animate('0.5s')
]),
```

Note: Some additional notes on using styles within `state` and `transition` functions.

- Use `state()` to define styles that are applied at the end of each transition, they persist after the animation has completed.
- Use `transition()` to define intermediate styles, which create the illusion of motion during the animation.

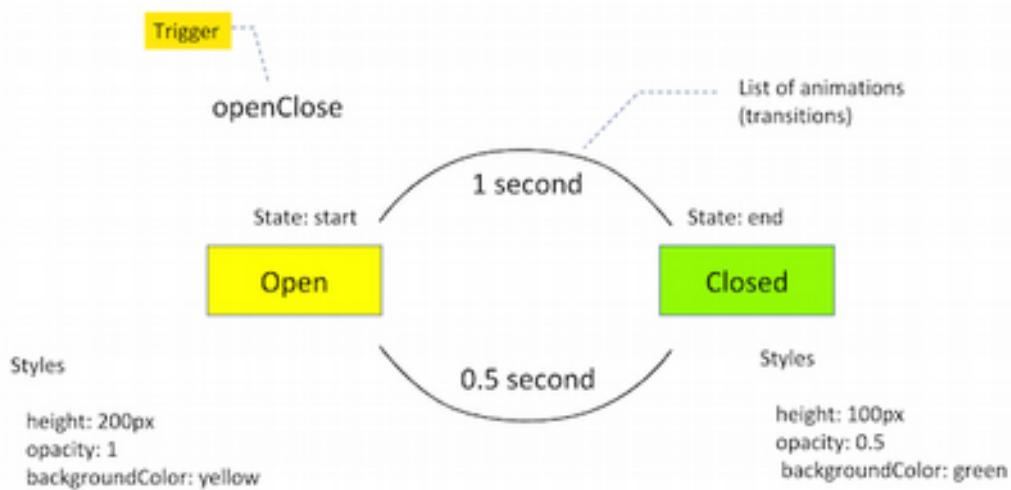
- When animations are disabled, `transition()` styles can be skipped, but `state()` styles can't.
- You can include multiple state pairs within the same `transition()` argument:
- `transition('on => off, off => void')`.

## Triggering the animation

An animation requires a trigger, so that it knows when to start. The `trigger()` function collects the states and transitions, and gives the animation a name, so that you can attach it to the triggering element in the HTML template.

The `trigger()` function describes the property name to watch for changes. When a change occurs, the trigger initiates the actions included in its definition. These actions can be transitions or other functions, as we'll see later on.

In this example, we'll name the trigger `openClose`, and attach it to the button element. The trigger describes the open and closed states, and the timings for the two transitions.



Note: Within each `trigger()` function call, an element can only be in one state at any given time. However, it's possible for multiple triggers to be active at once.

## Defining animations and attaching them to the HTML template

Animations are defined in the metadata of the component that controls the HTML element to be animated. Put the code that defines your animations under the `animations`: property within the `@Component()` decorator.

src/app/open-close.component.ts

```

@Component({
  selector: 'app-open-close',
  animations: [
    trigger('openClose', [
      
```

```

// ...
state('open', style({
  height: '200px',
  opacity: 1,
  backgroundColor: 'yellow'
))),
state('closed', style({
  height: '100px',
  opacity: 0.5,
  backgroundColor: 'green'
))),
transition('open => closed', [
  animate('1s')
]),
transition('closed => open', [
  animate('0.5s')
]),
],
templateUrl: 'open-close.component.html',
styleUrls: ['open-close.component.css']
})
export class OpenCloseComponent {
  isOpen = true;

  toggle() {
    this.isOpen = !this.isOpen;
  }
}

```

When you've defined an animation trigger for a component, you can attach it to an element in that component's template by wrapping the trigger name in brackets and preceding it with an @ symbol. Then, you can bind the trigger to a template expression using standard Angular property binding syntax as shown below, where `triggerName` is the name of the trigger, and expression evaluates to a defined animation state.

```
<div [@triggerName]="expression">...</div>;
```

The animation is executed or triggered when the expression value changes to a new state.

The following code snippet binds the trigger to the value of the `isOpen` property.

```
src/app/open-close.component.html
```

```
<div [@openClose]="isOpen ? 'open' : 'closed'" class="open-close-container">
  <p>The box is now {{ isOpen ? Open } : 'Closed' }}!</p>
</div>
```

In this example, when the isOpen expression evaluates to a defined state of open or closed, it notifies the trigger openClose of a state change. Then it's up to the openClose code to handle the state change and kick off a state change animation.

For elements entering or leaving a page (inserted or removed from the DOM), you can make the animations conditional. For example, use `*ngIf` with the animation trigger in the HTML template.

Note: In the component file, set the trigger that defines the animations as the value of the animations: property in the [@Component\(\)](#) decorator.

In the HTML template file, use the trigger name to attach the defined animations to the HTML element to be animated.

## Code review

Here are the code files discussed in the transition example.

`src/app/open-close.component.ts`

`src/app/open-close.component.html`

`src/app/open-close.component.css`

```
1. @Component\({
2.   selector: 'app-open-close',
3.   animations: [
4.     trigger('openClose', [
5.       // ...
6.       state('open', style({
7.         height: '200px',
8.         opacity: 1,
9.         backgroundColor: 'yellow'
10.      })),
11.      state('closed', style({
12.        height: '100px',
13.        opacity: 0.5,
14.        backgroundColor: 'green'
15.      })),
16.      transition('open => closed', [
17.        animate('1s')
18.      ]),
19.      transition('closed => open', [
20.        animate('0.5s')
```

```

21. ]),
22. ]),
23. ],
24. templateUrl: 'open-close.component.html',
25. styleUrls: ['open-close.component.css']
26.})
27.export class OpenCloseComponent {
28. isOpen = true;
29.
30. toggle() {
31. this.isOpen = !this.isOpen;
32. }
33.
34.}

```

## Summary

You learned to add animation to a simple transition between two states, using [style\(\)](#) and [state\(\)](#) along with [animate\(\)](#) for the timing.

You can learn about more advanced features in Angular animations under the Animation section, beginning with advanced techniques in [transition and triggers](#).

## Animations API summary

The functional API provided by the `@angular/animations` module provides a domain-specific language (DSL) for creating and controlling animations in Angular applications. See the [API reference](#) for a complete listing and syntax details of the core functions and related data structures.

Function name	What it does
<a href="#">trigger()</a>	Kicks off the animation and serves as a container for all other animation function calls. HTML template binds to <a href="#">triggerName</a> . Use the first argument to declare a unique trigger name. Uses array syntax.
<a href="#">style()</a>	Defines one or more CSS styles to use in animations. Controls the visual appearance of HTML elements during animations. Uses object syntax.
<a href="#">state()</a>	Creates a named set of CSS styles that should be applied on successful transition to a given state. The state can then be referenced by name within other animation functions.
<a href="#">animate()</a>	Specifies the timing information for a transition. Optional values for delay and

	<a href="#">easing</a> . Can contain <a href="#">style()</a> calls within.
<a href="#">transition()</a>	Defines the animation sequence between two named states. Uses array syntax.
<a href="#">keyframes()</a>	Allows a sequential change between styles within a specified time interval. Use within <a href="#">animate()</a> . Can include multiple <a href="#">style()</a> calls within each <a href="#">keyframe()</a> . Uses array syntax.
<a href="#">group()</a>	Specifies a group of animation steps (inner animations) to be run in parallel. Animation continues only after all inner animation steps have completed. Used within <a href="#">sequence()</a> or <a href="#">transition()</a> .
<a href="#">query()</a>	Use to find one or more inner HTML elements within the current element.
<a href="#">sequence()</a>	Specifies a list of animation steps that are run sequentially, one by one.
<a href="#">stagger()</a>	Staggers the starting time for animations for multiple elements.
<a href="#">animation()</a>	Produces a reusable animation that can be invoked from elsewhere. Used together with <a href="#">useAnimation()</a> .
<a href="#">useAnimation()</a> )	Activates a reusable animation. Used with <a href="#">animation()</a> .
<a href="#">animateChild()</a>	Allows animations on child components to be run within the same timeframe as the parent.

## More on Angular animations

You may also be interested in the following:

- [Transition and triggers](#)
- [Complex animation sequences](#)
- [Reusable animations](#)
- [Route transition animations](#)

Check out this full animation [demo](#) with accompanying [presentation](#), shown at the AngularConnect conference in November 2017.

You learned the basics of Angular animations in the [introduction](#) page.

In this guide, we go 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. The chapter also explores on multiple animation triggers, animation callbacks and sequence-based animation using keyframes.

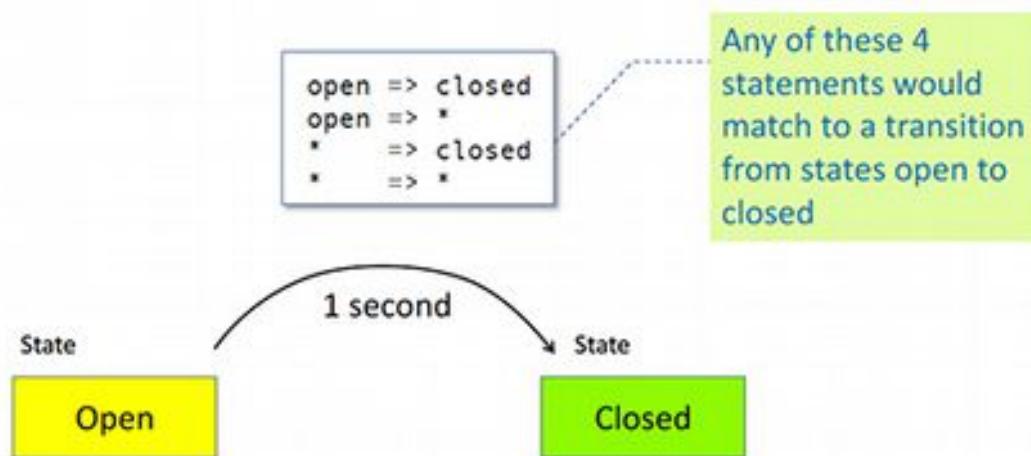
# Predefined states and wildcard matching

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.



Here's another code sample using the wildcard state together with our previous example using the open and closed states. Instead of defining each state-to-state transition pair, we're now saying that 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

```
animations: [
  trigger('openClose', [
    // ...
    state('open', style({
      height: '200px',
      opacity: 1,
      backgroundColor: 'yellow'
    })),
    state('closed', style({
      height: '100px',
      opacity: 0.5,
      backgroundColor: 'green'
    }))
]
```

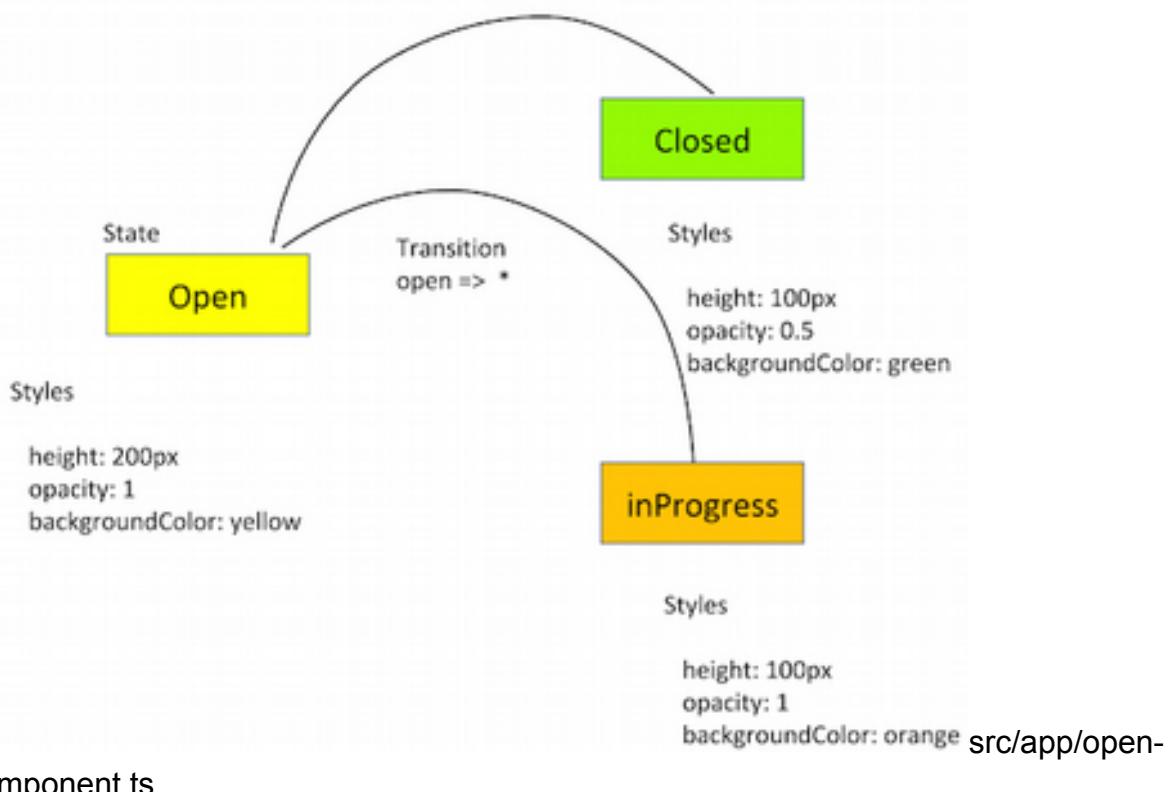
Use a double arrow syntax to specify state-to-state transitions in both directions.

src/app/open-close.component.ts

```
transition('open <=> closed', [  
  animate('0.5s')  
],  
);
```

## Using wildcard state with multiple transition states

In our 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 our button can change from open to either closed or something like inProgress, using a wildcard state could reduce the amount of coding needed.



```
animations: [
  trigger('openClose', [
    // ...
    state('open', style({
      height: '200px',
      opacity: 1,
      backgroundColor: 'yellow'
    })),
    state('closed', style({
      height: '100px',
      opacity: 0.5,
      backgroundColor: 'green'
    })),
    transition('open => closed', [
      animate('1s')
    ]),
    transition('closed => open', [
      animate('0.5s')
    ]),
    transition('* => closed', [
      animate('1s')
    ]),
    transition('* => open', [
      animate('0.5s')
    ]),
    transition('open <=> closed', [
      animate('0.5s')
    ]),
    transition('* => open', [
      animate('1s',
        style ({ opacity: '*' })
      ),
    ]),
    transition('* => *', [
      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

```
transition ('* => open', [
  animate ('1s',
    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](#).

## 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 our purposes, an element entering or leaving a view is equivalent to being inserted or removed from the DOM.

Now we'll 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

```
animations: [
  trigger('flyInOut', [
    state('in', style({ transform: 'translateX(0)' })),
  ]),
],
```

```

transition('void => *', [
  style({ transform: 'translateX(-100%)' }),
  animate(100)
]),
transition(* => void', [
  animate(100, style({ transform: 'translateX(100%)' })))
])
])
]

```

In the above code, we 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 ( ':enter', [ ... ] ); // alias for void => *
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 :leaveto 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` or `*ngFor` views are placed on the page, and :leave runs when those views are removed from the page.

In this example, we have a special trigger for the enter and leave animation called myInsertRemoveTrigger. The HTML template contains the following code.

src/app/insert-remove.component.html

```

<div @myInsertRemoveTrigger *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

```

trigger('myInsertRemoveTrigger', [
  transition(':enter', [
    style({ opacity: 0 }),
    animate('5s', style({ opacity: 1 })),
  ]),
])

```

```
transition(':leave', [
  animate('5s', style({ opacity: 0 })),
])
]),
```

Note that this example doesn't need to use state().

## :increment and :decrement in transitions

The 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() and stagger() methods, which is discussed in the complex sequences page.

src/app/hero-list-page.component.ts

```
trigger('filterAnimation', [
  transition(':enter, * => 0, * => -1', []),
  transition(':increment', [
    query(':enter', [
      style({ opacity: 0, width: '0px' }),
      stagger(50, [
        animate('300ms ease-out', style({ opacity: 1, width: '*' })),
      ]),
    ], { optional: true })
  ],
  transition(':decrement', [
    query(':leave', [
      stagger(50, [
        animate('300ms ease-out', 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() expression that compares true and false, or 1 and 0.

src/app/open-close.component.html

```
<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](#) 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

```
animations: [
  trigger('openClose', [
    state('true', style({ height: '*' })),
    state('false', style({ height: '0px' })),
    transition('false <=> true', 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\(\)](#) 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

```
<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\(\)](#) 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\(\)](#) 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

```
@Component({
  selector: 'app-root',
  templateUrl: 'app.component.html',
  styleUrls: ['app.component.css'],
  animations: [
    slideInAnimation
    // animation triggers go here
  ]
})
export class AppComponent {
  @HostBinding('@.disabled')
  public animationsDisabled = false;
}
```

Note: Disabling animations application-wide is useful during end-to-end (E2E) testing.

## Animation callbacks

The animation [trigger\(\)](#) function emits callbacks when it starts and when it finishes. In the example below we have a component that contains an `openClose` trigger.

src/app/open-close.component.ts

```

@Component({
  selector: 'app-open-close',
  animations: [
    trigger('openClose', [
      // ...
    ]),
  ],
  templateUrl: 'open-close.component.html',
  styleUrls: ['open-close.component.css']
})
export class OpenCloseComponent {
  onAnimationEvent ( event: AnimationEvent ) {
  }
}

```

In the HTML template, the animation event is passed back via \$event, as @trigger.start and @trigger.done, where `trigger` is the name of the trigger being used. In our example, the trigger `openClose` appears as follows.

`src/app/open-close.component.html`

```

<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 our original example, a button with the two states of open and closed.

`src/app/open-close.component.ts`

```

export class OpenCloseComponent {
  onAnimationEvent ( event: AnimationEvent ) {
    // openClose is 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}`);

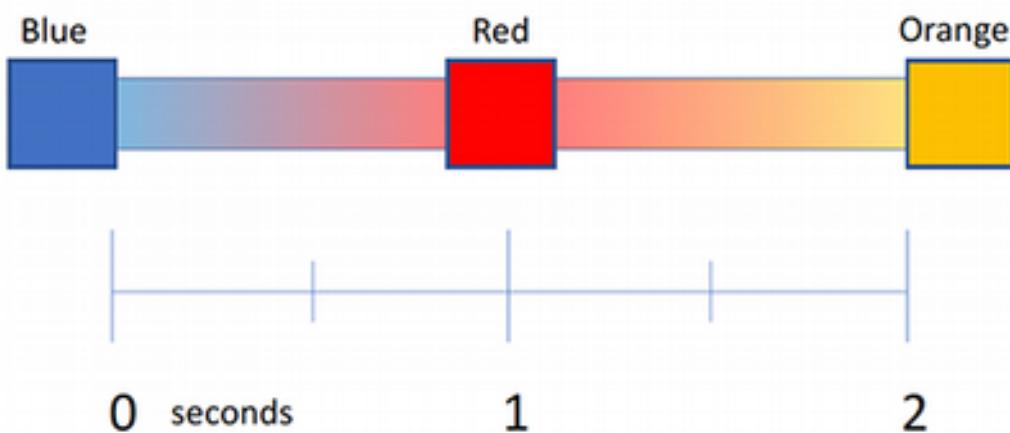
    // the HTML element itself, the button in this case
    console.warn(`Element: ${event.element}`);
  }
}

```

## Keyframes

In the previous section, we saw a simple two-state transition. Now we'll 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, our 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.

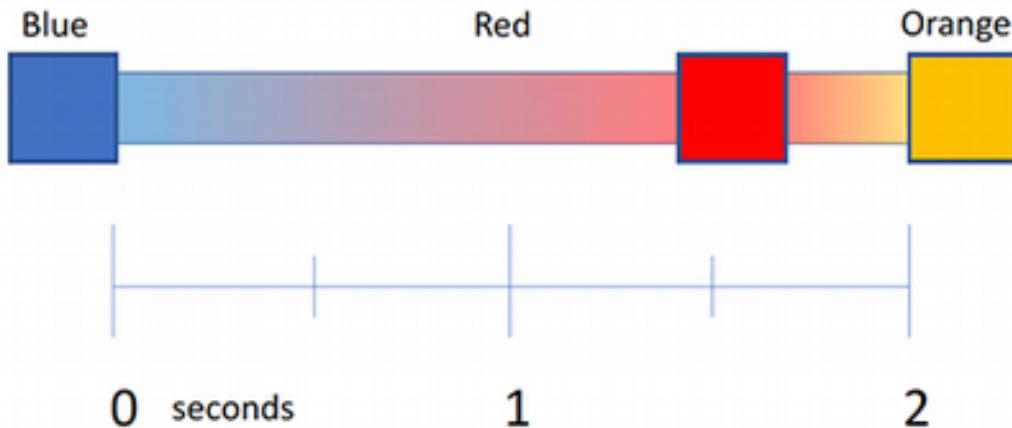
src/app/status-slider.component.ts

```
transition('* => active', [
  animate('2s', keyframes([
    style({ backgroundColor: 'blue' }),
    style({ backgroundColor: 'red' }),
    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.

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

```
1. transition('* => active', [
2.   animate('2s', keyframes([
3.     style({ backgroundColor: 'blue', offset: 0}),
4.     style({ backgroundColor: 'red', offset: 0.8}),
5.     style({ backgroundColor: 'orange', offset: 1.0})
6.   ])),
7. ]),
8. transition('* => inactive', [
```

```

9. animate('2s', keyframes[
10.   style({ backgroundColor: 'orange', offset: 0}),
11.   style({ backgroundColor: 'red', offset: 0.2}),
12.   style({ backgroundColor: 'blue', offset: 1.0})
13. ]),
14.),

```

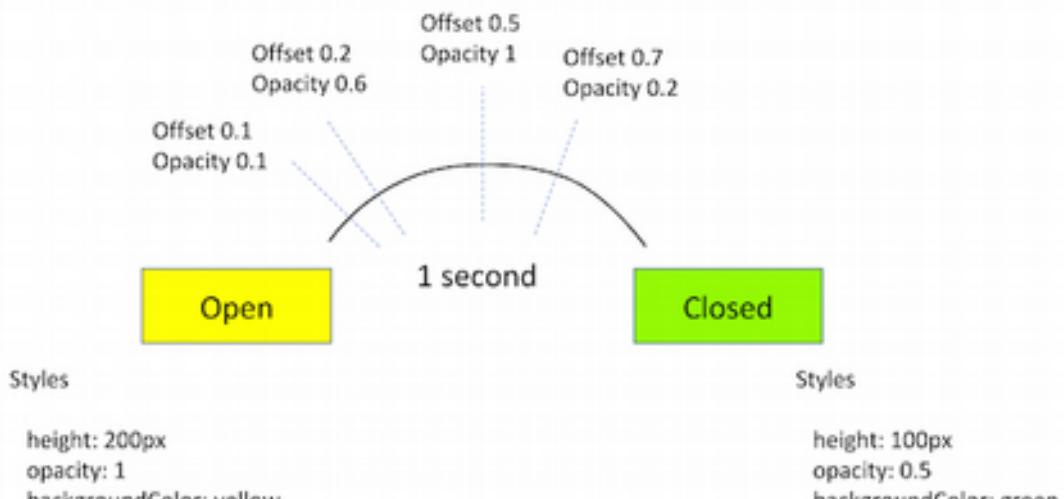
You can combine keyframes with duration, delay, and easing within a single animation.

## 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

```

trigger('openClose', [
  state('open', style({
    height: '200px',
    opacity: 1,
    backgroundColor: 'yellow'
  })),
  state('close', style({
    height: '100px',

```

```
    opacity: 0.5,  
    backgroundColor: 'green'  
}),  
// ...  
transition(* => *, [  
  animate('1s', keyframes ( [  
    style({ opacity: 0.1, offset: 0.1 }),  
    style({ opacity: 0.6, offset: 0.2 }),  
    style({ opacity: 1, offset: 0.5 }),  
    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](#) 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(), so that the value of that particular style property is computed at runtime and then plugged into the animation.

In this example, we have 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

```
animations: [
  trigger('shrinkOut', [
    state('in', style({ height: '*' })),
    transition('* => void', [
      style({ height: '*' }),
      animate(250, style({ height: 0 }))
    ])
  ])
]
```

## Keyframes summary

The [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](#)
- [Complex animation sequences](#)
- [Reusable animations](#)
- [Route transition animations](#)

### Prerequisites

A basic understanding of the following concepts:

- [Introduction to Angular animations](#)
- [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\(\)](#) finds one or more inner HTML elements.
- [stagger\(\)](#) applies a cascading delay to animations for multiple elements.
- [group\(\)](#) runs multiple animation steps in parallel.
- [sequence\(\)](#) runs animation steps one after another.

## Animate multiple elements using query() and stagger() functions

The `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\(\)](#) 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\(\)](#) and [stagger\(\)](#) functions on the entry of an animated element.

- Use `query()` to look for any element entering or leaving the page. The query specifies elements meeting certain CSS class criteria.
  - For each of these elements, use `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()` 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

```
animations: [
  trigger('pageAnimations', [
    transition(':enter', [
      query('.hero, form', [
        style({opacity: 0, transform: 'translateY(-100px)'}),
        stagger(-30, [
          animate('500ms cubic-bezier(0.35, 0, 0.25, 1)', style({ opacity: 1, transform: 'none' }))
        ])
      ])
    ])
  ]),
  ],
],
})
}

export class HeroListPageComponent implements OnInit {
  @HostBinding('@pageAnimations')
  public animatePage = true;

  _heroes = [];
  heroTotal = -1;
```

```

get heroes() {
  return this._heroes;
}

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\(\)](#) function.

Note: The [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)

```

animations: [
  trigger('flyInOut', [
    state('in', style({
      width: 120,
      transform: 'translateX(0)', opacity: 1
    })),
    transition('void => *', [
      style({ width: 10, transform: 'translateX(50px)', opacity: 0 })
    ])
  ])
]

```

```

group([
  animate('0.3s 0.1s ease', style({
    transform: 'translateX(0)',
    width: 120
  })),
  animate('0.3s ease', style({
    opacity: 1
  }))
])
],
transition('* => void', [
  group([
    animate('0.3s ease', style({
      transform: 'translateX(50px)',
      width: 10
    })),
    animate('0.3s 0.2s ease', 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()` to run multiple animations all at the same time, in parallel.

A second function called `sequence()` lets you run those same animations one after the other. Within `sequence()`, the animation steps consist of either `style()` or `animate()` function calls.

- Use `style()` to apply the provided styling data immediately.
- Use `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
```

```
<ul class="heroes" [@filterAnimation]="heroTotal">  
</ul>
```

The component file contains three transitions.

```
src/app/hero-list-page.component.ts
```

```
@Component({  
  animations: [  
    trigger('filterAnimation', [  
      transition(':enter, * => 0, * => -1', []),  
      transition(':increment', [  
        query(':enter', [  
          style({ opacity: 0, width: '0px' }),  
          stagger(50, [  
            animate('300ms ease-out', style({ opacity: 1, width: '*' })),  
          ]),  
        ], { optional: true })  
      ]),  
      transition(':decrement', [  
        query(':leave', [  
          stagger(50, [  
            animate('300ms ease-out', style({ opacity: 0, width: '0px' })),  
          ]),  
        ])  
      ]),  
    ]  
  ]  
})  
export class HeroListPageComponent implements 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\(\)](#) to find inner elements, for example gathering all images within a `<div>`. The remaining functions, [stagger\(\)](#), [group\(\)](#), and [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](#)
- [Transition and triggers](#)
- [Reusable animations](#)
- [Route transition animations](#)

### Prerequisites

A basic understanding of the following concepts:

- [Introduction to Angular animations](#)
- [Transition and triggers](#)

The [AnimationOptions](#) interface in Angular animations enables you to create animations that you can reuse across different components.

## Creating reusable animations

To create a reusable animation, use the [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 app components using the [useAnimation\(\)](#) API.

`src/app/animations.ts`

```
import {
  animation, trigger, animateChild, group,
  transition, animate, style, query
} from '@angular/animations';
```

```
export const transAnimation = animation([
```

```
style({
  height: '{{ height }}',
  opacity: '{{ opacity }}',
  backgroundColor: '{{ backgroundColor }}'
}),
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 import the reusable transAnimation variable in your component class and reuse it using the [useAnimation\(\)](#) method as shown below.

src/app/open-close.component.ts

```
import { Component } from '@angular/core';
import { useAnimation, transition, trigger, style, animate } from '@angular/animations';
import { transAnimation } from './animations';
```

```
@Component({
  trigger('openClose', [
    transition('open => closed', [
      useAnimation(transAnimation, {
        params: {
          height: 0,
          opacity: 1,
          backgroundColor: 'red',
          time: '1s'
        }
      })
    ])
  ]),
  ],
})
```

## More on Angular animations

You may also be interested in the following:

- [Introduction to Angular animations](#)
- [Transition and triggers](#)
- [Complex animation Sequences](#)
- [Route transition animations](#)

## Prerequisites

A basic understanding of the following concepts:

- [Introduction to Angular animations](#)
  - [Transition and triggers](#)
  - [Reusable animations](#)
- 

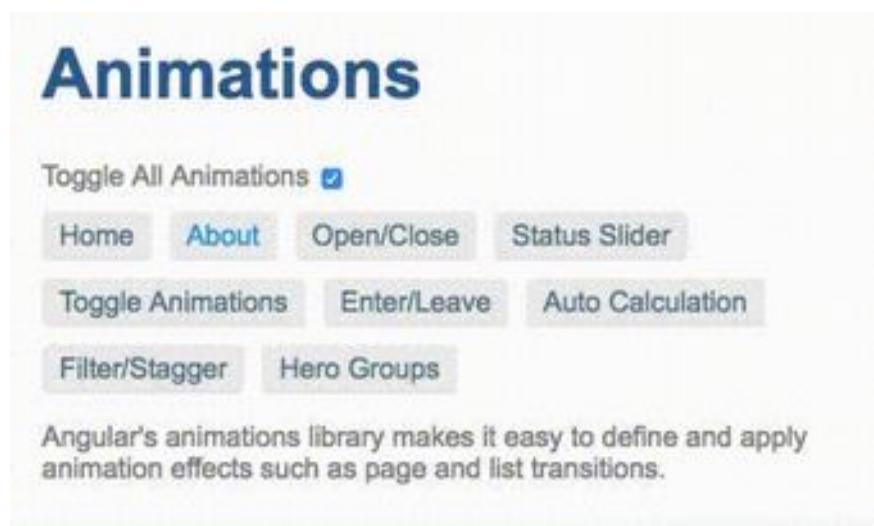
Routing enables users to navigate between different routes in an application. When a user navigates from one route to another, the Angular router maps the URL path to a relevant component and displays its view. Animating this route transition can greatly enhance the user experience.

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](#) 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](#) 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`

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

```
@NgModule({
  imports: [
    BrowserModule,
    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: 'FilterPage'} },
    { path: 'hero-groups', component: HeroListGroupPageComponent },
    { path: 'enter-leave', component: HeroListEnterLeavePageComponent },
    { path: 'auto', component: HeroListAutoCalcPageComponent },
    { path: 'home', component: HomeComponent, data: {animation: 'HomePage'} },
    { path: 'about', component: AboutComponent, data: {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 config 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>` container inside the root AppComponent template.

The `<router-outlet>` 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`

```

<div [@routeAnimations]="prepareRoute(outlet)" >
  <router-outlet #outlet="outlet"></router-outlet>
</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`

```
prepareRoute(outlet: RouterOutlet) {
  return outlet && outlet.activatedRouteData && outlet.activatedRouteData['animation'];
}
```

Here, the `prepareRoute()` method takes the value of the output 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

```
export const slideInAnimation =
  trigger('routeAnimations', [
    transition('HomePage <=> AboutPage', [
      style({position: 'relative'}),
      query(':enter, :leave', [
        style({
          position: 'absolute',
          top: 0,
          left: 0,
          width: '100%'
        })
      ]),
      query(':enter', [
        style({left: '-100%'})
      ]),
      query(':leave', animateChild()),
      group([
        query(':leave', [
          animate('300ms ease-out', style({left: '100%'}))
        ]),
        query(':enter', [
          animate('300ms ease-out', style({left: '0%'}))
        ])
      ]),
      query(':enter', animateChild()),
    ]),
    transition('* <=> FilterPage', [
```

```

    style({ position: 'relative' }),
    query(':enter, :leave', [
      style({
        position: 'absolute',
        top: 0,
        left: 0,
        width: '100%'
      })
    ]),
    query(':enter', [
      style({ left: '-100%' })
    ]),
    query(':leave', animateChild()),
    group([
      query(':leave', [
        animate('200ms ease-out', style({ left: '100%' }))
      ]),
      query(':enter', [
        animate('300ms ease-out', style({ left: '0%' }))
      ])
    ]),
    query(':enter', 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()` 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`

```

@Component({
  selector: 'app-root',

```

```
templateUrl: 'app.component.html',
styleUrls: ['app.component.css'],
animations: [
  slideInAnimation
  // 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

```
trigger('routeAnimations', [
  transition('HomePage <=> AboutPage', [
    style({ position: 'relative' }),
    query(':enter, :leave', [
      style({
        position: 'absolute',
        top: 0,
        left: 0,
        width: '100%'
      })
    ]),
  ]),
]),
```

## Querying the view containers

Use the [query\(\)](#) method to find and animate elements within the current host component. The [query\(":enter"\)](#) statement returns the view that is being inserted, and [query\(":leave"\)](#) returns the view that is being removed.

Let's assume that we are routing from the Home => About.

src/app/animations.ts

```
query(':enter', [
  style({ left: '-100%' })
]),
query(':leave', animateChild\(\)),
group([
  query(':leave', [
    animate\('300ms ease-out', style\({ left: '100%' }\)\)
  ])
]),
```

```

]),
query(':enter', [
  animate('300ms ease-out', style({ left: '0%'}))
])
],
query(':enter', animateChild()),
]),
transition('* <=> FilterPage', [
  style({ position: 'relative' }),
  query(':enter, :leave', [
    style({
      position: 'absolute',
      top: 0,
      left: 0,
      width: '100%'
    })
],
query(':enter', [
  style({ left: '-100%'})
]),
query(':leave', animateChild()),
group([
  query(':leave', [
    animate('200ms ease-out', style({ left: '100%'}))
]),
  query(':enter', [
    animate('300ms ease-out', style({ left: '0%'}))
])
]),
query(':enter', animateChild()),
])
)

```

The animation code does the following after styling the views:

- `query(':enter style({ left: '-100%'})` matches the view that is added and hides the newly added view by positioning it to the far left.
- Calls `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\(\)](#) 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](#)
- [Transition and triggers](#)
- [Complex animation sequences](#)
- [Reusable animations](#)

This guide offers tips and techniques for unit and integration testing Angular applications.

The guide presents tests of a sample CLI application that is much like the [Tour of Heroes tutorial](#). The sample application and all tests in this guide are available for inspection and experimentation:

- [Sample app / download example](#)
  - [Tests / download example](#)
- 

## Setup

The Angular CLI downloads and install everything you need to test an Angular application with the [Jasmine test framework](#).

The project you create with the CLI is immediately ready to test. Just run this one CLI command:

```
ng test
```

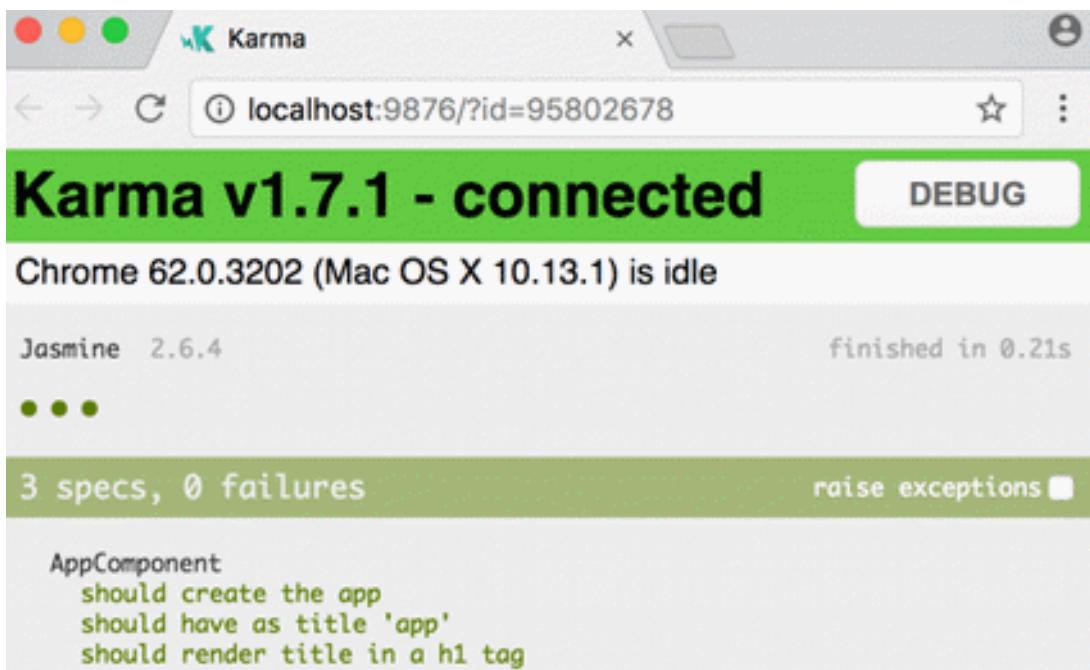
The ng test command builds the app in watch mode, and launches the [karma test runner](#).

The console output looks a bit like this:

```
10% building modules 1/1 modules 0 active
...INFO [karma]: Karma v1.7.1 server started at http://0.0.0.0:9876/
...INFO [launcher]: Launching browser Chrome ...
...INFO [launcher]: Starting browser Chrome
...INFO [Chrome ...]: Connected on socket ...
Chrome ...: Executed 3 of 3 SUCCESS (0.135 secs / 0.205 secs)
```

The last line of the log is the most important. It shows that Karma ran three tests that all passed.

A chrome browser also opens and displays the test output in the "Jasmine HTML Reporter" like this.



Most people find this browser output easier to read than the console log. You can click on a test row to re-run just that test or click on a description to re-run the tests in the selected test group ("test suite").

Meanwhile, the ng test command is watching for changes.

To see this in action, make a small change to app.component.ts and save. The tests run again, the browser refreshes, and the new test results appear.

## Configuration

The CLI takes care of Jasmine and karma configuration for you.

You can fine-tune many options by editing the karma.conf.js and the test.ts files in the src/ folder.

The karma.conf.js file is a partial karma configuration file. The CLI constructs the full runtime configuration in memory, based on application structure specified in the angular.json file, supplemented by karma.conf.js.

Search the web for more details about Jasmine and karma configuration.

## Other test frameworks

You can also unit test an Angular app with other testing libraries and test runners. Each library and runner has its own distinctive installation procedures, configuration, and syntax.

Search the web to learn more.

## Test file name and location

Look inside the src/app folder.

The CLI generated a test file for the AppComponent named app.component.spec.ts.

The test file extension must be .spec.ts so that tooling can identify it as a file with tests (AKA, a spec file).

The app.component.ts and app.component.spec.ts files are siblings in the same folder. The root file names (app.component) are the same for both files.

Adopt these two conventions in your own projects for every kind of test file.

## Service Tests

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

```
1. // Straight Jasmine testing without Angular's testing support
2. describe('ValueService', () => {
3.   let service: ValueService;
4.   beforeEach(() => { service = new ValueService(); });
5.
6.   it('#getValue should return real value', () => {
7.     expect(service.getValue()).toBe('real value');
8.   });
9.
10.  it('#getObservableValue should return value from observable',
11.    (done: DoneFn) => {
12.      service.getObservableValue().subscribe(value => {
13.        expect(value).toBe('observable value');
14.        done();
15.      });
16.    });
17.
18.  it('#getPromiseValue should return value from a promise',
19.    (done: DoneFn) => {
20.      service.getPromiseValue().then(value => {
21.        expect(value).toBe('promise value');
22.        done();
23.      });
24.    });
25.});
```

## 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

```
@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

```
1. describe('MasterService without Angular testing support', () => {
2.   let masterService: MasterService;
3.
4.   it('#getValue should return real value from the real service', () => {
5.     masterService = new MasterService(new ValueService());
6.     expect(masterService.getValue()).toBe('real value');
7.   });
8.
9.   it('#getValue should return faked value from a fakeService', () => {
10.    masterService = new MasterService(new FakeValueService());
11.    expect(masterService.getValue()).toBe('faked service value');
12.  });
13.
14.  it('#getValue should return faked value from a fake object', () => {
15.    const fake = { getValue: () => 'fake value' };
16.    masterService = new MasterService(fake as ValueService);
17.    expect(masterService.getValue()).toBe('fake value');
18.  });
19.
20.  it('#getValue should return stubbed value from a spy', () => {
21.    // create `getValue` spy on an object representing the ValueService
22.    const valueServiceSpy =
23.      jasmine.createSpyObj('ValueService', ['getValue']);
24.
25.    // set the value to return when the `getValue` spy is called.
```

```
26. const stubValue = 'stub value';
27. valueServiceSpy.getValue.and.returnValue(stubValue);
28.
29. masterService = new MasterService(valueServiceSpy);
30.
31. expect(masterService.getValue())
32.   .toBe(stubValue, 'service returned stub value');
33. expect(valueServiceSpy.getValue.calls.count())
34.   .toBe(1, 'spy method was called once');
35. expect(valueServiceSpy.getValue.calls.mostRecent().returnValue)
36.   .toBe(stubValue);
37. });
38.});
```

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](#) 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 service 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 app relies on Angular [dependency injection \(DI\)](#) 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](#) testing utility to provide and create services.

## Angular TestBed

The [TestBed](#) is the most important of the Angular testing utilities. The [TestBed](#) creates a dynamically-constructed Angular test module that emulates an Angular [@NgModule](#).

The TestBed.configureTestingModule() method takes a metadata object that can have most of the properties of an [@NgModule](#).

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)
```

```
let service: ValueService;
```

```
beforeEach(() => {
```

```
  TestBed.configureTestingModule({ providers: [ValueService] });
});
```

Then inject it inside a test by calling TestBed.get() with the service class as the argument.

```
it('should use ValueService', () => {
  service = TestBed.get(ValueService);
  expect(service.getValue()).toBe('real value');
});
```

Or inside the beforeEach() if you prefer to inject the service as part of your setup.

```
beforeEach(() => {
  TestBed.configureTestingModule({ providers: [ValueService] });
  service = TestBed.get(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.

```
let 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 both the service-to-test and its (spy) dependency
masterService = TestBed.get(MasterService);
```

```
    valueServiceSpy = TestBed.get(ValueService);
});
```

The test consumes that spy in the same way it did earlier.

```
it('#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](#) 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](#).

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)

```
function 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 (e.g., `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.

```
it('#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](#) to extract the setup variables that it needs.

```
const { 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 tradition style and the default [CLI schematics](#) generate test files with `beforeEach()` and  [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](#) service for XHR calls.

You can test a data service with an injected [HttpClient](#) spy as you would test any service with a dependency.

```
app/model/hero.service.spec.ts (tests with spies)
```

content\_copy

1. let httpClientSpy: { get: jasmine.Spy };
2. let heroService: HeroService;
- 3.
4. beforeEach(() => {
5. // TODO: spy on other methods too
6. httpClientSpy = jasmine.createSpyObj('HttpClient', ['get']);
7. heroService = new HeroService(<any> httpClientSpy);
8. });
- 9.
10. it('should return expected heroes ([HttpClient](#) called once)', () => {
11. const expectedHeroes: Hero[] =
12. [{ id: 1, name: 'A' }, { id: 2, name: 'B' }];
- 13.
14. httpClientSpy.get.and.returnValue(asyncData(expectedHeroes));
- 15.

```

16. heroService.getHeroes().subscribe(
17.   heroes => expect(heroes).toEqual(expectedHeroes, 'expected heroes'),
18.   fail
19. );
20. expect(httpClientSpy.get.calls.count()).toBe(1, 'one call');
21.});
22.
23.it('should return an error when the server returns a 404', () => {
24. const errorResponse = new HttpErrorResponse({
25.   error: 'test 404 error',
26.   status: 404, statusText: 'Not Found'
27. });
28.
29. httpClientSpy.get.and.returnValue(asyncError(errorResponse));
30.
31. heroService.getHeroes().subscribe(
32.   heroes => fail('expected an error, not heroes'),
33.   error => expect(error.message).toContain('test 404 error')
34. );
35.});

```

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](#) can be complex and difficult to mock with spies.

The [HttpClientTestingModule](#) can make these testing scenarios more manageable.

While the code sample accompanying this guide demonstrates [HttpClientTestingModule](#), this page defers to the [Http guide](#), which covers testing with the [HttpClientTestingModule](#) in detail.

This guide's sample code also demonstrates testing of the legacy [HttpModule](#) in app/[model/http-hero.service.spec.ts](#).

# Component Test Basics

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, and 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](#) 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.

## Component class testing

Test a component class on its own as you would test a service class.

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)

```
@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 a service with no dependencies, you create it with `new`, poke at its API, and assert expectations on its public state. Do the same with the component class.

app/demo/demo.spec.ts (Lightswitch tests)

```
describe('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)

```
export class DashboardHeroComponent {
  @Input\(\) hero: Hero;
  @Output\(\) selected = new EventEmitter<Hero>();
  click() { this.selected.emit(this.hero); }
}
```

It appears within the template of a parent component, which binds a hero to the [@Input](#) property and listens for an event raised through the selected [@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)

```
it('raises the selected event when clicked', () => {
  const comp = new DashboardHeroComponent();
  const hero: Hero = { id: 42, name: 'Test' };
  comp.hero = hero;

  comp.selected.subscribe(selectedHero => expect(selectedHero).toBe(hero));
  comp.click();
});
```

When a component has dependencies, you may wish to use the [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

```
export class WelcomeComponent implements OnInit {
  welcome: string;
  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)

```
class MockUserService {
  isLoggedIn = true;
  user = { name: 'Test User'};
};
```

Then provide and inject both the component and the service in the [TestBed](#) configuration.

app/welcome/welcome.component.spec.ts (class-only setup)

```
beforeEach(() => {
  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.get(WelcomeComponent);
  userService = TestBed.get(UserService);
});
```

Then exercise the component class, remembering to call the [lifecycle hook methods](#) as Angular does when running the app.

app/welcome/welcome.component.spec.ts (class-only tests)

```
it('should not have welcome message after construction', () => {
  expect(comp.welcome).toBeUndefined();
});
```

```
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.

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](#) 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):

```
ng 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)
```

```
import { async,  ComponentFixture,  TestBed } from '@angular/core/testing';
import { BannerComponent } from './banner.component';
```

```
describe('BannerComponent', () => {
  let component: BannerComponent;
  let fixture:  ComponentFixture<BannerComponent>;
```

```
beforeEach(async() => {
  TestBed.configureTestingModule({
     declarations: [ BannerComponent ]
  })
  .compileComponents();
});
```

```
beforeEach(() => {
  fixture = TestBed.createComponent(BannerComponent);
  component = fixture.componentInstance;
  fixture.detectChanges();
});
```

```
it('should create', () => {
  expect(component).toBeDefined();
});
```

## 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)
```

```
describe('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.

```
TestBed.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](#) 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](#), you call its `createComponent()` method.

```
const 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](#).

Do not re-configure [TestBed](#) after calling `createComponent`.

The `createComponent` method freezes the current [TestBed](#) definition, closing it to further configuration.

You cannot call any more [TestBed](#) configuration methods, not `configureTestingModule()`, nor `get()`, nor any of the `override...` methods. If you try, [TestBed](#) throws an error.

### `ComponentFixture`

The [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:

```
const component = fixture.componentInstance;
expect(component.toBeDefined());
```

## **beforeEach()**

You will add more tests as this component evolves. Rather than duplicate the [TestBed](#) configuration for each test, you refactor to pull the setup into a Jasmine `beforeEach()` and some supporting variables:

```
describe('BannerComponent (with beforeEach)', () => {
  let component: BannerComponent;
  let fixture: 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.

```
it('should contain "banner works!"', () => {
  const bannerElement: HTMLElement = fixture.nativeElement;
  expect(bannerElement.textContent).toContain('banner works!');
});
```

## **nativeElement**

The value of [ComponentFixture.nativeElement](#) has the `any` type. Later you'll encounter the [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 app might be running on a non-browser platform, such as the server or a [Web Worker](#), 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:

```
it('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`.

```
const bannerElement: HTMLElement = fixture.nativeElement;
```

This is actually a convenience method, implemented as `fixture.debugElement.nativeElement`.

```
const bannerDe: 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](#) abstraction to work safely across all supported platforms. Instead of creating an HTML element tree, Angular creates a [DebugElement](#) tree that wraps the native elements for the runtime platform. The `nativeElement` property unwraps the [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`:

```
it('should find the <p> with fixture.debugElement.nativeElement', () => {
  const bannerDe: DebugElement = fixture.debugElement;
  const bannerEl: HTMLElement = bannerDe.nativeElement;
  const p = bannerEl.querySelector('p');
  expect(p.textContent).toEqual('banner works!');
});
```

The [DebugElement](#) has other methods and properties that are useful in tests, as you'll see elsewhere in this guide.

You import the [DebugElement](#) symbol from the Angular core library.

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

## By.css()

Although the tests in this guide all run in the browser, some apps 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](#) offers query methods that work for all supported platforms. These query methods take a predicatefunction that returns true when a node in the [DebugElement](#) tree matches the selection criteria.

You create a predicate with the help of a [By](#) class imported from a library for the runtime platform. Here's the [By](#) import for the browser platform:

```
import { By } from '@angular/platform-browser';
```

The following example re-implements the previous test with [DebugElement.query\(\)](#) and the browser's By.css method.

```
it('should find the <p> with fixture.debugElement.query(By.css)', () => {
  const bannerDe: 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\(\)](#) static method selects [DebugElement](#) nodes with a [standard CSS selector](#).
- The query returns a [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(), as you'll see in the next set of tests.

---

## Component Test Scenarios

The following sections, comprising most of this guide, explore common component testing scenarios

## Component binding

The current 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

```
@Component({
  selector: 'app-banner',
  template: '<h1>{{title}}</h1>',
  styles: ['h1 { color: green; font-size: 350% }']
})
export class BannerComponent {
  title = 'Test Tour of Heroes';
}
```

Simple 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)

```
let component: BannerComponent;
let fixture: 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:

```
it('should display original title', () => {
  expect(h1.textContent).toContain(component.title);
});
```

That test fails with the message:

```
expected " 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 (e.g., AJAX) completes.

The TestBed.createComponent does not trigger change detection. a fact confirmed in the revised test:

```
it('no title in the DOM after createComponent()', () => {
  expect(h1.textContent).toEqual("");
});
```

### **detectChanges()**

You must tell the [TestBed](#) to perform data binding by calling fixture.detectChanges(). Only then does the <h1> have the expected title.

```
it('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](#).

Here's another test that changes the component's title property before calling fixture.detectChanges().

```
it('should display a different test title', () => {
  component.title = 'Test Title';
  fixture.detectChanges();
  expect(h1.textContent).toContain('Test 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](#) with the [ComponentFixtureAutoDetect](#) provider. First import it from the testing utility library:

```
app/banner/banner.component.detect-changes.spec.ts (import)
import { 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)
```

```
TestBed.configureTestingModule({
  declarations: [ BannerComponent ],
  providers: [
    { provide: ComponentFixtureAutoDetect, useValue: true }
  ]
});
```

Here are three tests that illustrate how automatic change detection works.

```
app/banner/banner.component.detect-changes.spec.ts (AutoDetect Tests)
```

```
it('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';
  // 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';
  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](#) 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)

```
1. it('should convert hero name to Title Case', () => {
  2. // get the name's input and display elements from the DOM
  3. const hostElement = fixture.nativeElement;
  4. const nameInput: HTMLInputElement = hostElement.querySelector('input');
  5. const nameDisplay: HTMLElement = hostElement.querySelector('span');
  6.
  7. // simulate user entering a new name into the input box
  8. nameInput.value = 'quick BROWN fOx';
  9.
 10. // dispatch a DOM event so that Angular learns of input value change.
 11. nameInput.dispatchEvent(newEvent('input'));
 12.
 13. // Tell Angular to update the display binding through the title pipe
 14. fixture.detectChanges();
 15.
 16. expect(nameDisplay.textContent).toBe('Quick Brown Fox');
 17.});
```

---

### Component with external files

The `BannerComponent` above is defined with an inline template and inline css, specified in the `@Component.template` and `@Component.styles` properties respectively.

Many components specify external templates and external css with the `@Component templateUrl` and `@Component.styleUrls` properties respectively, as the following variant of `BannerComponent` does.

```
app/banner/banner-external.component.ts (metadata)
```

```
@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 app 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](#), 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](#).

## 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
```

```
import { Component, OnInit } from '@angular/core';
import { UserService } from '../model/user.service';
```

```
@Component({
  selector: 'app-welcome',
  template: '<h3 class="welcome"><i>{{welcome}}</i></h3>'
})
export class WelcomeComponent implements OnInit {
  welcome: string;
  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:

```
app/welcome/welcome.component.spec.ts
```

```
TestBed.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 providerslist. 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
```

```
let 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](#) 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](#).

WelcomeComponent's injector

```
// UserService actually injected into the component
userService = fixture.debugElement.injector.get(UserService);
```

### TestBed.get()

You may also be able to get the service from the root injector via `TestBed.get()`. 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.get()` as follows:

TestBed injector

```
// UserService from the root injector
userService = TestBed.get(UserService);
```

For a use case in which `TestBed.get()` does not work, see the [Override component providers](#) section that explains when and why you must get the service from the component's injector instead.

### Always get the service from an injector

Do not reference the `userServiceStub` object that's provided to the testing module in the body of your test. It does not work! The `userService` instance injected into the component is a completely different object, a clone of the provided `userServiceStub`.

app/welcome/welcome.component.spec.ts

```
it('stub object and injected UserService should not be the same', () => {
  expect(userServiceStub === userService).toBe(false);
```

```
// Changing the stub object has no effect on the injected service
```

```
userServiceStub.isLoggedIn = false;
expect(userService.isLoggedIn).toBe(true);
});
```

### Final setup and tests

Here's the complete `beforeEach()`, using `TestBed.get()`:

app/welcome/welcome.component.spec.ts

```
let 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.get(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

it('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 (e.g., 'expected name') is an optional failure label. If the expectation fails, Jasmine displays 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)

```
template: `
<p class="twain"><i>{{quote | async}}</i></p>
<button (click)="getQuote()">Next quote</button>
<p class="error" *ngIf="errorMessage">{{ errorMessage }}</p>`,
```

Note that value of the component's quote property passes through an [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)

```
getQuote() {
  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 returns 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)
```

```
beforeEach(() => {
```

```
  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.

```
// 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.

```
it('should show quote after component initialized', () => {
  fixture.detectChanges(); // ngOnInit()

  // 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\(\)](#) functionality, you need to import zone-testing, for details, please read [setup guide](#).

The following test confirms the expected behavior when the service returns an ErrorObservable.

1. it('should display error when TwainService fails', [fakeAsync\(\)](#) => {
2. // tell spy to return an error observable
3. getQuoteSpy.and.returnValue(
4. throwError('TwainService test failure'));
- 5.
6. fixture.detectChanges(); // ngOnInit()
7. // sync spy errors immediately after init
- 8.
9. [tick\(\)](#); // [flush](#) the component's setTimeout()
- 10.
11. fixture.detectChanges(); // [update](#) errorMessage within setTimeout()
- 12.

```
13. expect(errorMessage()).toMatch(/test failure/, 'should display error');
14. expect(quoteEl.textContent).toBe('...', 'should show placeholder');
15.});
```

Note that the `it()` function receives an argument of the following form.

```
fakeAsync() => { /* test body */ }
```

The `fakeAsync()` function enables a linear coding style by running the test body in a special `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.

### The `tick()` function

You do have to call `tick()` to advance the (virtual) clock.

Calling `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()` function accepts milliseconds as parameter (defaults to 0 if not provided). The parameter represents how much the virtual clock advances. For example, if you have a `setTimeout(fn, 100)` in a `fakeAsync()` test, you need to use `tick(100)` to trigger the fn callback.

```
it('should run timeout callback with delay after call tick with millis', fakeAsync() => {
  let called = false;
  setTimeout(() => { called = true; }, 100);
  tick(100);
  expect(called).toBe(true);
});
```

The `tick()` function is one of the Angular testing utilities that you import with  `TestBed`. It's a companion to `fakeAsync()` and you can only call it within a `fakeAsync()` body.

### Comparing dates inside `fakeAsync()`

`fakeAsync()` simulates passage of time, which allows you to calculate the difference between dates inside `fakeAsync()`.

```
it('should get Date diff correctly in fakeAsync', fakeAsync() => {
  const start = Date.now();
  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()` method until `jasmine.clock().uninstall()` is called. `fakeAsync()` is not needed and throws an error if nested.

By default, this feature is disabled. To enable it, set a global flag before import zone-testing.

If you use the Angular CLI, configure this flag in src/test.ts.

```
(window as any).__zone_symbol__fakeAsyncPatchLock' = true;  
import 'zone.js/dist/zone-testing';
```

```
1. describe('use jasmine.clock()', () => {  
2.   // need to config __zone_symbol__fakeAsyncPatchLock flag  
3.   // before loading zone.js/dist/zone-testing  
4.   beforeEach(() => { jasmine.clock().install(); });  
5.   afterEach(() => { jasmine.clock().uninstall(); });  
6.   it('should auto enter fakeAsync', () => {  
7.     // is in fakeAsync now, don't need to call fakeAsync(testFn)  
8.     let called = false;  
9.     setTimeout(() => { called = true; }, 100);  
10.    jasmine.clock().tick(100);  
11.    expect(called).toBe(true);  
12.  });  
13.});
```

### Using the RxJS scheduler inside fakeAsync()

You can also use RxJS scheduler in fakeAsync() just like using setTimeout() or setInterval(), but you need to import zone.js/dist/zone-patch-rxjs-fake-async to patch RxJS scheduler.

content\_copy

```
1. it('should get Date diff correctly in fakeAsync with rxjs scheduler', fakeAsync(() => {  
2.   // need to add `import 'zone.js/dist/zone-patch-rxjs-fake-async'  
3.   // to patch rxjs scheduler  
4.   let result = null;  
5.   of ('hello').pipe(delay(1000)).subscribe(v => { result = v; });  
6.   expect(result).toBeNull();  
7.   tick(1000);  
8.   expect(result).toBe('hello');  
9.  
10.  const start = new Date().getTime();  
11.  let dateDiff = 0;  
12.  interval(1000).pipe(take(2)).subscribe(() => dateDiff  
   = (new Date().getTime() - start));  
13.  
14.  tick(1000);  
15.  expect(dateDiff).toBe(1000);  
16.  tick(1000);  
17.  expect(dateDiff).toBe(2000);
```

```
18. }));
```

## Support more macroTasks

By default [fakeAsync\(\)](#) supports the following macroTasks.

- setTimeout
- setInterval
- requestAnimationFrame
- webkitRequestAnimationFrame
- mozRequestAnimationFrame

If you run other macroTask such as HTMLCanvasElement.toBlob(), Unknown macroTask scheduled in fake [async](#) test error will be thrown.

src/app/shared/canvas.component.spec.ts

src/app/shared/canvas.component.ts

```
import { TestBed, async, tick, fakeAsync } from '@angular/core/testing';
import { CanvasComponent } from './canvas.component';
describe('CanvasComponent', () => {
beforeEach(async() => {
  TestBed.configureTestingModule({
    declarations: [
      CanvasComponent
    ],
    }).compileComponents();
  }));
beforeEach(() => {
  window['__zone_symbol__FakeAsyncTestMacroTask'] = [
    {
      source: 'HTMLCanvasElement.toBlob',
      callbackArgs: [{ size: 200 }]
    }
  ];
});
it('should be able to generate blob data from canvas', fakeAsync(() => {
  const fixture = TestBed.createComponent(CanvasComponent);
  fixture.detectChanges();
  tick();
  const app = fixture.debugElement.componentInstance;
  expect(app.blobSize).toBeGreaterThanOrEqual(0);
}));
```

If you want to support such case, you need to define the macroTask you want to support in `beforeEach()`. For example:

```
1. beforeEach(() => {
2.   window['__zone_symbol__FakeAsyncTestMacroTask'] = [
3.     {
4.       source: 'HTMLCanvasElement.toBlob',
5.       callbackArgs: [{ size: 200 }]
6.     }
7.   ];
8. });
9.
10.it('toBlob should be able to run in fakeAsync', fakeAsync((() => {
11.   const canvas: HTMLCanvasElement = document.getElementById('canvas') as
    HTMLCanvasElement;
12.   let blob = null;
13.   canvas.toBlob(function(b) {
14.     blob = b;
15.   });
16.   tick();
17.   expect(blob.size).toBe(200);
18. })
19.);
```

## Async observables

You might be satisfied with the test coverage of these tests.

But 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.

```
// 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`

```
/** 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 [RxJS defer\(\) operator](#) 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](#), emits once and completes. Subscribers are unsubscribed after they receive the data value.

There's a similar helper for producing an async error.

```
/** Create async observable error that errors
 * after a JS engine turn */
export function throwError<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\(\)](#) test that demonstrates the data flow you'd expect in the real world.

```
it('should show quote after getQuote (fakeAsync)', fakeAsync((done) => {
  fixture.detectChanges(); // ngOnInit()
  expect(quoteEl.textContent).toBe('...', 'should show placeholder');

  tick(); // 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\(\)](#). 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 `async()`

To use `async()` functionality, you need to import zone-testing, for details, please read [setup guide](#).

The `fakeAsync()` utility function has a few limitations. In particular, it won't work if the test body makes an XHR call.

XHR calls within a test are rare so you can generally stick with `fakeAsync()`. But if you ever do need to call XHR, you'll want to know about `async()`.

The `TestBed.compileComponents()` method (see [below](#)) calls XHR to read external template and css files during "just-in-time" compilation. Write tests that call `compileComponents()` with the `async()` utility.

Here's the previous `fakeAsync()` test, re-written with the `async()` utility.

```
it('should show quote after getQuote (async)', async(() => {
  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 `async()` 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 `async()`, remember to cancel the timer with `clearInterval()` after the test, otherwise the `async()` never ends.

### `whenStable`

The test must wait for the `getQuote()` observable to emit the next quote. Instead of calling `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 `async()` and `fakeAsync()` functions greatly simplify Angular asynchronous testing, you can still fall back to the traditional technique and pass it a function that takes a `done` callback.

You can't call `done()` in `async()` or `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 `async()` and `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.

```
it('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 subscribecallback 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.

```
it('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](#) 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)
```

```
import { cold, getTestScheduler } from 'jasmine-marbles';
```

Here's the complete test for getting a quote:

```
it('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 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\(\)`](#). 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](#) that waits three [frames](#) (---), emits a value (x), and completes (|). In the second argument you map the value marker (x) to the emitted value (testQuote).

```
const 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.

```
getTestScheduler().flush(); // flush the observables
```

This step serves a purpose analogous to [tick\(\)](#) and [whenStable\(\)](#) in the earlier [fakeAsync\(\)](#) and [async\(\)](#) 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.

```
it('should display error when TwainService fails', 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 the observables
  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\(\)](#) and [tick\(\)](#), because the component itself calls [setTimeout\(\)](#) when processing errors.

Look at the marble observable definition.

```
const 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](#) methods return cold observables.

A hot observable is already producing values before you subscribe to it. The [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](#).

---

## 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)

```
<dashboard-hero *ngFor="let hero of heroes" class="col-1-4"
[hero]=hero (selected)="gotoDetail($event)">
</dashboard-hero>
```

The DashboardHeroComponent appears in an [\\*ngFor](#) 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)

```
@Component({
  selector: 'dashboard-hero',
  template: `
    <div (click)="click()" class="hero">
      {{hero.name | uppercase}}
    </div>`,
  styleUrls: [ './dashboard-hero.component.css' ]
```

```
})
export class DashboardHeroComponent {
  @Input() hero: Hero;
  @Output() selected = new 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)

```
constructor(
  private 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](#) 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)

```
TestBed.configureTestingModule({
  declarations: [ DashboardHeroComponent ]
})
fixture = TestBed.createComponent(DashboardHeroComponent);
comp = fixture.componentInstance;

// find the hero's 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 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 via the property binding in its repeater.

The following test verifies that the hero name is propagated to the template via a binding.

```
it('should display hero name in uppercase', () => {
  const expectedPipedName = expectedHero.name.toUpperCase();
  expect(heroEl.textContent).toContain(expectedPipedName);
});
```

Because the [template](#) passes the hero name through the Angular [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](#)—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:

```
it('should raise selected event when clicked (triggerEventHandler)', () => {
  let selectedHero: Hero;
  comp.selected.subscribe((hero: Hero) => selectedHero = hero);

  heroDe.triggerEventHandler('click', null);
  expect(selectedHero).toBe(expectedHero);
});
```

The component's selected property returns an [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](#) 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.

```
heroDe.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](#) directive expects an object with a button property that identifies which mouse button (if any) was pressed during the click. The [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.

```
it('should raise selected event when clicked (element.click)', () => {
  let selectedHero: Hero;
  comp.selected.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)

```
/** Button events to pass to 'DebugElement.triggerEventHandler' for 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 | 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 wish, you can pass a custom event object as the second parameter. The default is a (partial) [left-button mouse event object](#) accepted by many handlers including the [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)

```
it('should raise selected event when clicked (click helper)', () => {
  let selectedHero: Hero;
  comp.selected.subscribe(hero => selectedHero = hero);

  click(heroDe); // click helper with DebugElement
  click(heroEl); // click helper with native element

  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` 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)
```

```
@Component({
  template: `
    <dashboard-hero
      [hero]="hero" (selected)="onSelected($event)">
    </dashboard-hero>`
})
class TestHostComponent {
  hero: Hero = {id: 42, name: 'Test Name'};
  selectedHero: Hero;
  onSelected(hero: Hero) { this.selectedHero = hero; }
}
```

This test host binds to DashboardHeroComponent as the DashboardComponent would but without the noise of the [Router](#), the HeroService, or the `*ngFor` 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)
```

```
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 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)
```

```
it('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](#) 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](#), which it injects together with the HeroService.

```
app/dashboard/dashboard.component.ts (constructor)
```

```
constructor(
  private router: Router,
  private heroService: HeroService) {
```

Mocking the HeroService with a spy is a [familiar story](#). But the [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](#)

```
app/dashboard/dashboard.component.ts (goToDetail)
```

```
gotoDetail(hero: Hero) {
  let 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)
const routerSpy = jasmine.createSpyObj('Router', ['navigateByUrl']);
const heroServiceSpy = jasmine.createSpyObj('HeroService', ['getHeroes']);
```

```
TestBed.configureTestingModule({
  providers: [
    { provide: HeroService, useValue: heroServiceSpy },
    { provide: 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)
it('should tell ROUTER to navigate when hero clicked', () => {

  heroClick(); // 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](#) 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](#) to navigate to `heroes/:id`. The `:id` is a route parameter whose value is the id of the hero to edit.

The [Router](#) matches that URL to a route to the `HeroDetailComponent`. It creates an [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)

constructor(
  private heroDetailsService: HeroDetailService,
  private route: ActivatedRoute,
  private router: Router) {
}
```

The `HeroDetail` component needs the `id` parameter so it can fetch the corresponding hero via the `HeroDetailService`. The component has to get the `id` from the [ActivatedRoute.paramMap](#) property which is an Observable.

It can't just reference the `id` property of the [ActivatedRoute.paramMap](#). The component has to subscribe to the [ActivatedRoute.paramMap](#) observable and be prepared for the `id` to change during its lifetime.

```
app/hero/hero-detail.component.ts (ngOnInit)

ngOnInit(): void {
  // get hero when 'id' param changes
  this.route.paramMap.subscribe(pmap => this.getHero(pmap.get('id')));
}
```

The [Router](#) guide covers [ActivatedRoute.paramMap](#) in more detail.

Tests can explore how the `HeroDetailComponent` responds to different `id` parameter values by manipulating the [ActivatedRoute](#) injected into the component's constructor.

You know how to spy on the [Router](#) and a data service.

You'll take a different approach with [ActivatedRoute](#) because

- `paramMap` returns an Observable that can emit more than one value during a test.
- You need the router helper function, [convertToParamMap\(\)](#), to create a [ParamMap](#).
- Other routed components tests need a test double for [ActivatedRoute](#).

These differences argue for a re-usable stub class.

### ActivatedRouteStub

The following ActivatedRouteStub class serves as a test double for [ActivatedRoute](#).

testing/activated-route-stub.ts (ActivatedRouteStub)

```
import { convertToParamMap, ParamMap, 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>();

  constructor(initialParams?: Params) {
    this.setParamMap(initialParams);
  }

  /** The mock paramMap observable */
  readonly paramMap = this.subject.asObservable();

  /** Set the paramMap observable's next value */
  setParamMap(params?: Params) {
    this.subject.next(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](#).

### 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)

```
describe('when navigate to existing hero', () => {
  let expectedHero: Hero;
```

```

beforeEach(async(() => {
  expectedHero = firstHero;
  activatedRoute.setParamMap({ id: expectedHero.id });
  createComponent();
}));

it('should display that hero\'s name', () => {
  expect(page.nameDisplay.textContent).toBe(expectedHero.name);
});
});

```

The `createComponent()` method and `page` object are discussed [below](#). 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](#) 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)

describe('when navigate to non-existent hero id', () => {
  beforeEach(async(() => {
    activatedRoute.setParamMap({ id: 99999 });
    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 app 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)

describe('when navigate with no hero id', () => {
  beforeEach(async( 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](#) directives.

app/app.component.html

```
<app-banner></app-banner>
<app-welcome></app-welcome>
<nav>
  <a routerLink="/dashboard">Dashboard</a>
  <a routerLink="/heroes">Heroes</a>
  <a routerLink="/about">About</a>
</nav>
<router-outlet></router-outlet>
```

While the AppComponent class is empty, you may want to write unit tests to confirm that the links are wired properly to the [RouterLink](#) directives, perhaps for the reasons [explained below](#).

To validate the links, you don't need the [Router](#) to navigate and you don't need the [<router-outlet>](#) to mark where the [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](#) to create them.

If you neglect to declare them, the Angular compiler won't recognize the `<app-banner>`, `<app>Welcome>`, and `<router-outlet>`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 the 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)

```
@Component({selector: 'app-banner', template: ''})  
class BannerStubComponent {}
```

```
@Component({selector: 'router-outlet', template: ''})  
class RouterOutletStubComponent {}
```

```
@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](#) configuration next to the components, directives, and pipes that need to be real.

app/app.component.spec.ts (TestBed stubs)

```
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](#), is a test version of the real `RouterLink` that helps with the link tests.

The rest are stubs.

## **NO\_ERRORS\_SCHEMA**

In the second approach, add **NO\_ERRORS\_SCHEMA** to the TestBed.schemas metadata.

app/app.component.spec.ts (NO\_ERRORS\_SCHEMA)

```
TestBed.configureTestingModule({
```

```
  declarations: [
```

```
    AppComponent,
```

```
    RouterLinkDirectiveStub
```

```
  ],
```

```
  schemas: [ NO_ERRORS_SCHEMA ]
```

```
})
```

The **NO\_ERRORS\_SCHEMA** tells the Angular compiler to ignore unrecognized elements and attributes.

The compiler will recognize the <app-root> element and the [routerLink](#) attribute because you declared a corresponding AppComponent and RouterLinkDirectiveStub in the [TestBed](#) configuration.

But the compiler won't throw an error when it encounters <app-banner>, <app-welcome>, or <[router-outlet](#)>. 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** approach is the easier of the two but don't overuse it.

The **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)

```
TestBed.configureTestingModule({
```

```
  declarations: [
```

```
    AppComponent,
```

```
    BannerStubComponent,
```

```
    RouterLinkDirectiveStub
```

```
],
schemas: [ NO_ERRORS_SCHEMA ]
})
```

The Angular compiler creates the BannerComponentStub for the <app-banner> element and applies the RouterLinkStubDirective to the anchors with the `routerLink` attribute, but it ignores the <app-welcome> and <`router-outlet`>tags.

---

## Components with RouterLink

The real RouterLinkDirective is quite complicated and entangled with other components and directives of the `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)

```
@Directive({
  selector: '[routerLink]',
  host: { '(click)': 'onClick()' }
})
export class RouterLinkDirectiveStub {
  @Input('routerLink') linkParams: any;
  navigatedTo: any = null;

  onClick() {
    this.navigatedTo = this.linkParams;
  }
}
```

The URL bound to the `[routerLink]` attribute flows in to the directive's `linkParams` property.

The `host` metadata property 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 set 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)

beforeEach(() => {
  fixture.detectChanges(); // 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'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](#) wrappers around the matching elements.
3. Each [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)

<nav>
  <a routerLink="/dashboard">Dashboard</a>
  <a routerLink="/heroes">Heroes</a>
  <a routerLink="/about">About</a>
</nav>

```

Here are some tests that confirm those links are wired to the [routerLink](#) directives as expected:

```

app/app.component.spec.ts (selected tests)

it('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
  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](#) 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](#) 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 app 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](#).

---

### Use a page object

The HeroDetailComponent is a simple view with a title, two hero fields, and two buttons.

# Narco Details

**id:** 12

**name:**

**Save**

**Cancel**

But there's plenty of template complexity even in this simple form.

app/hero/hero-detail.component.html

```
<div *ngIf="hero">
  <h2><span>{{hero.name | titlecase}}</span> Details</h2>
  <div>
    <label>id: </label>{{hero.id}}</div>
  <div>
    <label for="name">name: </label>
    <input id="name" [(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)

```
class Page {
  // getter properties wait to query the DOM until called.
  get buttons() { return this.querySelectorAll('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(fixture: ComponentFixture<HeroDetailComponent>) {
  // get the navigate spy from the injected router spy object
  const routerSpy = <any> fixture.debugElement.injector.get(Router);
  this.navigateSpy = routerSpy.navigate;

  // spy on component's `gotoList()` method
  const component = fixture.componentInstance;
  this.gotoListSpy = spyOn(component, 'gotoList').and.callThrough();
}

<:///query helpers /////
private 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)

/** 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](#) 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)

```
it('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(() => {
  click(page.saveBtn);
  tick(); // wait for async save to complete
  expect(page.navigateSpy.calls.any()).toBe(true, 'router.navigate called');
}));

it('should convert hero name to Title Case', () => {
  // get the name's input and display elements from the DOM
  const hostElement = 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.
nameInput.dispatchEvent(newEvent('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)
```

```
import { Component } from '@angular/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` tries to create the component.

```
app/banner/banner.component.spec.ts (setup that fails)
```

```
beforeEach(() => {
  TestBed.configureTestingModule({
    declarations: [ BannerComponent ],
  });
});
```

```
fixture = TestBed.createComponent(BannerComponent);
});
```

Recall that the app hasn't been compiled. So when you call `createComponent()`, the [TestBed](#) compiles implicitly.

That's not a problem when the source code is in memory. But the `BannerComponent` requires external files that the compile must read from the file system, an inherently asynchronous operation.

If the [TestBed](#) were allowed to continue, the tests would run and fail mysteriously before the compiler could finish.

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` (e.g., forget to use [`async`\(\)](#) 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.

To follow this pattern, import the [`async`\(\)](#) helper with the other testing symbols.

```
import { async,  ComponentFixture,  TestBed } from '@angular/core/testing';
```

### **The `async` `beforeEach`**

Write the first `async` `beforeEach` like this.

app/banner/banner-external.component.spec.ts (async `beforeEach`)

```
beforeEach(async() => {
  TestBed.configureTestingModule({
    declarations: [ BannerComponent ],
  })
  .compileComponents(); // compile template and css
});
```

The [`async`\(\)](#) helper function takes a parameterless function with the body of the setup.

The `TestBed.configureTestingModule()` method returns the [`TestBed`](#) class so you can chain calls to other [`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 be require external files.

The TestBed.compileComponents method asynchronously compiles all components configured in the testing module.

Do not re-configure the [TestBed](#) after calling compileComponents().

Calling compileComponents() closes the current [TestBed](#) instance to further configuration. You cannot call any more [TestBed](#) configuration methods, not configureTestingModule() nor any of the override... methods. The [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)

```
beforeEach(() => {
  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)

```
beforeEach(async(() => {
  TestBed.configureTestingModule({
    declarations: [ BannerComponent ],
  })
  .compileComponents()
  .then(() => {
    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)
```

```
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](#) 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](#), it needs:

- [NgModel](#) and friends in the [FormsModule](#) to enable two-way data binding.
- The [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)
```

```
beforeEach(async(() => {  
  const routerSpy = createRouterSpy();
```

```
  TestBed.configureTestingModule({  
    imports: [ FormsModule ],  
    declarations: [ HeroDetailComponent, TitleCasePipe ],  
    providers: [
```

```

    { provide: ActivatedRoute, useValue: activatedRoute },
    { provide: HeroService, useClass: TestHeroService },
    { provide: 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\(\)`](#) 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 app components need the [FormsModule](#) and the [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)

```
beforeEach(async(() => {
  const routerSpy = createRouterSpy();
```

```
TestBed.configureTestingModule({
  imports: [ SharedModule ],
  declarations: [ HeroDetailComponent ],
  providers: [
    { provide: ActivatedRoute, useValue: activatedRoute },
    { provide: HeroService, useClass: TestHeroService },
    { provide: 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](#) 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)

```

beforeEach(async(() => {
  const routerSpy = createRouterSpy();

  TestBed.configureTestingModule({
    imports: [ HeroModule ],
    providers: [
      { provide: ActivatedRoute, useValue: activatedRoute },
      { provide: HeroService, useClass: TestHeroService },
      { provide: 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.
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)

```

@Component({
  selector: 'app-hero-detail',
  templateUrl: './hero-detail.component.html',
  styleUrls: ['./hero-detail.component.css'],
  providers: [ HeroDetailService ]
})
export class HeroDetailComponent implements OnInit {
  constructor(
    private heroDetailsService: HeroDetailService,
    private route: ActivatedRoute,
    private router: Router) {
  }
}

```

It's not possible to stub the component's HeroDetailService in the providers of the TestBed.configureTestingModuleTestingModule. 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.configureTestingModuleTestingModule 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)

```
@Injectable()
export class HeroDetailsService {
  constructor(private heroService: HeroService) { }
/* . . . */
}
```

The [previous test configuration](#) 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)

```
beforeEach(async(() => {
  const routerSpy = createRouterSpy();

  TestBed.configureTestingModule({
    imports: [ HeroModule ],
    providers: [
      { provide: ActivatedRoute, useValue: activatedRoute },
      { provide: 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.

### The overrideComponent method

Focus on the overrideComponent method.

app/hero/hero-detail.component.spec.ts (overrideComponent)

```

.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 is a generic defined as follows:

```

type MetadataOverride = {
  add?: Partial;
  remove?: Partial;
  set?: Partial;
};

```

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 decorator:

```

selector?: 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 HeroDetailsService 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 HeroDetailsService were called by spying on the service methods. Accordingly, the stub implements its methods as spies:

```
app/hero/hero-detail.component.spec.ts (HeroDetailServiceSpy)

class 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)

let hdsSpy: HeroDetailServiceSpy;

beforeEach(async() => {
  createComponent();
  // get the component's injected HeroDetailServiceSpy
  hdsSpy = fixture.debugElement.injector.get(HeroDetailsService) 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() => {
  const origName = hdsSpy.testHero.name;
  const newName = 'New Name';

  page.nameInput.value = newName;
  page.nameInput.dispatchEvent(newEvent('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(); // 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](#) 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.

---

## Attribute Directive Testing

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.

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

```
import { Directive, ElementRef, Input, OnChanges } from '@angular/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 {

  defaultColor = 'rgb(211, 211, 211)'; // lightgray

  @Input('highlight') bgColor: string;

  constructor(private el:  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

```

import { Component } from '@angular/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 above (in particular the "[Shallow test](#)" approach).

app/about/about.component.spec.ts

```

beforeEach(() => {
  fixture = TestBed.configureTestingModule({
    declarations: [ AboutComponent, HighlightDirective],
    schemas: [ NO\_ERRORS\_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)

```
@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 {}
```

**Something Yellow**

**The Default (Gray)**

**No Highlight**

cyan

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)

1. `beforeEach(() => {`
2. `fixture = TestBed.configureTestingModule({`
3.  `declarations: [ HighlightDirective, TestComponent ]`
4. `})`
5. `.createComponent(TestComponent);`

```
6.
7. fixture.detectChanges(); // initial binding
8.
9. // all elements with an attached HighlightDirective
10. des = fixture.debugElement.queryAll(By.directive(HighlightDirective));
11.
12. // the h2 without the HighlightDirective
13. bareH2 = fixture.debugElement.query(By.css('h2:not([highlight])'));
14.});
15.
16.// color tests
17.it('should have three highlighted elements', () => {
18. expect(des.length).toBe(3);
19.});
20.
21.it('should color 1st <h2> background "yellow"', () => {
22. const bgColor = des[0].nativeElement.style.backgroundColor;
23. expect(bgColor).toBe('yellow');
24.});
25.
26.it('should color 2nd <h2> background w/ default color', () => {
27. const dir = des[1].injector.get(HighlightDirective) as HighlightDirective;
28. const bgColor = des[1].nativeElement.style.backgroundColor;
29. expect(bgColor).toBe(dir.defaultColor);
30.});
31.
32.it('should bind <input> background to value color', () => {
33. // easier to work with nativeElement
34. const input = des[2].nativeElement as HTMLInputElement;
35. expect(input.style.backgroundColor).toBe('cyan', 'initial backgroundColor');
36.
37. // dispatch a DOM event so that Angular responds to the input value change.
38. input.value = 'green';
39. input.dispatchEvent(newEvent('input'));
40. fixture.detectChanges();
41.
42. expect(input.style.backgroundColor).toBe('green', 'changed backgroundColor');
43.});
44.
45.
46.it('bare <h2> should not have a customProperty', () => {
47. expect(bareH2.properties['customProperty']).toBeUndefined();
```

```
48.});
```

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 [:not pseudo-class](#) 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](#) affords access to element styles even in the absence of a real browser, thanks to the [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](#) affords access to the artificial custom property that is set by the directive.
- 

## Pipe Testing

Pipes are easy to test without the Angular testing utilities.

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](#) metadata and an interface.

Consider a [TitleCasePipe](#) that capitalizes the first letter of each word. Here's a naive implementation with a regular expression.

app/shared/title-case.pipe.ts

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({name: 'titlecase', pure: true})
/** Transform to Title Case: uppercase the first letter of the words in a string.*/
export class TitleCasePipe implements 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

```
1. describe('TitleCasePipe', () => {
2.   // This pipe is a pure, stateless function so no need for BeforeEach
3.   let pipe = new TitleCasePipe();
4.
5.   it('transforms "abc" to "Abc"', () => {
6.     expect(pipe.transform('abc')).toBe('Abc');
7.   });
8.
9.   it('transforms "abc def" to "Abc Def"', () => {
10.    expect(pipe.transform('abc def')).toBe('Abc Def');
11.  });
12.
13. // ... more tests ...
14.});
```

## Write DOM tests too

These are tests of the pipe in isolation. They can't tell if the [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)

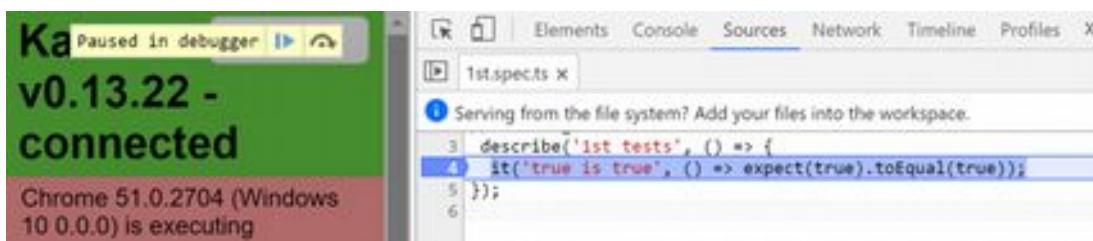
```
1. it('should convert hero name to Title Case', () => {
2.   // get the name's input and display elements from the DOM
3.   const hostElement = fixture.nativeElement;
4.   const nameInput: HTMLInputElement = hostElement.querySelector('input');
5.   const nameDisplay: HTMLElement = hostElement.querySelector('span');
6.
7.   // simulate user entering a new name into the input box
8.   nameInput.value = 'quick BROWN fOx';
9.
10. // dispatch a DOM event so that Angular learns of input value change.
11. nameInput.dispatchEvent(newEvent('input'));
12.
13. // Tell Angular to update the display binding through the title pipe
14. fixture.detectChanges();
15.
16. expect(nameDisplay.textContent).toBe('Quick Brown Fox');
17.});
```

---

## Test debugging

Debug specs in the browser in the same way that you debug an application.

1. Reveal the karma browser window (hidden earlier).
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 OSX).
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 section takes inventory of the most useful Angular testing features and summarizes what they do.

The Angular testing utilities include the [TestBed](#), the [ComponentFixture](#), and a handful of functions that control the test environment. The [TestBed](#) and [ComponentFixture](#) classes are covered separately.

Here's a summary of the stand-alone functions, in order of likely utility:

Function	Description
<a href="#">async</a>	Runs the body of a test (it) or setup (beforeEach) function within a special async test zone. See <a href="#">discussion above</a> .
<a href="#">fakeAsync</a>	Runs the body of a test (it) within a special fakeAsync test zone, enabling a linear control flow coding style. See <a href="#">discussion above</a> .
<a href="#">tick</a>	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

	<p>post, "<a href="#">Tasks, microtasks, queues and schedules</a>".</p>
	<p>Accepts an optional argument that moves the virtual clock forward by the specified number of milliseconds, clearing asynchronous activities scheduled within that timeframe. See <a href="#">discussion above</a>.</p>
inject	<p>Injects one or more services from the current <a href="#"> TestBed </a> injector into a test function. It cannot inject a service provided by the component itself. See discussion of the <a href="#"> debugElement.injector </a>.</p>
<a href="#"> discardPeriodicTasks </a>	<p>When a <a href="#"> fakeAsync </a>() test ends with pending timer event tasks (queued setTimeOut and setInterval callbacks), the test fails with a clear error message.</p>
<a href="#"> flushMicrotasks </a>	<p>In general, a test should end with no queued tasks. When pending timer tasks are expected, call <a href="#"> discardPeriodicTasks </a> to flush the task queue and avoid the error.</p>
<a href="#"> ComponentFixtureAutoDetect </a>	<p>When a <a href="#"> fakeAsync </a>() test ends with pending micro-tasks such as unresolved promises, the test fails with a clear error message.</p>
<a href="#"> get TestBed </a>	<p>In general, a test should wait for micro-tasks to finish. When pending microtasks are expected, call <a href="#"> flushMicrotasks </a> to flush the micro-task queue and avoid the error.</p>

---

## TestBed class summary

The  [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](#)  metadata properties.

```
type TestModuleMetadata = {
  providers?: any[];
  declarations?: any[];
  imports?: any[];
  schemas?: Array<SchemaMetadata | any[]>;
};
```

Each override method takes a `MetadataOverride`<T> where T is the kind of metadata appropriate to the method, that is, the parameter of an `@NgModule`, `@Component`, `@Directive`, or `@Pipe`.

```
type MetadataOverride = {
  add?: Partial;
  remove?: Partial;
  set?: Partial;
};
```

The `TestBed` API consists of static class methods that either update or reference a global instance of the `TestBed`.

Internally, all static methods cover methods of the current runtime `TestBed` instance, which is also returned by the `get TestBed()` function.

Call `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
	The testing shims ( <code>karma-test-shim</code> , <code>browser-test-shim</code> ) establish the <code>initial test environment</code> and a default testing module. The default testing module is configured with basic declaratives and some Angular service substitutes that every tester needs.
configureTestingModule	Call <code>configureTestingModule</code> 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 <code>templateUrl</code> or <code>styleUrls</code> because fetching component template and style files is necessarily asynchronous. See <a href="#">above</a> .

After calling `compileComponents`, the `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 <a href="#"> TestBed </a> configuration. After calling compileComponent, the <a href="#"> TestBed </a> configuration is frozen for the duration of the current spec.
overrideModule	Replace metadata for the given <a href="#"> NgModule </a> . 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.
get	<p>Retrieve a service from the current <a href="#"> TestBed </a> injector.</p> <p>The inject function is often adequate for this purpose. But inject throws an error if it can't provide the service.</p> <p>What if the service is optional?</p> <p>The TestBed.get() method takes an optional second parameter, the object to return if Angular can't find the provider (null in this example):</p> <pre>app/demo/demo.testbed.spec.ts</pre> <pre>service = TestBed.get(NotProvided, null); // service is null</pre> <p>After calling get, the <a href="#"> TestBed </a> configuration is frozen for the duration of the current spec.</p> <p>initTestEnvironment</p> <p>Initialize the testing environment for the entire test run.</p> <p>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.</p> <p>You may call this method exactly once. If you must change this default in the middle of your test run, call resetTestEnvironment first.</p> <p>Specify the Angular compiler factory, a <a href="#"> PlatformRef </a>, and a default Angular testing module. Alternatives for non-browser platforms are available in the general</p>

	form@angular/platform-<platform_name>/testing/<platform_name>.
resetTestEnvironment	Reset the initial test environment, including the default testing module.

A few of the  [TestBed](#) instance methods are not covered by static  [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](#) for that component.

The [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
<code>componentInstance</code>	The instance of the component class created by <code>TestBed.createComponent</code> .
<code>debugElement</code>	The <a href="#">DebugElement</a> associated with the root element of the component.
<code>nativeElement</code>	The <code>debugElement</code> 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 <a href="#">below</a> .
<code>changeDetectorRef</code>	The native DOM element at the root of the component.
<code>changeDetectorRef</code>	The <a href="#">ChangeDetectorRef</a> for the component.
<code>changeDetectorRef</code>	The <a href="#">ChangeDetectorRef</a> is most valuable when testing a component that has the <a href="#">ChangeDetectionStrategy.OnPush</a> 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
---------	-------------

	Trigger a change detection cycle for the component.
detectChanges	Call it to initialize the component (it calls <code>ngOnInit</code> ) and after your test code, change the component's data bound property values. Angular can't see that you've changed <code>personComponent.name</code> and won't update the name binding until you call <code>detectChanges</code> .
	Runs <code>checkNoChanges</code> afterwards to confirm that there are no circular updates unless called as <code>detectChanges(false)</code> ;
	Set this to true when you want the fixture to detect changes automatically.
autoDetectChanges	When <code>autodetect</code> is true, the test fixture calls <code>detectChanges</code> immediately after creating the component. Then it listens for pertinent zone events and calls <code>detectChanges</code> accordingly. When your test code modifies component property values directly, you probably still have to call <code>fixture.detectChanges</code> 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.
	Returns a promise that resolves when the fixture is stable.
whenStable	To resume testing after completion of asynchronous activity or asynchronous change detection, hook that promise. See <a href="#">above</a> .
destroy	Trigger component destruction.

## DebugElement

The [DebugElement](#) provides crucial insights into the component's DOM representation.

From the test root component's [DebugElement](#) returned by `fixture.debugElement`, you can walk (and query) the fixture's entire element and component subtrees.

Here are the most useful [DebugElement](#) members for testers, in approximate order of utility:

Member	Description
nativeElement	The corresponding DOM element in the browser (null for WebWorkers).

<a href="#">query</a>	Calling <code>query(predicate: Predicate&lt;DebugElement&gt;)</code> returns the first <code>DebugElement</code> that matches the <a href="#">predicate</a> at any depth in the subtree.
<a href="#">queryAll</a>	Calling <code>queryAll(predicate: Predicate&lt;DebugElement&gt;)</code> returns all <code>DebugElements</code> that matches the <a href="#">predicate</a> at any depth in subtree.
<a href="#">injector</a>	The host dependency injector. For example, the root element's component instance injector.
<a href="#">componentInstance</a>	The element's own component instance, if it has one.
	An object that provides parent context for this element. Often an ancestor component instance that governs this element.
<a href="#">context</a>	When an element is repeated within <code>*ngFor</code> , the context is an <code>NgForRow</code> whose <code>\$implicitproperty</code> is the value of the row instance value. For example, the hero in <code>*ngFor="let hero of heroes"</code> .
	The immediate <code>DebugElement</code> children. Walk the tree by descending through children.
<a href="#">children</a>	<code>DebugElement</code> also has <a href="#">childNodes</a> , a list of <code>DebugNode</code> objects. <code>DebugElement</code> derives from <code>DebugNode</code> objects and there are often more nodes than elements. Testers can usually ignore plain nodes.
<a href="#">parent</a>	The <code>DebugElement</code> parent. Null if this is the root element.
<a href="#">name</a>	The element tag name, if it is an element.
<a href="#">triggerEventHandler</a>	Triggers the event by its name if there is a corresponding listener in the element's <a href="#">listeners</a> collection. The second parameter is the event object expected by the handler. See <a href="#">above</a> .  If the event lacks a listener or there's some other problem, consider calling <code>nativeElement.dispatchEvent(eventObject)</code> .
<a href="#">listeners</a>	The callbacks attached to the component's <a href="#">@Output</a> properties and/or the element's event properties.
<a href="#">providerTokens</a>	This component's injector lookup tokens. Includes the component itself plus the tokens that the component lists in its providers metadata.
<a href="#">source</a>	Where to find this element in the source component template.

<a href="#">references</a>	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](#).

The predicate is any method that takes a [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

```
// Filter for DebugElements with a #content reference
const contentRefs = el.queryAll( de => de.references['content']);
```

The Angular [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

```
// Can find DebugElement either by css selector or by directive
const h2      = fixture.debugElement.query(By.css('h2'));
const directive = fixture.debugElement.query(By.directive(HighlightDirective));
```

---

## Frequently Asked Questions

### Why put spec file next to the file it tests?

It's a good idea to put unit test spec files in the same folder as the application source code files that they test:

- Such tests are easy to find.
  - You see at a glance if a part of your application lacks tests.
  - Nearby tests can reveal how a part works in context.
  - When you move the source (inevitable), you remember to move the test.
  - When you rename the source file (inevitable), you remember to rename the test file.
- 

### When would I put specs in a test folder?

Application integration specs can test the interactions of multiple parts spread across folders and modules. They don't really belong to any part in particular, so they don't have a natural home next to any one file.

It's often better to create an appropriate folder for them in the tests directory.

Of course specs that test the test helpers belong in the test folder, next to their corresponding helper files.

### Why not rely on E2E tests of DOM integration?

The component DOM tests described in this guide often require extensive setup and advanced techniques whereas the [unit tests](#) are comparatively simple.

### Why not defer DOM integration tests to end-to-end (E2E) testing?

E2E tests are great for high-level validation of the entire system. But they can't give you the comprehensive test coverage that you'd expect from unit tests.

E2E tests are difficult to write and perform poorly compared to unit tests. They break easily, often due to changes or misbehavior far removed from the site of breakage.

E2E tests can't easily reveal how your components behave when things go wrong, such as missing or bad data, lost connectivity, and remote service failures.

E2E tests for apps that update a database, send an invoice, or charge a credit card require special tricks and back-doors to prevent accidental corruption of remote resources. It can even be hard to navigate to the component you want to test.

Because of these many obstacles, you should test DOM interaction with unit testing techniques as much as possible.

Bootstrapping

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
```

[platformBrowserDynamic\(\)](#).bootstrapModule(AppModule);

Bootstraps the app, using the root component from the specified [NgModule](#).

NgModules

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

[@NgModule\({ declarations: ..., imports: ...,](#)

exports: ..., providers: ..., bootstrap: ...})

Defines a module that contains components, directives, pipes, and providers.

class MyModule {}

[declarations: \[MyRedComponent, MyBlueComponent, MyDatePipe\]](#)

List of components, directives, and pipes that belong to this module.

[imports: \[BrowserModule, SomeOtherModule\]](#)

List of modules to import into this module. Everything from the imported modules is available to [declarations](#) of this module.

exports: [MyRedComponent, MyDatePipe]

List of components, directives, and pipes visible to modules that import this module.

providers: [MyService, { provide: ... }]

List of dependency injection providers visible both to the contents of this module and to importers of this module.

entryComponents: [SomeComponent, OtherComponent]

List of components not referenced in any reachable template, for example dynamically created from code.

bootstrap: [MyAppComponent]

List of components to bootstrap when this module is bootstrapped.

## Template syntax

<input [value]="firstName">

Binds property value to the result of expression firstName.

<div [attr.role]="myAriaRole">

Binds attribute role to the result of expression myAriaRole.

<div [class.extra-sparkle]="isDelightful">

Binds the presence of the CSS class extra-sparkle on the element to the truthiness of the expression isDelightful.

<div [style.width.px]="mySize">

Binds style property width to the result of expression mySize in pixels. Units are optional.

<button  
(click)="readRainbow(\$event)">

Calls method readRainbow when a click event is triggered on this button element (or its children) and passes in the event object.

<div title="Hello {{ponyName}}">

Binds a property to an interpolated string, for example, "Hello Seabiscuit". Equivalent to: <div [title]=""Hello '+ ponyName">

<p>Hello {{ponyName}}</p>

Binds text content to an interpolated string, for example, "Hello Seabiscuit".

<my-cmp [(title)]="name">

Sets up two-way data binding. Equivalent to: <my-cmp [title]=""name" (titleChange)="name=\$event">

<video #movieplayer ...>

Creates a local variable movieplayer that provides access to the videoelement instance in data-binding and

```
<button  
(click)="movieplayer.play()">  
</video>
```

event-binding expressions in the current template.

```
<p  
*myUnless="myExpression">...</p>
```

The \* symbol turns the current element into an embedded template. Equivalent to: <ng-template [myUnless]="myExpression"><p>...</p></ng-template>

```
<p>Card No.: {{cardNumber |  
myCardNumberFormatter}}</p>
```

Transforms the current value of expression cardNumber via the pipe called myCardNumberFormatter.

```
<p>Employer:  
&{{employer?.companyName}}</p>
```

The safe navigation operator (?) means that the employer field is optional and if undefined, the rest of the expression should be ignored.

```
<svg:rect x="0" y="0" width="100"  
height="100"/>
```

An SVG snippet template needs an svg: prefix on its root element to disambiguate the SVG element from an HTML component.

```
<svg>  
  
<rect x="0" y="0" width="100"  
height="100"/>  
  
</svg>
```

An <svg> root element is detected as an SVG element automatically, without the prefix.

Built-in directives

```
<section *ngIf="showSection">
```

Removes or recreates a portion of the DOM tree based on the showSectionexpression.

```
<li *ngFor="let item of list">
```

Turns the li element and its contents into a template, and uses that to instantiate a view for each item in list.

```
<div  
[ngSwitch]="conditionExpression">
```

Conditionally swaps the contents of the div by selecting one of the embedded templates based on the current value of conditionExpression.

```
<ng-template  
[ngSwitchCase]="case1Exp">...<n  
g-template>
```

```
<ng-template  
ngSwitchCase="case2LiteralString
```

>...</ng-template>

<ng-template
ngSwitchDefault>...</ng-template>

</div>

<div [ngClass]="{'active': isActive,
'disabled': isDisabled}">      Binds the presence of CSS classes on the element to the truthiness of the associated map values. The right-hand expression should return {class-name: true/false} map.

<div [ngStyle]="{'property':
'value'}">      Allows you to assign styles to an HTML element using CSS. You can use CSS directly, as in the first example, or you can call a method from the component.

<div [ngStyle]="dynamicStyles()">

Forms                  import { [FormsModule](#) } from '@angular/forms';

<input
[(ngModel)]="userName">      Provides two-way data-binding, parsing, and validation for form controls.

Class decorators      import { [Directive](#), ... } from '@angular/core';

[@Component](#)(...)      Declares that a class is a component and provides metadata about the component.

class  
MyComponent() {}

[@Directive](#)(...)      Declares that a class is a directive and provides metadata about the directive.

class MyDirective() {}

[@Pipe](#)(...)      Declares that a class is a pipe and provides metadata about the pipe.

class MyPipe() {}

[@Injectable](#)()      Declares that a class has dependencies that should be injected into the constructor when the dependency injector is creating an instance of this class.

class MyService() {}

Directive configuration      [@Directive](#)({ property1: value1, ... })

selector: '.cool-button:not([a](#))'      Specifies a CSS selector that identifies this directive within a template. Supported selectors include element,

	[attribute], .class, and :not().
	Does not support parent-child relationship selectors.
providers: [MyService, { provide: ... }]	List of dependency injection providers for this directive and its children.
Component configuration	<a href="#">@Component</a> extends <a href="#">@Directive</a> , so the <a href="#">@Directive</a> configuration applies to components as well
<a href="#">moduleId</a> : module.id	If set, the templateUrl and styleUrls are resolved relative to the component.
<a href="#">viewProviders</a> : [MyService, { provide: ... }]	List of dependency injection providers scoped to this component's view.
<a href="#">template</a> : 'Hello {{name}}' templateUrl: 'my-component.html'	Inline template or external template URL of the component's view.
styles: ['.primary {color: red}'] styleUrls: ['my-component.css']	List of inline CSS styles or external stylesheet URLs for styling the component's view.
Class field decorators for directives and components	import { <a href="#">Input</a> , ... } from '@angular/core';
<a href="#">@Input()</a> myProperty;	Declares an input property that you can update via property binding (example: <my-cmp [myProperty]="someExpression">).
<a href="#">@Output()</a> myEvent = new EventEmitter();	Declares an output property that fires events that you can subscribe to with an event binding (example: <my-cmp (myEvent)="doSomething()">).
<a href="#">@HostBinding</a> ('class.valid') isValid;	Binds a host element property (here, the CSS class valid) to a directive/component property (isValid).
<a href="#">@HostListener</a> ('click', ['\$event']) onClick(e) {...}	Subscribes to a host element event (click) with a directive/component method (onClick), optionally passing an argument (\$event).
<a href="#">@ContentChild</a> (myPredicate) myChildC omponent;	Binds the first result of the component content query (myPredicate) to a property (myChildComponent) of

	the class.
<code>@ContentChildren(myPredicate) myChildComponents;</code>	Binds the results of the component content query (myPredicate) to a property (myChildComponents) of the class.
<code>@ViewChild(myPredicate) myChildComponent;</code>	Binds the first result of the component view query (myPredicate) to a property (myChildComponent) of the class. Not available for directives.
<code>@ViewChildren(myPredicate) myChildComponents;</code>	Binds the results of the component view query (myPredicate) to a property (myChildComponents) of the class. Not available for directives.
Directive and component change detection and lifecycle hooks	(implemented as class methods)
<code>constructor(myService: MyService,</code> <code>...)</code> { ... }	Called before any other lifecycle hook. Use it to inject dependencies, but avoid any serious work here.
<code>ngOnChanges(changeRecord) { ... }</code>	Called after every change to input properties and before processing content or child views.
<code>ngOnInit() { ... }</code>	Called after the constructor, initializing input properties, and the first call to ngOnChanges.
<code>ngDoCheck() { ... }</code>	Called every time that the input properties of a component or a directive are checked. Use it to extend change detection by performing a custom check.
<code>ngAfterContentInit() { ... }</code>	Called after ngOnInit when the component's or directive's content has been initialized.
<code><u>ngAfterContentChecked()</u> { ... }</code>	Called after every check of the component's or directive's content.
<code>ngAfterViewInit() { ... }</code>	Called after ngAfterContentInit when the component's views and child views / the view that a directive is in has been initialized.
<code><u>ngAfterViewChecked()</u> { ... }</code>	Called after every check of the component's views and child views / the view that a directive is in.
<code>ngOnDestroy() { ... }</code>	Called once, before the instance is destroyed.

## Dependency injection configuration

{ provide: MyService, useClass: MyMockService }	Sets or overrides the provider for MyService to the MyMockService class.
{ provide: MyService, useFactory: myFactory }	Sets or overrides the provider for MyService to the myFactory factory function.
{ provide: MyValue, useValue: 41 }	Sets or overrides the provider for MyValue to the value 41.

Routing and navigation      import { [Routes](#), [RouterModule](#), ... } from '@angular/router';

```
const routes: Routes = [  
  
{ path: '', component:  
HomeComponent },  
  
{ path: 'path/:routeParam',  
component: MyComponent },  
  
{ path: 'staticPath', component:  
... },  
  
{ path: '**', component: ... },  
  
{ path: 'oldPath', redirectTo:  
'/staticPath' },  
  
{ path: ..., component: ..., data:  
{ message: 'Custom' } }  
]);
```

```
const routing =  
RouterModule.forRoot(routes);
```

<[router-outlet](#)></[router-outlet](#)>

<[router-outlet](#) name="aux"></  
router-outlet>

<[a](#) [routerLink](#)="/path">

<[a](#) [[routerLink](#)]="[ '/path',  
routeParam ]">

Configures routes for the application. Supports static, parameterized, redirect, and wildcard routes. Also supports custom route data and resolve.

Creates a link to a different view based on a route instruction consisting of a route path, required and optional parameters, query parameters, and a fragment. To navigate to a root route, use the / prefix; for a child route, use the ./prefix; for a

```
<a [routerLink]="[ '/path',  
{ matrixParam: 'value' } ]">
```

```
<a [routerLink]="[ '/path' ]"  
[queryParams]="{{ page: 1 }}>
```

sibling or parent, use the ../ prefix.

```
<a [routerLink]="[ '/path' ]"  
fragment="anchor">
```

```
<a [routerLink]=[ '/path' ]  
routerLinkActive="active">
```

The provided classes are added to the element when the routerLink becomes the current active route.

```
class CanActivateGuard  
implements CanActivate {
```

```
canActivate(
```

route:

```
ActivatedRouteSnapshot,
```

```
state: RouterStateSnapshot
```

```
): Observable<boolean>|  
Promise<boolean>|boolean  
{ ... }
```

```
}
```

```
{ path: ..., canActivate:  
[CanActivateGuard] }
```

An interface for defining a class that the router should call first to determine if it should activate this component. Should return a boolean or an Observable/Promise that resolves to a boolean.

```
class CanDeactivateGuard  
implements CanDeactivate<T>  
{
```

```
canDeactivate(
```

```
component: T,
```

route:

```
ActivatedRouteSnapshot,
```

```
state: RouterStateSnapshot
```

```
): Observable<boolean>|  
Promise<boolean>|boolean
```

An interface for defining a class that the router should call first to determine if it should deactivate this component after a navigation. Should return a boolean or an Observable/Promise that resolves to a boolean.

```
{ ... }  
}  
  
{ path: ..., canDeactivate:  
[CanDeactivateGuard] }  
  
class CanActivateChildGuard  
implements CanActivateChild {  
  
  canActivateChild(  
  
    route:  
    ActivatedRouteSnapshot,  
  
    state: RouterStateSnapshot  
  ): Observable<boolean>|  
  Promise<boolean>|boolean  
  { ... }  
  
}  
  
{ path: ..., canActivateChild:  
[CanActivateGuard],  
  
  children: ... }  
  
class ResolveGuard  
implements Resolve<T> {  
  
  resolve(  
  
    route:  
    ActivatedRouteSnapshot,  
  
    state: RouterStateSnapshot  
  ): Observable<any>|  
  Promise<any>|any { ... }  
  
}  
  
{ path: ..., resolve:  
[ResolveGuard] }  
  
class CanLoadGuard
```

An interface for defining a class that the router should call first to determine if it should activate the child route. Should return a boolean or an Observable/Promise that resolves to a boolean.

An interface for defining a class that the router should call first to resolve route data before rendering the route. Should return a value or an Observable/Promise that resolves to a value.

An interface for defining a class that the router should call

```
implements CanLoad {  
  canLoad(  
    route: Route  
    ): Observable<boolean>|  
    Promise<boolean>|boolean  
    { ... }  
  }  
  
{ path: ..., canLoad:  
  [CanLoadGuard], loadChildren:  
  ... }
```