AngularJS to Angular Quick Reference

{@a top}

Angular is the name for the Angular of today and tomorrow. *AngularJS* is the name for all v1.x versions of Angular.

This guide helps you transition from AngularJS to Angular by mapping AngularJS syntax to the equivalent Angular syntax.

See the Angular syntax in this .

Template basics

Templates are the user-facing part of an Angular application and are written in HTML. The following table lists some of the key AngularJS template features with their equivalent Angular template syntax.

AngularJS	Angular
### Bindings/interpolation Your favorite hero is: {{vm.favoriteHero}} In AngularJS, an expression in curly braces denotes one-way binding. This binds the value of the element to a property in the controller associated with this template. When using the `controller as` syntax, the binding is prefixed with the controller alias (`vm` or `$ctrl`) because you have to be specific about the source of the binding.	### Bindings/interpolation In Angular, a template expression in curly braces still denotes one-way binding. This binds the value of the element to a property of the component. The context of the binding is implied and is always the associated component, so it needs no reference variable. For more information, see the [Interpolation](guide/template-syntax#interpolation) section of the [Template Syntax](guide/template-syntax) page.
### Filters <td>{{movie.title | uppercase}}</td> To filter output in AngularJS templates, use the pipe character (|) and one or more filters. This example filters the `title` property to uppercase.	### Pipes In Angular you use similar syntax with the pipe (|) character to filter output, but now you call them **pipes**. Many (but not all) of the built-in filters from AngularJS are built-in pipes in Angular. For more information, see [Filters/pipes](guide/ajs-quick-reference#filters-pipes) below.
### Local variables <tr ng-repeat="movie in vm.movies"> <td>{{movie.title}}</td> </tr> Here, `movie` is a user-defined local variable.	### Input variables Angular has true template input variables that are explicitly defined using the `let` keyword. For more information, see the [ngFor micro-syntax](guide/template-syntax#microsyntax) section of the [Template Syntax](guide/template-syntax) page.

Template directives

AngularJS provides more than seventy built-in directives for templates. Many of them aren't needed in Angular because of its more capable and expressive binding system. The following are some of the key AngularJS built-in directives and their equivalents in Angular.

AngularJS	Angular
### ng-app <body ng-app="movieHunter"> The application startup process is called **bootstrapping**. Although you can bootstrap an AngularJS app in code, many applications bootstrap declaratively with the `ng-app` directive,	### Bootstrapping Angular doesn't have a bootstrap directive. To launch the app in code, explicitly bootstrap the application's root module (`AppModule`) in `main.ts` and

giving it the name of the application's module (`movieHunter`).	the application's root component (`AppComponent`) in `app.module.ts`.
### ng-class <div ng-class="{active: isActive}"> <div ng-class="{active: isActive, shazam: isImportant}"> In AngularJS, the `ng-class` directive includes/excludes CSS classes based on an expression. That expression is often a key-value control object with each key of the object defined as a CSS class name, and each value defined as a template expression that evaluates to a Boolean value. In the first example, the `active` class is applied to the element if `isActive` is true. You can specify multiple classes, as shown in the second example.	### ngClass In Angular, the `ngClass` directive works similarly. It includes/excludes CSS classes based on an expression. In the first example, the `active` class is applied to the element if `isActive` is true. You can specify multiple classes, as shown in the second example. Angular also has **class binding**, which is a good way to add or remove a single class, as shown in the third example. For more information see the [Attribute, class, and style bindings](guide/template-syntax#other-bindings) section of the [Template Syntax](guide/template-syntax) page.
### ng-click <button ng-click="vm.toggleImage()"> <button ng-click="vm.toggleImage($event)"> In AngularJS, the `ng-click` directive allows you to specify custom behavior when an element is clicked. In the first example, when the user clicks the button, the `toggleImage()` method in the	### Bind to the `click` event AngularJS event-based directives do not exist in Angular. Rather, define one-way binding from the template view to the component using **event binding**. For event binding, define the name of the target event within parenthesis and specify a template statement, in quotes, to the right of the equals. Angular then sets up an event handler for the target event. When the event is raised, the handler executes the template statement. In the first example, when a user clicks

controller referenced by the `vm` `controller as` alias is executed. The second example demonstrates passing in the `$event` object, which provides details about the event to the controller.	the button, the `toggleImage()` method in the associated component is executed. The second example demonstrates passing in the `$event` object, which provides details about the event to the component. For a list of DOM events, see: https://developer.mozilla.org/en-US/docs/Web/Events. For more information, see the [Event binding](guide/template-syntax#event-binding) section of the [Template Syntax](guide/template-syntax) page.
### ng-controller <div ng-controller="MovieListCtrl as vm"> In AngularJS, the `ng-controller` directive attaches a controller to the view. Using the `ng-controller` (or defining the controller as part of the routing) ties the view to the controller code associated with that view.	### Component decorator In Angular, the template no longer specifies its associated controller. Rather, the component specifies its associated template as part of the component class decorator. For more information, see [Architecture Overview](guide/architecture#components).
### ng-hide In AngularJS, the `ng-hide` directive shows or hides the associated HTML element based on an expression. For more information, see [ng-show](guide/ajs-quick-reference#ng-show).	### Bind to the `hidden` property In Angular, you use property binding; there is no built-in *hide* directive. For more information, see [ng-show](guide/ajs-quick-reference#ng-show).
### ng-href <a ng-href="{{ angularDocsUrl }}">Angular Docs</a> The `ng-	### Bind to the `href` property Angular uses property binding; there is no built-in *href* directive. Place the element's `href` property in square brackets and set it to a quoted template expression. For more information see the [Property binding]

<table>
<tr>
<td>href` directive allows AngularJS to preprocess the `href` property so that it can replace the binding expression with the appropriate URL before the browser fetches from that URL. In AngularJS, the `ng-href` is often used to activate a route as part of navigation. <a ng-href="#{{ moviesHash }}">Movies</a> Routing is handled differently in Angular.</td>
<td>(guide/template-syntax#property-binding) section of the [Template Syntax](guide/template-syntax) page. In Angular, `href` is no longer used for routing. Routing uses `routerLink`, as shown in the following example. For more information on routing, see the [RouterLink binding](guide/router#router-link) section of the [Routing & Navigation](guide/router) page.</td>
</tr>
<tr>
<td>### ng-if <table ng-if="movies.length"> In AngularJS, the `ng-if` directive removes or recreates a portion of the DOM, based on an expression. If the expression is false, the element is removed from the DOM. In this example, the `` element is removed from the DOM unless the `movies` array has a length greater than zero.
<table>
<tr>
<td>### *ngIf The `*ngIf` directive in Angular works the same as the `ng-if` directive in AngularJS. It removes or recreates a portion of the DOM based on an expression. In this example, the `` element is removed from the DOM unless the `movies` array has a length. The (*) before `ngIf` is required in this example. For more information, see [Structural Directives](guide/structural-directives). `) element repeats for each movie object in the collection of movies. ` in this example) and its contents into a template and uses that template to instantiate a view for each item in the list. Notice the other syntax differences: The (*) before `ngFor` is required; the `let` keyword identifies `movie` as an input variable; the list preposition is `of`, not `in`. For more information, see [Structural Directives](guide/structural-directives).
<table>
<tr>
<td></td>
<td>### ngModel In Angular, **two-way binding** is denoted by `[()]`, descriptively referred to as a "banana in a box". This syntax is a shortcut for defining both property binding (from</td>
</tr>
</table>
</td>
</tr>
</table>
</td>
<td></td>
</tr>
</table>

### ng-model <input ng-model="vm.favoriteHero"/> In AngularJS, the `ng-model` directive binds a form control to a property in the controller associated with the template. This provides **two-way binding**, whereby any change made to the value in the view is synchronized with the model, and any change to the model is synchronized with the value in the view.	the component to the view) and event binding (from the view to the component), thereby providing two-way binding. For more information on two-way binding with `ngModel`, see the [NgModel—Two-way binding to form elements with `[(ngModel)]`](../guide/template-syntax.html#ngModel) section of the [Template Syntax](guide/template-syntax) page.
### ng-repeat <tr ng-repeat="movie in vm.movies"> In AngularJS, the `ng-repeat` directive repeats the associated DOM element for each item in the specified collection. In this example, the table row (`	
### *ngFor The `*ngFor` directive in Angular is similar to the `ng-repeat` directive in AngularJS. It repeats the associated DOM element for each item in the specified collection. More accurately, it turns the defined element (`	
	### Bind to the `hidden` property Angular uses property binding; there is no built-in *show* directive. For hiding and showing elements, bind to the HTML `hidden`

### ng-show <h3 ng-show="vm.favoriteHero"> Your favorite hero is: {{vm.favoriteHero}} </h3> In AngularJS, the `ng-show` directive shows or hides the associated DOM element, based on an expression. In this example, the ` ` element is shown if the `favoriteHero` variable is truthy.	property. To conditionally display an element, place the element's `hidden` property in square brackets and set it to a quoted template expression that evaluates to the *opposite* of *show*. In this example, the ` ` element is hidden if the `favoriteHero` variable is not truthy. For more information on property binding, see the [Property binding](guide/template-syntax#property-binding) section of the [Template Syntax](guide/template-syntax) page.
### ng-src <img ng-src="{{movie.imageurl}}"> The `ng-src` directive allows AngularJS to preprocess the `src` property so that it can replace the binding expression with the appropriate URL before the browser fetches from that URL.	### Bind to the `src` property Angular uses property binding; there is no built-in *src* directive. Place the `src` property in square brackets and set it to a quoted template expression. For more information on property binding, see the [Property binding](guide/template-syntax#property-binding) section of the [Template Syntax]

	(guide/template-syntax) page.
### ng-style <div ng-style="{color: colorPreference}"> In AngularJS, the `ng-style` directive sets a CSS style on an HTML element based on an expression. That expression is often a key-value control object with each key of the object defined as a CSS property, and each value defined as an expression that evaluates to a value appropriate for the style. In the example, the `color` style is set to the current value of the `colorPreference` variable.	### ngStyle In Angular, the `ngStyle` directive works similarly. It sets a CSS style on an HTML element based on an expression. In the first example, the `color` style is set to the current value of the `colorPreference` variable. Angular also has **style binding**, which is good way to set a single style. This is shown in the second example. For more information on style binding, see the [Style binding](guide/template-syntax#style-binding) section of the [Template Syntax](guide/template-syntax) page. For more information on the `ngStyle` directive, see [NgStyle](guide/template-syntax#ngStyle) section of the [Template Syntax](guide/template-syntax) page.
	### ngSwitch In Angular, the `ngSwitch` directive works similarly. It displays an element whose `*ngSwitchCase` matches the current

### ng-switch <div ng-switch="vm.favoriteHero && vm.checkMovieHero(vm.favoriteHero)"> <div ng-switch-when="true"> Excellent choice! </div> <div ng-switch-when="false"> No movie, sorry! </div> <div ng-switch-default> Please enter your favorite hero. </div> </div> In AngularJS, the `ng-switch` directive swaps the contents of an element by selecting one of the templates based on the current value of an expression. In this example, if `favoriteHero` is not set, the template displays "Please enter ...". If `favoriteHero` is set, it checks the movie hero by calling a controller method. If that method returns `true`, the template displays "Excellent choice!". If that methods returns `false`, the template displays "No movie, sorry!".	`ngSwitch` expression value. In this example, if `favoriteHero` is not set, the `ngSwitch` value is `null` and `*ngSwitchDefault` displays, "Please enter ...". If `favoriteHero` is set, the app checks the movie hero by calling a component method. If that method returns `true`, the app selects `*ngSwitchCase="true"` and displays: "Excellent choice!" If that methods returns `false`, the app selects `*ngSwitchCase="false"` and displays: "No movie, sorry!" The (*) before `ngSwitchCase` and `ngSwitchDefault` is required in this example. For more information, see [The NgSwitch directives](guide/template-syntax#ngSwitch) section of the [Template Syntax](guide/template-syntax) page.

{@a filters-pipes}

Filters/pipes

Angular **pipes** provide formatting and transformation for data in the template, similar to AngularJS **filters**. Many of the built-in filters in

AngularJS have corresponding pipes in Angular. For more information on pipes, see Pipes.

AngularJS	Angular
### currency <td>{{movie.price | currency}}</td> Formats a number as currency.	### currency The Angular `currency` pipe is similar although some of the parameters have changed.
### date <td> {{movie.releaseDate | date}} </td> Formats a date to a string based on the requested format.	### date The Angular `date` pipe is similar.
### filter <tr ng-repeat="movie in movieList | filter: {title:listFilter}"> Selects a subset of items from the defined collection, based on the filter criteria.	### none For performance reasons, no comparable pipe exists in Angular. Do all your filtering in the component. If you need the same filtering code in several templates, consider building a custom pipe.
### json <pre>{{movie | json}} </pre> Converts a JavaScript object into a JSON string. This is useful for debugging.	### json The Angular `json` pipe does the same thing.
### limitTo <tr ng-repeat="movie in movieList | limitTo:2:0"> Selects up to the first parameter (2) number of items from the collection starting (optionally) at the beginning index (0).	### slice The `SlicePipe` does the same thing but the *order of the parameters is reversed*, in keeping with the JavaScript `Slice` method. The first parameter is the starting index; the second is the limit. As in AngularJS, coding this operation within the component instead could improve performance.
### lowercase <div> {{movie.title | lowercase}}</div>	### lowercase The Angular `lowercase` pipe does the same

Converts the string to lowercase.	thing.
### number <td> {{movie.starRating | number}} </td> Formats a number as text.	### number The Angular `number` pipe is similar. It provides more functionality when defining the decimal places, as shown in the second example above. Angular also has a `percent` pipe, which formats a number as a local percentage as shown in the third example.
### orderBy <tr ng-repeat="movie in movieList | orderBy : 'title'"> Displays the collection in the order specified by the expression. In this example, the movie title orders the `movieList`.	### none For performance reasons, no comparable pipe exists in Angular. Instead, use component code to order or sort results. If you need the same ordering or sorting code in several templates, consider building a custom pipe.

{@a controllers-components}

Modules/controllers/components

In both AngularJS and Angular, modules help you organize your application into cohesive blocks of functionality.

In AngularJS, you write the code that provides the model and the methods for the view in a **controller**. In Angular, you build a **component**.

Because much AngularJS code is in JavaScript, JavaScript code is shown in the AngularJS column. The Angular code is shown using TypeScript.

AngularJS	Angular
### IIFE (function () { ... }()); In	### none This is a nonissue in Angular because ES 2015

AngularJS, an immediately invoked function expression (or IIFE) around controller code keeps it out of the global namespace.	modules handle the namespacing for you. For more information on modules, see the [Modules](guide/architecture#modules) section of the [Architecture Overview](guide/architecture).
### Angular modules angular.module("movieHunter", ["ngRoute"]); In AngularJS, an Angular module keeps track of controllers, services, and other code. The second argument defines the list of other modules that this module depends upon.	### NgModules NgModules, defined with the `NgModule` decorator, serve the same purpose: * `imports`: specifies the list of other modules that this module depends upon * `declaration`: keeps track of your components, pipes, and directives. For more information on modules, see [NgModules](guide/ngmodule).
### Controller registration angular .module("movieHunter") .controller("MovieListCtrl", ["movieService", MovieListCtrl]); AngularJS has code in each controller that looks up an appropriate Angular module and registers the controller with that module. The first argument is the controller name. The second argument defines the string names of all dependencies injected into this controller, and a reference to the controller function.	### Component decorator Angular adds a decorator to the component class to provide any required metadata. The `@Component` decorator declares that the class is a component and provides metadata about that component such as its selector (or tag) and its template. This is how you associate a template with logic, which is defined in the component class. For more information, see the [Components](guide/architecture#components) section of the [Architecture Overview](guide/architecture) page.
	### Component class In

### Controller function function MovieListCtrl(movieService) { } In AngularJS, you write the code for the model and methods in a controller function.	Angular, you create a component class. NOTE: If you are using TypeScript with AngularJS, you must use the `export` keyword to export the component class. For more information, see the [Components](guide/architecture#components) section of the [Architecture Overview](guide/architecture) page.
### Dependency injection MovieListCtrl.$inject = ['MovieService']; function MovieListCtrl(movieService) { } In AngularJS, you pass in any dependencies as controller function arguments. This example injects a `MovieService`. To guard against minification problems, tell Angular explicitly that it should inject an instance of the `MovieService` in the first parameter.	### Dependency injection In Angular, you pass in dependencies as arguments to the component class constructor. This example injects a `MovieService`. The first parameter's TypeScript type tells Angular what to inject, even after minification. For more information, see the [Dependency injection](guide/architecture#dependency-injection) section of the [Architecture Overview](guide/architecture).

{@a style-sheets}

Style sheets

Style sheets give your application a nice look. In AngularJS, you specify the style sheets for your entire application. As the application grows over time, the styles for the many parts of the application merge, which can cause unexpected results. In Angular, you can still define style sheets for your entire application. But now you can also encapsulate a style sheet within a specific component.

AngularJS	Angular
### Link tag <link href="styles.css" rel="stylesheet" /> AngularJS, uses a `link` tag in the head section of the `index.html` file to define the styles for the application.	### Styles configuration With the Angular CLI, you can configure your global styles in the `.angular-cli.json` file. You can rename the extension to `.scss` to use sass. ### StyleUrls In Angular, you can use the `styles` or `styleUrls` property of the `@Component` metadata to define a style sheet for a particular component. This allows you to set appropriate styles for individual components that won’t leak into other parts of the application.

Animations

Motion is an important aspect in the design of modern web applications. Good user interfaces transition smoothly between states with engaging animations that call attention where it's needed. Well-designed animations can make a UI not only more fun but also easier to use.

Overview

Angular's animation system lets you build animations that run with the same kind of native performance found in pure CSS animations. You can also tightly integrate your animation logic with the rest of your application code, for ease of control.

Angular animations are built on top of the standard [Web Animations API](https://w3c.github.io/web-animations/) and run natively on [browsers that support it](http://caniuse.com/#feat=web-animation). For other browsers, a polyfill is required. Uncomment the `web-animations-js` polyfill from the `polyfills.ts` file.
The examples in this page are available as a .

Setup

Before you can add animations to your application, you need to import a few animation-specific modules and functions to the root application module.

Example basics

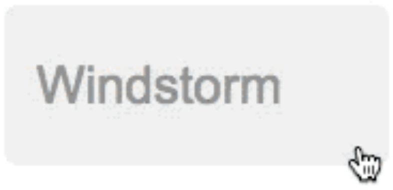
The animations examples in this guide animate a list of heroes.

A `Hero` class has a `name` property, a `state` property that indicates if the hero is active or not, and a `toggleState()` method to switch between the states.

Across the top of the screen (`app.hero-team-builder.component.ts`) are a series of buttons that add and remove heroes from the list (via the `HeroService`). The buttons trigger changes to the list that all of the example components see at the same time.

{@a example-transitioning-between-states}

Transitioning between two states



Windstorm

You can build a simple animation that transitions an element between two states driven by a model attribute.

Animations can be defined inside `@Component` metadata.

With these, you can define an *animation trigger* called `heroState` in the component metadata. It uses animations to transition between two states: `active` and `inactive`. When a hero is active, the element appears in a slightly larger size and lighter color.

In this example, you are defining animation styles (color and transform) inline in the animation metadata.

Now, using the `[@triggerName]` syntax, attach the animation that you just defined to one or more elements in the component's template.

Here, the animation trigger applies to every element repeated by an `ngFor`. Each of the repeated elements animates independently. The value of the attribute is bound to the expression `hero.state` and is always either `active` or `inactive`.

With this setup, an animated transition appears whenever a hero object changes state. Here's the full component implementation:

States and transitions

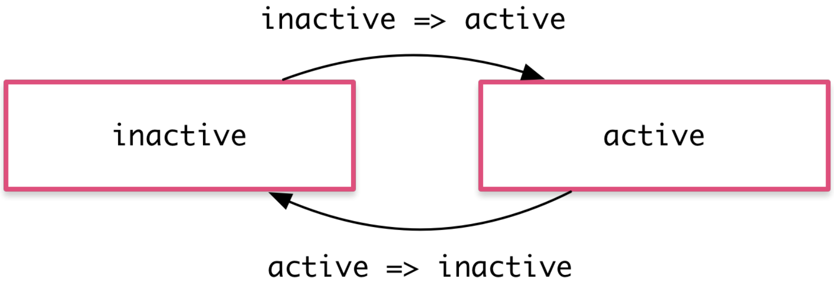
Angular animations are defined as logical **states** and **transitions** between states.

An animation state is a string value that you define in your application code. In the example above, the states `'active'` and `'inactive'` are based on the logical state of hero objects. The source of the state can be a simple object attribute, as it was in this case, or it can be a value computed in a method. The important thing is that you can read it into the component's template.

You can define *styles* for each animation state:

These `state` definitions specify the *end styles* of each state. They are applied to the element once it has transitioned to that state, and stay *as long as it remains in that state*. In effect, you're defining what styles the element has in different states.

After you define states, you can define *transitions* between the states. Each transition controls the timing of switching between one set of styles and the next:



If several transitions have the same timing configuration, you can combine them into the same `transition` definition:

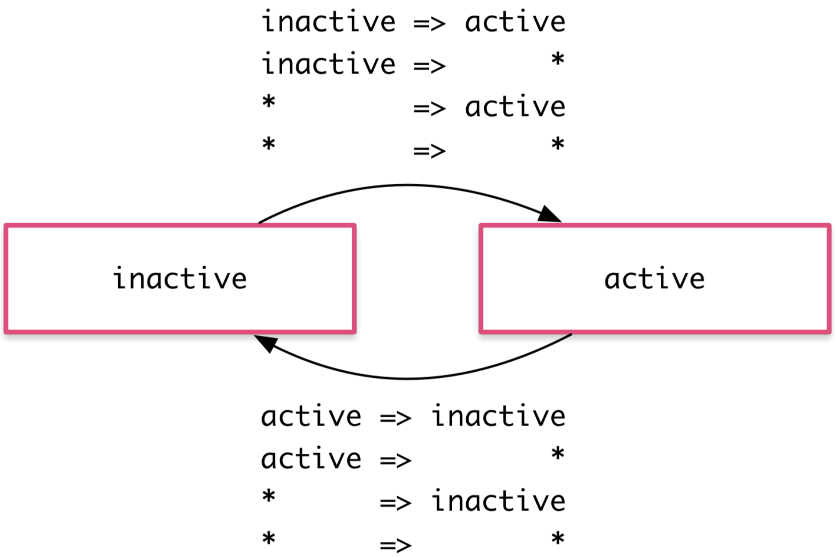
When both directions of a transition have the same timing, as in the previous example, you can use the shorthand syntax `<=>`:

You can also apply a style during an animation but not keep it around after the animation finishes. You can define such styles inline, in the `transition`. In this example, the element receives one set of styles immediately and is then animated to the next. When the transition finishes, none of these styles are kept because they're not defined in a `state`.

The wildcard state `*`

The `*` ("wildcard") state matches *any* animation state. This is useful for defining styles and transitions that apply regardless of which state the animation is in. For example:

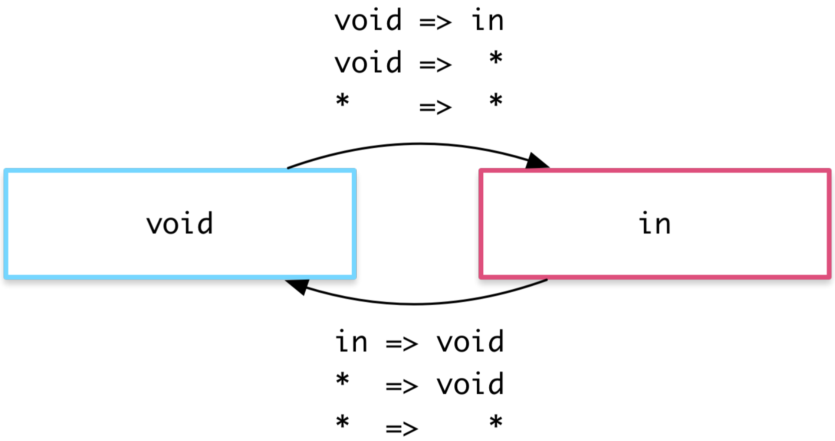
- The `active => *` transition applies when the element's state changes from `active` to anything else.
- The `* => *` transition applies when *any* change between two states takes place.



The `void` state

The special state called `void` can apply to any animation. It applies when the element is *not* attached to a view, perhaps because it has not yet been added or because it has been removed. The `void` state is useful for defining enter and leave animations.

For example the `* => void` transition applies when the element leaves the view, regardless of what state it was in before it left.



The wildcard state `*` also matches `void`.

Example: Entering and leaving

Using the `void` and `*` states you can define transitions that animate the entering and leaving of elements:

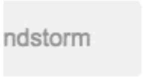
- Enter: `void => *`
- Leave: `* => void`

For example, in the `animations` array below there are two transitions that use the `void => *` and `* => void` syntax to animate the element in and out of the view.

Note that in this case the styles are applied to the void state directly in the transition definitions, and not in a separate `state(void)` definition. Thus, the transforms are different on enter and leave: the element enters from the left and leaves to the right.

These two common animations have their own aliases: transition(':enter', [...]); // void => * transition(':leave', [...]); // * => void

Example: Entering and leaving from different states

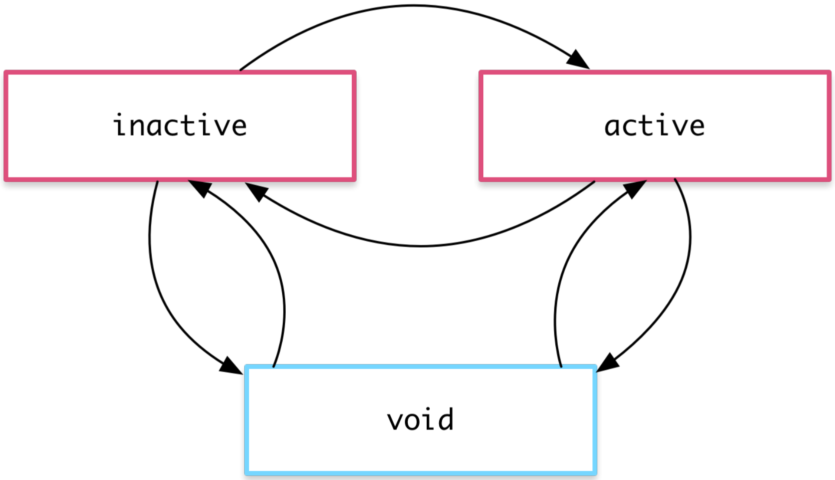


You can also combine this animation with the earlier state transition animation by using the hero state as the animation state. This lets you configure different transitions for entering and leaving based on what the state of the hero is:

- Inactive hero enter: `void => inactive`
- Active hero enter: `void => active`
- Inactive hero leave: `inactive => void`

- Active hero leave: `active => void`

This gives you fine-grained control over each transition:



Animatable properties and units

Since Angular's animation support builds on top of Web Animations, you can animate any property that the browser considers *animatable*. This includes positions, sizes, transforms, colors, borders, and many others. The W3C maintains a list of animatable properties on its CSS Transitions page.

For positional properties that have a numeric value, you can define a unit by providing the value as a string with the appropriate suffix:

- `'50px'`
- `'3em'`
- `'100%'`

If you don't provide a unit when specifying dimension, Angular assumes the default of `px` :

- `50` is the same as saying `'50px'`

Automatic property calculation

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 tricky to animate with CSS.

In these cases, you can use a special `*` property value so that the value of the property is computed at runtime and then plugged into the animation.

In this example, the leave animation takes whatever height the element has before it leaves and animates from that height to zero:

Animation timing

There are three timing properties you can tune for every animated transition: the duration, the delay, and the easing function. They are all combined into a single transition *timing string*.

Duration

The duration controls how long the animation takes to run from start to finish. You can define a duration in three ways:

- As a plain number, in milliseconds: `100`
- In a string, as milliseconds: `'100ms'`
- In a string, as seconds: `'0.1s'`

Delay

The delay controls the length of time between the animation trigger and the beginning of the transition. You can define one by adding it to the same string following the duration. It also has the same format options as the duration:

- Wait for 100ms and then run for 200ms: `'0.2s 100ms'`

Easing

The easing function controls how the animation accelerates and decelerates during its runtime. For example, an `ease-in` function causes the animation to begin relatively slowly but pick up speed as it progresses. You can control the easing by adding it as a *third* value in the string after the duration and the delay (or as the *second* value when there is no delay):

- Wait for 100ms and then run for 200ms, with easing: `'0.2s 100ms ease-out'`
- Run for 200ms, with easing: `'0.2s ease-in-out'`

Example

Here are a couple of custom timings in action. Both enter and leave last for 200 milliseconds, that is `0.2s`, but they have different easings. The leave begins after a slight delay of 10 milliseconds as specified in `'0.2s 10 ease-out'`:

Multi-step animations with keyframes

Animation *keyframes* go beyond a simple transition to a more intricate animation that goes through one or more intermediate styles when transitioning between two sets of styles.

For each keyframe, you specify an *offset* that defines at which point in the animation that keyframe applies. The offset is a number between zero, which marks the beginning of the animation, and one, which marks the end.

This example adds some "bounce" to the enter and leave animations with keyframes:

Note that the offsets are *not* defined in terms of absolute time. They are relative measures from zero to one. The final timeline of the animation is based on the combination of keyframe offsets, duration, delay, and easing.

Defining offsets for keyframes is optional. If you omit them, offsets with even spacing are automatically assigned. For example, three keyframes without predefined offsets receive offsets `0`, `0.5`, and `1`.

Parallel animation groups

You've seen how to animate multiple style properties at the same time: just put all of them into the same `style()` definition.

But you may also want to configure different *timings* for animations that happen in parallel. For example, you may want to animate two CSS properties but use a different easing function for each one.

For this you can use animation *groups*. In this example, using groups both on enter and leave allows for two different timing configurations. Both are applied to the same element in parallel, but run independently of each other:

One group animates the element transform and width; the other group animates the opacity.

Animation callbacks

A callback is fired when an animation is started and also when it is done.

In the keyframes example, you have a `trigger` called `@flyInOut`. You can hook those callbacks like this:

The callbacks receive an `AnimationEvent` that contains useful properties such as `fromState`, `toState` and `totalTime`.

Those callbacks will fire whether or not an animation is picked up.

The Ahead-of-Time (AOT) Compiler

The Angular Ahead-of-Time (AOT) compiler converts your Angular HTML and TypeScript code into efficient JavaScript code during the build phase *before* the browser downloads and runs that code.

This guide explains how to build with the AOT compiler and how to write Angular metadata that AOT can compile.

Watch compiler author Tobias Bosch explain the Angular Compiler at AngularConnect 2016.

{@a overview}

Angular compilation

An Angular application consists largely of components and their HTML templates. Before the browser can render the application, the components and templates must be converted to executable JavaScript by an *Angular compiler*.

Angular offers two ways to compile your application:

1. ***Just-in-Time*** **(JIT)**, which compiles your app in the browser at runtime
2. ***Ahead-of-Time*** **(AOT)**, which compiles your app at build time.

JIT compilation is the default when you run the *build-only* or the *build-and-serve-locally* CLI commands:

ng build ng serve

{@a compile}

For AOT compilation, append the `--aot` flags to the *build-only* or the *build-and-serve-locally* CLI commands:

ng build --aot ng serve --aot

The `--prod` meta-flag compiles with AOT by default. See the [CLI documentation](https://github.com/angular/angular-cli/wiki) for details, especially the [`build` topic](https://github.com/angular/angular-cli/wiki/build).

{@a why-aot}

Why compile with AOT?

Faster rendering

With AOT, the browser downloads a pre-compiled version of the application. The browser loads executable code so it can render the application immediately, without waiting to compile the app first.

Fewer asynchronous requests

The compiler *inlines* external HTML templates and CSS style sheets within the application JavaScript, eliminating separate ajax requests for those source files.

Smaller Angular framework download size

There's no need to download the Angular compiler if the app is already compiled. The compiler is roughly half of Angular itself, so omitting it dramatically reduces the application payload.

Detect template errors earlier

The AOT compiler detects and reports template binding errors during the build step before users can see them.

Better security

AOT compiles HTML templates and components into JavaScript files long before they are served to the client. With no templates to read and no risky client-side HTML or JavaScript evaluation, there are fewer opportunities for injection attacks.

Angular Metadata and AOT

The Angular **AOT compiler** extracts and interprets **metadata** about the parts of the application that Angular is supposed to manage.

Angular metadata tells Angular how to construct instances of your application classes and interact with them at runtime.

You specify the metadata with **decorators** such as `@Component()` and `@Input()`. You also specify metadata implicitly in the constructor declarations of these decorated classes.

In the following example, the `@Component()` metadata object and the class constructor tell Angular how to create and display an instance of `TypicalComponent`.

```
@Component({
  selector: 'app-typical',
  template: '<div>A typical component for {{data.name}}</div>'
)}
export class TypicalComponent {
  @Input() data: TypicalData;
  constructor(private someService: SomeService) { ... }
}
```

The Angular compiler extracts the metadata *once* and generates a *factory* for `TypicalComponent`. When it needs to create a `TypicalComponent` instance, Angular calls the factory, which produces a new visual element, bound to a new instance of the component class with its injected dependency.

Metadata restrictions

You write metadata in a *subset* of TypeScript that must conform to the following general constraints:

1. Limit expression syntax to the supported subset of JavaScript.
2. Only reference exported symbols after code folding.
3. Only call functions supported by the compiler.
4. Decorated and data-bound class members must be public.

The next sections elaborate on these points.

How AOT works

It helps to think of the AOT compiler as having two phases: a code analysis phase in which it simply records a representation of the source; and a code generation phase in which the compiler's `StaticReflector` handles the interpretation as well as places restrictions on what it interprets.

Phase 1: analysis

The TypeScript compiler does some of the analytic work of the first phase. It emits the `.d.ts` *type definition files* with type information that the AOT compiler needs to generate application code.

At the same time, the AOT ***collector*** analyzes the metadata recorded in the Angular decorators and outputs metadata information in **`.metadata.json`** files, one per `.d.ts` file.

You can think of `.metadata.json` as a diagram of the overall structure of a decorator's metadata, represented as an abstract syntax tree (AST).

Angular's [schema.ts](https://github.com/angular/angular/blob/master/packages/compiler-cli/src/metadata/schema.ts) describes the JSON format as a collection of TypeScript interfaces.

{@a expression-syntax}

Expression syntax

The *collector* only understands a subset of JavaScript. Define metadata objects with the following limited syntax:

Syntax	Example
Literal object	`{cherry: true, apple: true, mincemeat: false}`
Literal array	`['cherries', 'flour', 'sugar']`
Spread in literal array	`['apples', 'flour', ...the_rest]`
Calls	`bake(ingredients)`
New	`new Oven()`
Property access	`pie.slice`
Array index	`ingredients[0]`
Identifier reference	`Component`
A template string	`` `pie is ${multiplier} times better than cake` ``
Literal string	`'pi'`
Literal number	`3.14153265`
Literal boolean	`true`
Literal null	`null`
Supported prefix operator	`!cake`
Supported Binary operator	`a + b`
Conditional operator	`a ? b : c`
Parentheses	`(a + b)`

If an expression uses unsupported syntax, the *collector* writes an error node to the `.metadata.json` file. The compiler later reports the error if it needs that piece of metadata to generate the application code.

If you want `ngc` to report syntax errors immediately rather than produce a `.metadata.json` file with errors, set the `strictMetadataEmit` option in `tsconfig`. ``` "angularCompilerOptions": { ... "strictMetadataEmit" : true } ``` Angular libraries have this option to ensure that all Angular `.metadata.json` files are clean and it is a best practice to do the same when building your own libraries.

{@a function-expression} {@a arrow-functions}

No arrow functions

The AOT compiler does not support function expressions and arrow functions, also called *lambda* functions.

Consider the following component decorator:

```
@Component({
  ...
  providers: [{provide: server, useFactory: () => new Server()}]
})
```

The AOT *collector* does not support the arrow function, `() => new Server()`, in a metadata expression. It generates an error node in place of the function.

When the compiler later interprets this node, it reports an error that invites you to turn the arrow function into an *exported function*.

You can fix the error by converting to this:

```
export function serverFactory() {
  return new Server();
}

@Component({
  ...
  providers: [{provide: server, useFactory: serverFactory}]
})
```

Beginning in version 5, the compiler automatically performs this rewritting while emitting the `.js` file.

Limited function calls

The *collector* can represent a function call or object creation with `new` as long as the syntax is valid. The *collector* only cares about proper syntax.

But beware. The compiler may later refuse to generate a call to a *particular* function or creation of a *particular* object. The compiler only supports calls to a small set of functions and will use `new` for only a few designated classes. These functions and classes are in a table of below.

Folding

{@a exported-symbols} The compiler can only resolve references to ***exported*** symbols. Fortunately, the *collector* enables limited use of non-exported symbols through *folding*.

The *collector* may be able to evaluate an expression during collection and record the result in the `.metadata.json` instead of the original expression.

For example, the *collector* can evaluate the expression `1 + 2 + 3 + 4` and replace it with the result, `10`.

This process is called *folding*. An expression that can be reduced in this manner is *foldable*.

{@a var-declaration} The collector can evaluate references to module-local `const` declarations and initialized `var` and `let` declarations, effectively removing them from the `.metadata.json` file.

Consider the following component definition:

```
const template = '<div>{{hero.name}}</div>';

@Component({
  selector: 'app-hero',
  template: template
})
export class HeroComponent {
  @Input() hero: Hero;
}
```

The compiler could not refer to the `template` constant because it isn't exported.

But the *collector* can *fold* the `template` constant into the metadata definition by inlining its contents. The effect is the same as if you had written:

```
@Component({
  selector: 'app-hero',
  template: '<div>{{hero.name}}</div>'
})
export class HeroComponent {
  @Input() hero: Hero;
}
```

There is no longer a reference to `template` and, therefore, nothing to trouble the compiler when it later interprets the *collector's* output in `.metadata.json`.

You can take this example a step further by including the `template` constant in another expression:

```
const template = '<div>{{hero.name}}</div>';

@Component({
  selector: 'app-hero',
  template: template + '<div>{{hero.title}}</div>'
})
export class HeroComponent {
  @Input() hero: Hero;
}
```

The *collector* reduces this expression to its equivalent *folded* string:

`'<div>{{hero.name}}</div><div>{{hero.title}}</div>'`.

Foldable syntax

The following table describes which expressions the *collector* can and cannot fold:

Syntax	Foldable
Literal object	yes
Literal array	yes
Spread in literal array	no
Calls	no
New	no
Property access	yes, if target is foldable
Array index	yes, if target and index are foldable
Identifier reference	yes, if it is a reference to a local
A template with no substitutions	yes
A template with substitutions	yes, if the substitutions are foldable
Literal string	yes
Literal number	yes
Literal boolean	yes
Literal null	yes
Supported prefix operator	yes, if operand is foldable
Supported binary operator	yes, if both left and right are foldable
Conditional operator	yes, if condition is foldable
Parentheses	yes, if the expression is foldable

If an expression is not foldable, the collector writes it to `.metadata.json` as an AST for the compiler to resolve.

Phase 2: code generation

The *collector* makes no attempt to understand the metadata that it collects and outputs to `.metadata.json`. It represents the metadata as best it can and records errors when it detects a metadata syntax violation.

It's the compiler's job to interpret the `.metadata.json` in the code generation phase.

The compiler understands all syntax forms that the *collector* supports, but it may reject *syntactically* correct metadata if the *semantics* violate compiler rules.

The compiler can only reference *exported symbols*.

Decorated component class members must be public. You cannot make an `@Input()` property private or internal.

Data bound properties must also be public.

```
// BAD CODE - title is private
@Component({
  selector: 'app-root',
  template: '<h1>{{title}}</h1>'
})
export class AppComponent {
  private title = 'My App'; // Bad
}
```

{@a supported-functions} Most importantly, the compiler only generates code to create instances of certain classes, support certain decorators, and call certain functions from the following lists.

New instances

The compiler only allows metadata that create instances of the class `InjectionToken` from `@angular/core`.

Annotations/Decorators

The compiler only supports metadata for these Angular decorators.

Decorator	Module
`Attribute`	`@angular/core`
`Component`	`@angular/core`
`ContentChild`	`@angular/core`
`ContentChildren`	`@angular/core`
`Directive`	`@angular/core`
`Host`	`@angular/core`
`HostBinding`	`@angular/core`
`HostListener`	`@angular/core`
`Inject`	`@angular/core`
`Injectable`	`@angular/core`
`Input`	`@angular/core`
`NgModule`	`@angular/core`
`Optional`	`@angular/core`
`Output`	`@angular/core`
`Pipe`	`@angular/core`
`Self`	`@angular/core`
`SkipSelf`	`@angular/core`
`ViewChild`	`@angular/core`

Macro-functions and macro-static methods

The compiler also supports *macros* in the form of functions or static methods that return an expression.

For example, consider the following function:

```
export function wrapInArray<T>(value: T): T[] {
  return [value];
}
```

You can call the `wrapInArray` in a metadata definition because it returns the value of an expression that conforms to the compiler's restrictive JavaScript subset.

You might use `wrapInArray()` like this:

```
@NgModule({
  declarations: wrapInArray(TypicalComponent)
})
export class TypicalModule {}
```

The compiler treats this usage as if you had written:

```
@NgModule({
  declarations: [TypicalComponent]
})
export class TypicalModule {}
```

The collector is simplistic in its determination of what qualifies as a macro function; it can only contain a single `return` statement.

The Angular `RouterModule` exports two macro static methods, `forRoot` and `forChild`, to help declare root and child routes. Review the source code for these methods to see how macros can simplify configuration of complex NgModules.

Metadata rewriting

The compiler treats object literals containing the fields `useClass` , `useValue` , `useFactory` , and `data` specially. The compiler converts the expression initializing one of these fields into an exported variable, which replaces the expression. This process of rewriting these expressions removes all the restrictions on what can be in them because the compiler doesn't need to know the expression's value—it just needs to be able to generate a reference to the value.

You might write something like:

```
class TypicalServer {

}

@NgModule({
  providers: [{provide: SERVER, useFactory: () => TypicalServer}]
})
export class TypicalModule {}
```

Without rewriting, this would be invalid because lambdas are not supported and `TypicalServer` is not exported.

To allow this, the compiler automatically rewrites this to something like:

```
class TypicalServer {

}

export const ɵ0 = () => new TypicalServer();

@NgModule({
  providers: [{provide: SERVER, useFactory: ɵ0}]
})
export class TypicalModule {}
```

This allows the compiler to generate a reference to `ɵ0` in the factory without having to know what the value of `ɵ0` contains.

The compiler does the rewriting during the emit of the `.js` file. This doesn't rewrite the `.d.ts` file, however, so TypeScript doesn't recognize it as being an export. Thus, it does not pollute the ES module's exported API.

Metadata Errors

The following are metadata errors you may encounter, with explanations and suggested corrections.

Expression form not supported
Reference to a local (non-exported) symbol
Only initialized variables and constants
Reference to a non-exported class
Reference to a non-exported function
Function calls are not supported
Destructured variable or constant not supported
Could not resolve type
Name expected
Unsupported enum member name
Tagged template expressions are not supported
Symbol reference expected

Expression form not supported

The compiler encountered an expression it didn't understand while evalutating Angular metadata.

Language features outside of the compiler's restricted expression syntax can produce this error, as seen in the following example:

```
// ERROR
export class Fooish { ... }
...
const prop = typeof Fooish; // typeof is not valid in metadata
  ...
  // bracket notation is not valid in metadata
  { provide: 'token', useValue: { [prop]: 'value' } };
  ...
```

You can use `typeof` and bracket notation in normal application code. You just can't use those features within expressions that define Angular metadata.

Avoid this error by sticking to the compiler's restricted expression syntax when writing Angular metadata and be wary of new or unusual TypeScript features.

{@a reference-to-a-local-symbol}

Reference to a local (non-exported) symbol

Reference to a local (non-exported) symbol 'symbol name'. Consider exporting the symbol.

The compiler encountered a referenced to a locally defined symbol that either wasn't exported or wasn't initialized.

Here's a `provider` example of the problem.

```
// ERROR
let foo: number; // neither exported nor initialized

@Component({
  selector: 'my-component',
  template: ... ,
  providers: [
    { provide: Foo, useValue: foo }
  ]
})
export class MyComponent {}
```

The compiler generates the component factory, which includes the `useValue` provider code, in a separate module. *That* factory module can't reach back to *this* source module to access the local (non-exported) `foo` variable.

You could fix the problem by initializing `foo`.

```
let foo = 42; // initialized
```

The compiler will fold the expression into the provider as if you had written this.

```
  providers: [
    { provide: Foo, useValue: 42 }
  ]
```

Alternatively, you can fix it by exporting `foo` with the expectation that `foo` will be assigned at runtime when you actually know its value.

```
// CORRECTED
export let foo: number; // exported

@Component({
  selector: 'my-component',
  template: ... ,
  providers: [
    { provide: Foo, useValue: foo }
  ]
})
export class MyComponent {}
```

Adding `export` often works for variables referenced in metadata such as `providers` and `animations` because the compiler can generate *references* to the exported variables in these expressions. It doesn't need the *values* of those variables.

Adding `export` doesn't work when the compiler needs the *actual value* in order to generate code. For example, it doesn't work for the `template` property.

```
// ERROR
export let someTemplate: string; // exported but not initialized

@Component({
  selector: 'my-component',
  template: someTemplate
})
export class MyComponent {}
```

The compiler needs the value of the `template` property *right now* to generate the component factory. The variable reference alone is insufficient. Prefixing the declaration with `export` merely produces a new error, "`Only initialized variables and constants can be referenced`".

{@a only-initialized-variables}

Only initialized variables and constants

Only initialized variables and constants can be referenced because the value of this variable is needed by the template compiler.

The compiler found a reference to an exported variable or static field that wasn't initialized. It needs the value of that variable to generate code.

The following example tries to set the component's `template` property to the value of the exported `someTemplate` variable which is declared but *unassigned*.

```
// ERROR
export let someTemplate: string;

@Component({
  selector: 'my-component',
  template: someTemplate
})
export class MyComponent {}
```

You'd also get this error if you imported `someTemplate` from some other module and neglected to initialize it there.

```
// ERROR - not initialized there either
import { someTemplate } from './config';

@Component({
  selector: 'my-component',
  template: someTemplate
})
export class MyComponent {}
```

The compiler cannot wait until runtime to get the template information. It must statically derive the value of the `someTemplate` variable from the source code so that it can generate the component factory, which includes instructions for building the element based on the template.

To correct this error, provide the initial value of the variable in an initializer clause *on the same line*.

```
// CORRECTED
export let someTemplate = '<h1>Greetings from Angular</h1>';

@Component({
  selector: 'my-component',
  template: someTemplate
})
export class MyComponent {}
```

Reference to a non-exported class

Reference to a non-exported class . Consider exporting the class.

Metadata referenced a class that wasn't exported.

For example, you may have defined a class and used it as an injection token in a providers array but neglected to export that class.

```
// ERROR
abstract class MyStrategy { }

  ...
  providers: [
    { provide: MyStrategy, useValue: ... }
  ]
  ...
```

Angular generates a class factory in a separate module and that factory can only access exported classes. To correct this error, export the referenced class.

```
// CORRECTED
export abstract class MyStrategy { }

  ...
  providers: [
    { provide: MyStrategy, useValue: ... }
  ]
  ...
```

Reference to a non-exported function

Metadata referenced a function that wasn't exported.

For example, you may have set a providers `useFactory` property to a locally defined function that you neglected to export.

```
// ERROR
function myStrategy() { ... }

  ...
  providers: [
    { provide: MyStrategy, useFactory: myStrategy }
  ]
  ...
```

Angular generates a class factory in a separate module and that factory can only access exported functions. To correct this error, export the function.

```
// CORRECTED
export function myStrategy() { ... }

  ...
  providers: [
    { provide: MyStrategy, useFactory: myStrategy }
  ]
  ...
```

{@a function-calls-not-supported}

Function calls are not supported

Function calls are not supported. Consider replacing the function or lambda with a reference to an exported function.

The compiler does not currently support function expressions or lambda functions. For example, you cannot set a provider's `useFactory` to an anonymous function or arrow function like this.

```
// ERROR
  ...
  providers: [
    { provide: MyStrategy, useFactory: function() { ... } },
    { provide: OtherStrategy, useFactory: () => { ... } }
  ]
  ...
```

You also get this error if you call a function or method in a provider's `useValue` . ``` // ERROR import { calculateValue } from './utilities';

... providers: [{ provide: SomeValue, useValue: calculateValue() }] ... ```

To correct this error, export a function from the module and refer to the function in a `useFactory` provider instead.

// CORRECTED import { calculateValue } from './utilities';

export function myStrategy() { ... } export function otherStrategy() { ... } export function someValueFactory() { return calculateValue(); } ... providers: [{ provide: MyStrategy, useFactory: myStrategy }, { provide: OtherStrategy, useFactory: otherStrategy }, { provide: SomeValue, useFactory: someValueFactory }] ...

{@a destructured-variable-not-supported}

Destructured variable or constant not supported

Referencing an exported destructured variable or constant is not supported by the template compiler. Consider simplifying this to avoid destructuring.

The compiler does not support references to variables assigned by destructuring.

For example, you cannot write something like this:

// ERROR import { configuration } from './configuration';

// destructured assignment to foo and bar const {foo, bar} = configuration; ... providers: [{provide: Foo, useValue: foo}, {provide: Bar, useValue: bar},] ...

To correct this error, refer to non-destructured values.

// CORRECTED import { configuration } from './configuration'; ... providers: [{provide: Foo, useValue: configuration.foo}, {provide: Bar, useValue: configuration.bar},] ...

Could not resolve type

The compiler encountered a type and can't determine which module exports that type.

This can happen if you refer to an ambient type. For example, the `Window` type is an ambiant type declared in the global `.d.ts` file.

You'll get an error if you reference it in the component constructor, which the compiler must statically analyze.

```
// ERROR
@Component({ })
export class MyComponent {
  constructor (private win: Window) { ... }
}
```

TypeScript understands ambiant types so you don't import them. The Angular compiler does not understand a type that you neglect to export or import.

In this case, the compiler doesn't understand how to inject something with the `Window` token.

Do not refer to ambient types in metadata expressions.

If you must inject an instance of an ambiant type, you can finesse the problem in four steps:

1. Create an injection token for an instance of the ambiant type.
2. Create a factory function that returns that instance.
3. Add a `useFactory` provider with that factory function.
4. Use `@Inject` to inject the instance.

Here's an illustrative example.

// CORRECTED import { Inject } from '@angular/core';

export const WINDOW = new InjectionToken('Window'); export function _window() { return window; }

@Component({ ... providers: [{ provide: WINDOW, useFactory: _window }] }) export class MyComponent { constructor (@Inject(WINDOW) private win: Window) { ... } }

The `Window` type in the constructor is no longer a problem for the compiler because it uses the `@Inject(WINDOW)` to generate the injection code.

Angular does something similar with the `DOCUMENT` token so you can inject the browser's `document` object (or an abstraction of it, depending upon the platform in which the application runs).

import { Inject } from '@angular/core'; import { DOCUMENT } from '@angular/platform-browser';

@Component({ ... }) export class MyComponent { constructor (@Inject(DOCUMENT) private doc: Document) { ... } }

Name expected

The compiler expected a name in an expression it was evaluating. This can happen if you use a number as a property name as in the following example.

```
// ERROR
provider: [{ provide: Foo, useValue: { 0: 'test' } }]
```

Change the name of the property to something non-numeric.

```
// CORRECTED
provider: [{ provide: Foo, useValue: { '0': 'test' } }]
```

Unsupported enum member name

Angular couldn't determine the value of the enum member that you referenced in metadata.

The compiler can understand simple enum values but not complex values such as those derived from computed properties.

// ERROR enum Colors { Red = 1, White, Blue = "Blue".length // computed }

... providers: [{ provide: BaseColor, useValue: Colors.White } // ok { provide: DangerColor, useValue: Colors.Red } // ok { provide: StrongColor, useValue: Colors.Blue } // bad] ...

Avoid referring to enums with complicated initializers or computed properties.

{@a tagged-template-expressions-not-supported}

Tagged template expressions are not supported

Tagged template expressions are not supported in metadata.

The compiler encountered a JavaScript ES2015 tagged template expression such as,
`` // ERROR const expression = 'funky'; const raw = String.raw`A tagged template ${expression} string`; ... template: '<div>' + raw + '</div>' ``
`String.raw()` is a *tag function* native to JavaScript ES2015.

The AOT compiler does not support tagged template expressions; avoid them in metadata expressions.

Symbol reference expected

The compiler expected a reference to a symbol at the location specified in the error message.

This error can occur if you use an expression in the `extends` clause of a class.

Summary

- What the AOT compiler does and why it is important.
- Why metadata must be written in a subset of JavaScript.
- What that subset is.
- Other restrictions on metadata definition.
- Macro-functions and macro-static methods.
- Compiler errors related to metadata.

Architecture Overview

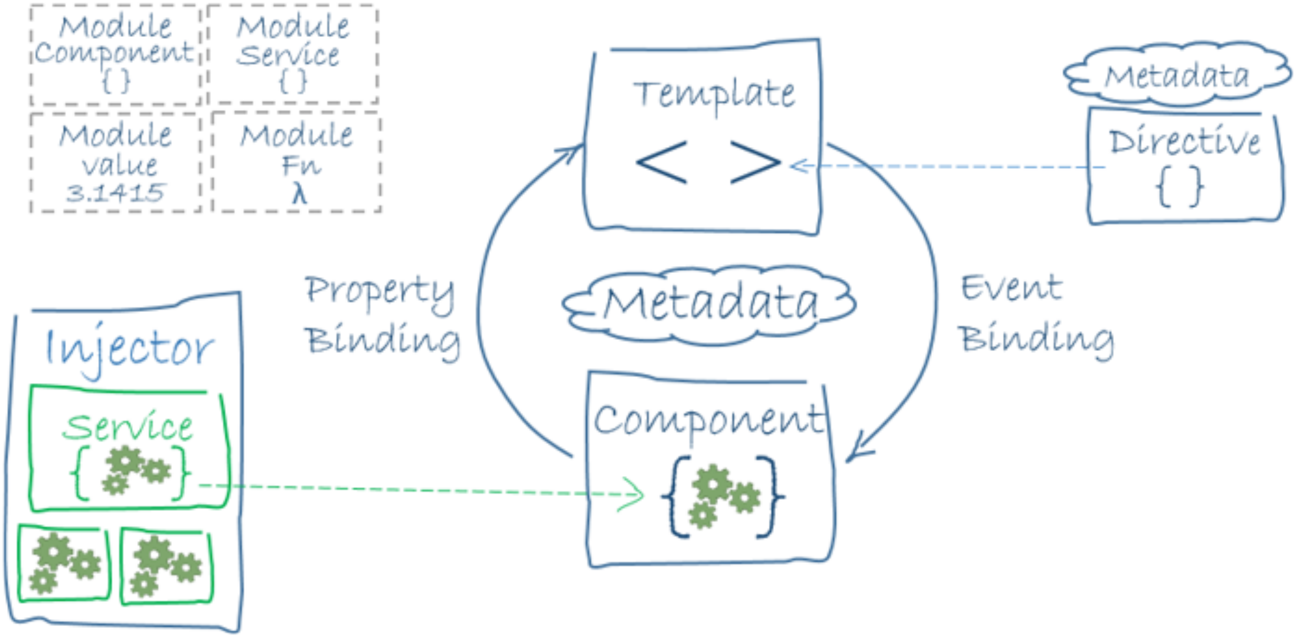
Angular is a framework for building client applications in HTML and either JavaScript or a language like TypeScript that compiles to JavaScript.

The framework consists of several libraries, some of them core and some optional.

You write Angular applications by composing HTML *templates* with Angularized markup, writing *component* classes to manage those templates, adding application logic in *services*, and boxing components and services in *modules*.

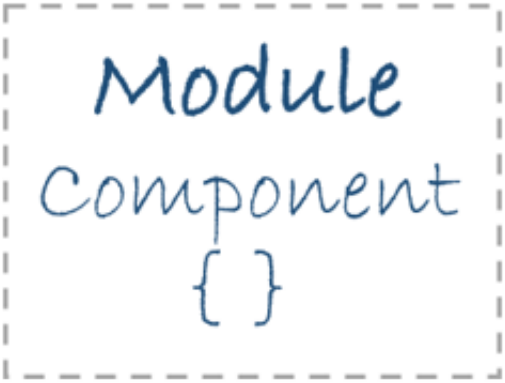
Then you launch the app by *bootstrapping* the *root module*. Angular takes over, presenting your application content in a browser and responding to user interactions according to the instructions you've provided.

Of course, there is more to it than this. You'll learn the details in the pages that follow. For now, focus on the big picture.



The code referenced on this page is available as a .

Modules



Angular apps are modular and Angular has its own modularity system called *NgModules*.

NgModules are a big deal. This page introduces modules; the NgModules page covers them in depth.

Every Angular app has at least one NgModule class, [the *root module*], conventionally named `AppModule`.

While the *root module* may be the only module in a small application, most apps have many more *feature modules*, each a cohesive block of code dedicated to an application domain, a workflow, or a closely related set of capabilities.

An NgModule, whether a *root* or *feature*, is a class with an `@NgModule` decorator.

Decorators are functions that modify JavaScript classes. Angular has many decorators that attach metadata to classes so that it knows what those classes mean and how they should work. Learn more about decorators on the web.

`NgModule` is a decorator function that takes a single metadata object whose properties describe the module. The most important properties are: * `declarations` - the *view classes* that belong to this module. Angular has three kinds of view classes: components, directives, and pipes.

- `exports` - the subset of declarations that should be visible and usable in the component templates of other modules.
- `imports` - other modules whose exported classes are needed by component templates declared in *this* module.
- `providers` - creators of services that this module contributes to the global collection of services; they become accessible in all parts of the app.
- `bootstrap` - the main application view, called the *root component*, that hosts all other app views. Only the *root module* should set this `bootstrap` property.

Here's a simple root module:

The `export` of `AppComponent` is just to show how to use the `exports` array to export a component; it isn't actually necessary in this example. A root module has no reason to _export_ anything because other components don't need to _import_ the root module.

Launch an application by *bootstrapping* its root module. During development you're likely to bootstrap the `AppModule` in a `main.ts` file like this one.

NgModules vs. JavaScript modules

The NgModule — a class decorated with `@NgModule` — is a fundamental feature of Angular.

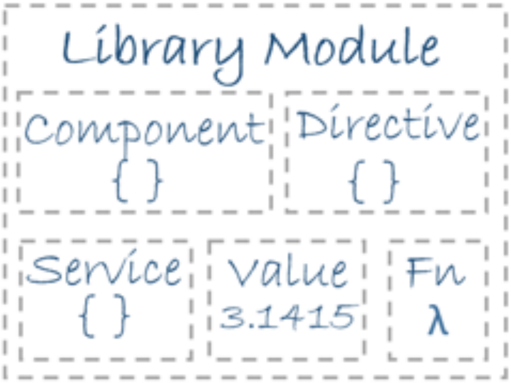
JavaScript also has its own module system for managing collections of JavaScript objects. It's completely different and unrelated to the NgModule system.

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.

Learn more about the JavaScript module system on the web.

These are two different and *complementary* module systems. Use them both to write your apps.

Angular libraries



Angular ships as a collection of JavaScript modules. You can think of them as library modules.

Each Angular library name begins with the `@angular` prefix.

You 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:

You also import NgModules from Angular *libraries* using JavaScript import statements:

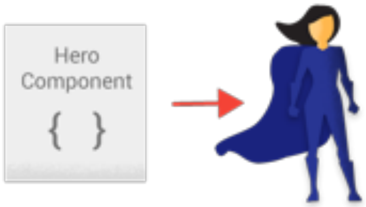
In the example of the simple root module above, the application module needs material from within that `BrowserModule`. To access that material, add it to the `@NgModule` metadata `imports` like this.

In this way you're using both the Angular and JavaScript module systems *together*.

It's easy to confuse the two systems because they share the common vocabulary of "imports" and "exports". Hang in there. The confusion yields to clarity with time and experience.

Learn more from the [NgModules](guide/ngmodule) page.

Components



A *component* controls a patch of screen called a *view*.

For example, the following views are controlled by components:

- 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.

{@a component-code}

For example, this `HeroListComponent` has a `heroes` property that returns an array of heroes that it acquires from a service. `HeroListComponent` also has a `selectHero()` method that sets a `selectedHero` property when the user clicks to choose a hero from that list.

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()` declared above.

Templates

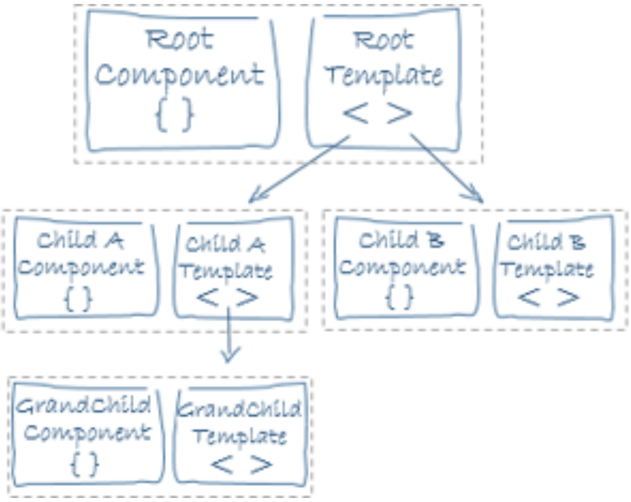
You define a component's view with its companion **template**. A template is a form of HTML that tells Angular how to render the component.

A template looks like regular HTML, except for a few differences. Here is a template for our `HeroListComponent`:

Although this template uses typical HTML elements like `<h2>` and `<p>`, it also has some differences. Code like `*ngFor`, `{{hero.name}}`, `(click)`, `[hero]`, and `<hero-detail>` uses Angular's template syntax.

In the last line of the template, the `<hero-detail>` tag is a custom element that represents a new component, `HeroDetailComponent`.

The `HeroDetailComponent` is a *different* component than the `HeroListComponent` you've been reviewing. The `HeroDetailComponent` (code not shown) presents facts about a particular hero, the hero that the user selects from the list presented by the `HeroListComponent`. The `HeroDetailComponent` is a **child** of the `HeroListComponent`.



Notice how `<hero-detail>` rests comfortably among native HTML elements. Custom components mix seamlessly with native HTML in the same layouts.

Metadata

Metadata tells Angular how to process a class.

Looking back at the code for `HeroListComponent`, you can see that it's just a class. There is no evidence of a framework, no "Angular" in it at all.

In fact, `HeroListComponent` really is *just a class*. It's not a component until you *tell Angular about it*.

To tell Angular that `HeroListComponent` is a component, attach **metadata** to the class.

In TypeScript, you attach metadata by using a **decorator**. Here's some metadata for `HeroListComponent`:

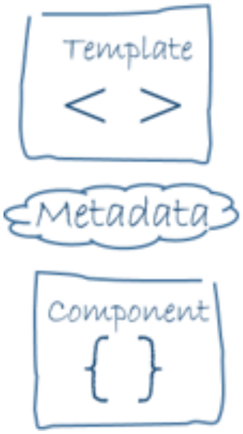
Here is the `@Component` decorator, which identifies the class immediately below it as a component class.

The `@Component` decorator takes a required configuration object with the information Angular needs to create and present the component and its view.

Here are a few of the most useful `@Component` configuration options:

- `selector`: CSS selector that tells Angular to create and insert an instance of this component where it finds a `<hero-list>` tag in *parent* HTML. For example, if an app's HTML contains `<hero-list></hero-list>`, then Angular inserts an instance of the `HeroListComponent` view between those tags.

- `templateUrl` : module-relative address of this component's HTML template, shown above.
- `providers` : array of **dependency injection providers** for services that the component requires. This is one way to tell Angular that the component's constructor requires a `HeroService` so it can get the list of heroes to display.



The metadata in the `@Component` tells Angular where to get the major building blocks you specify for the component.

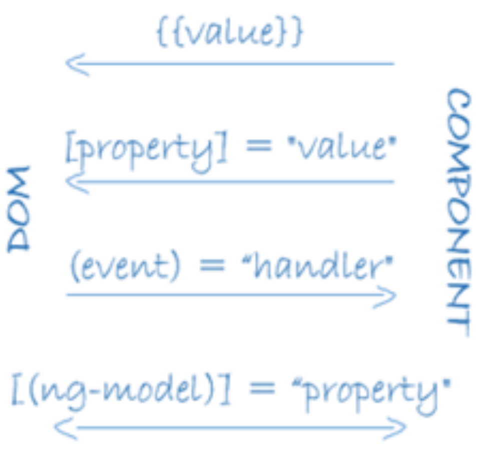
The template, metadata, and component together describe a view.

Apply other metadata decorators in a similar fashion to guide Angular behavior. `@Injectable` , `@Input` , and `@Output` are a few of the more popular decorators.

The architectural takeaway is that you must add metadata to your code so that Angular knows what to do.

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/pull logic by hand is tedious, error-prone, and a nightmare to read as any experienced jQuery programmer can attest.



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

As the diagram shows, there are four forms of data binding syntax. Each form has a direction — to the DOM, from the DOM, or in both directions.

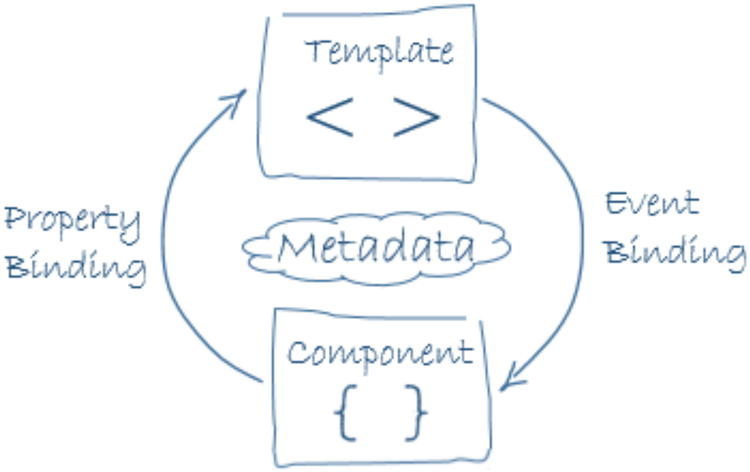
The `HeroListComponent` example template has three forms:

- 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.

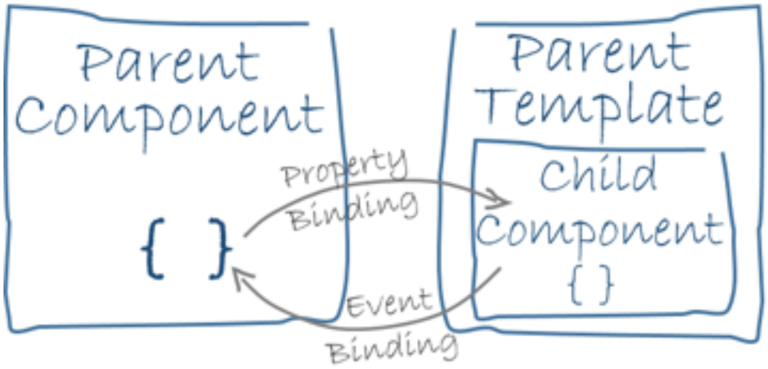
Two-way data binding is an important fourth form that combines property and event binding in a single notation, using the `ngModel` directive. Here's an example from the `HeroDetailComponent` template:

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 per 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.



Data binding is also important for communication between parent and child components.

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 a *directive-with-a-template*; a `@Component` decorator is actually a `@Directive` decorator extended with template-oriented features.

While **a component is technically a directive**, components are so distinctive and central to Angular applications that this architectural overview separates components from directives.

Two *other* kinds of directives exist: *structural* and *attribute* directives.

They tend to appear within an element tag as attributes do, sometimes by name but more often as the target of an assignment or a binding.

Structural directives alter layout by adding, removing, and replacing elements in DOM.

The example template uses two built-in structural directives:

- `*ngFor` tells Angular to stamp out one `<li>` per hero in the `heroes` list.
- `*ngIf` includes the `HeroDetail` component only if a selected hero exists.

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 an `<input>`) by setting its display value property and responding to change events.

Angular has a few more directives that either alter the layout structure (for example, ngSwitch) or modify aspects of DOM elements and components (for example, ngStyle and ngClass).

Of course, you can also write your own directives. Components such as `HeroListComponent` are one kind of custom directive.

Services

Service is a broad category encompassing any value, function, or feature that your application needs.

Almost anything can be a service. A service is typically a class with a narrow, well-defined purpose. It should do something specific and do it well.

Examples include:

- logging service
- data service
- message bus
- tax calculator
- application configuration

There is nothing specifically *Angular* about services. Angular has no definition of a service. There is no service base class, and no place to register a service.

Yet services are fundamental to any Angular application. Components are big consumers of services.

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

Here's a `HeroService` that uses a Promise to fetch heroes. The `HeroService` depends on the `Logger` service and another `BackendService` that handles the server communication grunt work.

Services are everywhere.

Component classes should be lean. They don't fetch data from the server, validate user input, or log directly to the console. They delegate such tasks to services.

A component's job is to enable the user experience and nothing more. It mediates between the view (rendered by the template) and the application logic (which often includes some notion of a *model*). A good component presents properties and methods for data binding. It delegates everything nontrivial to services.

Angular doesn't *enforce* these principles. It won't complain if you write a "kitchen sink" component with 3000 lines.

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*.

Dependency injection

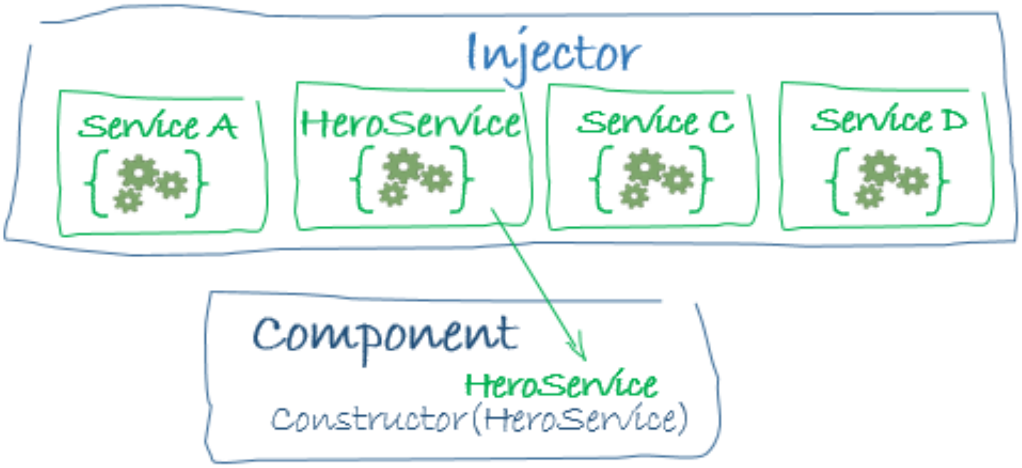
Dependency injection is a way to supply a new instance of a class with the fully-formed dependencies it requires. Most dependencies are services. Angular uses dependency injection to provide new components with the services they need.

Angular can tell which services a component needs by looking at the types of its constructor parameters. For example, the constructor of your `HeroListComponent` needs a `HeroService`:

When Angular creates a component, it first asks an **injector** for the services that the component requires.

An injector maintains a container of service instances that it has previously created. If a requested service instance is not in the container, the injector makes one and adds it to the container 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. This is *dependency injection*.

The process of `HeroService` injection looks a bit like this:



If the injector doesn't have a `HeroService`, how does it know how to make one?

In brief, you must have previously registered a **provider** of the `HeroService` with the injector. A provider is something that can create or return a service, typically the service class itself.

You can register providers in modules or in components.

In general, add providers to the root module so that the same instance of a service is available everywhere.

Alternatively, register at a component level in the `providers` property of the `@Component` metadata:

Registering at a component level means you get a new instance of the service with each new instance of that component.

Points to remember about dependency injection:

- Dependency injection is wired into the Angular framework and used everywhere.

- The *injector* is the main mechanism.

  - An injector maintains a *container* of service instances that it created.
  - An injector can create a new service instance from a *provider*.

- A *provider* is a recipe for creating a service.

- Register *providers* with injectors.

Wrap up

You've learned the basics about the eight main building blocks of an Angular application:

- Modules
- Components
- Templates
- Metadata
- Data binding
- Directives
- Services
- Dependency injection

That's a foundation for everything else in an Angular application, and it's more than enough to get going. But it doesn't include everything you need to know.

Here is a brief, alphabetical list of other important Angular features and services. Most of them are covered in this documentation (or soon will be).

> **Animations**: Animate component behavior without deep knowledge of animation techniques or CSS with Angular's animation library.
>
> **Change detection**: The change detection documentation will cover how Angular decides that a component property value has changed, when to update the screen, and how it uses **zones** to intercept asynchronous activity and run its change detection strategies.
>
> **Events**: The events documentation will cover how to use components and services to raise events with mechanisms for publishing and subscribing to events.
>
> **Forms**: Support complex data entry scenarios with HTML-based validation and dirty checking.

> **HTTP**: Communicate with a server to get data, save data, and invoke server-side actions with an HTTP client.
>
> **Lifecycle hooks**: Tap into key moments in the lifetime of a component, from its creation to its destruction, by implementing the lifecycle hook interfaces.
>
> **Pipes**: Use pipes in your templates to improve the user experience by transforming values for display. Consider this `currency` pipe expression:
>
> > `price | currency:'USD':true`
>
> It displays a price of 42.33 as `$42.33`.
>
> **Router**: Navigate from page to page within the client application and never leave the browser.
>
> **Testing**: Run unit tests on your application parts as they interact with the Angular framework using the *Angular Testing Platform*.

Attribute Directives

An **Attribute** directive changes the appearance or behavior of a DOM element.

Try the .

{@a directive-overview}

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 QuickStart guide.

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:

{@a write-directive}

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](guide/ngmodule) in the same manner as _components_. The generated `src/app/highlight.directive.ts` is as follows: The imported `Directive` symbol provides the Angular the `@Directive` decorator. The `@Directive` decorator's lone configuration property specifies the directive's [CSS attribute selector](https://developer.mozilla.org/en-US/docs/Web/CSS/Attribute_selectors), `[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:

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.

{@a apply-directive}

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.

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.

{@a respond-to-user}

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.

Then add two eventhandlers that respond when the mouse enters or leaves, each adorned by the `HostListener` decorator.

The `@HostListener` decorator lets you subscribe to events of the DOM element that hosts an attribute directive, the `<p>` in this case.

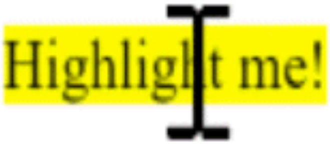
Of course you could reach into the DOM with standard JavaScript and attach event listeners manually. There are at least three problems with _that_ approach: 1. You have to write the listeners correctly. 1. The code must *detach* the listener when the directive is destroyed to avoid memory leaks. 1. Talking 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`.

Here's the updated directive in full:

Run the app and confirm that the background color appears when the mouse hovers over the `p` and disappears as it moves out.



{@a bindings}

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`.

Add a `highlightColor` property to the directive class like this:

{@a input}

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:

Add a `color` property to the `AppComponent`.

Let it control the highlight color with a property binding.

That's good, but it would be nice to *simultaneously* apply the directive and set the color *in the same attribute* like this.

The `[appHighlight]` attribute binding both applies the highlighting directive to the `<p>` element and sets the directive's highlight color 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.

This is disagreeable. The word, `appHighlight`, is a terrible property name and it doesn't convey the property's intent.

{@a input-alias}

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`.

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:

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:

Here's the latest version of the directive class.

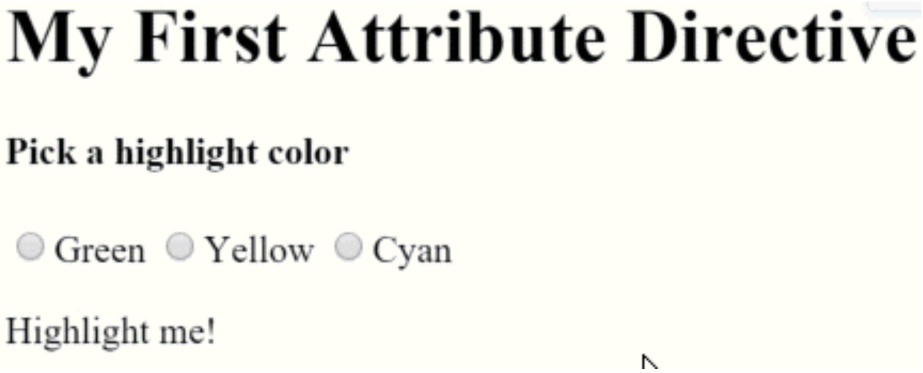
Write a harness to try it

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:

Revise the `AppComponent.color` so that it has no initial value.

Here are the harness and directive in action.



My First Attribute Directive

Pick a highlight color

○ Green ○ Yellow ○ Cyan

Highlight me!

{@a second-property}

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`:

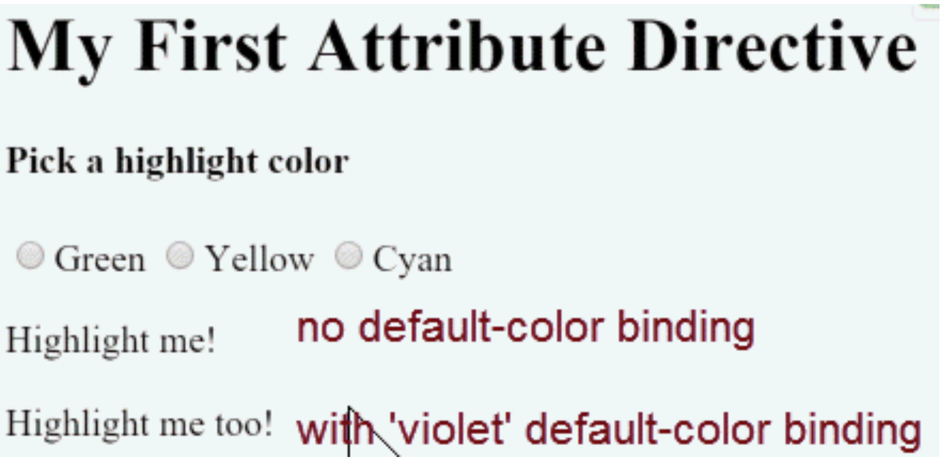
Revise the directive's `onMouseEnter` so that it first tries to highlight with the `highlightColor`, then with the `defaultColor`, and falls back to "red" if both properties are undefined.

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.

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:

You can also experience and download the .

{@a why-input}

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:

You've seen it with an alias:

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:

- 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.

Bootstrapping

An Angular Module (NgModule) class describes how the application parts fit together. Every application has at least one Angular Module, the *root* module that you bootstrap to launch the application. You can call the class anything you want. The conventional name is `AppModule`.

The **Angular CLI** produces a new project with the following minimal `AppModule`. You evolve this module as your application grows.

After the `import` statements, you come to a class adorned 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.

The `@NgModule` properties for the minimal `AppModule` generated by the CLI are as follows:

- ***declarations*** — declares the application components. At the moment, there is only the `AppComponent`.
- ***imports*** — the `BrowserModule`, which this and every application must import in order to run the app in a browser.
- ***providers*** — there are none to start but you are likely to add some soon.
- ***bootstrap*** — the *root* `AppComponent` that Angular creates and inserts into the `index.html` host web page.

The Angular Modules (NgModules) guide dives deeply into the details of `@NgModule`. All you need to know at the moment is a few basics about these four properties.

{@a declarations}

The *declarations* array

You tell Angular which components belong to the `AppModule` by listing it in the module's `declarations` array. As you create more components, you'll add them to `declarations`.

You must declare *every* component in an Angular Module class. If you use a component without declaring it, you'll see a clear error message in the browser console.

You'll learn to create two other kinds of classes — directives and pipes — that you must also add to the `declarations` array.

Only _declarables_ — _components_, _directives_ and _pipes_ — belong in the `declarations` array. Do not put any other kind of class in `declarations`. Do _not_ declare `NgModule` classes. Do _not_ declare service classes. Do _not_ declare model classes.

{@a imports}

The *imports* array

Angular Modules are a way to consolidate features that belong together into discrete units. Many features of Angular itself are organized as Angular Modules. HTTP services are in the `HttpClientModule`. The router is in the `RouterModule`. Eventually you may create your own modules.

Add a module to the `imports` array when the application requires its features.

This application, like most applications, executes in a browser. Every application that executes in a browser needs the `BrowserModule` from `@angular/platform-browser`. So every such application includes the `BrowserModule` in its *root* `AppModule`'s `imports` array. Other guide pages will tell you when you need to add additional modules to this array.

Only `@NgModule` classes go in the `imports` array. Do not put any other kind of class in `imports`. The `import` statements at the top of the file and the NgModule's `imports` array are unrelated and have completely different jobs. The _JavaScript_ `import` statements give you access to symbols _exported_ by other files so you can reference them within _this_ file. You add `import` statements to almost every application file. They have nothing to do with Angular and Angular knows nothing about them. The _module's_ `imports` array appears _exclusively_ in the `@NgModule` metadata object. It tells Angular about specific _other_ Angular Modules—all of them classes decorated with `@NgModule`—that the application needs to function properly.

{@a providers}

The *providers* array

Angular apps rely on *dependency injection (DI)* to deliver services to various parts of the application.

Before DI can inject a service, it must create that service with the help of a *provider*. You can tell DI about a service's *provider* in a number of ways. Among the most popular ways is to register the service in the root `ngModule.providers` array, which will make that service available *everywhere*.

For example, a data service provided in the `AppModule`s *providers* can be injected into any component anywhere in the application.

You don't have any services to provide yet. But you will create some before long and you may chose to provide many of them here.

{@a bootstrap-array}

The *bootstrap* array

You launch the application by *bootstrapping* the root `AppModule`. 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, that's not typical. Most applications have only one component tree and they bootstrap a single *root* component.

You can call the one *root* component anything you want but most developers call it `AppComponent`.

Which brings us to the *bootstrapping* process itself.

{@a main}

Bootstrap in *main.ts*

While there are many ways to bootstrap an application, most applications do so in the `src/main.ts` that is generated by the Angular CLI.

This code creates a browser platform for dynamic compilation and bootstraps the `AppModule` described above.

The *bootstrapping* process sets up the execution environment, digs the *root* `AppComponent` out of the module's `bootstrap` array, creates an instance of the component and inserts it within the element tag identified by the component's `selector`.

The `AppComponent` selector — here and in most documentation samples — is `app-root` so Angular looks for a `<app-root>` tag in the `index.html` like this one ...

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

... and displays the `AppComponent` there.

The `main.ts` file is very stable. Once you've set it up, you may never change it again.

More about Angular Modules

Your initial app has only a single module, the *root* module. As your app grows, you'll consider subdividing it into multiple "feature" modules, some of which can be loaded later ("lazy loaded") if and when the user chooses to visit those features.

When you're ready to explore these possibilities, visit the Angular Modules guide.

Browser support

Angular supports most recent browsers. This includes the following specific versions:

Chrome	Firefox	Edge	IE	Safari	iOS	Android	IE Mobile
latest	latest	14	11	10	10	Nougat (7.0) Marshmallow (6.0)	11
		13	10	9	9	Lollipop (5.0, 5.1)	
			9	8	8	KitKat (4.4)	
				7	7	Jelly Bean (4.1, 4.2, 4.3)	

Angular's continuous integration process runs unit tests of the framework on all of these browsers for every pull request, using SauceLabs and Browserstack.

Polyfills

Angular is built on the latest standards of the web platform. Targeting such a wide range of browsers is challenging because they do not support all features of modern browsers.

You compensate by loading polyfill scripts ("polyfills") for the browsers that you must support. The table below identifies most of the polyfills you might need.

The suggested polyfills are the ones that run full Angular applications. You may need additional polyfills to support features not covered by this list. Note that polyfills cannot magically transform an old, slow browser into a modern, fast one.

Enabling polyfills

Angular CLI users enable polyfills through the `src/polyfills.ts` file that the CLI created with your project.

This file incorporates the mandatory and many of the optional polyfills as JavaScript `import` statements.

The npm packages for the *mandatory* polyfills (such as `zone.js`) were installed automatically for you when you created your project and their corresponding `import` statements are ready to go. You probably won't touch these.

But if you need an optional polyfill, you'll have to install its npm package with `npm` or `yarn`. For example, if you need the web animations polyfill, you could install it with either of the following commands:

npm install --save web-animations-js yarn add web-animations-js

Then open the `polyfills.ts` file and un-comment the corresponding `import` statement as in the following example:

/** * Required to support Web Animations `@angular/platform-browser/animations`. * Needed for: All but Chrome, Firefox and Opera. http://caniuse.com/#feat=web-animation **/ import 'web-animations-js'; // Run `npm install --save web-animations-js`.

If you can't find the polyfill you want in `polyfills.ts`, add it yourself, following the same pattern:

1. install the npm package
2. `import` the file in `polyfills.ts`

Non-CLI users should follow the instructions [below](#non-cli).

{@a polyfill-libs}

Mandatory polyfills

These are the polyfills required to run an Angular application on each supported browser:

Browsers (Desktop & Mobile)	Polyfills Required
Chrome, Firefox, Edge, Safari 9+	[ES7/reflect](guide/browser-support#core-es7-reflect) (JIT only)
Safari 7 & 8, IE10 & 11, Android 4.1+	[ES6](guide/browser-support#core-es6)
IE9	[ES6 classList](guide/browser-support#classlist)

Optional browser features to polyfill

Some features of Angular may require additional polyfills.

For example, the animations library relies on the standard web animation API, which is only available in Chrome and Firefox today. You'll need a polyfill to use animations in other browsers.

Here are the features which may require additional polyfills:

Feature	Polyfill	Browsers (Desktop & Mobile)
[JIT compilation](guide/aot-compiler). Required to reflect for metadata.	[ES7/reflect](guide/browser-support#core-es7-reflect)	All current browsers. Enabled by default. Can remove If you always use AOT and only use Angular decorators.
[Animations](guide/animations)	[Web Animations](guide/browser-support#web-animations)	All but Chrome and Firefox Not supported in IE9
If you use the following deprecated i18n pipes: [date](api/common/DeprecatedDatePipe), [currency](api/common/DeprecatedCurrencyPipe), [decimal](api/common/DeprecatedDecimalPipe) and [percent](api/common/DeprecatedPercentPipe)	[Intl API](guide/browser-support#intl)	All but Chrome, Firefox, Edge, IE11 and Safari 10
[NgClass](api/common/NgClass) on SVG elements	[classList](guide/browser-support#classlist)	IE10, IE11
[Http](guide/http) when sending and receiving binary data	[Typed Array](guide/browser-support#typedarray) [Blob](guide/browser-support#blob) [FormData](guide/browser-support#formdata)	IE 9

Suggested polyfills

Below are the polyfills which are used to test the framework itself. They are a good starting point for an application.

Polyfill	License	Size*
ES7/reflect	MIT	0.5KB
ES6	MIT	27.4KB
classList	Public domain	1KB
Intl	MIT / Unicode license	13.5KB
Web Animations	Apache	14.8KB
Typed Array	MIT	4KB
Blob	MIT	1.3KB
FormData	MIT	0.4KB

* Figures are for minified and gzipped code, computed with the closure compiler.

{@a non-cli}

Polyfills for non-CLI users

If you aren't using the CLI, you should add your polyfill scripts directly to the host web page (`index.html`), perhaps like this.

<!-- pre-zone polyfills --> <script src="node*modules/core-js/client/shim.min.js"></script> <script src="node*modules/web-animations-js/web-animations.min.js"></script>

<!-- zone.js required by Angular --> <script src="node_modules/zone.js/dist/zone.js"></script>

<!-- application polyfills -->

Change Log

The Angular documentation is a living document with continuous improvements. This log calls attention to recent significant changes.

Updated to Angular 4.0. Documentation for Angular 2.x can be found at v2.angular.io.

All mention of moduleId removed. "Component relative paths" guide deleted (2017-03-13)

We added a new SystemJS plugin (systemjs-angular-loader.js) to our recommended SystemJS configuration. This plugin dynamically converts "component-relative" paths in templateUrl and styleUrls to "absolute paths" for you.

We strongly encourage you to only write component-relative paths. That is the only form of URL discussed in these docs. You no longer need to write @Component({ moduleId: module.id }), nor should you.

NEW: Downloadable examples for each guide (2017-02-28)

Now you can download the sample code for any guide and run it locally. Look for the new download links next to the "live example" links.

Template Syntax/Structural Directives: refreshed (2017-02-06)

The *Template-Syntax* and *Structural Directives* guides were significantly revised for clarity, accuracy, and current recommended practices. Discusses `<ng-container>`. Revised samples are more clear and cover all topics discussed.

NEW: Samples re-structured with `src/` folder (2017-02-02)

All documentation samples have been realigned with the default folder structure of the Angular CLI. That's a step along the road to basing the sample in the Angular CLI. But it's also good in its own right. It helps clearly separate app code from setup and configuration files.

All samples now have a `src/` folder at the project root. The former `app/` folder moves under `src/`. Read about moving your existing project to this structure in the QuickStart repo update instructions.

Notably:

- `app/main.ts` moved to `src/main.ts`.
- `app/` moved to `src/app/`.
- `index.html`, `styles.css` and `tsconfig.json` moved inside `src/`.
- `systemjs.config.js` now imports `main.js` instead of `app`.
- Added `lite-server` configuration (`bs-config.json`) to serve `src/`.

NEW: Reactive Forms guide (2017-01-31)

The new **Reactive Forms** guide explains how and why to build a "reactive form". "Reactive Forms" are the code-based counterpart to the declarative "Template Driven" forms approach introduced in the Forms guide. Check it out before you decide how to add forms to your app. Remember also that you can use both techniques in the same app, choosing the approach that best fits each scenario.

NEW: Deployment guide (2017-01-30)

The new Deployment guide describes techniques for putting your application on a server. It includes important advice on optimizing for production.

Hierarchical Dependency Injection: refreshed (2017-01-13)

Hierarchical Dependency Injection guide is significantly revised. Closes issue #3086. Revised samples are clearer and cover all topics discussed.

Miscellaneous (2017-01-05)

- Setup guide: added (optional) instructions on how to remove *non-essential* files.
- No longer consolidate RxJS operator imports in `rxjs-extensions` file; each file should import what it needs.
- All samples prepend template/style URLs with `./` as a best practice.

- Style Guide: copy edits and revised rules.

Router: more detail (2016-12-21)

Added more information to the Router guide including sections named outlets, wildcard routes, and preload strategies.

HTTP: how to set default request headers (and other request options) (2016-12-14)

Added section on how to set default request headers (and other request options) to HTTP guide.

Testing: added component test plunkers (2016-12-02)

Added two plunkers that each test *one simple component* so you can write a component test plunker of your own: one for the QuickStart seed's `AppComponent` and another for the Testing guide's `BannerComponent`. Linked to these plunkers in Testing and Setup anatomy guides.

Internationalization: pluralization and *select* (2016-11-30)

The Internationalization (i18n) guide explains how to handle pluralization and translation of alternative texts with `select`. The sample demonstrates these features too.

Testing: karma file updates (2016-11-30)

- `karma.config` + `karma-test-shim` can handle multiple spec source paths; see quickstart issue: angular/quickstart#294.
- Displays Jasmine Runner output in the karma-launched browser.

QuickStart Rewrite (2016-11-18)

The QuickStart is completely rewritten so that it actually is quick. It references a minimal "Hello Angular" app running in Plunker. The new Setup page tells you how to install a local development environment by downloading (or cloning) the QuickStart github repository. You are no longer asked to copy-and-paste code into setup files that were not explained anyway.

Sync with Angular v.2.2.0 (2016-11-14)

Docs and code samples updated and tested with Angular v.2.2.0.

UPDATE: NgUpgrade Guide for the AOT friendly *upgrade/static* module (2016-11-14)

The updated NgUpgrade Guide guide covers the new AOT friendly `upgrade/static` module released in v.2.2.0, which is the recommended facility for migrating from AngularJS to Angular. The documentation for the version prior to v.2.2.0 has been removed.

ES6 described in "TypeScript to JavaScript" (2016-11-14)

The updated TypeScript to JavaScript guide (removed August 2017, PR #18694) explains how to write apps in ES6/7 by translating the common idioms in the TypeScript documentation examples (and elsewhere on the web) to ES6/7 and ES5.

Sync with Angular v.2.1.1 (2016-10-21)

Docs and code samples updated and tested with Angular v.2.1.1.

npm *@types* packages replace *typings* (2016-10-20)

Documentation samples now get TypeScript type information for 3rd party libraries from npm `@types` packages rather than with the *typings* tooling. The `typings.json` file is gone.

The AngularJS Upgrade guide reflects this change. The `package.json` installs `@types/angular` and several `@types/angular-...` packages in support of upgrade; these are not needed for pure Angular development.

"Template Syntax" explains two-way data binding syntax (2016-10-20)

Demonstrates how to two-way data bind to a custom Angular component and re-explains `[(ngModel)]` in terms of the basic `[()]` syntax.

BREAKING CHANGE: `in-memory-web-api` (v.0.1.11) delivered as esm umd (2016-10-19)

This change supports ES6 developers and aligns better with typical Angular libraries. It does not affect the module's API but it does affect how you load and import it. See the change note in the `in-memory-web-api` repo.

"Router" *preload* syntax and *:enter*/*:leave* animations (2016-10-19)

The router can lazily *preload* modules *after* the app starts and *before* the user navigates to them for improved perceived performance.

New `:enter` and `:leave` aliases make animation more natural.

Sync with Angular v.2.1.0 (2016-10-12)

Docs and code samples updated and tested with Angular v.2.1.0.

NEW "Ahead of time (AOT) Compilation" guide (2016-10-11)

The NEW Ahead of time (AOT) Compilation guide explains what AOT compilation is and why you'd want it. It demonstrates the basics with a QuickStart app followed by the more advanced considerations of compiling and bundling the Tour of Heroes.

Sync with Angular v.2.0.2 (2016-10-6)

Docs and code samples updated and tested with Angular v.2.0.2.

"Routing and Navigation" guide with the *Router Module* (2016-10-5)

The Routing and Navigation guide now locates route configuration in a *Routing Module*. The *Routing Module* replaces the previous *routing object* involving the `ModuleWithProviders`.

All guided samples with routing use the *Routing Module* and prose content has been updated, most conspicuously in the NgModule guide and NgModule FAQ guide.

New "Internationalization" guide (2016-09-30)

Added a new Internationalization (i18n) guide that shows how to use Angular "i18n" facilities to translate template text into multiple languages.

"angular-in-memory-web-api" package rename (2016-09-27)

Many samples use the `angular-in-memory-web-api` to simulate a remote server. This library is also useful to you during early development before you have a server to talk to.

The package name was changed from "angular2-in-memory-web-api" which is still frozen-in-time on npm. The new "angular-in-memory-web-api" has new features. Read about them on github.

"Style Guide" with *NgModules* (2016-09-27)

StyleGuide explains recommended conventions for NgModules. Barrels now are far less useful and have been removed from the style guide; they remain valuable but are not a matter of Angular style. Also relaxed the rule that discouraged use of the `@Component.host` property.

moduleId: module.id everywhere (2016-09-25)

Sample components that get their templates or styles with `templateUrl` or `styleUrls` have been converted to *module-relative* URLs. Added the `moduleId: module.id` property-and-value to their `@Component` metadata.

This change is a requirement for compilation with AOT compiler when the app loads modules with SystemJS as the samples currently do.

"Lifecycle Hooks" guide simplified (2016-09-24)

The Lifecycle Hooks guide is shorter, simpler, and draws more attention to the order in which Angular calls the hooks.

Cheat Sheet

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: ...})`
`class MyModule {}`
	Defines a module that contains components, directives, pipes, and providers.

`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.
`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 ...>`
`<button (click)="movieplayer.play()">`
`</video>`
	Creates a local variable `movieplayer` that provides access to the `video` element instance in data-binding and 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	`import { CommonModule } from '@angular/common';`
`<section *ngIf="showSection">`	Removes or recreates a portion of the DOM tree based on the `showSection` expression.
`<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">`
`<ng-template [ngSwitchCase]="case1Exp">...</ng-template>`
`<ng-template ngSwitchCase="case2LiteralString">...</ng-template>`
`<ng-template ngSwitchDefault>...</ng-template>`
`</div>`
	Conditionally swaps the contents of the div by selecting one of the embedded templates based on the current value of `conditionExpression` .

`<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.

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({...})`
`class MyComponent() {}`
	Declares that a class is a component and provides metadata about the component.

`@Directive({...})`
`class MyDirective() {}`
	Declares that a class is a directive and provides metadata about the directive.

`@Pipe({...})`
`class MyPipe() {}`
	Declares that a class is a pipe and provides metadata about the pipe.

`@Injectable()`
`class MyService() {}`
	Declares that a class has dependencies that should be injected into the constructor when the dependency injector is creating an instance of this class.

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	`@Component` **extends** `@Directive` **, so the** `@Directive` **configuration applies to components as well**
`moduleId: module.id`	If set, the `templateUrl` and `styleUrl` are resolved relative to the component.
`viewProviders: [MyService, { provide: ... }]`	List of dependency injection providers scoped to this component's view.

`template: '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 { Input, ... } from '@angular/core';`
`@Input() myProperty;`	Declares an input property that you can update via property binding (example: `<my-cmp [myProperty]="someExpression">`).
`@Output() myEvent = new EventEmitter();`	Declares an output property that fires events that you can subscribe to with an event binding (example: `<my-cmp (myEvent)="doSomething()">`).
`@HostBinding('class.valid') isValid;`	Binds a host element property (here, the CSS class `valid`) to a directive/component property (`isValid`).
`@HostListener('click', ['$event']) onClick(e) {...}`	Subscribes to a host element event (`click`) with a directive/component method (`onClick`), optionally passing an argument (`$event`).
`@ContentChild(myPredicate) myChildComponent;`	Binds the first result of the component content query (`myPredicate`) to a property (`myChildComponent`) of the class.
`@ContentChildren(myPredicate) myChildComponents;`	Binds the results of the component content query (`myPredicate`) to a property (`myChildComponents`) of the class.
`@ViewChild(myPredicate) myChildComponent;`	Binds the first result of the component view query (`myPredicate`) to a property (`myChildComponent`) of the class. Not available for directives.
`@ViewChildren(myPredicate) myChildComponents;`	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)
`constructor(myService: MyService, ...) { ... }`	Called before any other lifecycle hook. Use it to inject dependencies, but avoid any serious work here.
`ngOnChanges(changeRecord) { ... }`	Called after every change to input properties and before processing content or child views.
`ngOnInit() { ... }`	Called after the constructor, initializing input properties, and the first call to `ngOnChanges`.
`ngDoCheck() { ... }`	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.
`ngAfterContentInit() { ... }`	Called after `ngOnInit` when the component's or directive's content has been initialized.
`ngAfterContentChecked() { ... }`	Called after every check of the component's or directive's content.
`ngAfterViewInit() { ... }`	Called after `ngAfterContentInit` when the component's view has been initialized. Applies to components only.
`ngAfterViewChecked() { ... }`	Called after every check of the component's view. Applies to components only.
`ngOnDestroy() { ... }`	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' } }`
`]);`
	Configures routes for the application. Supports static, parameterized, redirect, and wildcard routes. Also supports custom route data and resolve.

`const routing = RouterModule.forRoot(routes);`	

`<router-outlet></router-outlet>`
`<router-outlet name="aux"></router-outlet>`
	Marks the location to load the component of the active route.

`<a routerLink="/path">`
`<a [routerLink]="[ '/path', routeParam ]">`
`<a [routerLink]="[ '/path', { matrixParam: 'value' } ]">`
`<a [routerLink]="[ '/path' ]" [queryParams]="{ page: 1 }">`
`<a [routerLink]="[ '/path' ]" fragment="anchor">`
	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 sibling or parent, use the `../` prefix.

`<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 { ... }`
`}`
 	
`{ path: ..., canDeactivate: [CanDeactivateGuard] }`
	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.

`class CanActivateChildGuard implements CanActivateChild {`
`canActivateChild(`
`route: ActivatedRouteSnapshot,`
`state: RouterStateSnapshot`
`): Observable<boolean>|Promise<boolean>|boolean { ... }`
`}`
 	
`{ path: ..., canActivateChild: [CanActivateGuard],`
`children: ... }`
	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.

`class ResolveGuard implements Resolve<T> {`
`resolve(`
`route: ActivatedRouteSnapshot,`
`state: RouterStateSnapshot`
`): Observable<any>|Promise<any>|any { ... }`
`}`
 	
`{ path: ..., resolve: [ResolveGuard] }`
	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.

`class CanLoadGuard implements CanLoad {`
`canLoad(`
`route: Route`
`): Observable<boolean>|Promise<boolean>|boolean { ... }`
`}`
 	
`{ path: ..., canLoad: [CanLoadGuard], loadChildren: ... }`
	An interface for defining a class that the router should call first to check if the lazy loaded module should be loaded. Should return a boolean or an Observable/Promise that resolves to a boolean.

Component Interaction

{@a top}

This cookbook contains recipes for common component communication scenarios in which two or more components share information. {@a toc}

See the .

{@a parent-to-child}

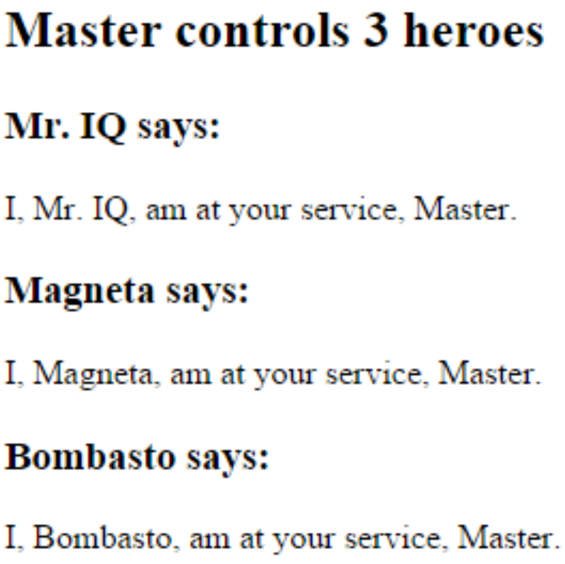
Pass data from parent to child with input binding

`HeroChildComponent` has two ***input properties***, typically adorned with @Input decorations.

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.

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:

Back to top

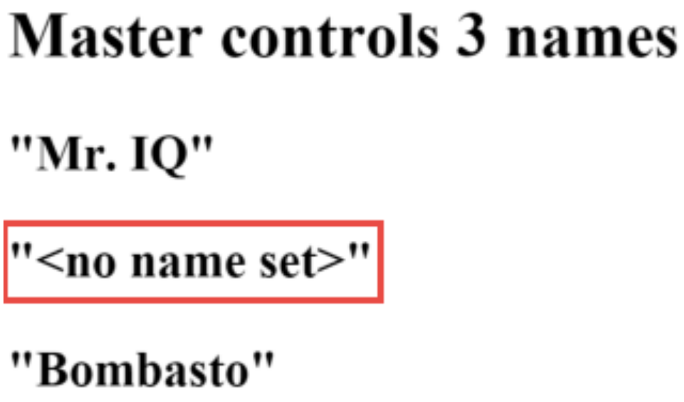
{@a parent-to-child-setter}

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.

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



Master controls 3 names

"Mr. IQ"

"<no name set>"

"Bombasto"

Test it

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

Back to top

{@a parent-to-child-on-changes}

Intercept input property changes with *ngOnChanges()*

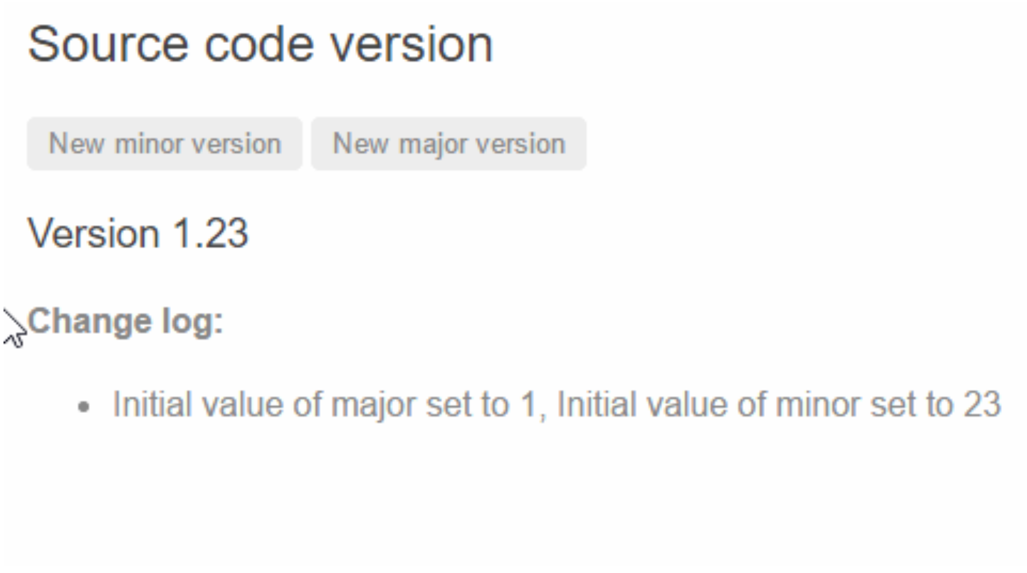
Detect and act upon changes to input property values with the `ngOnChanges()` method of the `OnChanges` lifecycle hook interface.

You may prefer this approach to the property setter when watching multiple, interacting input properties. Learn about `ngOnChanges()` in the [LifeCycle Hooks](guide/lifecycle-hooks) chapter.

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

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

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



Source code 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:

[Back to top]

{@a child-to-parent}

Parent listens for child event

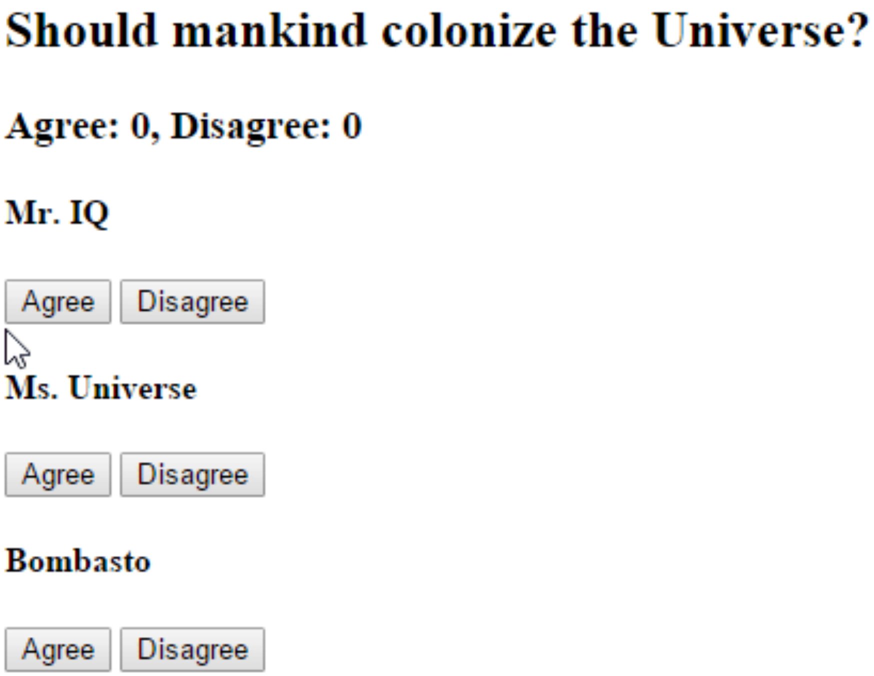
The child component exposes an `EventEmitter` property with which it `emits` events when something happens. The parent binds to that event property and reacts to those events.

The child's `EventEmitter` property is an ***output property***, typically adorned with an @Output decoration as seen in this `VoterComponent` :

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.

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:

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.

{@a countdown-timer-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.

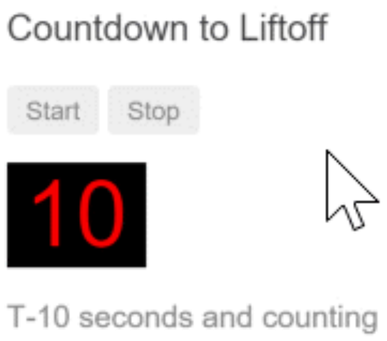
The `CountdownLocalVarParentComponent` that hosts the timer component is as follows:

The parent component cannot data bind to the child's `start` and `stop` methods nor to its `seconds` property.

You can place a local variable, `#timer`, on the tag `<countdown-timer>` representing the child component. That gives you a reference to the child component and the ability to access *any of its properties or methods* from within the parent template.

This example wires parent buttons to the child's `start` and `stop` and uses interpolation to display the child's `seconds` property.

Here we see the parent and child working together.


Countdown to Liftoff

T-10 seconds and counting

{@a countdown-tests}

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:

Back to top

{@a parent-to-view-child}

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` :

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

{@a bidirectional-service}

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.

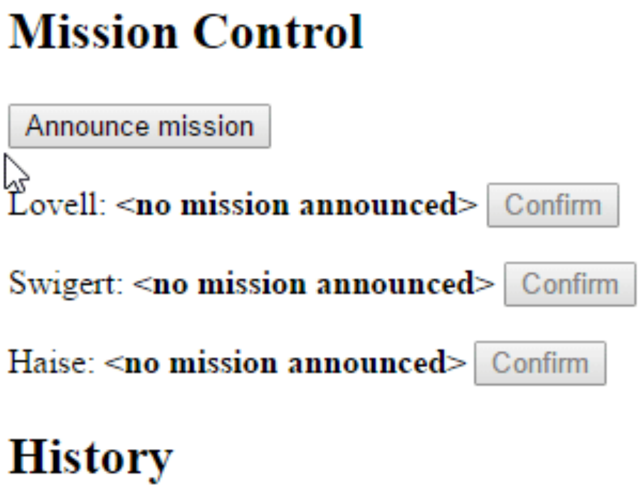
This `MissionService` connects the `MissionControlComponent` to multiple `AstronautComponent` children.

The `MissionControlComponent` both provides the instance of the service that it shares with its children (through the `providers` metadata array) and injects that instance into itself through its constructor:

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:

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

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:

Back to top

Component Styles

Angular applications are styled with standard CSS. That means you can apply everything you know about CSS stylesheets, selectors, rules, and media queries directly to Angular applications.

Additionally, Angular can bundle *component styles* with components, enabling a more modular design than regular stylesheets.

This page describes how to load and apply these component styles.

You can run the in Plunker 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:

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.

{@a special-selectors}

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).

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.

: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`.

(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.

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](guide/component-styles#view-encapsulation) section.
The shadow-piercing descendant combinator is deprecated and [support is being removed from major browsers](https://www.chromestatus.com/features/6750456638341120) and tools. As such we plan to drop support in Angular (for all 3 of `/deep/`, `>>>` and `::ng-deep`). Until then `::ng-deep` should be preferred for a broader compatibility with the tools.

{@a loading-styles}

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.

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-styles` 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:

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 `style` 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.

Template link tags

You can also write `<link>` tags into the component's HTML template.

The link tag's `href` URL must be relative to the _**application root**_, not relative to the component file. 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](https://github.com/angular/angular-cli/wiki/stories-asset-configuration).

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.

External and global style files

When building with the CLI, you must configure the `.angular-cli.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.

{@a view-encapsulation}

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:

- `Native` 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.

- `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:

`Native` view encapsulation only works on browsers that have native support for shadow DOM (see Shadow DOM v0 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.

{@a inspect-generated-css}

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.

Dependency Injection

Dependency Injection is a powerful pattern for managing code dependencies. This cookbook explores many of the features of Dependency Injection (DI) in Angular. {@a toc}

See the of the code in this cookbook.

{@a app-wide-dependencies}

Application-wide dependencies

Register providers for dependencies used throughout the application in the root application component, `AppComponent`.

The following example shows importing and registering the `LoggerService`, `UserContext`, and the `UserService` in the `@Component` metadata `providers` array.

All of these services are implemented as classes. Service classes can act as their own providers which is why listing them in the `providers` array is all the registration you need.

A *provider* is something that can create or deliver a service. Angular creates a service instance from a class provider by using `new`. Read more about providers in the [Dependency Injection](guide/dependency-injection#register-providers-ngmodule) guide.

Now that you've registered these services, Angular can inject them into the constructor of *any* component or service, *anywhere* in the application.

{@a external-module-configuration}

External module configuration

Generally, register providers in the `NgModule` rather than in the root application component.

Do this when you expect the service to be injectable everywhere, or you are configuring another application global service *before the application starts*.

Here is an example of the second case, where the component router configuration includes a non-default location strategy by listing its provider in the `providers` list of the `AppModule`.

{@a injectable}

{@a nested-dependencies}

@Injectable() and nested service dependencies

The consumer of an injected service does not know how to create that service. It shouldn't care. It's the dependency injection's job to create and cache that service.

Sometimes a service depends on other services, which may depend on yet other services. Resolving these nested dependencies in the correct order is also the framework's job. At each step, the consumer of dependencies simply declares what it requires in its constructor and the framework takes over.

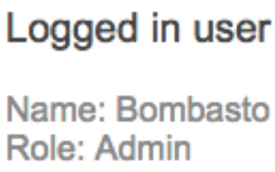
The following example shows injecting both the `LoggerService` and the `UserContext` in the `AppComponent`.

The `UserContext` in turn has its own dependencies on both the `LoggerService` and a `UserService` that gathers information about a particular user.

When Angular creates the `AppComponent`, the dependency injection framework creates an instance of the `LoggerService` and starts to create the `UserContextService`. The `UserContextService` needs the `LoggerService`, which the framework already has, and the `UserService`, which it has yet to create. The `UserService` has no dependencies so the dependency injection framework can just use `new` to instantiate one.

The beauty of dependency injection is that `AppComponent` doesn't care about any of this. You simply declare what is needed in the constructor (`LoggerService` and `UserContextService`) and the framework does the rest.

Once all the dependencies are in place, the `AppComponent` displays the user information:



Logged in user

Name: Bombasto
Role: Admin

{@a injectable-1}

@Injectable()

Notice the `@Injectable()` decorator on the `UserContextService` class.

That decorator makes it possible for Angular to identify the types of its two dependencies, `LoggerService` and `UserService`.

Technically, the `@Injectable()` decorator is only required for a service class that has *its own dependencies*. The `LoggerService` doesn't depend on anything. The logger would work if you omitted `@Injectable()` and the generated code would be slightly smaller.

But the service would break the moment you gave it a dependency and you'd have to go back and add `@Injectable()` to fix it. Add `@Injectable()` from the start for the sake of consistency and to avoid future pain.

Although this site recommends applying \`@Injectable()\` to all service classes, don't feel bound by it. Some developers prefer to add it only where needed and that's a reasonable policy too.
The \`AppComponent\` class had two dependencies as well but no \`@Injectable()\`. It didn't need \`@Injectable()\` because that component class has the \`@Component\` decorator. In Angular with TypeScript, a *single* decorator—*any* decorator—is sufficient to identify dependency types.

{@a service-scope}

Limit service scope to a component subtree

All injected service dependencies are singletons meaning that, for a given dependency injector, there is only one instance of service.

But an Angular application has multiple dependency injectors, arranged in a tree hierarchy that parallels the component tree. So a particular service can be *provided* and created at any component level and multiple times if provided in multiple components.

By default, a service dependency provided in one component is visible to all of its child components and Angular injects the same service instance into all child components that ask for that service.

Accordingly, dependencies provided in the root `AppComponent` can be injected into *any* component *anywhere* in the application.

That isn't always desirable. Sometimes you want to restrict service availability to a particular region of the application.

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 similar providing a service to a sub-root component is to providing a service in the root `AppComponent`. The syntax is the same. Here, the `HeroService` is available to the `HeroesBaseComponent` because it is in the `providers` array:

When Angular creates the `HeroesBaseComponent`, it also creates a new instance of `HeroService` that is visible only to the component and its children, if any.

You could also provide the `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 the `HeroBiosComponent`, `HeroOfTheMonthComponent`, and `HeroesBaseComponent`. Each of these components has its own `HeroService` instance managing its own independent collection of heroes. ### Take a break! This much Dependency Injection knowledge may be all that many Angular developers ever need to build their applications. It doesn't always have to be more complicated.

{@a multiple-service-instances}

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.

{@a hero-bios-component} Imagine a `HeroBiosComponent` that presents three instances of the `HeroBioComponent`.

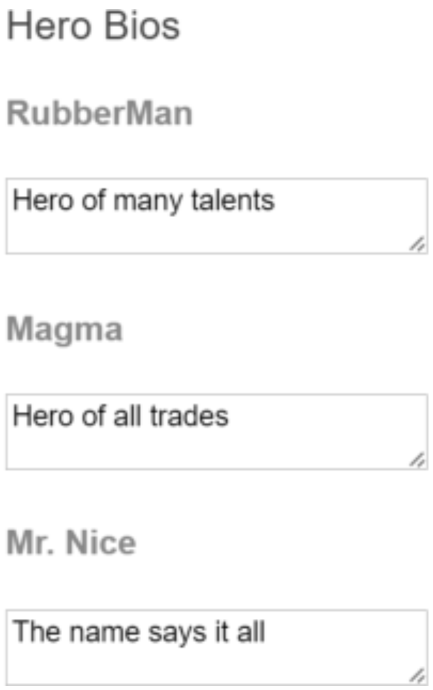
Each `HeroBioComponent` can edit a single hero's biography. A `HeroBioComponent` relies on a `HeroCacheService` to fetch, cache, and perform other persistence operations on that hero.

Clearly the three instances of the `HeroBioComponent` can't share the same `HeroCacheService`. They'd be competing with each other to determine which hero to cache.

Each `HeroBioComponent` gets its *own* `HeroCacheService` instance by listing the `HeroCacheService` in its metadata `providers` array.

The parent `HeroBiosComponent` binds a value to the `heroId`. The `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. And the template displays this data-bound property.

Find this example in live code 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

{@a optional}

{@a qualify-dependency-lookup}

Qualify dependency lookup with *@Optional()* and `@Host()`

As you now know, 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.

You *want* this behavior most of the time. But sometimes you need to limit the search and/or accommodate a missing dependency. You can modify Angular's search behavior with the `@Host` and `@Optional` qualifying decorators, used individually or together.

The `@Optional` decorator tells Angular to continue when it can't find the dependency. Angular sets the injection parameter to `null` instead.

The `@Host` decorator stops the upward search at the *host component*.

The host component is typically the component requesting the dependency. But when this component is projected into a *parent* component, that parent component becomes the host. The next example covers this second case.

{@a demonstration}

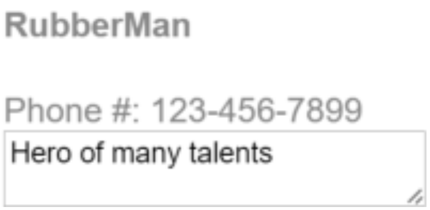
Demonstration

The `HeroBiosAndContactsComponent` is a revision of the `HeroBiosComponent` that you looked at above.

Focus on the template:

Now there is 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:

It looks like this, with the hero's telephone number from `HeroContactComponent` projected above the hero description:



RubberMan

Phone #: 123-456-7899

Hero of many talents

Here's the `HeroContactComponent` which demonstrates the qualifying decorators:

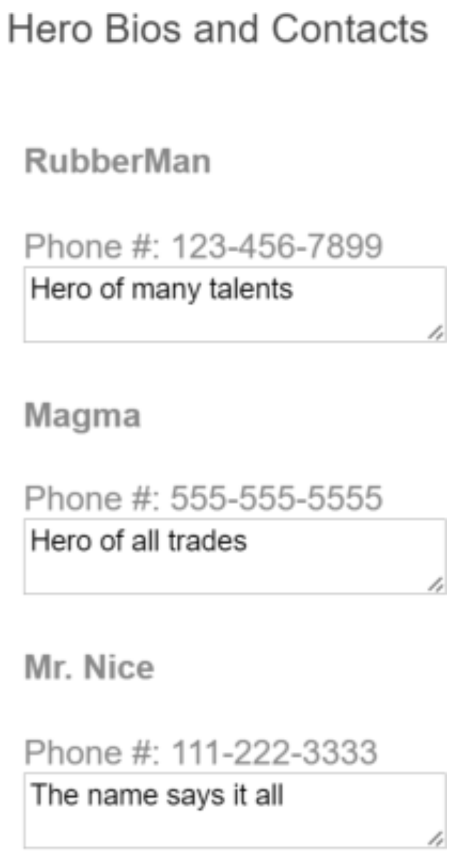
Focus on the constructor parameters:

The `@Host()` function decorating the `heroCache` 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 happens to have it.

A second `@Host()` function decorates the `loggerService` 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 would throw an error if you hadn't also decorated the property with the `@Optional()` function. Thanks to `@Optional()` , Angular sets the `loggerService` to null and the rest of the component adapts.

Here's the `HeroBiosAndContactsComponent` in action.



Hero Bios and Contacts

RubberMan

Phone #: 123-456-7899

Hero of many talents

Magma

Phone #: 555-555-5555

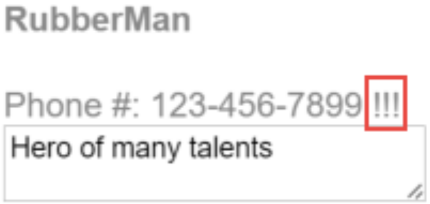
Hero of all trades

Mr. Nice

Phone #: 111-222-3333

The name says it all

If you comment out the `@Host()` decorator, Angular now 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 gratuitous "!!!", indicating that the logger was found.



RubberMan

Phone #: 123-456-7899!!!

Hero of many talents

On the other hand, if you restore the `@Host()` decorator and comment out `@Optional`, the application fails for lack of the required logger at the host component level.
`EXCEPTION: No provider for LoggerService! (HeroContactComponent -> LoggerService)`
{@a component-element}

Inject the component's DOM element

On occasion you might need to access a component's corresponding DOM element. Although developers strive to avoid it, many visual effects and 3rd party tools, such as jQuery, require DOM access.

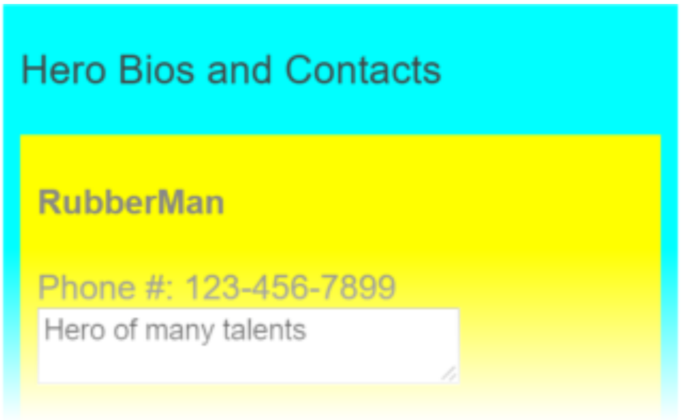
To illustrate, here's a simplified version of the `HighlightDirective` from the [Attribute Directives](#) page.

The directive sets the background to a highlight color when the user mouses over the DOM element to which it is applied.

Angular sets the constructor's `el` parameter to the injected `ElementRef`, which is a wrapper around that DOM element. Its `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.

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

{@a providers}

Define dependencies with providers

This section demonstrates how to write providers that deliver dependent services.

Get a service from a dependency injector by giving it a ***token***.

You usually let Angular handle 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. Here's a typical example:

Angular asks the injector for the service associated with the `LoggerService` and assigns the returned value to the `logger` parameter.

Where did the injector get that value? It may already have that value in its internal container. If it doesn't, it may be able to make one with the help of a ***provider***. A *provider* is a recipe for delivering a service associated with a *token*.

If the injector doesn't have a provider for the requested *token*, it delegates the request to its parent injector, where the process repeats until there are no more injectors. If the search is futile, the injector throws an error—unless the request was [optional](guide/dependency-injection-in-action#optional).

A new injector has no providers. Angular initializes the injectors it creates with some providers it cares about. You have to register your *own* application providers manually, usually in the `providers` array of the `Component` or `Directive` metadata:

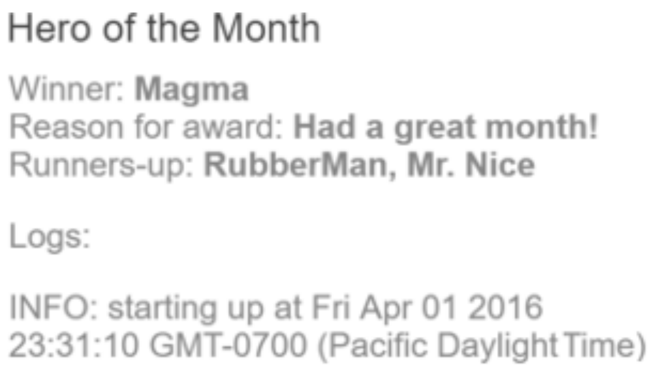
{@a defining-providers}

Defining providers

The simple class provider is the most typical by far. You mention the class in the `providers` array and you're done.

It's that simple because the most common injected service is an instance of a class. But not every dependency can be satisfied by creating a new instance of a class. You need other ways to deliver dependency values and that means you need other ways to specify a provider.

The `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 gives you plenty to think about.

{@a provide}

The *provide* object literal

The `provide` object literal takes a *token* and a *definition object*. The *token* is usually a class but it doesn't have to be.

The *definition* object has a required property that specifies how to create the singleton instance of the service. In this case, the property.

{@a usevalue}

useValue—the *value provider*

Set the `useValue` property to a ***fixed value*** that the provider can return as the service instance (AKA, the "dependency object").

Use this technique to provide *runtime configuration constants* such as website base addresses and feature flags. You can use a *value provider* in a unit test to replace a production service with a fake or mock.

The `HeroOfTheMonthComponent` example has two *value providers*. The first provides an instance of the `Hero` class; the second specifies a literal string resource:

The `Hero` provider token is a class which makes sense because the value is a `Hero` and the consumer of the injected hero would want the type information.

The `TITLE` provider token is *not a class*. It's a special kind of provider lookup key called an [InjectionToken](). You can use an `InjectionToken` for any kind of provider but it's particular 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 *now*. You can't create the value later. Obviously the title string literal is immediately available. The `someHero` variable in this example was set earlier in the file:

The other providers create their values *lazily* when they're needed for injection.

{@a useclass}

useClass—the *class provider*

The `useClass` provider creates and returns new instance of the specified class.

Use this technique to ***substitute an alternative implementation*** for a common or default class. The alternative could implement a different strategy, extend the default class, or fake the behavior of the real class in a test case.

Here are two examples in the `HeroOfTheMonthComponent`:

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. It's in this long form to de-mystify the preferred short form.

The second provider substitutes the `DateLoggerService` for the `LoggerService`. The `LoggerService` is already registered at the `AppComponent` level. When *this component* requests the

`LoggerService` , it receives the `DateLoggerService` instead.

This component and its tree of child components receive the `DateLoggerService` instance. Components outside the tree continue to receive the original `LoggerService` instance.

The `DateLoggerService` inherits from `LoggerService` ; it appends the current date/time to each message:

{@a useexisting}

useExisting—the *alias provider*

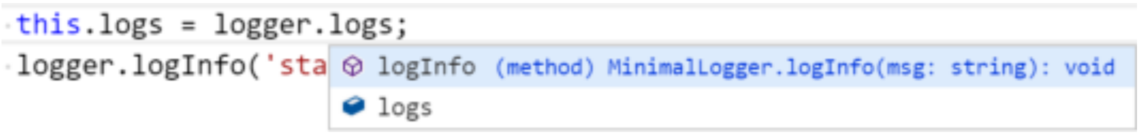
The `useExisting` provider maps one token to another. In effect, the first token is an ***alias*** for the service associated with the second token, creating ***two ways to access the same service object***.

Narrowing an API through an aliasing interface is *one* important use case for this technique. The following example shows aliasing for that purpose.

Imagine that the `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. Here the `MinimalLogger` [*class-interface*]() reduces the API to two members:

Now put it to use in a simplified version of the `HeroOfTheMonthComponent` .

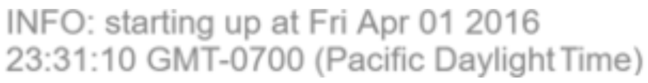
The `HeroOfTheMonthComponent` constructor's `logger` parameter is typed as `MinimalLogger` so only the `logs` and `logInfo` members are visible in a TypeScript-aware editor:



```
this.logs = logger.logs;
logger.logInfo('sta
```

Behind the scenes, Angular actually sets the `logger` parameter to the full service registered under the `LoggingService` token which happens to be the `DateLoggerService` that was [provided above]().

The following image, which displays the logging date, confirms the point:



```
INFO: starting up at Fri Apr 01 2016
23:31:10 GMT-0700 (Pacific Daylight Time)
```

{@a usefactory}

useFactory—the *factory provider*

The `useFactory` provider creates a dependency object by calling a factory function as in this example.

Use this technique to ***create a dependency object*** with a factory function whose inputs are some ***combination of injected services and local state***.

The *dependency object* doesn't have to be a class instance. It could be anything. In this example, the *dependency object* is a string of the names of the runners-up to the "Hero of the Month" contest.

The local state is the number `2` , the number of runners-up this component should show. It executes `runnersUpFactory` immediately with `2` .

The `runnersUpFactory` itself isn't the provider factory function. The true provider factory function is the function that `runnersUpFactory` returns.

That returned 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 two `deps` values are *tokens* that the injector uses to provide these factory function dependencies.

After some undisclosed work, the function returns the string of names and Angular injects it into the `runnersUp` parameter of the `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 for the full source code.

{@a tokens}

Provider token alternatives: the *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 what you usually call the *service*.

But the 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. {@a class-interface}

class-interface

The previous *Hero of the Month* example used the `MinimalLogger` class as the token for a provider of a `LoggerService` .

The `MinimalLogger` is an abstract class.

You usually inherit from an abstract class. But *no class* in this application inherits from `MinimalLogger`.

The `LoggerService` and the `DateLoggerService` *could* have inherited from `MinimalLogger`. They could have *implemented* it instead in the manner of an interface. But they did neither. The `MinimalLogger` is used exclusively as a dependency injection token.

When you use a class this way, it's called a ***class-interface***. The key benefit of a *class-interface* is that you can get the strong-typing of an interface and you can ***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.

#### Why *MinimalLogger* is a class and not a TypeScript interface You can't use an interface as a provider token because interfaces are not JavaScript objects. They exist only in the TypeScript design space. They disappear after the code is transpiled to JavaScript. A provider token must be a real JavaScript object of some kind: such as a function, an object, a string, or a class. Using a class as an interface gives you the characteristics of an interface in a real JavaScript object. Of course a real object occupies memory. To minimize memory cost, the class should have *no implementation*. The `MinimalLogger` transpiles to this unoptimized, pre-minified JavaScript for a constructor function: Notice that it doesn't have a single member. 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.

{@a injection-token}

InjectionToken

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.

The `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.

You created the `TITLE` token like this:

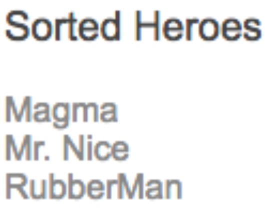
The type parameter, while optional, conveys the dependency's type to developers and tooling. The token description is another developer aid.

{@a di-inheritance}

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` could stand on its own. It demands its own instance of the `HeroService` to get heroes and displays them in the order they arrive from the database.

Keep constructors simple. They 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.

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*.

{@a find-parent}

Find a parent component by injection

Application components often need to share information. More loosely coupled techniques such as data binding and service sharing are preferable. But sometimes it makes sense for one component to have a direct reference to another component perhaps to access values or call methods on that component.

Obtaining a component reference is a bit tricky in Angular. Although an Angular application is a tree of components, there is no public API for inspecting and traversing that tree.

There is an API for acquiring a child reference. Check out `Query`, `QueryList`, `ViewChildren`, and `ContentChildren` in the API Reference.

There is no public API for acquiring a parent reference. But 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.

{@a known-parent}

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`:

{@a alex}

Cathy reports whether or not she has access to *Alex* after injecting an `AlexComponent` into her constructor:

Notice that even though the @Optional qualifier is there for safety, the confirms that the `alex` parameter is set.

{@a base-parent}

Cannot 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`.

The `CraigComponent` tries to inject `Base` into its `alex` constructor parameter and reports if it succeeded.

Unfortunately, this does not work. The confirms that the `alex` parameter is null. *You cannot inject a parent by its base class.*

{@a class-interface-parent}

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.

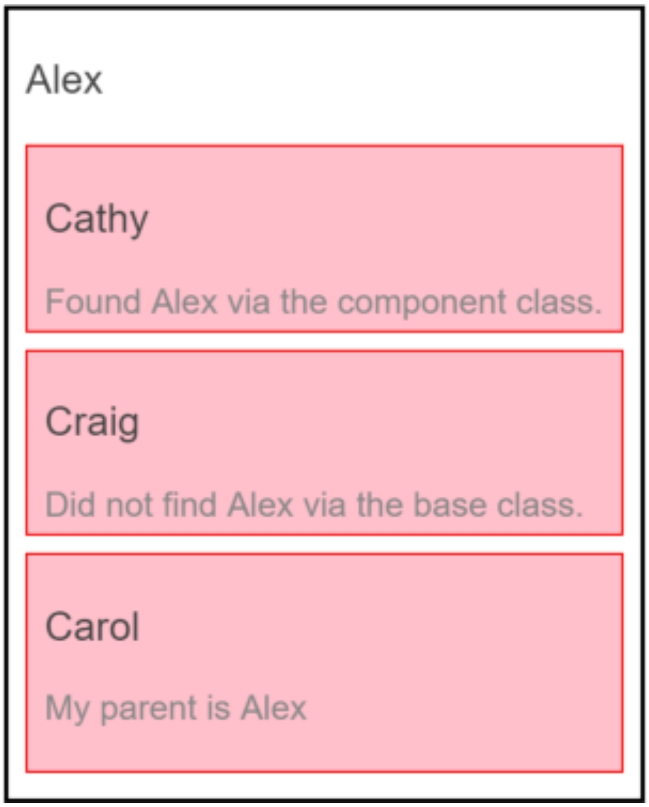
Write an *alias provider*—a `provide` object literal with a `useExisting` definition—that creates an *alternative* way to inject the same component instance and add that provider to the `providers` array of the `@Component` metadata for the `AlexComponent`:

{@a alex-providers}

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:

Here's *Alex* and family in action:



Alex

Cathy

Found Alex via the component class.

Craig

Did not find Alex via the base class.

Carol

My parent is Alex

{@a parent-tree}

Find the parent in a tree of parents 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*:

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:

It's identical to *Carol*'s constructor except for the additional `@SkipSelf` decorator.

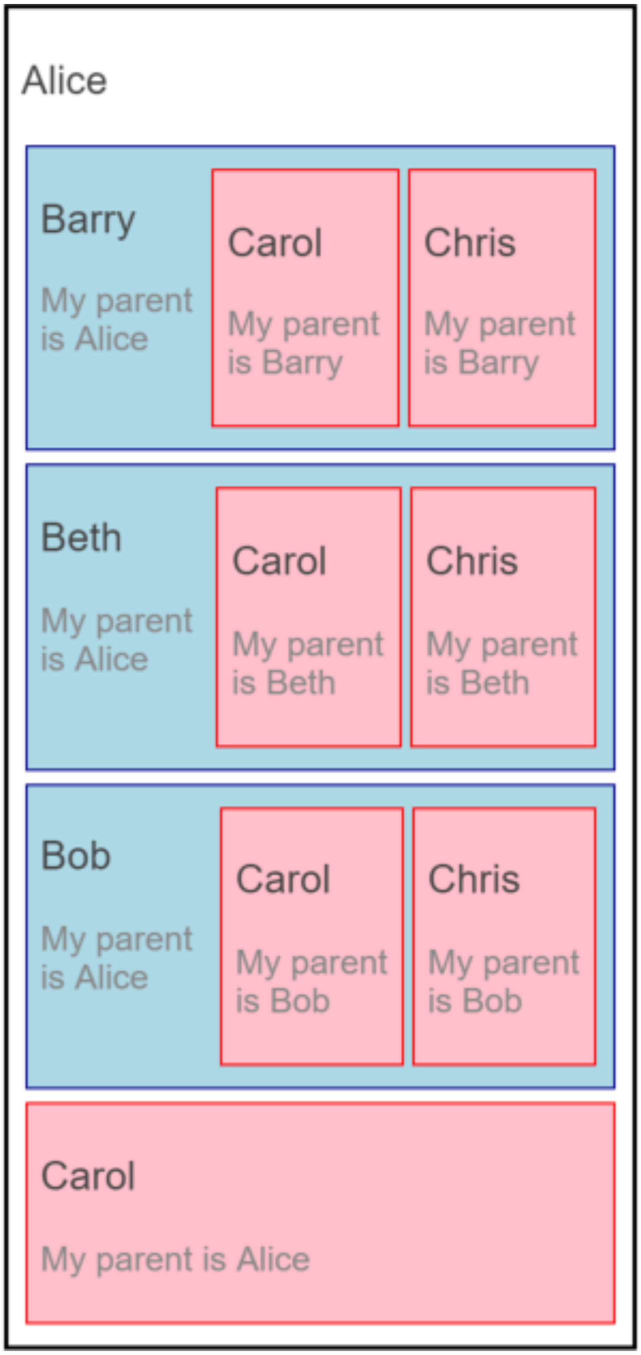
`@SkipSelf` is essential for two reasons:

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.

```
Cannot instantiate cyclic dependency! (BethComponent -> Parent -> BethComponent)
```

Here's *Alice*, *Barry* and family in action:



Alice

Barry

My parent is Alice

Carol

My parent is Barry

Chris

My parent is Barry

Beth

My parent is Alice

Carol

My parent is Beth

Chris

My parent is Beth

Bob

My parent is Alice

Carol

My parent is Bob

Chris

My parent is Bob

Carol

My parent is Alice

{@a parent-token}

The *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*.

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:

Doing so adds clarity to the code. But it's not technically necessary. Although the `AlexComponent` has a `name` property, as required by its `Base` class, its class signature doesn't mention `Parent`:

The \`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

{@a provideparent}

A *provideParent()* helper function

Writing variations of the same parent *alias provider* gets old quickly, especially this awful mouthful with a *forwardRef*:

You can extract that logic into a helper function like this:

Now you can add a simpler, more meaningful parent provider to your components:

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*.

And here's how you could use it with a different parent type:

{@a forwardref}

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 the `AlexComponent` in its `providers` array. The `providers` array is a property of the `@Component` decorator function which must appear *above* the class definition.

Break the circularity with `forwardRef`:

The Dependency Injection pattern

Dependency injection is an important application design pattern. It's used so widely that almost everyone just calls it *DI*.

Angular has its own dependency injection framework, and you really can't build an Angular application without it.

This page covers what DI is and why it's useful.

When you've learned the general pattern, you're ready to turn to the Angular Dependency Injection guide to see how it works in an Angular app.

{@a why-di }

Why dependency injection?

To understand why dependency injection is so important, consider an example without it. Imagine writing the following code:

The `Car` class creates everything it needs inside its constructor. What's the problem? The problem is that the `Car` class is brittle, inflexible, and hard to test.

This `Car` needs an engine and tires. Instead of asking for them, the `Car` constructor instantiates its own copies from the very specific classes `Engine` and `Tires`.

What if the `Engine` class evolves and its constructor requires a parameter? That would break the `Car` class and it would stay broken until you rewrote it along the lines of `this.engine = new Engine(theNewParameter)`. The `Engine` constructor parameters weren't even a consideration when you first wrote `Car`. You may not anticipate them even now. But you'll *have* to start caring because when the definition of `Engine` changes, the `Car` class must change. That makes `Car` brittle.

What if you want to put a different brand of tires on your `Car`? Too bad. You're locked into whatever brand the `Tires` class creates. That makes the `Car` class inflexible.

Right now each new car gets its own `engine`. It can't share an `engine` with other cars. While that makes sense for an automobile engine, surely you can think of other dependencies that should be shared, such as the onboard wireless connection to the manufacturer's service center. This `Car` lacks the flexibility to share services that have been created previously for other consumers.

When you write tests for `Car` you're at the mercy of its hidden dependencies. Is it even possible to create a new `Engine` in a test environment? What does `Engine` depend upon? What does that dependency depend on? Will a new instance of `Engine` make an asynchronous call to the server? You certainly don't want that going on during tests.

What if the `Car` should flash a warning signal when tire pressure is low? How do you confirm that it actually does flash a warning if you can't swap in low-pressure tires during the test?

You have no control over the car's hidden dependencies. When you can't control the dependencies, a class becomes difficult to test.

How can you make `Car` more robust, flexible, and testable?

{@a ctor-injection} That's super easy. Change the `Car` constructor to a version with DI:

See what happened? The definition of the dependencies are now in the constructor. The `Car` class no longer creates an `engine` or `tires`. It just consumes them.

This example leverages TypeScript's constructor syntax for declaring parameters and properties simultaneously.

Now you can create a car by passing the engine and tires to the constructor.

How cool is that? The definition of the `engine` and `tire` dependencies are decoupled from the `Car` class. You can pass in any kind of `engine` or `tires` you like, as long as they conform to the general API requirements of an `engine` or `tires`.

Now, if someone extends the `Engine` class, that is not `Car`'s problem.

The _consumer_ of `Car` has the problem. The consumer must update the car creation code to something like this: The critical point is this: the `Car` class did not have to change. You'll take care of the consumer's problem shortly.

The `Car` class is much easier to test now because you are in complete control of its dependencies. You can pass mocks to the constructor that do exactly what you want them to do during each test:

You just learned what dependency injection is.

It's a coding pattern in which a class receives its dependencies from external sources rather than creating them itself.

Cool! But what about that poor consumer? Anyone who wants a `Car` must now create all three parts: the

`Car`, `Engine`, and `Tires`. The `Car` class shed its problems at the consumer's expense. You need something that takes care of assembling these parts.

You *could* write a giant class to do that:

It's not so bad now with only three creation methods. But maintaining it will be hairy as the application grows. This factory is going to become a huge spiderweb of interdependent factory methods!

Wouldn't it be nice if you could simply list the things you want to build without having to define which dependency gets injected into what?

This is where the dependency injection framework comes into play. Imagine the framework had something called an *injector*. You register some classes with this injector, and it figures out how to create them.

When you need a `Car`, you simply ask the injector to get it for you and you're good to go.

Everyone wins. The `Car` knows nothing about creating an `Engine` or `Tires`. The consumer knows nothing about creating a `Car`. You don't have a gigantic factory class to maintain. Both `Car` and consumer simply ask for what they need and the injector delivers.

This is what a **dependency injection framework** is all about.

Now that you know what dependency injection is and appreciate its benefits, turn to the Angular Dependency Injection guide to see how it is implemented in Angular.

Angular Dependency Injection

Dependency Injection (DI) is a way to create objects that depend upon other objects. A Dependency Injection system supplies the dependent objects (called the *dependencies*) when it creates an instance of an object.

The Dependency Injection pattern page describes this general approach. *The guide you're reading now* explains how Angular's own Dependency Injection system works.

DI by example

You'll learn Angular Dependency Injection through a discussion of the sample app that accompanies this guide. Run the anytime.

Start by reviewing this simplified version of the *heroes* feature from the The Tour of Heroes.

The `HeroesComponent` is the top-level heroes component. It's only purpose is to display the `HeroListComponent` which displays a list of hero names.

This version of the `HeroListComponent` gets its `heroes` from the `HEROES` array, an in-memory collection defined in a separate `mock-heroes` file.

That may suffice in the early stages of development, but it's far from ideal. As soon as you try to test this component or get heroes from a remote server, you'll have to change the implementation of `HerosListComponent` and replace every other use of the `HEROES` mock data.

It's better to hide these details inside a *service* class, defined in its own file.

Create an injectable *HeroService*

The **Angular CLI** can generate a new `HeroService` class in the `src/app/heroes` folder with this command.

ng generate service heroes/hero

That command creates the following `HeroService` skeleton.

Assume for now that the `@Injectable` decorator is an essential ingredient in every Angular service definition. The rest of the class has been rewritten to expose a `getHeroes` method that returns the same mock data as before.

Of course, this isn't a real data service. If the app were actually getting data from a remote server, the `getHeroes` method signature would have to be asynchronous.

That's a defect we can safely ignore in this guide where our focus is on *injecting the service* into the `HeroList` component.

{@a injector-config} {@a bootstrap}

Register a service provider

A *service* is just a class in Angular until you register it with an Angular dependency injector.

An Angular injector is responsible for creating service instances and injecting them into classes like the `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.

You do have to register *providers* with an injector before the injector can create that service.

Providers tell the injector *how to create the service*. Without a provider, the injector would not know that it is responsible for injecting the service nor be able to create the service.

You'll learn much more about _providers_ [below](#providers). For now it is sufficient to know that they create services and must be registered with an injector.

You can register a provider with any Angular decorator that supports the **`providers` array property**.

Many Angular decorators accept metadata with a `providers` property. The two most important examples are `@Component` and `@NgModule`.

{@a register-providers-component}

@Component providers

Here's a revised `HeroesComponent` that registers the `HeroService` in its `providers` array.

{@a register-providers-ngmodule}

@NgModule providers

In the following excerpt, the root `AppModule` registers two providers in its `providers` array.

The first entry registers the `UserService` class (*not shown*) under the `UserService` *injection token*. The second registers a value (`HERO_DI_CONFIG`) under the `APP_CONFIG` *injection token*.

Thanks to these registrations, Angular can inject the `UserService` or the `HERO_DI_CONFIG` value into any class that it creates.

You'll learn about _injection tokens_ and _provider_ syntax [below](#providers).

{@a ngmodule-vs-comp}

@NgModule or *@Component*?

Should you register a service with an Angular module or with a component? The two choices lead to differences in service *scope* and service *lifetime*.

Angular module providers (`@NgModule.providers`) are registered with the application's root injector. Angular can inject the corresponding services in any class it creates. Once created, a service instance lives for the life of the app and Angular injects this one service instance in every class that needs it.

You're likely to inject the `UserService` in many places throughout the app and will want to inject the same service instance every time. Providing the `UserService` with an Angular module is a good choice.

To be precise, Angular module providers are registered with the root injector _unless the module is_ [lazy loaded](guide/ngmodule#lazy-load-DI). In this sample, all modules are _eagerly loaded_ when the application starts, so all module providers are registered with the app's root injector.

A component's providers (`@Component.providers`) are registered with 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 cannot inject the same service instance anywhere else. Note that a component-provided service may have a limited lifetime. Each new instance of the component gets its own instance of the service and, when the component instance is destroyed, so is that service instance. In this sample app, the `HeroComponent` is created when the application starts and is never destroyed so the `HeroService` created for the `HeroComponent` also live for the life of the app. If you want to restrict `HeroService` access to the `HeroComponent` and its nested `HeroListComponent`, providing the `HeroService` in the `HeroComponent` may be a good choice.
The scope and lifetime of component-provided services is a consequence of [the way Angular creates component instances](#component-child-injectors).

Inject a service

The `HeroListComponent` should get heroes from the `HeroService`.

The component shouldn't create the `HeroService` with `new`. It should ask for the `HeroService` to be injected.

You can tell Angular to inject a dependency in the component's constructor by specifying a **constructor parameter with the dependency type**. Here's the `HeroListComponent` constructor, asking for the `HeroService` to be injected.

Of course, the `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.

Notice that the `HeroListComponent` doesn't know where the `HeroService` comes from. *You* know that it comes from the parent `HeroesComponent`. But if you decided instead to provide the `HeroService` in the `AppModule`, the `HeroListComponent` wouldn't change at all. The *only thing that matters* is that the `HeroService` is provided in some parent injector.

{@a singleton-services}

Singleton services

Services are singletons *within the scope of an injector*. There is at most one instance of a service in a given injector.

There is only one root injector and the `UserService` is registered with that injector. Therefore, there can be just one `UserService` instance in the entire app and every class that injects `UserService` get this service instance.

However, Angular DI is a hierarchical injection system, which means that nested injectors can create their own service instances. Angular creates nested injectors all the time.

{@a component-child-injectors}

Component child injectors

For example, when Angular creates a new instance of a component that has `@Component.providers`, it also creates a new *child injector* for that instance.

Component injectors are independent of each other and each of them creates its own instances of the component-provided services.

When Angular destroys one of these component instance, it also destroys the component's 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 could inject a `HeroListComponent` with both the `HeroService` provided in `HeroComponent` and the `UserService` provided in `AppModule`.

{@a testing-the-component}

Testing the component

Earlier you saw that designing a class for 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:

Learn more in the [Testing](guide/testing) guide.

{@a service-needs-service}

When the service needs a service

The `HeroService` is very simple. It doesn't have any dependencies of its own.

What if it had a dependency? What if it reported its activities through a logging service? You'd apply the same *constructor injection* pattern, adding a constructor that takes a `Logger` parameter.

Here is the revised `HeroService` that injects the `Logger`, side-by-side with the previous service for comparison.

The constructor asks for an injected instance of a `Logger` and stores it in a private field called `logger`. The `getHeroes()` method logs a message when asked to fetch heroes.

{@a logger-service}

The dependent *Logger* service

The sample app's `Logger` service is quite simple:

If the app didn't provide this `Logger`, Angular would throw an exception when it looked for a `Logger` to inject into the `HeroService`.

ERROR Error: No provider for Logger!

Because a singleton logger service is useful everywhere, it's provided in the root `AppModule`.

{@a injectable}

@Injectable()

The [@Injectable()] decorator identifies a service class that *might* require injected dependencies.

The `HeroService` must be annotated with `@Injectable()` because it requires an injected `Logger`.

Always write `@Injectable()` with parentheses, not just `@Injectable`.

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 right 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*. While *any* decorator will do, 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. It 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` It's your job to put `@Injectable()` on your service classes.

The `Logger` service is annotated with `@Injectable()` decorator too, although it has no constructor and no dependencies.

In fact, *every* Angular service class in this app is annotated with the `@Injectable()` decorator, whether or not it has a constructor and dependencies. `@Injectable()` is a required coding style for services.

{@a providers}

Providers

A service provider *provides* the concrete, runtime version of a dependency value. The injector relies on **providers** to create instances of the services that the injector injects into components, directives, pipes, and other services.

You must register a service *provider* with an injector, or it won't know how to create the service.

The next few sections explain the many ways you can specify a provider.

Almost all of the accompanying code snippets are extracts from the sample app's `providers.component.ts` file.

The class as its own provider

There are many ways to *provide* something that looks and behaves like a `Logger`. The `Logger` class itself is an obvious and natural provider.

But it's not the only way.

You can configure the injector with alternative providers that can deliver an object that behaves like a `Logger`. You could provide a substitute class. You could provide a logger-like object. You could give it a provider that calls a logger factory function. Any of these approaches might be a good choice under the right circumstances.

What matters is that the injector has a provider to go to when it needs a `Logger`.

{@a provide}

The *provide* object literal

Here's the class-provider syntax again.

This is actually a shorthand expression for a provider registration using a *provider* object literal with two properties:

The `provide` property holds the [token](#) that serves as the key for both locating a dependency value and registering the provider.

The second property is always a provider definition object, which you can think of as a *recipe* for creating the dependency value. There are many ways to create dependency values just as there are many ways to write a recipe.

{@a class-provider}

Alternative class providers

Occasionally you'll ask a different class to provide the service. The following code tells the injector to return a `BetterLogger` when something asks for the `Logger`.

{@a class-provider-dependencies}

Class provider with dependencies

Maybe an `EvenBetterLogger` could display the user name in the log message. This logger gets the user from the injected `UserService`, which is also injected at the application level.

Configure it like `BetterLogger`.

{@a aliased-class-providers}

Aliased class providers

Suppose an old component depends upon an `OldLogger` class. `OldLogger` has the same interface as the `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'd like the singleton instance of `NewLogger` to handle it instead.

The dependency injector should inject that singleton instance when a component asks for either the new or the old logger. The `OldLogger` should be an alias for `NewLogger`.

You certainly do not want two different `NewLogger` instances in your app. Unfortunately, that's what you get if you try to alias `OldLogger` to `NewLogger` with `useClass`.

The solution: alias with the `useExisting` option.

{@a value-provider}

Value providers

Sometimes it's easier to provide a ready-made object rather than ask the injector to create it from a class.

Then you register a provider with the `useValue` option, which makes this object play the logger role.

See more `useValue` examples in the Non-class dependencies and InjectionToken sections.

{@a factory-provider}

Factory providers

Sometimes you need to create the dependent value dynamically, based on information you won't have until the last possible moment. Maybe the information changes repeatedly in the course of the browser session.

Suppose also that the injectable service has no independent access to the source of this information.

This situation calls for a **factory provider**.

To illustrate the point, add a new business requirement: the `HeroService` must hide *secret* heroes from normal users. Only authorized users should see secret heroes.

Like the `EvenBetterLogger` , the `HeroService` needs a fact about the user. It 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.

Unlike `EvenBetterLogger` , you can't inject the `UserService` into the `HeroService` . The `HeroService` won't have direct access to the user information to decide who is authorized and who is not.

Instead, the `HeroService` constructor takes a boolean flag to control display of secret heroes.

You can inject the `Logger` , but you can't inject the boolean `isAuthorized` . You'll have to take over the creation of new instances of this `HeroService` with a factory provider.

A factory provider needs a factory function:

Although the `HeroService` has no access to the `UserService` , the factory function does.

You inject both the `Logger` and the `UserService` into the factory provider and let the injector pass them along to the factory function:

The \`useFactory\` field tells Angular that the provider is a factory function whose implementation is the \`heroServiceFactory\`. The \`deps\` property is an array of [provider tokens](guide/dependency-injection#token). The \`Logger\` and \`UserService\` classes serve as tokens for their own class providers. The injector resolves these tokens and injects the corresponding services into the matching 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 register the `HeroService` with this variable wherever you need it.

In this sample, you need it only in the `HeroesComponent` , where it replaces the previous `HeroService` registration in the metadata `providers` array. Here you see the new and the old implementation side-by-side:

{@a token}

Dependency injection tokens

When you register a provider with an injector, you associate that provider with a dependency injection 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 all previous examples, the dependency value has been a class *instance*, and the class *type* served as its own lookup key. Here you get a `HeroService` directly from the injector by supplying the `HeroService` type as the token:

You have similar good fortune 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:

This is especially convenient when you consider that most dependency values are provided by classes.

{@a non-class-dependencies}

Non-class dependencies

What if the dependency value isn't a class? Sometimes the thing you want to inject is a string, function, or object.

Applications often define configuration objects with lots of small facts (like the title of the application or the address of a web API endpoint) but these configuration objects aren't always instances of a class. They can be object literals such as this one:

What if you'd like to make this configuration object available for injection? You know you can register an object with a value provider.

But what should you use as the token? You don't have a class to serve as a token. There is no `AppConfig` class.

### TypeScript interfaces aren't valid tokens The `HERO_DI_CONFIG` constant conforms to the `AppConfig` interface. Unfortunately, you cannot use a TypeScript interface as a token: That seems strange if you're used to dependency injection in strongly typed languages, where an interface is the preferred dependency lookup key. It's not Angular's doing. An interface is a TypeScript design-time artifact. JavaScript doesn't have interfaces. The TypeScript interface disappears from the generated JavaScript. There is no interface type information left for Angular to find at runtime.

{@a injection-token}

InjectionToken

One solution to choosing a provider token for non-class dependencies is to define and use an *InjectionToken*. The definition of such a token looks like this:

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:

Now you can inject the configuration object into any constructor that needs it, with the help of an `@Inject` decorator:

Although the `AppConfig` interface plays no role in dependency injection, it supports typing of the configuration object within the class.

Alternatively, you can provide and inject the configuration object in an ngModule like `AppModule`.

{@a optional}

Optional dependencies

The `HeroService` *requires* a `Logger`, but what if it could get by without a `logger`? You can tell Angular that the dependency is optional by annotating the constructor argument with `@Optional()`:

When using `@Optional()`, your code must be prepared for a null value. If you don't register a `logger` somewhere up the line, the injector will set the value of `logger` to null.

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.

Angular dependency injection is more capable than this guide has described. You can learn more about its advanced features, beginning with its support for nested injectors, in Hierarchical Dependency Injection.

{@a explicit-injector}

Appendix: Working with injectors directly

Developers rarely work directly with an injector, but here's an `InjectorComponent` that does.

An `Injector` is itself an injectable service.

In this example, Angular injects the component's own `Injector` into the component's constructor. The component then asks the injected injector for the services it wants in `ngOnInit()`.

Note that the services themselves are not injected into the component. They are retrieved by calling `injector.get()`.

The `get()` method throws an error if it can't resolve the requested service. You can call `get()` with a second parameter, which is the value to return if the service is not found. Angular can't find the service if it's not registered with this or any ancestor injector.

The technique is an example of the [service locator pattern](https://en.wikipedia.org/wiki/Service_locator_pattern). **Avoid** this technique unless you genuinely need it. It encourages a careless grab-bag approach such as you see here. It's difficult to explain, understand, and test. You can't know by inspecting the constructor what this class requires or what it will do. It could acquire services from any ancestor component, not just its own. You're forced to spelunk the implementation to discover what it does. Framework developers may take this approach when they must acquire services generically and dynamically.

{@a one-class-per-file}

Appendix: one class per file

Having multiple classes in the same file is confusing and best avoided. Developers expect one class per file. Keep them happy.

If you combine the `HeroService` class with the `HeroesComponent` in the same file, **define the component last**. If you define the component before the service, you'll get a runtime null reference error.

You actually can define the component first with the help of the `forwardRef()` method as explained in this [blog post](http://blog.thoughtram.io/angular/2015/09/03/forward-references-in-angular-2.html). But it's best to avoid the problem altogether by defining components and services in separate files.

Deployment

This page describes techniques for deploying your Angular application to a remote server.

{@a dev-deploy} {@a copy-files}

Simplest deployment possible

For the simplest deployment, build for development and copy the output directory to a web server.

1. Start with the development build

   ```
   ng build
   ```

2. Copy *everything* within the output folder (`dist/` by default) to a folder on the server.

3. If you copy the files into a server *sub-folder*, append the build flag, `--base-href` and set the `<base href>` appropriately.

   For example, if the `index.html` is on the server at `/my/app/index.html` , set the *base href* to `<base href="/my/app/">` like this.

   ```
   ng build --base-href=/my/app/
   ```

   You'll see that the `<base href>` is set properly in the generated `dist/index.html` .

   If you copy to the server's root directory, omit this step and leave the `<base href>` alone.

   Learn more about the role of `<base href>` below.

4. Configure the server to redirect requests for missing files to `index.html` . Learn more about server-side redirects below.

This is *not* a production deployment. It's not optimized and it won't be fast for users. It might be good enough for sharing your progress and ideas internally with managers, teammates, and other stakeholders.

{@a optimize}

Optimize for production

Although deploying directly from the development environment works, you can generate an optimized build with additional CLI command line flags, starting with `--prod`.

Build with *--prod*

```
ng build --prod
```

The `--prod` *meta-flag* engages the following optimization features.

- Ahead-of-Time (AOT) Compilation: pre-compiles Angular component templates.
- Production mode: deploys the production environment which enables *production mode*.
- Bundling: concatenates your many application and library files into a few bundles.
- Minification: removes excess whitespace, comments, and optional tokens.
- Uglification: rewrites code to use short, cryptic variable and function names.
- Dead code elimination: removes unreferenced modules and much unused code.

The remaining copy deployment steps are the same as before.

You may further reduce bundle sizes by adding the `build-optimizer` flag.

```
ng build --prod --build-optimizer
```

See the CLI Documentation for details about available build options and what they do.

{@a enable-prod-mode}

Enable production mode

Angular apps run in development mode by default, as you can see by the following message on the browser console:

```
Angular is running in the development mode. Call enableProdMode() to enable the production mode.
```

Switching to *production mode* can make it run faster by disabling development specific checks such as the dual change detection cycles.

Building for production (or appending the `--environment=prod` flag) enables *production mode* Look at the CLI-generated `main.ts` to see how this works.

{@a lazy-loading}

Lazy loading

You can dramatically reduce launch time by only loading the application modules that absolutely must be present when the app starts.

Configure the Angular Router to defer loading of all other modules (and their associated code), either by waiting until the app has launched or by *lazy loading* them on demand.

Don't eagerly import something from a lazy loaded module

It's a common mistake. You've arranged to lazy load a module. But you unintentionally import it, with a JavaScript `import` statement, in a file that's eagerly loaded when the app starts, a file such as the root `AppModule`. If you do that, the module will be loaded immediately.

The bundling configuration must take lazy loading into consideration. Because lazy loaded modules aren't imported in JavaScript (as just noted), bundlers exclude them by default. Bundlers don't know about the router configuration and won't create separate bundles for lazy loaded modules. You have to create these bundles manually.

The CLI runs the Angular Ahead-of-Time Webpack Plugin which automatically recognizes lazy loaded `NgModules` and creates separate bundles for them.

{@a measure}

Measure performance

You can make better decisions about what to optimize and how when you have a clear and accurate understanding of what's making the application slow. The cause may not be what you think it is. You can waste a lot of time and money optimizing something that has no tangible benefit or even makes the app slower. You should measure the app's actual behavior when running in the environments that are important to you.

The Chrome DevTools Network Performance page is a good place to start learning about measuring performance.

The WebPageTest tool is another good choice that can also help verify that your deployment was successful.

{@a inspect-bundle}

Inspect the bundles

The source-map-explorer tool is a great way to inspect the generated JavaScript bundles after a production build.

Install `source-map-explorer` :

```
npm install source-map-explorer --save-dev
```

Build your app for production *including the source maps*

```
ng build --prod --sourcemaps
```

List the generated bundles in the `dist/` folder.

```
ls dist/*.bundle.js
```

Run the explorer to generate a graphical representation of one of the bundles. The following example displays the graph for the *main* bundle.

```
node_modules/.bin/source-map-explorer dist/main.*.bundle.js
```

The `source-map-explorer` analyzes the source map generated with the bundle and draws a map of all dependencies, showing exactly which classes are included in the bundle.

Here's the output for the *main* bundle of the QuickStart.

{@a base-tag}

The `base` tag

The HTML *<base href="..."/>* specifies a base path for resolving relative URLs to assets such as images, scripts, and style sheets. For example, given the `<base href="/my/app/">` , the browser resolves a URL such as `some/place/foo.jpg` into a server request for `my/app/some/place/foo.jpg` . During navigation, the Angular router uses the *base href* as the base path to component, template, and module files.

See also the [*APP_BASE_HREF*](api/common/APP_BASE_HREF "API: APP_BASE_HREF") alternative.

In development, you typically start the server in the folder that holds `index.html` . That's the root folder and you'd add `<base href="/">` near the top of `index.html` because `/` is the root of the app.

But on the shared or production server, you might serve the app from a subfolder. For example, when the URL to load the app is something like `http://www.mysite.com/my/app/` , the subfolder is `my/app/` and you should add `<base href="/my/app/">` to the server version of the `index.html` .

When the `base` tag is mis-configured, the app fails to load and the browser console displays

`404 - Not Found` errors for the missing files. Look at where it *tried* to find those files and adjust the base tag appropriately.

build vs. *serve*

You'll probably prefer `ng build` for deployments.

The **ng build** command is intended for building the app and deploying the build artifacts elsewhere. The **ng serve** command is intended for fast, local, iterative development.

Both `ng build` and `ng serve` **clear the output folder** before they build the project. The `ng build` command writes generated build artifacts to the output folder. The `ng serve` command does not. It serves build artifacts from memory instead for a faster development experience.

The output folder is \`dist/\` by default. To output to a different folder, change the \`outDir\` in \`.angular-cli.json\`.

The `ng serve` command builds, watches, and serves the application from a local CLI development server.

The `ng build` command generates output files just once and does not serve them. The `ng build --watch` command will regenerate output files when source files change. This `--watch` flag is useful if you're building during development and are automatically re-deploying changes to another server.

See the CLI `build` topic for more details and options.

{@a server-configuration}

Server configuration

This section covers changes you may have make to the server or to files deployed to the server.

{@a fallback}

Routed apps must fallback to `index.html`

Angular apps are perfect candidates for serving with a simple static HTML server. You don't need a server-side engine to dynamically compose application pages because Angular does that on the client-side.

If the app uses the Angular router, you must configure the server to return the application's host page (`index.html`) when asked for a file that it does not have.

{@a deep-link}

A routed application should support "deep links". A *deep link* is a URL that specifies a path to a component inside the app. For example, `http://www.mysite.com/heroes/42` is a *deep link* to the hero detail page that displays the hero with `id: 42`.

There is no issue when the user navigates to that URL from within a running client. The Angular router interprets the URL and routes to that page and hero.

But clicking a link in an email, entering it in the browser address bar, or merely refreshing the browser while on the hero detail page — all of these actions are handled by the browser itself, *outside* the running application. The browser makes a direct request to the server for that URL, bypassing the router.

A static server routinely returns `index.html` when it receives a request for `http://www.mysite.com/`. But it rejects `http://www.mysite.com/heroes/42` and returns a `404 - Not Found` error *unless* it is configured to return `index.html` instead.

Fallback configuration examples

There is no single configuration that works for every server. The following sections describe configurations for some of the most popular servers. The list is by no means exhaustive, but should provide you with a good starting point.

Development servers

- Lite-Server: the default dev server installed with the Quickstart repo is pre-configured to fallback to `index.html`.

- Webpack-Dev-Server: setup the `historyApiFallback` entry in the dev server options as follows:

  historyApiFallback: { disableDotRule: true, htmlAcceptHeaders: ['text/html', 'application/xhtml+xml'] }

Production servers

- Apache: add a rewrite rule to the `.htaccess` file as shown (https://ngmilk.rocks/2015/03/09/angularjs-html5-mode-or-pretty-urls-on-apache-using-htaccess/):

  RewriteEngine On &#35 If an existing asset or directory is requested go to it as it is RewriteCond %{DOCUMENT*ROOT}%{REQUEST*URI} -f [OR] RewriteCond %{DOCUMENT*ROOT}%{REQUEST*URI} -d RewriteRule ^ - [L]

  &#35 If the requested resource doesn't exist, use index.html RewriteRule ^ /index.html

- NGinx: use `try_files`, as described in Front Controller Pattern Web Apps, modified to serve `index.html` :

try_files $uri $uri/ /index.html;

- IIS: add a rewrite rule to `web.config`, similar to the one shown here:

  <system.webServer> <rewrite> <rules> <rule name="Angular Routes" stopProcessing="true"> <match url=".*" /> <conditions logicalGrouping="MatchAll"> <add input="{REQUEST*FILENAME}" matchType="IsFile" negate="true" /> <add input="{REQUEST*FILENAME}" matchType="IsDirectory" negate="true" /> </conditions> <action type="Rewrite" url="/src/" /> </rule> </rules> </rewrite> </system.webServer>

- GitHub Pages: you can't directly configure the GitHub Pages server, but you can add a 404 page. Copy `index.html` into `404.html`. It will still be served as the 404 response, but the browser will process that page and load the app properly. It's also a good idea to serve from `docs/` on master and to create a `.nojekyll` file

- Firebase hosting: add a rewrite rule.

  "rewrites": [{ "source": "**", "destination": "/index.html" }]

{@a cors}

Requesting services from a different server (CORS)

Angular developers may encounter a *cross-origin resource sharing* error when making a service request (typically a data service request). to a server other than the application's own host server. Browsers forbid such requests unless the server permits them explicitly.

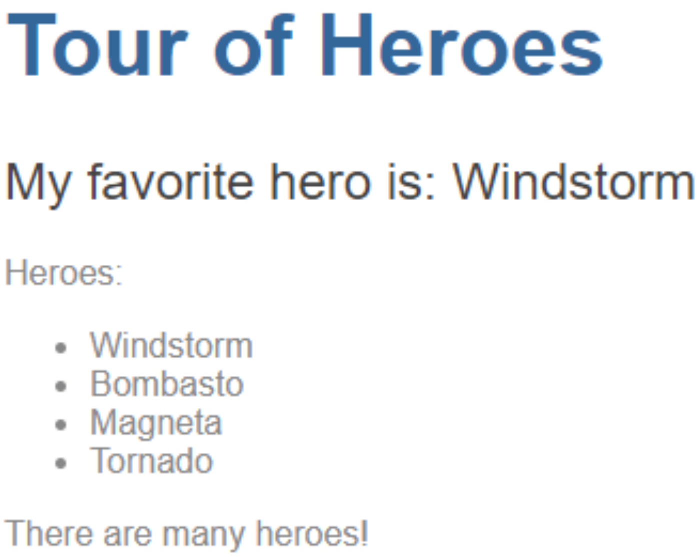
There isn't anything the client application can do about these errors. The server must be configured to accept the application's requests. Read about how to enable CORS for specific servers at enable-cors.org.

Displaying Data

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

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

The final UI looks like this:



Tour of Heroes

My favorite hero is: Windstorm

Heroes:

- Windstorm
- Bombasto
- Magneta
- Tornado

There are many heroes!

The demonstrates all of the syntax and code snippets described in this page.

{@a interpolation}

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 quickstart 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:

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

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

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.

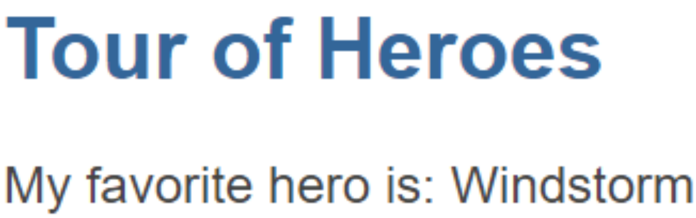
More precisely, the redisplay occurs after some kind of asynchronous event related to the view, such as a keystroke, a timer completion, or a response to an HTTP request.

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:

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:

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

{@a ngFor}

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.

Now use the Angular `ngFor` directive in the template to display each item in the `heroes` list.

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":

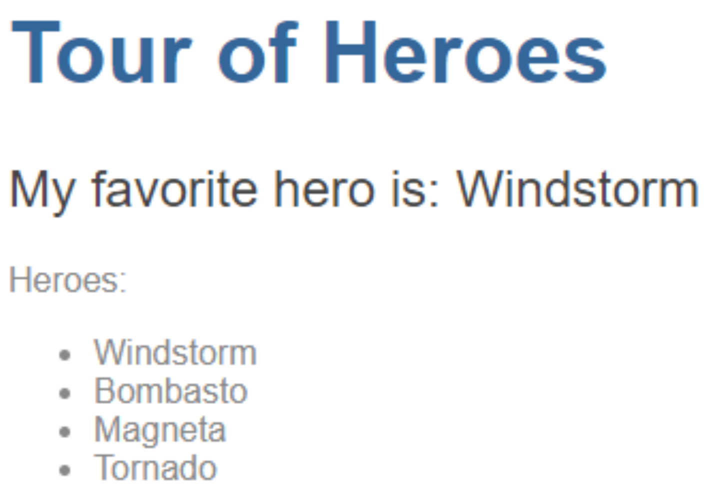
Don't forget the leading asterisk (*) in `*ngFor`. It is an essential part of the syntax. For more information, see the [Template Syntax](guide/template-syntax#ngFor) page.

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](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols) 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:

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:

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:

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.

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

{@a ngIf}

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:

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](guide/template-syntax#ngIf) of the [Template Syntax](guide/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.

Here's the final code:

Authors Style Guide

This page presents design and layout guidelines for Angular documentation pages. These guidelines should be followed by all guide page authors. Deviations must be approved by the documentation editor.

Most guide pages should have accompanying sample code with special markup for the code snippets on the page. Code samples should adhere to the style guide for Angular applications because readers expect consistency.

For clarity and precision, every guideline on *this* page is illustrated with a working example, followed by the page markup for that example ... as shown here.

```
followed by the page markup for that example ... as shown here.
```

Doc generation and tooling

To make changes to the documentation pages and sample code, clone the Angular github repository and go to the `aio/` folder.

The aio/README.md explains how to install and use the tools to edit and test your changes.

Here are a few essential commands for guide page authors.

1. `yarn setup` — installs packages; builds docs, plunkers, and zips.
2. `yarn docs-watch --watch-only` — watches for saved content changes and refreshes the browser. The (optional) `--watch-only` flag skips the initial docs rebuild.
3. `yarn start` — starts the doc viewer application so you can see your local changes in the browser.
4. http://localhost:4200/ — browse to the app running locally.

Guide pages

All but a few guide pages are markdown files with an `.md` extension.

Every guide page file is stored in the `content/guide` directory. Although the side navigation panel displays as a hierarchy, the directory is flat with no sub-folders. The flat folder approach allows us to shuffle the apparent navigation structure without moving page files or redirecting old page URLs.

The doc generation process consumes the markdown files in the `content/guide` directory and produces JSON files in the `src/generated/docs/guide` directory, which is also flat. Those JSON files contain a combination of document metadata and HTML content.

The reader requests a page by its Page URL. The doc viewer fetches the corresponding JSON file, interprets it, and renders it as fully-formed HTML page.

Page URLs mirror the `content` file structure. The URL for the page of a guide is in the form `guide/{page-name}`. The page for *this* "Authors Style Guide" is located at `content/guide/docs-style-guide.md` and its URL is `guide/docs-style-guide`.

Tutorial pages are exactly like guide pages. The only difference is that they reside in `content/tutorial` instead of `content/guide` and have URLs like `tutorial/{page-name}`. _API_ pages are generated from Angular source code into the `src/generated/docs/api` directory. The doc viewer translates URLs that begin `api/` into requests for document JSON files in that directory. This style guide does not discuss creation or maintenance of API pages. _Marketing_ pages are similar to guide pages. They're located in the `content/marketing` directory. While they can be markdown files, they may be static HTML pages or dynamic HTML pages that render with JSON data. Only a few people are authorized to write marketing pages. This style guide does not discuss creation or maintenance of marketing pages.

Markdown and HTML

While documentation guide pages ultimately render as HTML, almost all of them are written in markdown.

Markdown is easier to read and to edit than HTML. Many editors (including Visual Studio Code) can render markdown as you type it.

From time to time you'll have to step away from markdown and write a portion of the document in HTML. Markdown allows you to mix HTML and markdown in the same document.

Standard markdown processors don't allow you to put markdown *within* HTML tags. But the Angular documentation markdown processor **supports markdown within HTML**, as long as you follow one rule:

Always follow every opening and closing HTML tag with _a blank line_.

```
<div class="alert is-critical">

  **Always** follow every opening and closing HTML tag with _a blank line_.

</div>
```

It is customary but not required to _precede_ the _closing HTML_ tag with a blank line as well.

Title

Every guide document must have a title.

The title should appear at the top of the physical page. Begin the title with the markdown `#` character. Alternatively, you can write the equivalent `<h1>`.

```
  # Authors Style Guide
```

Only one title (`<h1>`) per document!

Title text should be in "Title Case", which means that you use capital letters to start the first words and all *principal* words. Use lower case letters for _secondary words such as "in", "of", and "the".

```
  # The Meat of the Matter
```

Always follow the title with at least one blank line.

Sections

A typical document is divided into sections.

All section heading text should be in "Sentence case", which means the first word is capitalized and all other words are lower case.

Always follow the section heading with at least one blank line.

Main section heading

There are usually one or more main sections that may be further divided into secondary sections.

Begin a main section heading with the markdown `##` characters. Alternatively, you can write the equivalent `<h2>` HTML tag.

The main section heading should be followed by a blank line and then the content for that heading.

```
  ## Sections

  A typical document is divided into sections.
```

Secondary section heading

A secondary section heading is related to a main heading and *falls textually within* the bounds of that main heading.

Begin a secondary heading with the markdown `###` characters. Alternatively, you can write the equivalent `<h3>` HTML tag.

The secondary heading should be followed by a blank line and then the content for that heading.

```
  ### Secondary section heading

  A secondary section ...
```

Additional section headings

Try to minimize the heading depth, preferably only two. But more headings, such as this one, are permitted if they make sense.

N.B.: The Table-of-contents generator only considers main (`<h2>`) and secondary (`<h3>`) headings.

```
  #### Additional section headings

  Try to minimize ...
```

Subsections

Subsections typically present extra detail and references to other pages.

Use subsections for commentary that *enriches* the reader's understanding of the text that precedes it.

A subsection *must not* contain anything *essential* to that understanding. Don't put a critical instruction or a tutorial step in a subsection.

A subsection is content within a `<div>` that has the `l-sub-section` CSS class. You should write the subsection content in markdown.

Here is an example of a subsection `<div>` surrounding the subsection content written in markdown.

You'll learn about styles for live examples in the [section below](guide/docs-style-guide#live-examples "Live examples").

```
<div class="l-sub-section">

You'll learn about styles for live examples in the [section below](guide/docs-style-guide#live-examples "Live examples").

</div>
```

Note that at least one blank line must follow the opening `<div>`. A blank line before the closing `</div>` is customary but not required.

Table of contents

Most pages display a table of contents (TOC). The TOC appears in the right panel when the viewport is wide. When narrow, the TOC appears in an expandable/collapsible region near the top of the page.

You should not create your own TOC by hand. The TOC is generated automatically from the page's main and secondary section headers.

To exclude a heading from the TOC, create the heading as an `<h2>` or `<h3>` element with a class called 'no-toc'. You can't do this with markdown.

```
<h3 class="no-toc">
This heading is not displayed in the TOC
</h3>
```

You can turn off TOC generation for the *entire* page by writing the title with an `<h1>` tag and the `no-toc` class.

```
<h1 class="no-toc">
A guide without a TOC
</h1>
```

Navigation

The navigation links at the top, left, and bottom of the screen are generated from the JSON configuration file, `content/navigation.json`.

The authority to change the `navigation.json` file is limited to a few core team members. But for a new guide page, you should suggest a navigation title and position in the left-side navigation panel called the "side nav".

Look for the `SideNav` node in `navigation.json`. The `SideNav` node is an array of navigation nodes. Each node is either an *item* node for a single document or a *header* node with child nodes.

Find the header for your page. For example, a guide page that describes an Angular feature is probably a child of the `Fundamentals` header.

```
{
  "title": "Fundamentals",
  "tooltip": "The fundamentals of Angular",
  "children": [ ... ]
}
```

A *header* node child can be an *item* node or another *header* node. If your guide page belongs under a sub-header, find that sub-header in the JSON.

Add an *item* node for your guide page as a child of the appropriate *header* node. It probably looks something like this one.

```
{
  "url": "guide/architecture",
  "title": "Architecture",
  "tooltip": "The basic building blocks of Angular applications."
}
```

A navigation node has the following properties:

- `url` - the URL of the guide page (*item node only*).
- `title` - the text displayed in the side nav.
- `tooltip` - text that appears when the reader hovers over the navigation link.
- `children` - an array of child nodes (*header node only*).
- `hidden` - defined and set true if this is a guide page that should *not* be displayed in the navigation panel. Rarely needed, it is a way to hide the page from navigation while making it available to readers who should know about it. *This* "Authors Style Guide" is a hidden page.

Do not create a node that is both a _header_ and an _item_ node. That is, do not specify the `url` property of a _header_ node.
The current guidelines allow for a three-level navigation structure with two header levels. Don't add a third header level.

Code snippets

Guides are rich in examples of working Angular code. Example code can be commands entered in a terminal window, a fragment of TypeScript or HTML, or an entire code file.

Whatever the source, the doc viewer renders them as "code snippets", either individually with the *code-example* component or as a tabbed collection with the *code-tabs* component.

{@a code-example}

Code example

You can display a simple, inline code snippet with the markdown backtick syntax. We generally prefer to display a code snippet with the Angular documentation *code-example* component represented by the `<code-example>` tag.

Inline code-snippets

You should source code snippets from working sample code when possible. But there are times when an inline snippet is the better choice.

For terminal input and output, put the content between `<code-example>` tags, set the CSS class to `code-shell`, and set the language attribute to `sh` as in this example.

npm start

```
<code-example language="sh" class="code-shell">
  npm start
</code-example>
```

Inline, hand-coded snippets like this one are *not* testable and, therefore, are intrinsically unreliable. This example belongs to the small set of pre-approved, inline snippets that includes user input in a command shell or the *output* of some process.

Do not write inline code snippets unless you have a good reason and the editor's permission to do so. In all other cases, code snippets should be generated automatically from tested code samples.

{@a from-code-samples}

Code snippets and code samples

One of the documentation design goals is that guide page code snippets should be examples of real, working code.

We meet this goal by displaying code snippets that are derived directly from standalone code samples, written specifically for these guide pages.

The author of a guide page is responsible for the code sample that supports that page. The author must also write end-to-end tests for the sample.

Code samples are located in sub-folders of the `content/examples` directory of the `angular/angular` repository. An example folder name should be the same as the guide page it supports.

A guide page might not have its own sample code. It might refer instead to a sample belonging to another page.

The Angular CI process runs all end-to-end tests for every Angular PR. Angular re-tests the samples after every new version of a sample and every new version of Angular itself.

When possible, every snippet of code on a guide page should be derived from a code sample file. You tell the Angular documentation engine which code file - or fragment of a code file - to display by configuring `<code-example>` attributes.

Code snippet from a file

This "Authors Doc Style Guide" has its own sample application, located in the `content/examples/docs-style-guide` folder.

The following *code-example* displays the sample's `app.module.ts`.

Here's the brief markup that produced that lengthy snippet:

```
<code-example
  path="docs-style-guide/src/app/app.module.ts"
  title="src/app/app.module.ts">
</code-example>
```

You identified the snippet's source file by setting the `path` attribute to sample folder's location *within* `content/examples`. In this example, that path is `docs-style-guide/src/app/app.module.ts`.

You added a header to tell the reader where to find the file by setting the `title` attribute. Following convention, you set the `title` attribute to the file's location within the sample's root folder.

Unless otherwise noted, all code snippets in this page are derived from sample source code located in the `content/examples/docs-style-guide` directory.
The doc tooling reports an error if the file identified in the path does not exist **or is _git_-ignored**. Most `.js` files are _git_-ignored. If you want to include an ignored code file in your project and display it in a guide you must _un-ignore_ it. The preferred way to un-ignore a file is to update the `content/examples/.gitignore` like this: # my-guide !my-guide/src/something.js !my-guide/more-javascript*.js

Code-example attributes

You control the *code-example* output by setting one or more of its attributes:

- `path` - the path to the file in the `content/examples` folder.
- `title` - the header of the code listing.
- `region` - displays the source file fragment with that region name; regions are identified by *docregion* markup in the source file, as explained below.
- `linenums` - value may be `true`, `false`, or a `number`. When not specified, line numbers are automatically displayed when there are greater than 10 lines of code. The rarely used `number` option starts line numbering at the given value. `linenums=4` sets the starting line number to 4.
- `class` - code snippets can be styled with the CSS classes `no-box`, `code-shell`, and `avoid`.
- `hideCopy` - hides the copy button
- `language` - the source code language such as `javascript`, `html`, `css`, `typescript`, `json`, or `sh`. This attribute only works for inline examples.

{@a region}

Displaying a code fragment

Often you want to focus on a fragment of code within a sample code file. In this example, you focus on the `AppModule` class and its `NgModule` metadata.

First you surround that fragment in the source file with a named *docregion* as described below. Then you reference that *docregion* in the `region` attribute of the `<code-example>` like this

```
<code-example
  path="docs-style-guide/src/app/app.module.ts"
  region="class">
</code-example>
```

A couple of observations:

1. The `region` value, `"class"`, is the name of the `#docregion` in the source file. Confirm that by looking at `content/examples/docs-style-guide/src/app/app.module.ts`
2. Omitting the `title` is fine when the source of the fragment is obvious. We just said that this is a fragment of the `app.module.ts` file which was displayed immediately above, in full, with a header. There's no need to repeat the header.
3. The line numbers disappeared. By default, the doc viewer omits line numbers when there are fewer than 10 lines of code; it adds line numbers after that. You can turn line numbers

on or off explicitly by setting the `linenums` attribute.

Example of bad code

Sometimes you want to display an example of bad code or bad design.

You should be careful. Readers don't always read carefully and are likely to copy and paste your example of bad code in their own applications. So don't display bad code often.

When you do, set the `class` to `avoid`. The code snippet will be framed in bright red to grab the reader's attention.

Here's the markup for an "avoid" example in the *Angular Style Guide*.

```
<code-example
  path="styleguide/src/05-03/app/heroes/shared/hero-button/hero-button.component.avoid.ts"
  region="example"
  title="app/heroes/hero-button/hero-button.component.ts">
</code-example>
```

{@a code-tabs}

Code Tabs

Code tabs display code much like *code examples* do. The added advantage is that they can display multiple code samples within a tabbed interface. Each tab is displayed using *code pane*.

Code-tabs attributes

- `linenums`: The value can be `true`, `false` or a number indicating the starting line number. If not specified, line numbers are enabled only when code for a tab pane has greater than 10 lines of code.

Code-pane attributes

- `path` - a file in the content/examples folder
- `title` - seen in the header of a tab
- `linenums` - overrides the `linenums` property at the `code-tabs` level for this particular pane. The value can be `true`, `false` or a number indicating the starting line number. If not specified, line numbers are enabled only when the number of lines of code are greater than 10.

The next example displays multiple code tabs, each with its own title. It demonstrates control over display of line numbers at both the `<code-tabs>` and `<code-pane>` levels.

Here's the markup for that example.

Note how the `linenums` attribute in the `<code-tabs>` explicitly disables numbering for all panes. The `linenums` attribute in the second pane restores line numbering for *itself only*.

```
<code-tabs linenums="false">
  <code-pane
    title="app.component.html"
    path="docs-style-guide/src/app/app.component.html">
  </code-pane>
  <code-pane
    title="app.component.ts"
    path="docs-style-guide/src/app/app.component.ts"
    linenums="true">
  </code-pane>
  <code-pane
    title="app.component.css (heroes)"
    path="docs-style-guide/src/app/app.component.css"
    region="heroes">
  </code-pane>
  <code-pane
    title="package.json (scripts)"
    path="docs-style-guide/package.1.json">
  </code-pane>
</code-tabs>
```

{@a source-code-markup}

Source code markup

You must add special code snippet markup to sample source code files before they can be displayed by `<code-example>` and `<code-tabs>` components.

The sample source code for this page, located in `context/examples/docs-style-guide`, contains examples of every code snippet markup described in this section.

Code snippet markup is always in the form of a comment. Here's the default *docregion* markup for a TypeScript or JavaScript file:

```
// #docregion
... some code ...
// #enddocregion
```

Different file types have different comment syntax so adjust accordingly.

```
<!-- #docregion -->
... some HTML ...
<!-- #enddocregion -->
```

```
/* #docregion */
... some CSS ...
/* #enddocregion */
```

The doc generation process erases these comments before displaying them in the doc viewer. It also strips them from plunkers and sample code downloads.

Code snippet markup is not supported in JSON files because comments are forbidden in JSON files. See [below](#json-files) for details and workarounds.

#docregion

The *#docregion* is the most important kind of code snippet markup.

The `<code-example>` and `<code-tabs>` components won't display a source code file unless it has a *#docregion*.

The *#docregion* comment begins a code snippet region. Every line of code *after* that comment belongs in the region *until* the code fragment processor encounters the end of the file or a closing *#enddocregion*.

The `src/main.ts` is a simple example of a file with a single _#docregion_ at the top of the file.

Named *#docregions*

You'll often display multiple snippets from different fragments within the same file. You distinguish among them by giving each fragment its own *#docregion name* as follows.

```
// #docregion region-name
... some code ...
// #enddocregion region-name
```

Remember to refer to this region by name in the `region` attribute of the `<code-example>` or `<code-pane>` as you did in an example above like this:

```
<code-example
  path="docs-style-guide/src/app/app.module.ts"
  region="class"></code-example>
```

The *#docregion* with no name is the *default region*. Do *not* set the `region` attribute when referring to the default *#docregion*.

Nested *#docregions*

You can nest *#docregions* within *#docregions*
`// #docregion ... some code ... // #docregion inner-region ... more code ... // #enddocregion inner-region ... yet more code ... /// #enddoc`

The `src/app/app.module.ts` file has a good example of a nested region.

Combining fragments

You can combine several fragments from the same file into a single code snippet by defining multiple *#docregions* with the *same region name*.

Examine the `src/app/app.component.ts` file which defines two nested *#docregions*.

The inner, `class-skeleton` region appears twice, once to capture the code that opens the class definition and once to capture the code that closes the class definition.

// #docplaster ... // #docregion class, class-skeleton export class AppComponent { // #enddocregion class-skeleton title = 'Authors Style Guide Sample'; heroes = HEROES; selectedHero: Hero;

onSelect(hero: Hero): void { this.selectedHero = hero; } // #docregion class-skeleton } // #enddocregion class, class-skeleton

Here's are the two corresponding code snippets displayed side-by-side.

Some observations:

- The `#docplaster` at the top is another bit of code snippet markup. It tells the processor how to join the fragments into a single snippet.

  In this example, we tell the processor to put the fragments together without anything in between - without any "plaster". Most sample files define this *empty plaster*.

  If we neglected to add, `#docplaster`, the processor would insert the *default* plaster - an ellipsis comment - between the fragments. Try removing the `#docplaster` comment yourself to see the effect.

- One `#docregion` comment mentions ***two*** region names as does an `#enddocregion` comment. This is a convenient way to start (or stop) multiple regions on the same code line. You could have put these comments on separate lines and many authors prefer to do so.

JSON files

Code snippet markup is not supported for JSON files because comments are forbidden in JSON files.

You can display an entire JSON file by referring to it in the `src` attribute. But you can't display JSON fragments because you can't add `#docregion` tags to the file.

If the JSON file is too big, you could copy the nodes-of-interest into markdown backticks.

Unfortunately, it's easy to mistakenly create invalid JSON that way. The preferred way is to create a JSON partial file with the fragment you want to display.

You can't test this partial file and you'll never use it in the application. But at least your IDE can confirm that it is syntactically correct.

Here's an example that excerpts certain scripts from `package.json` into a partial file named `package.1.json`.

```
<code-example
  path="docs-style-guide/package.1.json"
  title="package.json (selected scripts)"></code-example>
```

Partial file naming

Many guides tell a story. In that story, the app evolves incrementally, often with simplistic or incomplete code along the way.

To tell that story in code, you'll often need to create partial files or intermediate versions of the final source code file with fragments of code that don't appear in the final app.

Such partial and intermediate files need their own names. Follow the doc sample naming convention. Add a number before the file extension as illustrated here:

```
package.1.json
app.component.1.ts
app.component.2.ts
```

You'll find many such files among the samples in the Angular documentation.

Remember to exclude these files from plunkers by listing them in the `plnkr.json` as illustrated here.

{@a live-examples}

Live examples

By adding `<live-example>` to the page you generate links that run sample code in the Plunker live coding environment and download that code to the reader's file system.

Live examples (AKA "plunkers") are defined by one or more `plnkr.json` files in the root of a code sample folder. Each sample folder usually has a single unnamed definition file, the default `plnkr.json`.

You can create additional, named definition files in the form \`name.plnkr.json\`. See \`content/examples/testing\` for examples. The schema for a \`plnkr.json\` hasn't been documented yet but looking at the \`plnkr.json\` files in the example folders should tell you most of what you need to know.

Adding `<live-example></live-example>` to the page generates the two default links.

1. a link to the plunker defined by the default `plnkr.json` file located in the code sample folder with the same name as the guide page.
2. a link that downloads that sample.

Clicking the first link opens the code sample in a new browser tab in the "embedded plunker" style.

You can change the appearance and behavior of the live example with attributes and classes.

Custom label and tooltip

Give the live example anchor a custom label and tooltip by setting the `title` attribute.

```
<live-example title="Live Example with title"></live-example>
```

You can achieve the same effect by putting the label between the `<live-example>` tags:

Live example with content label

```
<live-example>Live example with content label</live-example>
```

Live example from another guide

To link to a plunker in a folder whose name is not the same as the current guide page, set the `name` attribute to the name of that folder.

Live Example from the Router guide

```
<live-example name="router">Live Example from the Router guide</live-example>
```

Live Example for named plunker

To link to a plunker defined by a named `plnkr.json` file, set the `plnkr` attribute. The following example links to the plunker defined by `second.plnkr.json` in the current guide's directory.

```
<live-example plnkr="second"></live-example>
```

Live Example without download

To skip the download link, add the `noDownload` attribute.

Just the plunker

```
<live-example noDownload>Just the plunker</live-example>
```

Live Example with download-only

To skip the live plunker link and only link to the download, add the `downloadOnly` attribute.

Download only

```
<live-example downloadOnly>Download only</live-example>
```

Embedded live example

By default, a live example link opens a plunker in a separate browser tab. You can embed the plunker within the guide page itself by adding the `embedded` attribute.

For performance reasons, the plunker does not start right away. The reader sees an image instead. Clicking the image starts the sometimes-slow process of launching the embedded plunker within an iframe on the page.

You usually replace the default plunker image with a custom image that better represents the sample. Store that image in the `content/images` directory in a folder with a name matching the corresponding example folder.

Here's an embedded live example for this guide. It has a custom image created from a snapshot of the running app, overlayed with `content/images/plunker/unused/click-to-run.png`.

```
<live-example embedded img="guide/docs-style-guide/docs-style-guide-plunker.png"></live-example>
```

Anchors

Every section header tag is also an anchor point. Another guide page could add a link to this section by writing:

See the ["Anchors"](guide/docs-style-guide#anchors "Style Guide - Anchors") section for details.

```
<div class="l-sub-section">

See the ["Anchors"](guide/docs-style-guide#anchors "Style Guide - Anchors") section for details.

</div>
```

When navigating within the page, you can omit the page URL when specifying the link that scrolls up to the beginning of this section.

```
... the link that [scrolls up](#anchors "Anchors") to ...
```

{@a ugly-anchors}

Ugly, long section header anchors

It is often a good idea to *lock-in* a good anchor name.

Sometimes the section header text makes for an unattractive anchor. This one is pretty bad.

```
[This one](#ugly-long-section-header-anchors) is pretty bad.
```

The greater danger is that **a future rewording of the header text would break** a link to this section.

For these reasons, it is often wise to add a custom anchor explicitly, just above the heading or text to which it applies, using the special `{@ name}` syntax like this.

{@a ugly-anchors}

Ugly, long section header anchors

Now link to that custom anchor name as you did before.

```
Now [link to that custom anchor name](#ugly-anchors) as you did before.
```

Alternatively, you can use the HTML `` tag. If you do, be sure to set the `id` attribute - not the `name` attribute! The docs generator will not convert the `name` to the proper link URL. ```html ## Anchors ```

Alerts

Alerts draw attention to important points. Alerts should not be used for multi-line content (use callouts insteads) or stacked on top of each other. Note that the content of an alert is indented to the right by two spaces.

A critical alert.
An important alert.
A helpful, informational alert.

Here is the markup for these alerts. ```html

A critical alert.

An important alert.
A helpful, informational alert.
``` Alerts are meant to grab the user's attention and should be used sparingly. They are not for casual asides or commentary. Use [subsections](#subsections "subsections") for commentary. ## Callouts Callouts (like alerts) are meant to draw attention to important points. Use a callout when you want a riveting header and multi-line content.
A critical point
**Pitchfork hoodie semiotics**, roof party pop-up _paleo_ messenger messenger bag cred Carles tousled Truffaut yr. Semiotics viral freegan VHS, Shoreditch disrupt McSweeney's. Intelligentsia kale chips Vice four dollar toast, Schlitz crucifix
An important point
**Pitchfork hoodie semiotics**, roof party pop-up _paleo_ messenger bag cred Carles tousled Truffaut yr. Semiotics viral freegan VHS, Shoreditch disrupt McSweeney's. Intelligentsia kale chips Vice four dollar toast, Schlitz crucifix
A helpful point
**Pitchfork hoodie semiotics**, roof party pop-up _paleo_ messenger bag cred Carles tousled Truffaut yr. Semiotics viral freegan VHS, Shoreditch disrupt McSweeney's. Intelligentsia kale chips Vice four dollar toast, Schlitz crucifix

Here is the markup for the first of these callouts. ```html

A critical point
```

Pitchfork hoodie semiotics, roof party pop-up *paleo* messenger bag cred Carles tousled Truffaut yr. Semiotics viral freegan VHS, Shoreditch disrupt McSweeney's. Intelligentsia kale chips Vice four dollar toast, Schlitz crucifix

\`\`\`

Notice that * the callout header text is forced to all upper case. * the callout body can be written in markdown. * a blank line separates the `</header>` tag from the markdown content.

Callouts are meant to grab the user's attention. They are not for casual asides. Please use them sparingly.

Trees

Trees can represent hierarchical data.

sample-dir
src
app
app.component.ts
app.module.ts
styles.css
tsconfig.json
node_modules ...
package.json

Here is the markup for this file tree.

```
<div class='filetree'>
    <div class='file'>
        sample-dir
    </div>
    <div class='children'>
        <div class='file'>
          src
        </div>
        <div class='children'>
            <div class='file'>
              app
            </div>
            <div class='children'>
                <div class='file'>
                  app.component.ts
                </div>
                <div class='file'>
                  app.module.ts
                </div>
            </div>
            <div class='file'>
              styles.css
            </div>
            <div class='file'>
              tsconfig.json
            </div>
        </div>
        <div class='file'>
          node_modules ...
        </div>
        <div class='file'>
          package.json
        </div>
    </div>
</div>
```

Tables

Use HTML tables to present tabular data.

Framework	Task	Speed
`AngularJS`	Routing	Fast
`Angular v2`	Routing	*Faster*
`Angular v4`	Routing	**Fastest :)**

Here is the markup for this table.

```
<style>
  td, th {vertical-align: top}
</style>

<table>
  <tr>
    <th>Framework</th>
    <th>Task</th>
    <th>Speed</th>
  </tr>
  <tr>
    <td><code>AngularJS</code></td>
    <td>Routing</td>
    <td>Fast</td>
  </tr>
  <tr>
    <td><code>Angular v2</code></td>
    <td>Routing</td>
    <!-- can use markdown too; remember blank lines -->
    <td>

      *Faster*

    </td>
  </tr>
  <tr>
    <td><code>Angular v4</code></td>
    <td>Routing</td>
    <td>

      **Fastest :)**

    </td>
  </tr>
</table>
```

Images

Image location

Store images in the `content/images` directory in a folder with the same URL as the guide page. Images for this "Authors Style Guide" page belong in the `content/images/guide/docs-style-guide` folder.

Angular doc generation copies these image folders to the *runtime* location, `generated/images`. Set the image `src` attribute to begin in *that* directory.

Here's the `src` attribute for the "flying hero" image belonging to this page. `src="/Users/nblavoie/Documents/projets/angular/aio/content/images/guide/docs-style-guide/flying-hero.png"`

Use the HTML *<img>* tag

Do not use the markdown image syntax, ![...](...).

Images should be specified in an `<img>` tag.

For accessibility, always set the `alt` attribute with a meaningful description of the image.

You should nest the `<img>` tag within a `<figure>` tag, which styles the image within a drop-shadow frame. You'll need the editor's permission to skip the `<figure>` tag.

Here's a conforming example

```
<figure>
  <img src="/Users/nblavoie/Documents/projets/angular/aio/content/images/guide/docs-style-guide/flying-hero.png"
       alt="flying hero">
</figure>
```

Note that the HTML image element does not have a closing tag.

Image dimensions

The doc generator reads the image dimensions from the file and adds width and height attributes to the `img` tag automatically. If you want to control the size of the image, supply your own width and height attributes.

Here's the "flying hero" at a more reasonable scale.

```
<figure>
 <img src="/Users/nblavoie/Documents/projets/angular/aio/content/images/guide/docs-style-guide/flying-hero.png"
   alt="flying Angular hero"
   width="200">
</figure>
```

Wide images can be a problem. Most browsers try to rescale the image but wide images may overflow the document in certain viewports.

Do not set a width greater than 700px. If you wish to display a larger image, provide a link to the actual image that the user can click on to see the full size image separately as in this example of `source-map-explorer` output from the "Ahead-of-time Compilation" guide:

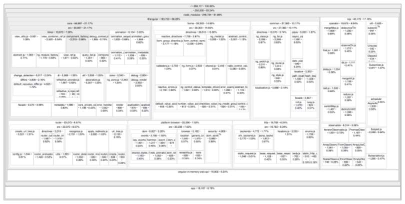


Image compression

Large image files can be slow to load, harming the user experience. Always compress the image. Consider using an image compression web site such as tinypng.

Floating images

You can float the image to the left or right of text by applying the class="left" or class="right" attributes respectively.

This text wraps around to the right of the floating "flying hero" image.

Headings and code-examples automatically clear a floating image. If you need to force a piece of text to clear a floating image, add `<br class="clear">` where the text should break.

The markup for the above example is:

```
<img src="/Users/nblavoie/Documents/projets/angular/aio/content/images/guide/docs-style-guide/flying-hero.png"
   alt="flying Angular hero"
   width="200"
   class="left">

This text wraps around to the right of the floating "flying hero" image.

Headings and code-examples automatically clear a floating image. If you need to force a piece of text to clear a floating image, add `<br class=
"clear">` where the text should break.

<br class="clear">
```

Note that you generally don't wrap a floating image in a `<figure>` element.

Floating within a subsection

If you have a floating image inside an alert, callout, or a subsection, it is a good idea to apply the `clear-fix` class to the `div` to ensure that the image doesn't overflow its container. For example:

A subsection with **markdown** formatted text.

```
<div class="l-sub-section clear-fix">

  <img src="/Users/nblavoie/Documents/projets/angular/aio/content/images/guide/docs-style-guide/flying-hero.png"
    alt="flying Angular hero"
    width="100"
    class="right">

  A subsection with **markdown** formatted text.

</div>
```

Dynamic Component Loader

Component templates are not always fixed. An application may need to load new components at runtime.

This cookbook shows you how to use `ComponentFactoryResolver` to add components dynamically.

See the of the code in this cookbook.

{@a dynamic-loading}

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.

{@a directive}

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.

`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.

{@a loading-components}

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.

The `<ng-template>` element is a good choice for dynamic components because it doesn't render any additional output.

{@a resolving-components}

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()`.

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 \`currentAddIndex\` by taking whatever it currently is plus one, dividing that by the length of the \`AdItem\` array, and using the _remainder_ as the new \`currentAddIndex\` 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.

{@a selector-references}

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:

{@a common-interface}

The *AdComponent* interface

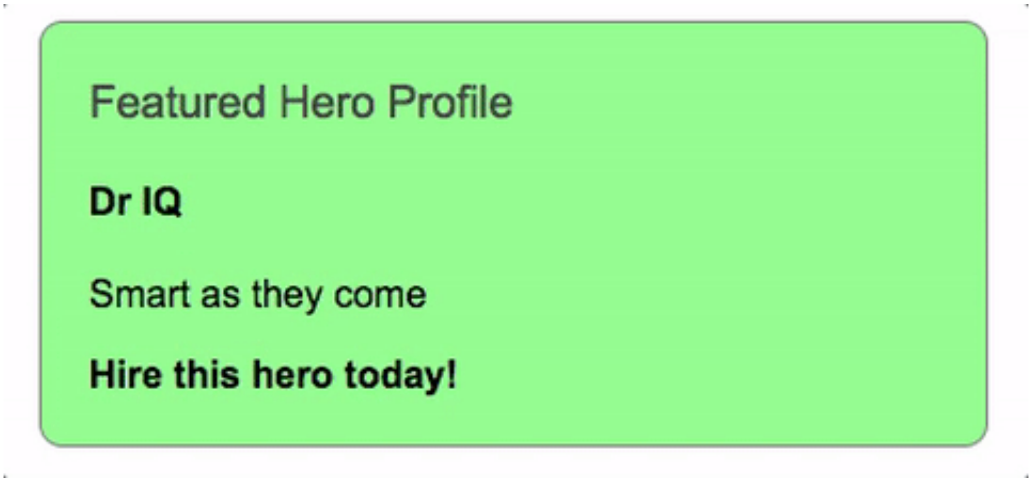
In the ad banner, all components implement a common `AdComponent` interface to standardize the API for passing data to the components.

Here are two sample components and the `AdComponent` interface for reference:

{@a final-ad-baner}

Final ad banner

The final ad banner looks like this:



Featured Hero Profile

Dr IQ

Smart as they come

Hire this hero today!

See the .

Dynamic Forms

{@a top}

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*. {@a toc}

See the .

{@a bootstrap}

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`.

{@a object-model}

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.

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.

`DropdownQuestion` presents a list of choices in a select box.

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.

{@a form-component}

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.

It presents a list of questions, each bound to a `<df-question>` component element. The `<df-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.

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`. {@a questionnaire-data}

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.

Finally, display an instance of the form in the `AppComponent` shell.

{@a dynamic-template}

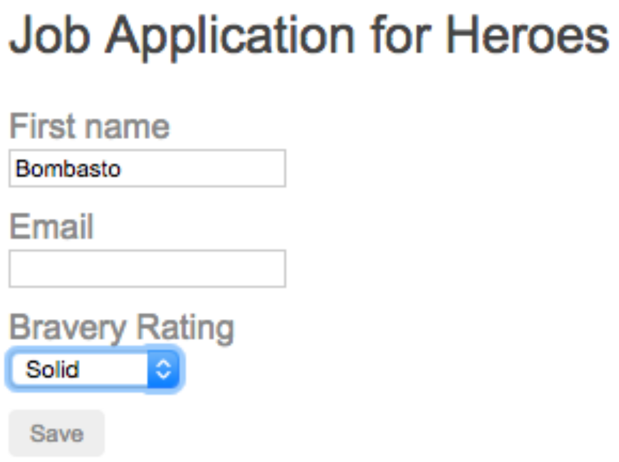
Dynamic Template

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:



Job Application for Heroes

First name: Bombasto

Email:

Bravery Rating: Solid

Back to top

Form Validation

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](guide/forms) and [Reactive Forms](guide/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` :

Note the following:

- The `<input>` element carries the HTML validation attributes: `required` and `minlength` . It also carries a custom validator directive, `forbiddenName` . 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:

{@a reactive-component-class}

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 `get` method 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.

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:

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`.

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.

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:

Once the `ForbiddenValidatorDirective` is ready, you can simply add its selector, `forbiddenName`, to any input element to activate it. For example:

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.

You can run the to see the complete reactive and template-driven example code.

Forms

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 in Plunker and download the code from there.

{@a template-driven}

Template-driven forms

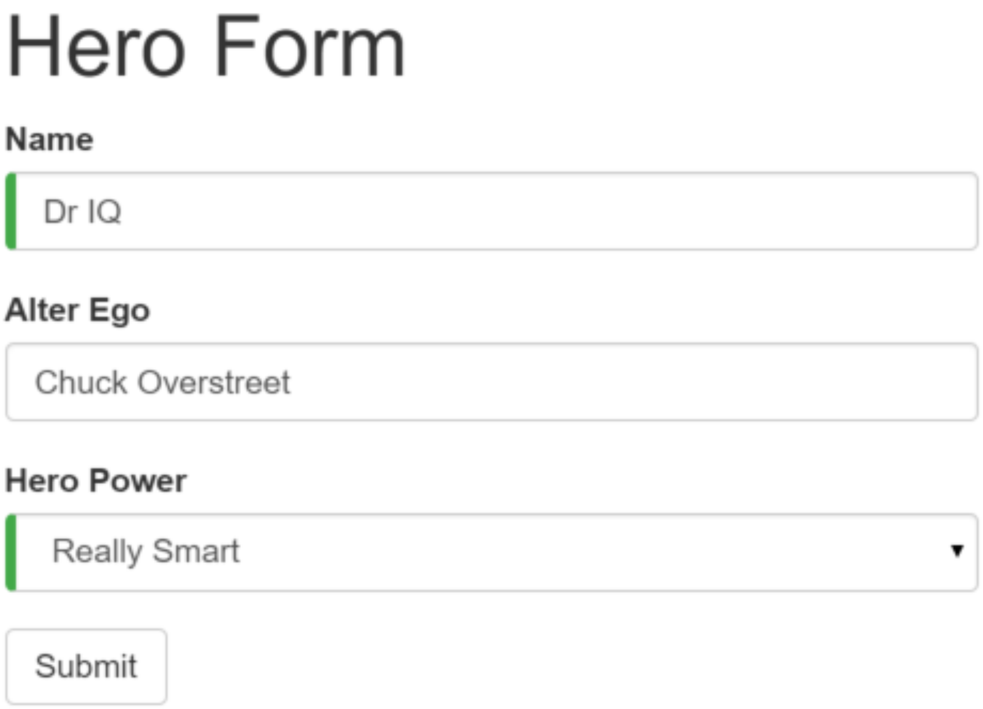
You can build forms by writing templates in the Angular template syntax with the form-specific directives and techniques described in this page.

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

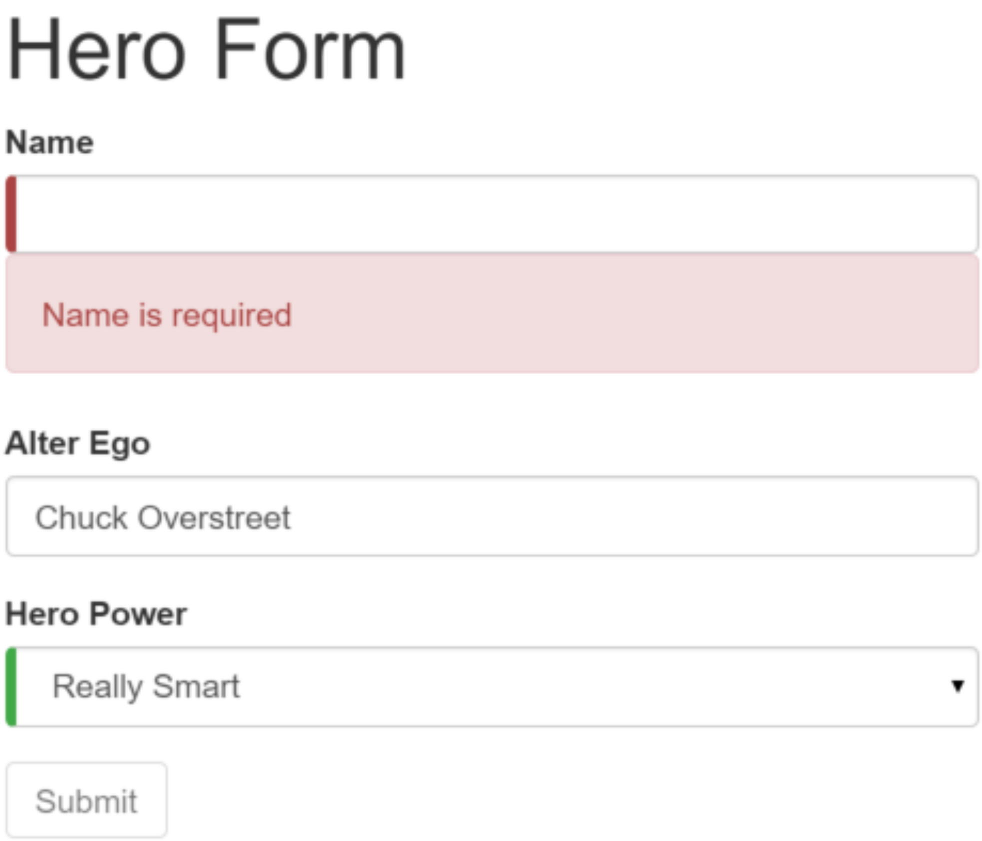
Hero Power

Really Smart

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

Chuck Overstreet

Hero Power

Really Smart

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:

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:

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:

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 "hero-form" means you can drop this form in a parent template with a `<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:

There are two changes: 1. You import `FormsModule`. 1. 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:

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 `name` field from the class body as well.

Create an initial HTML form template

Update the template file with the following contents:

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 \`
\` 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:

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.

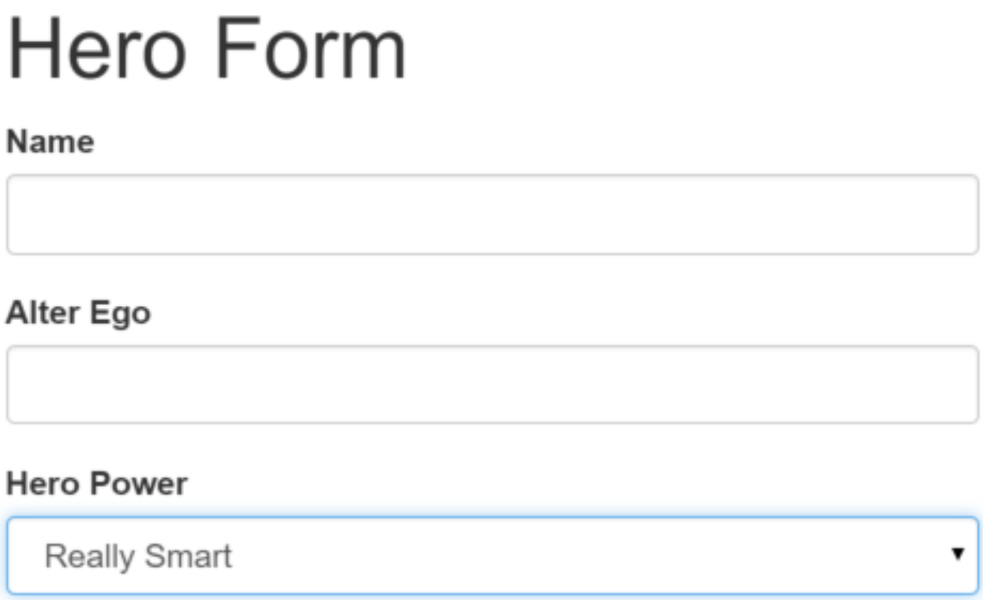
Add the following HTML *immediately below* the *Alter Ego* group:

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.

{@a ngModel}

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:

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.

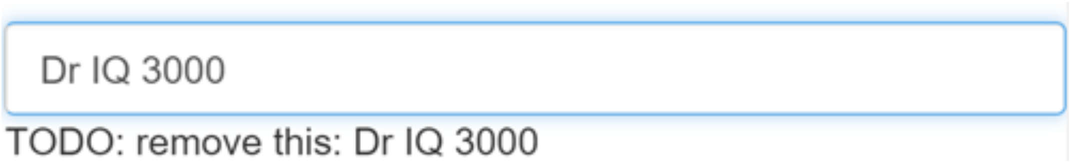
Focus on the binding syntax: `[(ngModel)]="..."`.

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:

The variable `heroForm` is now a reference to the `NgForm` directive that governs the form as a whole.

{@a ngForm} ### The _NgForm_ directive What `NgForm` directive? You didn't add an [NgForm](api/forms/NgForm) directive. Angular did. Angular automatically creates and attaches an `NgForm` directive to the `` 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](guide/template-syntax#ngModel) on the the [Template Syntax](guide/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.

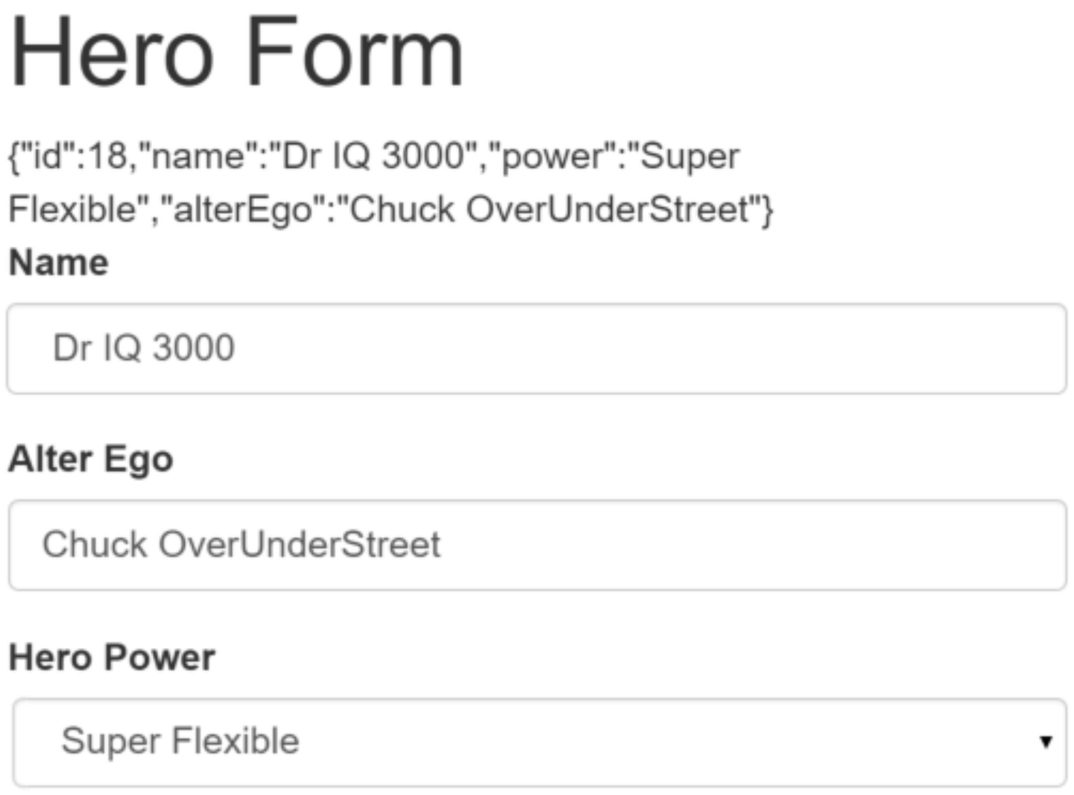
Internally, Angular creates `FormControl` instances and registers them with an `NgForm` directive that Angular attached to the `` tag. Each `FormControl` is registered under the name you assigned to the `name` attribute. Read more in the previous section, [The NgForm directive](guide/forms#ngForm).

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:

* 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.

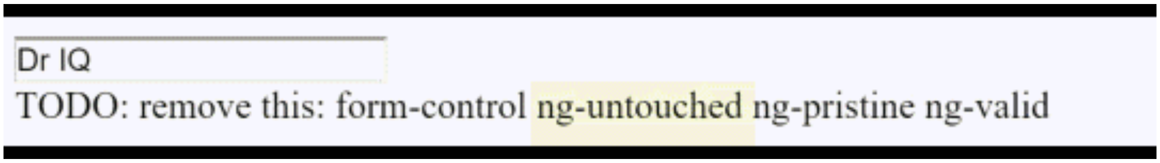
State	Class if true	Class if false
The control has been visited.	`ng-touched`	`ng-untouched`
The control's value has changed.	`ng-dirty`	`ng-pristine`
The control's value is valid.	`ng-valid`	`ng-invalid`

Temporarily add a template reference variable named `spy` to the *Name* `<input>` tag and use it to display the input's CSS classes.

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.

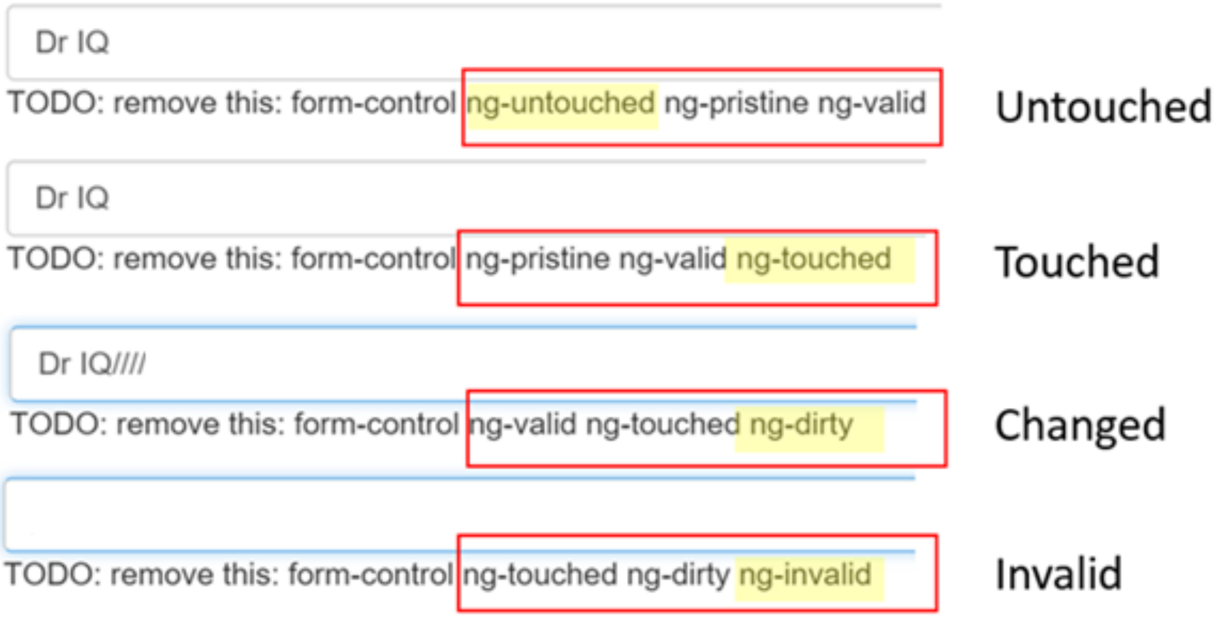
The actions and effects are as follows:



Dr IQ

TODO: remove this: form-control ng-untouched ng-pristine ng-valid

You should see the following transitions and class names:



Dr IQ

TODO: remove this: form-control ng-untouched ng-pristine ng-valid

Dr IQ

TODO: remove this: form-control ng-pristine ng-valid ng-touched

Dr IQ////

TODO: remove this: form-control ng-valid ng-touched ng-dirty

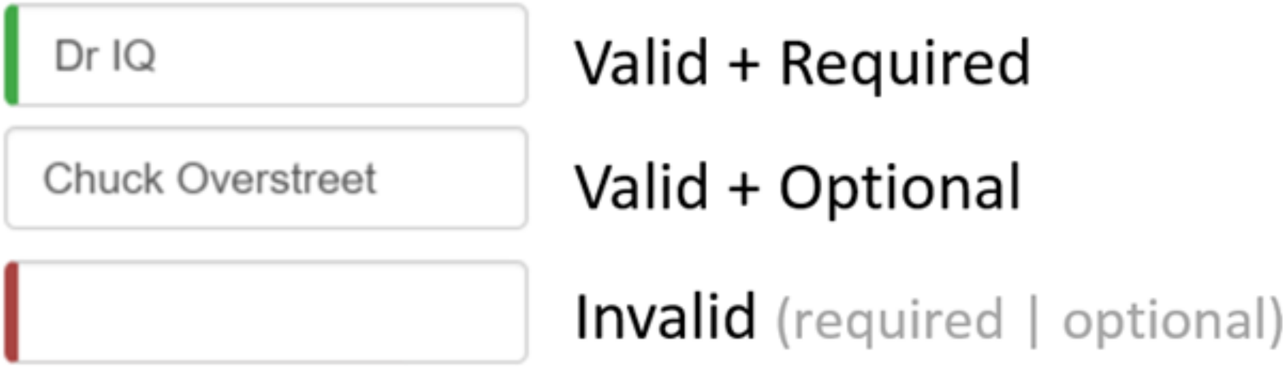
TODO: remove this: form-control ng-touched ng-dirty ng-invalid

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)

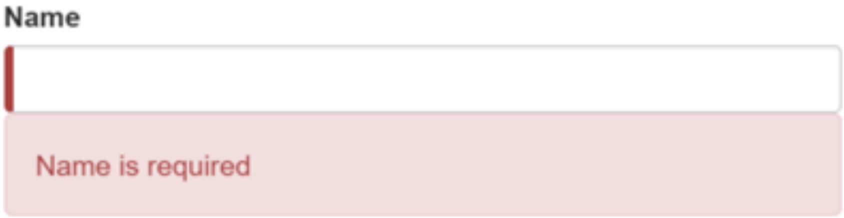
You achieve this effect by adding these class definitions to a new `forms.css` file that you add to the project as a sibling to `index.html` :

Update the `<head>` of `index.html` to include this style sheet:

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

Name is required

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:

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](api/core/Directive) 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.

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.

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.

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:

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:

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.

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:

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:

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:

Angular Glossary

Angular has its own vocabulary. Most Angular terms are common English words with a specific meaning within the Angular system.

This glossary lists the most prominent terms and a few less familiar ones that have unusual or unexpected definitions.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

{@a A} {@a aot}

Ahead-of-time (AOT) compilation

You can compile Angular applications at build time. By compiling your application using the compiler-cli, `ngc`, you can bootstrap directly to a module factory, meaning you don't need to include the Angular compiler in your JavaScript bundle. Ahead-of-time compiled applications also benefit from decreased load time and increased performance.

Annotation

In practice, a synonym for Decoration.

{@a attribute-directive}

{@a attribute-directives}

Attribute directives

A category of directive that can listen to and modify the behavior of other HTML elements, attributes, properties, and components. They are usually represented as HTML attributes, hence the name.

For example, you can use the `ngClass` directive to add and remove CSS class names.

Learn about them in the *Attribute Directives* guide.

{@a B}

Barrel

A way to *roll up exports* from several ES2015 modules into a single convenient ES2015 module. The barrel itself is an ES2015 module file that re-exports *selected* exports of other ES2015 modules.

For example, imagine three ES2015 modules in a `heroes` folder:

// heroes/hero.component.ts export class HeroComponent {}

// heroes/hero.model.ts export class Hero {}

// heroes/hero.service.ts export class HeroService {}

Without a barrel, a consumer needs three import statements:

import { HeroComponent } from '../heroes/hero.component.ts'; import { Hero } from '../heroes/hero.model.ts'; import { HeroService } from '../heroes/hero.service.ts';

You can add a barrel to the `heroes` folder (called `index`, by convention) that exports all of these items:

export * from './hero.model.ts'; // re-export all of its exports export * from './hero.service.ts'; // re-export all of its exports export { HeroComponent } from './hero.component.ts'; // re-export the named thing

Now a consumer can import what it needs from the barrel.

import { Hero, HeroService } from '../heroes'; // index is implied

The Angular scoped packages each have a barrel named `index`.

You can often achieve the same result using [NgModules](guide/glossary#ngmodule) instead.

Binding

Usually refers to data binding and the act of binding an HTML object property to a data object property.

Sometimes refers to a dependency-injection binding between a "token"—also referred to as a "key"—and a dependency provider.

Bootstrap

You launch an Angular application by "bootstrapping" it using the application root NgModule (`AppModule`). Bootstrapping identifies an application's top level "root" [component](guide/glossary#component), which is the first component that is loaded for the application. You can bootstrap multiple apps in the same `index.html`, each app with its own top-level root. {@a C} ## camelCase The practice of writing compound words or phrases such that each word or abbreviation begins with a capital letter _except the first letter, which is lowercase_. Function, property, and method names are typically spelled in camelCase. For example, `square`, `firstName`, and `getHeroes`. Notice that `square` is an example of how you write a single word in camelCase. camelCase is also known as *lower camel case* to distinguish it from *upper camel case*, or [PascalCase](guide/glossary#pascalcase). In Angular documentation, "camelCase" always means *lower camel case*. ## CLI The Angular CLI is a `command line interface` tool that can create a project, add files, and perform a variety of ongoing development tasks such as testing, bundling, and deployment. Learn more in the [Getting Started](guide/quickstart) guide. {@a component} ## Component An Angular class responsible for exposing data to a [view](guide/glossary#view) and handling most of the view's display and user-interaction logic. The *component* is one of the most important building blocks in the Angular system. It is, in fact, an Angular [directive](guide/glossary#directive) with a companion [template](guide/glossary#template). Apply the `@Component` [decorator](guide/glossary#decorator) to the component class, thereby attaching to the class the essential component metadata that Angular needs to create a component instance and render the component with its template as a view. Those familiar with "MVC" and "MVVM" patterns will recognize the component in the role of "controller" or "view model". {@a D} ## dash-case The practice of writing compound words or phrases such that each word is separated by a dash or hyphen (`-`). This form is also known as kebab-case. [Directive](guide/glossary#directive) selectors (like `my-app`) and the root of filenames (such as `hero-list.component.ts`) are often spelled in dash-case. ## Data binding Applications display data values to a user and respond to user actions (such as clicks, touches, and keystrokes). In data binding, you declare the relationship between an HTML widget and data source and let the framework handle the details. Data binding is an alternative to manually pushing application data values into HTML, attaching event listeners, pulling changed values from the screen, and updating application data values. Angular has a rich data-binding framework with a variety of data-binding operations and supporting declaration syntax. Read about the following forms of binding in the [Template Syntax](guide/template-syntax) page: * [Interpolation](guide/template-syntax#interpolation). * [Property binding](guide/template-syntax#property-binding). * [Event binding](guide/template-syntax#event-binding). * [Attribute binding](guide/template-syntax#attribute-binding). * [Class binding](guide/template-syntax#class-binding). * [Style binding](guide/template-syntax#style-binding). * [Two-way data binding with ngModel](guide/template-syntax#ngModel). {@a decorator} {@a decoration} ## Decorator | decoration A *function* that adds metadata to a class, its members (properties, methods) and function arguments. Decorators are an experimental (stage 2), JavaScript language [feature](https://github.com/wycats/javascript-decorators). TypeScript adds support for decorators. To apply a decorator, position it immediately above or to the left of the item it decorates. Angular has its own set of decorators to help it interoperate with your application parts. The following example is a `@Component` decorator that identifies a class as an Angular [component](guide/glossary#component) and an `@Input` decorator applied to the `name` property of that component. The elided object argument to the `@Component` decorator would contain the pertinent component metadata. ``` @Component({...}) export class AppComponent { constructor(@Inject('SpecialFoo') public foo:Foo) {} @Input() name:string; } ``` The scope of a decorator is

limited to the language feature that it decorates. None of the decorations shown here will "leak" to other classes that follow it in the file.
Always include parentheses `()` when applying a decorator.

Dependency injection

A design pattern and mechanism for creating and delivering parts of an application to other parts of an application that request them.

Angular developers prefer to build applications by defining many simple parts that each do one thing well and then wiring them together at runtime.

These parts often rely on other parts. An Angular component part might rely on a service part to get data or perform a calculation. When part "A" relies on another part "B," you say that "A" depends on "B" and that "B" is a dependency of "A."

You can ask a "dependency injection system" to create "A" for us and handle all the dependencies. If "A" needs "B" and "B" needs "C," the system resolves that chain of dependencies and returns a fully prepared instance of "A."

Angular provides and relies upon its own sophisticated dependency-injection system to assemble and run applications by "injecting" application parts into other application parts where and when needed.

At the core, an `injector` returns dependency values on request. The expression `injector.get(token)` returns the value associated with the given token.

A token is an Angular type (`InjectionToken`). You rarely need to work with tokens directly; most methods accept a class name (`Foo`) or a string ("foo") and Angular converts it to a token. When you write `injector.get(Foo)` , the injector returns the value associated with the token for the `Foo` class, typically an instance of `Foo` itself.

During many of its operations, Angular makes similar requests internally, such as when it creates a `component` for display.

The `Injector` maintains an internal map of tokens to dependency values. If the `Injector` can't find a value for a given token, it creates a new value using a `Provider` for that token.

A provider is a recipe for creating new instances of a dependency value associated with a particular token.

An injector can only create a value for a given token if it has a `provider` for that token in its internal provider registry. Registering providers is a critical preparatory step.

Angular registers some of its own providers with every injector. You can register your own providers.

Read more in the Dependency Injection page.

{@a directive}

{@a directives}

Directive

An Angular class responsible for creating, reshaping, and interacting with HTML elements in the browser DOM. The directive is Angular's most fundamental feature.

A directive is usually associated with an HTML element or attribute. This element or attribute is often referred to as the directive itself.

When Angular finds a directive in an HTML template, it creates the matching directive class instance and gives the instance control over that portion of the browser DOM.

You can invent custom HTML markup (for example, `<my-directive>`) to associate with your custom directives. You add this custom markup to HTML templates as if you were writing native HTML. In this way, directives become extensions of HTML itself.

Directives fall into one of the following categories:

- Components combine application logic with an HTML template to render application views. Components are usually represented as HTML elements. They are the building blocks of an Angular application.
- Attribute directives can listen to and modify the behavior of other HTML elements, attributes, properties, and components. They are usually represented as HTML attributes, hence the name.
- Structural directives are responsible for shaping or reshaping HTML layout, typically by adding, removing, or manipulating elements and their children.

{@a E}

ECMAScript

The official JavaScript language specification.

The latest approved version of JavaScript is ECMAScript 2017 (also known as "ES2017" or "ES8"). Many Angular developers write their applications in ES8 or a dialect that strives to be compatible with it, such as

TypeScript.

Most modern browsers only support the much older "ECMAScript 5" (also known as "ES5") standard. Applications written in ES2017, ES2016, ES2015, or one of their dialects must be transpiled to ES5 JavaScript.

Angular developers can write in ES5 directly.

ES2015

Short hand for ECMAScript 2015.

ES5

Short hand for ECMAScript 5, the version of JavaScript run by most modern browsers.

ES6

Short hand for ECMAScript 2015.

{@a F}

{@a G}

{@a H}

{@a I}

Injector

An object in the Angular dependency-injection system that can find a named dependency in its cache or create a dependency with a registered provider.

Input

A directive property that can be the *target* of a property binding (explained in detail in the Template Syntax page). Data values flow *into* this property from the data source identified in the template expression to the right of the equal sign.

See the Input and output properties section of the Template Syntax page.

Interpolation

A form of property data binding in which a template expression between double-curly braces renders as text. That text may be concatenated with neighboring text before it is assigned to an element property or displayed between element tags, as in this example.

My current hero is {{hero.name}}

Read more about interpolation in the Template Syntax page.

{@a J}

{@a jit}

Just-in-time (JIT) compilation

A bootstrapping method of compiling components and modules in the browser and launching the application dynamically. Just-in-time mode is a good choice during development. Consider using the ahead-of-time mode for production apps.

{@a K}

kebab-case

See dash-case.

{@a L}

Lifecycle hooks

Directives and components have a lifecycle managed by Angular as it creates, updates, and destroys them.

You can tap into key moments in that lifecycle by implementing one or more of the lifecycle hook interfaces.

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`.

Angular calls these hook methods in the following order:

- `ngOnChanges`: when an input/output binding value changes.

- `ngOnInit` : after the first `ngOnChanges` .
- `ngDoCheck` : developer's custom change detection.
- `ngAfterContentInit` : after component content initialized.
- `ngAfterContentChecked` : after every check of component content.
- `ngAfterViewInit` : after a component's views are initialized.
- `ngAfterViewChecked` : after every check of a component's views.
- `ngOnDestroy` : just before the directive is destroyed.

Read more in the Lifecycle Hooks page.

{@a M}

Module

Angular has the following types of modules: * [NgModules](guide/glossary#ngmodule). For details and examples, see the [NgModules](guide/ngmodule) page. * ES2015 modules, as described in this section.

A cohesive block of code dedicated to a single purpose.

Angular apps are modular.

In general, you assemble an application from many modules, both the ones you write and the ones you acquire from others.

A module *exports* something of value in that code, typically one thing such as a class; a module that needs that class *imports* it.

The structure of NgModules and the import/export syntax is based on the ES2015 module standard.

An application that adheres to this standard requires a module loader to load modules on request and resolve inter-module dependencies. Angular doesn't include a module loader and doesn't have a preference for any particular third-party library. You can use any module library that conforms to the standard.

Modules are typically named after the file in which the exported thing is defined. The Angular DatePipe class belongs to a feature module named `date_pipe` in the file `date_pipe.ts` .

You rarely access Angular feature modules directly. You usually import them from an Angular scoped package such as `@angular/core` .

{@a N}

NgModule

Helps you organize an application into cohesive blocks of functionality. An NgModule identifies the components, directives, and pipes that the application uses along with the list of external NgModules that the application needs, such as `FormsModule`. Every Angular application has an application root-module class. By convention, the class is called `AppModule` and resides in a file named `app.module.ts`. For details and examples, see [NgModules](guide/ngmodule).

{@a O}

Observable

An array whose items arrive asynchronously over time. Observables help you manage asynchronous data, such as data coming from a backend service. Observables are used within Angular itself, including Angular's event system and its HTTP client service.

To use observables, Angular uses a third-party library called Reactive Extensions (RxJS). Observables are a proposed feature for ES2016, the next version of JavaScript.

Output

A directive property that can be the *target* of event binding (read more in the event binding section of the Template Syntax page). Events stream *out* of this property to the receiver identified in the template expression to the right of the equal sign.

See the Input and output properties section of the Template Syntax page.

{@a P}

PascalCase

The practice of writing individual words, compound words, or phrases such that each word or abbreviation begins with a capital letter. Class names are typically spelled in PascalCase. For example, `Person` and `HeroDetailComponent`.

This form is also known as *upper camel case* to distinguish it from *lower camel case* or simply camelCase. In this documentation, "PascalCase" means *upper camel case* and "camelCase" means *lower camel case*.

Pipe

An Angular pipe is a function that transforms input values to output values for display in a view. Here's an example that uses the built-in `currency` pipe to display a numeric value in the local currency.

Price: {{product.price | currency}}

You can also write your own custom pipes. Read more in the page on pipes.

Provider

A *provider* creates a new instance of a dependency for the dependency injection system. It relates a lookup token to code—sometimes called a "recipe"—that can create a dependency value.

{@a Q}

{@a R}

Reactive forms

A technique for building Angular forms through code in a component. The alternative technique is template-driven forms.

When building reactive forms:

- The "source of truth" is the component. The validation is defined using code in the component.
- Each control is explicitly created in the component class with `new FormControl()` or with `FormBuilder`.
- The template input elements do *not* use `ngModel`.
- The associated Angular directives are all prefixed with `Form`, such as `FormGroup`, `FormControl`, and `FormControlName`.

Reactive forms are powerful, flexible, and a good choice for more complex data-entry form scenarios, such as dynamic generation of form controls.

Router

Most applications consist of many screens or views. The user navigates among them by clicking links and buttons, and performing other similar actions that cause the application to replace one view with another.

The Angular component router is a richly featured mechanism for configuring and managing the entire view navigation process, including the creation and destruction of views.

In most cases, components become attached to a router by means of a `RouterConfig` that defines routes to views.

A routing component's template has a `RouterOutlet` element where it can display views produced by the router.

Other views in the application likely have anchor tags or buttons with `RouterLink` directives that users can click to navigate.

For more information, see the Routing & Navigation page.

Router module

A separate NgModule that provides the necessary service providers and directives for navigating through application views.

For more information, see the Routing & Navigation page.

Routing component

An Angular component with a `RouterOutlet` that displays views based on router navigations.

For more information, see the Routing & Navigation page.

{@a S}

Scoped package

A way to group related *npm* packages. Read more at the npm-scope page.

NgModules are delivered within *scoped packages* such as `@angular/core`, `@angular/common`, `@angular/platform-browser-dynamic`, `@angular/http`, and `@angular/router`.

Import a scoped package the same way that you import a normal package. The only difference, from a consumer perspective, is that the scoped package name begins with the Angular *scope name*, `@angular`.

Service

For data or logic that is not associated with a specific view or that you want to share across components, build services.

Applications often require services such as a hero data service or a logging service.

A service is a class with a focused purpose. You often create a service to implement features that are independent from any specific view, provide shared data or logic across components, or encapsulate external interactions.

Applications often require services such as a data service or a logging service.

For more information, see the Services page of the Tour of Heroes tutorial.

{@a snake-case}

snake_case

The practice of writing compound words or phrases such that an underscore (`_`) separates one word from the next. This form is also known as *underscore case*.

{@a structural-directive}

{@a structural-directives}

Structural directives

A category of directive that can shape or reshape HTML layout, typically by adding and removing elements in the DOM. The `ngIf` "conditional element" directive and the `ngFor` "repeater" directive are well-known examples.

Read more in the Structural Directives page.

{@a T}

Template

A chunk of HTML that Angular uses to render a view with the support and guidance of an Angular directive, most notably a component.

Template-driven forms

A technique for building Angular forms using HTML forms and input elements in the view. The alternate technique is Reactive Forms.

When building template-driven forms:

- The "source of truth" is the template. The validation is defined using attributes on the individual input elements.
- Two-way binding with `ngModel` keeps the component model synchronized with the user's entry into the input elements.
- Behind the scenes, Angular creates a new control for each input element, provided you have set up a `name` attribute and two-way binding for each input.
- The associated Angular directives are all prefixed with `ng` such as `ngForm`, `ngModel`, and `ngModelGroup`.

Template-driven forms are convenient, quick, and simple. They are a good choice for many basic data-entry form scenarios.

Read about how to build template-driven forms in the Forms page.

Template expression

A TypeScript-like syntax that Angular evaluates within a data binding.

Read about how to write template expressions in the Template expressions section of the Template Syntax page.

Transpile

The process of transforming code written in one form of JavaScript (such as TypeScript) into another form of JavaScript (such as ES5).

TypeScript

A version of JavaScript that supports most ECMAScript 2015 language features such as decorators.

TypeScript is also notable for its optional typing system, which provides compile-time type checking and strong tooling support (such as "intellisense," code completion, refactoring, and intelligent search). Many code editors and IDEs support TypeScript either natively or with plugins.

TypeScript is the preferred language for Angular development, although you can use other JavaScript dialects such as ES5.

Read more about TypeScript at typescriptlang.org.

{@a U}

{@a V}

View

A portion of the screen that displays information and responds to user actions such as clicks, mouse moves, and keystrokes.

Angular renders a view under the control of one or more directives, especially component directives and their companion templates. The component plays such a prominent role that it's often convenient to refer to a component as a view.

Views often contain other views. Any view might be loaded and unloaded dynamically as the user navigates through the application, typically under the control of a router.

{@a W}

{@a X}

{@a Y}

{@a Z}

Zone

A mechanism for encapsulating and intercepting a JavaScript application's asynchronous activity.

The browser DOM and JavaScript have a limited number of asynchronous activities, such as DOM events (for example, clicks), promises, and XHR calls to remote servers.

Zones intercept all of these activities and give a "zone client" the opportunity to take action before and after the async activity finishes.

Angular runs your application in a zone where it can respond to asynchronous events by checking for data changes and updating the information it displays via data bindings.

Learn more about zones in this Brian Ford video.

Hierarchical Dependency Injectors

You learned the basics of Angular Dependency injection in the Dependency Injection guide.

Angular has a *Hierarchical Dependency Injection* system. There is actually a tree of injectors that parallel an application'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.

Try the .

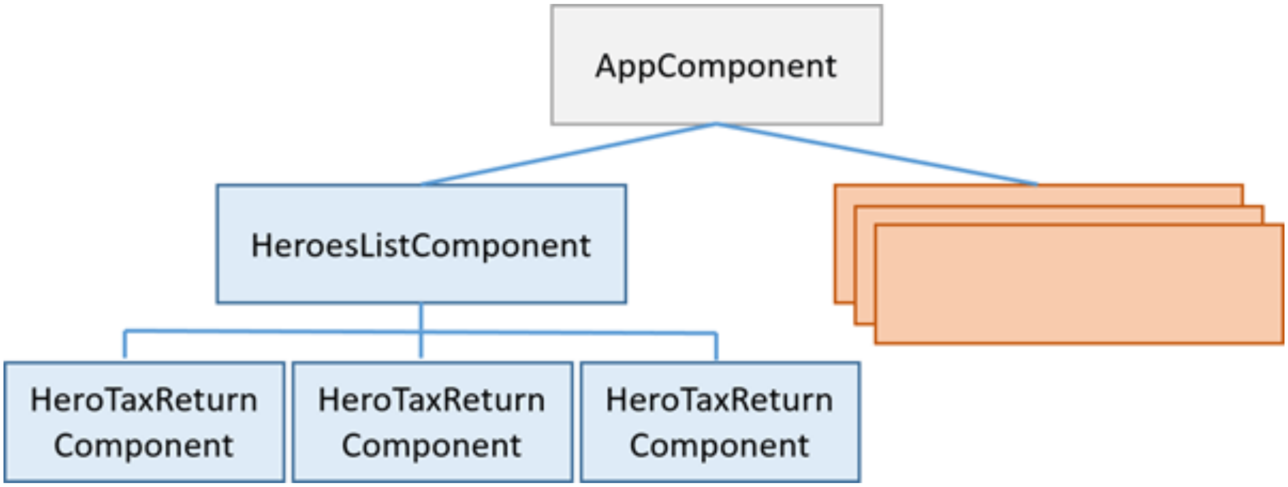
The injector tree

In the Dependency Injection guide, you learned how to configure a dependency injector and how to retrieve dependencies where you need them.

In fact, there is no such thing as ***the*** injector. An application may have multiple injectors. An Angular application is a tree of components. Each component instance has its own injector. The tree of components parallels the tree of injectors.

The component's injector may be a _proxy_ for an ancestor injector higher in the component tree. That's an implementation detail that improves efficiency. You won't notice the difference and your mental model should be that every component has its own injector.

Consider this guide's variation on the Tour of Heroes application. At the top is the `AppComponent` which has some sub-components. One of them is the `HeroesListComponent`. The `HeroesListComponent` holds and manages multiple instances of the `HeroTaxReturnComponent`. The following diagram represents the state of the this guide's three-level component tree when there are three instances of `HeroTaxReturnComponent` open simultaneously.



Injector bubbling

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 it along to *its* parent injector. 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.

You can cap the bubbling. An intermediate component can declare that it is the "host" component. The hunt for providers will climb no higher than the injector for that host component. This is a topic for another day.

Re-providing a service at different levels

You can re-register a provider for a particular dependency token at multiple levels of the injector tree. You don't *have* to re-register providers. You shouldn't do so unless you have a good reason. But you *can*.

As the resolution logic works upwards, the first provider encountered wins. Thus, a provider in an intermediate injector intercepts a request for a service from something lower in the tree. It effectively "reconfigures" and "shadows" a provider at a higher level in the tree.

If you only specify providers at the top level (typically the root `AppModule`), the tree of injectors appears to be flat. All requests bubble up to the root `NgModule` injector that you configured with the `bootstrapModule` method.

Component injectors

The ability to configure one or more providers at different levels opens up interesting and useful possibilities.

Scenario: service isolation

Architectural reasons may lead you to restrict access to a service to the application domain where it belongs.

The guide sample includes a `VillainsListComponent` that displays a list of villains. It gets those villains from a `VillainsService`.

While you *could* provide `VillainsService` in the root `AppModule` (that's where you'll find 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. That's not supposed to happen but providing the service in the root `AppModule` creates that risk.

Instead, provide the `VillainsService` in the `providers` metadata of the `VillainsListComponent` like this:

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 exist 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

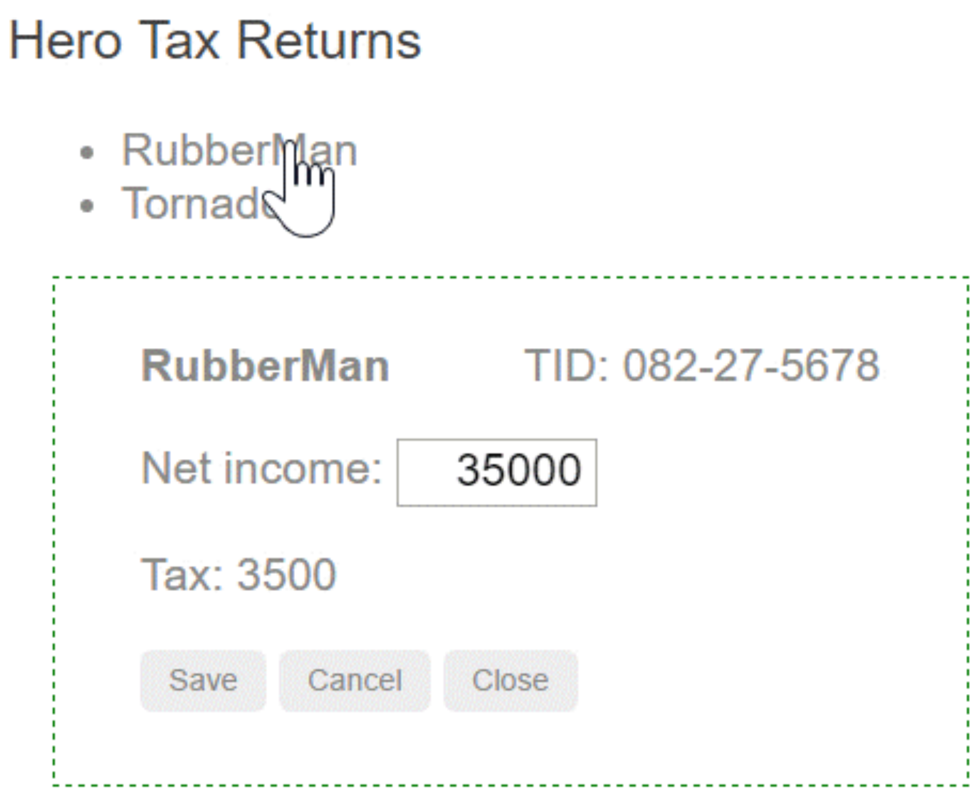
Many applications allow users to work on several open tasks at the same time. For example, in a tax preparation application, the preparer could be working on several tax returns, switching from one to the other throughout the day.

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

RubberMan TID: 082-27-5678

Net income: 35000

Tax: 3500

One might 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 might 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.

Here is the `HeroTaxReturnComponent` that makes use of it.

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.

There'd be big trouble if *this* service were an application-wide singleton. Every component would share the same service instance. Each component would overwrite the tax return that belonged to another hero. What a mess!

Look closely at the metadata for the `HeroTaxReturnComponent`. Notice the `providers` property.

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. No tax return overwriting. No mess.

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 .

Scenario: specialized providers

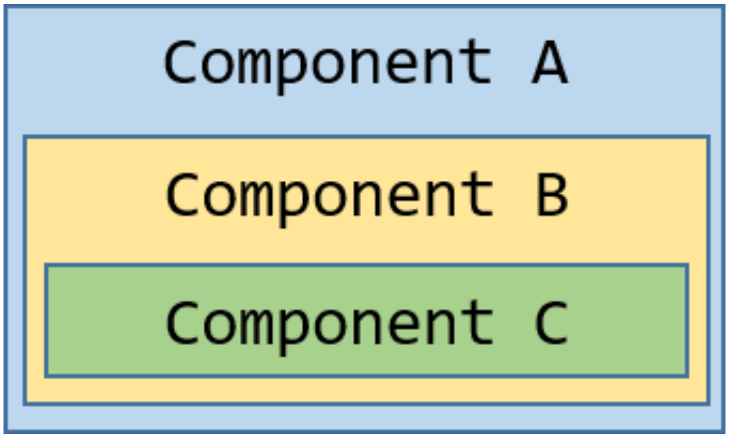
Another reason to re-provide a service is to substitute a *more specialized* implementation of that service, deeper in the component tree.

Consider again the Car example from the Dependency Injection guide. 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.

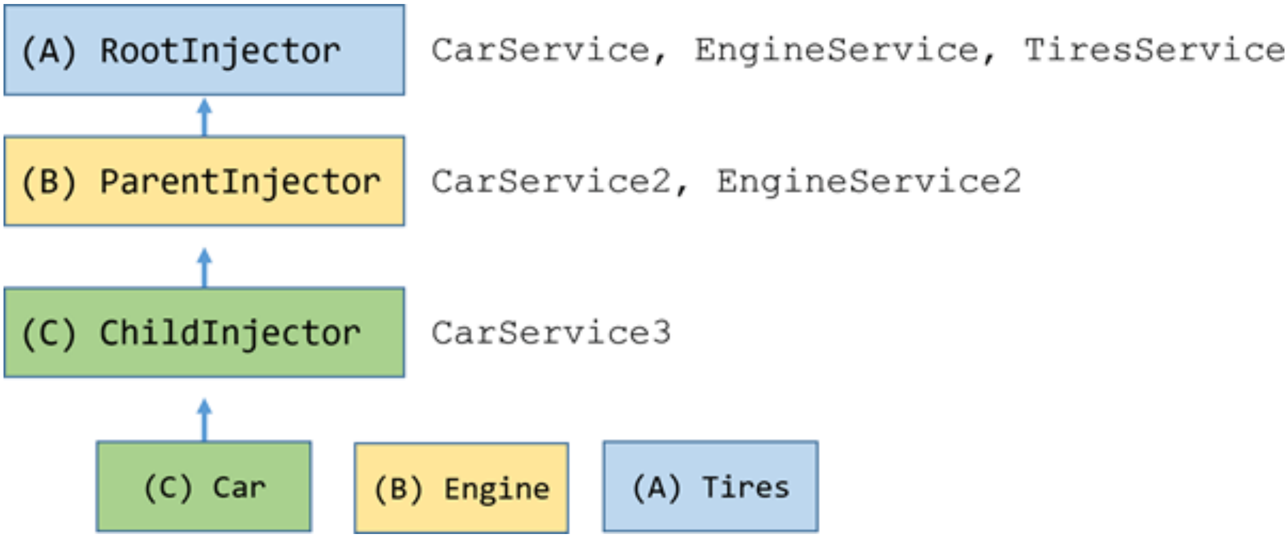
Then you create a child component (B) that defines its own, *specialized* providers for `CarService` and `EngineService` that have special capabilites suitable for whatever is going on in component (B).

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 .

HttpClient

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.

With `HttpClient`, `@angular/common/http` provides a simplified API for HTTP functionality for use with Angular applications, building on top of the `XMLHttpRequest` interface exposed by browsers. Additional benefits of `HttpClient` include testability support, strong typing of request and response objects, request and response interceptor support, and better error handling via apis based on Observables.

Setup: installing the module

Before you can use the `HttpClient`, you need to install the `HttpClientModule` which provides it. This can be done in your application module, and is only necessary once.

```
// app.module.ts:

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

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

@NgModule({
  imports: [
    BrowserModule,
    // Include it under 'imports' in your application module
    // after BrowserModule.
    HttpClientModule,
  ],
})
export class MyAppModule {}
```

Once you import `HttpClientModule` into your app module, you can inject `HttpClient` into your components and services.

Making a request for JSON data

The most common type of request applications make to a backend is to request JSON data. For example, suppose you have an API endpoint that lists items, `/api/items` , which returns a JSON object of the form:

```
{
  "results": [
    "Item 1",
    "Item 2",
  ]
}
```

The `get()` method on `HttpClient` makes accessing this data straightforward.

```
@Component(...)
export class MyComponent implements OnInit {

  results: string[];

  // Inject HttpClient into your component or service.
  constructor(private http: HttpClient) {}

  ngOnInit(): void {
    // Make the HTTP request:
    this.http.get('/api/items').subscribe(data => {
      // Read the result field from the JSON response.
      this.results = data['results'];
    });
  }
}
```

Typechecking the response

In the above example, the `data['results']` field access stands out because you use bracket notation to access the results field. If you tried to write `data.results` , TypeScript would correctly complain that the `Object` coming back from HTTP does not have a `results` property. That's because while `HttpClient` parsed the JSON response into an `Object` , it doesn't know what shape that object is.

You can, however, tell `HttpClient` what type the response will be, which is recommended. To do so, first you define an interface with the correct shape:

```
interface ItemsResponse {
  results: string[];
}
```

Then, when you make the `HttpClient.get` call, pass a type parameter:

```
http.get<ItemsResponse>('/api/items').subscribe(data => {
  // data is now an instance of type ItemsResponse, so you can do this:
  this.results = data.results;
});
```

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, and inspecting those can be necessary. To do this, you can tell `HttpClient` you want the full response instead of just the body with the `observe` option:

```
http
  .get<MyJsonData>('/data.json', {observe: 'response'})
  .subscribe(resp => {
    // Here, resp is of type HttpResponse<MyJsonData>.
    // You can inspect its headers:
    console.log(resp.headers.get('X-Custom-Header'));
    // And access the body directly, which is typed as MyJsonData as requested.
    console.log(resp.body.someField);
  });
```

As you can see, the resulting 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* instead of a successful response.

To handle it, add an error handler to your `.subscribe()` call:

```
http
  .get<ItemsResponse>('/api/items')
  .subscribe(
    // Successful responses call the first callback.
    data => {...},
    // Errors will call this callback instead:
    err => {
      console.log('Something went wrong!');
    }
  );
```

Getting error details

Detecting that an error occurred is one thing, but it's more useful to know what error actually occurred. The `err` parameter to the callback above is of type `HttpErrorResponse`, and contains useful information on what went wrong.

There are two types of errors that can occur. If the backend returns an unsuccessful response code (404, 500, etc.), it gets returned as an error. Also, if something goes wrong client-side, such as an exception gets thrown in an RxJS operator, or if a network error prevents the request from completing successfully, an actual `Error` will be thrown.

In both cases, you can look at the `HttpErrorResponse` to figure out what happened.

```
http
  .get<ItemsResponse>('/api/items')
  .subscribe(
    data => {...},
    (err: HttpErrorResponse) => {
      if (err.error instanceof Error) {
        // A client-side or network error occurred. Handle it accordingly.
        console.log('An error occurred:', err.error.message);
      } else {
        // The backend returned an unsuccessful response code.
        // The response body may contain clues as to what went wrong,
        console.log(`Backend returned code ${err.status}, body was: ${err.error}`);
      }
    }
  );
```

`.retry()`

One way to deal with errors is to simply retry the request. This strategy can be useful when the errors are transient and unlikely to repeat.

RxJS has a useful operator called `.retry()`, which automatically resubscribes to an Observable, thus reissuing the request, upon encountering an error.

First, import it:

```
import 'rxjs/add/operator/retry';
```

Then, you can use it with HTTP Observables like this:

```
http
  .get<ItemsResponse>('/api/items')
  // Retry this request up to 3 times.
  .retry(3)
  // Any errors after the 3rd retry will fall through to the app.
  .subscribe(...);
```

Requesting non-JSON data

Not all APIs return JSON data. Suppose you want to read a text file on the server. You have to tell `HttpClient` that you expect a textual response:

```
http
  .get('/textfile.txt', {responseType: 'text'})
  // 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().
  .subscribe(data => console.log(data));
```

Sending data to the server

In addition to fetching data from the server, `HttpClient` supports mutating requests, that is, sending data to the server in various forms.

Making a POST request

One common operation is to POST data to a server; for example when submitting a form. The code for sending a POST request is very similar to the code for GET:

```
const body = {name: 'Brad'};

http
  .post('/api/developers/add', body)
  // See below - subscribe() is still necessary when using post().
  .subscribe(...);
```

Note the `subscribe()` method. All Observables returned from `HttpClient` are _cold_, which is to say that they are _blueprints_ for making requests. Nothing will happen until you call `subscribe()`, and every such call will make a separate request. For example, this code sends a POST request with the same data twice: ```javascript const req = http.post('/api/items/add', body); // 0 requests made - .subscribe() not called.

req.subscribe(); // 1 request made. req.subscribe(); // 2 requests made. ```

Configuring other parts of the request

Besides the URL and a possible request body, there are other aspects of an outgoing request which you may wish to configure. All of these are available via an options object, which you pass to the request.

Headers

One common task is adding an `Authorization` header to outgoing requests. Here's how you do that:

```
http
  .post('/api/items/add', body, {
    headers: new HttpHeaders().set('Authorization', 'my-auth-token'),
  })
  .subscribe();
```

The `HttpHeaders` class is immutable, so every `set()` returns a new instance and applies the changes.

URL Parameters

Adding URL parameters works in the same way. To send a request with the `id` parameter set to `3`, you would do:

```
http
  .post('/api/items/add', body, {
    params: new HttpParams().set('id', '3'),
  })
  .subscribe();
```

In this way, you send the POST request to the URL `/api/items/add?id=3`.

Advanced usage

The above sections detail how to use the basic HTTP functionality in `@angular/common/http`, but sometimes you need to do more than just make requests and get data back.

Intercepting all requests or responses

A major feature of `@angular/common/http` is *interception*, the ability to declare interceptors which sit in between your application and the backend. When your application makes a request, interceptors transform it before sending it to the server, and the interceptors can transform the response on its way back before your application sees it. This is useful for everything from authentication to logging.

Writing an interceptor

To implement an interceptor, you declare a class that implements `HttpInterceptor`, which has a single `intercept()` method. Here is a simple interceptor which does nothing but forward the request through without altering it:

```
import {Injectable} from '@angular/core';
import {HttpEvent, HttpInterceptor, HttpHandler, HttpRequest} from '@angular/common/http';

@Injectable()
export class NoopInterceptor implements HttpInterceptor {
  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    return next.handle(req);
  }
}
```

`intercept` is a method which transforms a request into an Observable that eventually returns the response. In this sense, each interceptor is entirely responsible for handling the request by itself.

Most of the time, though, interceptors will make some minor change to the request and forward it to the rest of the chain. That's where the `next` parameter comes in. `next` is an `HttpHandler`, an interface that, similar to `intercept`, transforms a request into an Observable for the response. In an interceptor, `next` always represents the next interceptor in the chain, if any, or the final backend if there are no more interceptors. So most interceptors will end by calling `next` on the request they transformed.

Our do-nothing handler simply calls `next.handle` on the original request, forwarding it without mutating it at all.

This pattern is similar to those in middleware frameworks such as Express.js.

Providing your interceptor

Simply declaring the `NoopInterceptor` above doesn't cause your app to use it. You need to wire it up in your app module by providing it as an interceptor, as follows:

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

@NgModule({
  providers: [{
    provide: HTTP_INTERCEPTORS,
    useClass: NoopInterceptor,
    multi: true,
  }],
})
export class AppModule {}
```

Note the `multi: true` option. This is required and tells Angular that `HTTP_INTERCEPTORS` is an array of values, rather than a single value.

Events

You may have also noticed that the Observable returned by `intercept` and `HttpHandler.handle` is not an `Observable<HttpResponse<any>>` but an `Observable<HttpEvent<any>>`. That's because interceptors work at a lower level than the `HttpClient` interface. A single request can generate multiple events, including upload and download progress events. The `HttpResponse` class is actually an event itself, with a `type` of `HttpEventType.HttpResponseEvent`.

An interceptor must pass through all events that it does not understand or intend to modify. It must not filter out events it didn't expect to process. Many interceptors are only concerned with the outgoing request, though, and will simply return the event stream from `next` without modifying it.

Ordering

When you provide multiple interceptors in an application, Angular applies them in the order that you provided them.

Immutability

Interceptors exist to examine and mutate outgoing requests and incoming responses. However, it may be surprising to learn that the `HttpRequest` and `HttpResponse` classes are largely immutable.

This is for a reason: because the app may retry requests, the interceptor chain may process an individual request multiple times. If requests were mutable, a retried request would be different than the original request. Immutability ensures the interceptors see the same request for each try.

There is one case where type safety cannot protect you when writing interceptors—the request body. It is invalid to mutate a request body within an interceptor, but this is not checked by the type system.

If you have a need to mutate the request body, you need to copy the request body, mutate the copy, and then use `clone()` to copy the request and set the new body.

Since requests are immutable, they cannot be modified directly. To mutate them, use `clone()`:

```
intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
  // This is a duplicate. It is exactly the same as the original.
  const dupReq = req.clone();

  // Change the URL and replace 'http://' with 'https://'
  const secureReq = req.clone({url: req.url.replace('http://', 'https://')});
}
```

As you can see, the hash accepted by `clone()` allows you to mutate specific properties of the request while copying the others.

Setting new headers

A common use of interceptors is to set default headers on outgoing responses. For example, assuming you have an injectable `AuthService` which can provide an authentication token, here is how you would write an interceptor which adds it to all outgoing requests:

```
import {Injectable} from '@angular/core';
import {HttpEvent, HttpInterceptor, HttpHandler, HttpRequest} from '@angular/common/h
ttp';

@Injectable()
export class AuthInterceptor implements HttpInterceptor {
  constructor(private auth: AuthService) {}

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    // Get the auth header from the service.
    const authHeader = this.auth.getAuthorizationHeader();
    // Clone the request to add the new header.
    const authReq = req.clone({headers: req.headers.set('Authorization', authHeader)}
);
    // Pass on the cloned request instead of the original request.
    return next.handle(authReq);
  }
}
```

The practice of cloning a request to set new headers is so common that there's actually a shortcut for it:

```
const authReq = req.clone({setHeaders: {Authorization: authHeader}});
```

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 log or time requests. Consider this interceptor which uses `console.log` to show how long each request takes:

```
import 'rxjs/add/operator/do';

export class TimingInterceptor implements HttpInterceptor {
  constructor(private auth: AuthService) {}

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    const started = Date.now();
    return next
      .handle(req)
      .do(event => {
        if (event instanceof HttpResponse) {
          const elapsed = Date.now() - started;
          console.log(`Request for ${req.urlWithParams} took ${elapsed} ms.`);
        }
      });
  }
}
```

Notice the RxJS `do()` operator—it adds a side effect to an Observable without affecting the values on the stream. Here, it detects the `HttpResponse` event and logs the time the request took.

Caching

You can also use interceptors to implement caching. For this example, assume that you've written an HTTP cache with a simple interface:

```
abstract class HttpCache {
  /**
   * Returns a cached response, if any, or null if not present.
   */
  abstract get(req: HttpRequest<any>): HttpResponse<any>|null;

  /**
   * Adds or updates the response in the cache.
   */
  abstract put(req: HttpRequest<any>, resp: HttpResponse<any>): void;
}
```

An interceptor can apply this cache to outgoing requests.

```
@Injectable()
export class CachingInterceptor implements HttpInterceptor {
  constructor(private cache: HttpCache) {}

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    // Before doing anything, it's important to only cache GET requests.
    // Skip this interceptor if the request method isn't GET.
    if (req.method !== 'GET') {
      return next.handle(req);
    }

    // First, check the cache to see if this request exists.
    const cachedResponse = this.cache.get(req);
    if (cachedResponse) {
      // A cached response exists. Serve it instead of forwarding
      // the request to the next handler.
      return Observable.of(cachedResponse);
    }

    // No cached response exists. Go to the network, and cache
    // the response when it arrives.
    return next.handle(req).do(event => {
      // Remember, there may be other events besides just the response.
      if (event instanceof HttpResponse) {
        // Update the cache.
        this.cache.put(req, event);
      }
    });
  }
}
```

Obviously this example glosses over request matching, cache invalidation, etc., but it's easy to see that interceptors have a lot of power beyond just transforming requests. If desired, they can be used to completely take over the request flow.

To really demonstrate their flexibility, you can change the above example to return *two* response events if the request exists in cache—the cached response first, and an updated network response later.

```
intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
  // Still skip non-GET requests.
  if (req.method !== 'GET') {
    return next.handle(req);
  }

  // This will be an Observable of the cached value if there is one,
  // or an empty Observable otherwise. It starts out empty.
  let maybeCachedResponse: Observable<HttpEvent<any>> = Observable.empty();

  // Check the cache.
  const cachedResponse = this.cache.get(req);
  if (cachedResponse) {
    maybeCachedResponse = Observable.of(cachedResponse);
  }

  // Create an Observable (but don't subscribe) that represents making
  // the network request and caching the value.
  const networkResponse = next.handle(req).do(event => {
    // Just like before, check for the HttpResponse event and cache it.
    if (event instanceof HttpResponse) {
      this.cache.put(req, event);
    }
  });

  // Now, combine the two and send the cached response first (if there is
  // one), and the network response second.
  return Observable.concat(maybeCachedResponse, networkResponse);
}
```

Now anyone doing `http.get(url)` will receive *two* responses if that URL has been cached before.

Listening to progress events

Sometimes applications need to transfer large amounts of data, and those transfers can take time. It's a good user experience practice to provide feedback on the progress of such transfers; for example, uploading files—

and `@angular/common/http` supports this.

To make a request with progress events enabled, first create an instance of `HttpRequest` with the special `reportProgress` option set:

```
const req = new HttpRequest('POST', '/upload/file', file, {
  reportProgress: true,
});
```

This option enables tracking of progress events. Remember, every progress event triggers change detection, so only turn them on if you intend to actually update the UI on each event.

Next, make the request through the `request()` method of `HttpClient`. The result will be an Observable of events, just like with interceptors:

```
http.request(req).subscribe(event => {
  // Via this API, you get access to the raw event stream.
  // Look for upload progress events.
  if (event.type === HttpEventType.UploadProgress) {
    // This is an upload progress event. Compute and show the % done:
    const percentDone = Math.round(100 * event.loaded / event.total);
    console.log(`File is ${percentDone}% uploaded.`);
  } else if (event instanceof HttpResponse) {
    console.log('File is completely uploaded!');
  }
});
```

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`'s support is 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.withConfig()` to override the defaults.

```
imports: [
  HttpClientModule,
  HttpClientXsrfModule.withConfig({
    cookieName: 'My-Xsrf-Cookie',
    headerName: 'My-Xsrf-Header',
  }),
]
```

Testing HTTP requests

Like any external dependency, the HTTP backend needs to be mocked as part of good testing practice. `@angular/common/http` provides a testing library `@angular/common/http/testing` that makes setting up such mocking straightforward.

Mocking philosophy

Angular's HTTP testing library is designed for a pattern of testing where the app executes code and makes requests first. After that, tests expect that certain requests have or have not been made, perform assertions against those requests, and finally provide responses by "flushing" each expected request, which may trigger more new requests, etc. At the end, tests can optionally verify that the app has made no unexpected requests.

Setup

To begin testing requests made through `HttpClient`, import `HttpClientTestingModule` and add it to your `TestBed` setup, like so:

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

beforeEach(() => {
  TestBed.configureTestingModule({
    ...,
    imports: [
      HttpClientTestingModule,
    ],
  })
});
```

That's it. Now requests made in the course of your tests will hit the testing backend instead of the normal backend.

Expecting and answering requests

With the mock installed via the module, you can write a test that expects a GET Request to occur and provides a mock response. The following example does this by injecting both the `HttpClient` into the test and a class called `HttpTestingController`

```
it('expects a GET request', inject([HttpClient, HttpTestingController], (http: HttpCl
ient, httpMock: HttpTestingController) => {
  // Make an HTTP GET request, and expect that it return an object
  // of the form {name: 'Test Data'}.
  http
    .get('/data')
    .subscribe(data => expect(data['name']).toEqual('Test Data'));

  // At this point, the request is pending, and no response has been
  // sent. The next step is to expect that the request happened.
  const req = httpMock.expectOne('/data');

  // If no request with that URL was made, or if multiple requests match,
  // expectOne() would throw. However this test makes only one request to
  // this URL, so it will match and return a mock request. The mock request
  // can be used to deliver a response or make assertions against the
  // request. In this case, the test asserts that the request is a GET.
  expect(req.request.method).toEqual('GET');

  // Next, fulfill the request by transmitting a response.
  req.flush({name: 'Test Data'});

  // Finally, assert that there are no outstanding requests.
  httpMock.verify();
}));
```

The last step, verifying that no requests remain outstanding, is common enough for you to move it into an `afterEach()` step:

```
afterEach(inject([HttpTestingController], (httpMock: HttpTestingController) => {
  httpMock.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:

```
const req = httpMock.expectOne((req) => req.headers.has('Authorization'));
```

Just as with the `expectOne()` by URL in the test above, if 0 or 2+ requests match this expectation, it will throw.

Handling more than one request

If you need to respond to duplicate requests in your test, use the `match()` API instead of `expectOne()`, which takes the same arguments but returns an array of matching requests. Once returned, these requests are removed from future matching and are your responsibility to verify and flush.

```
// Expect that 5 pings have been made and flush them.
const reqs = httpMock.match('/ping');
expect(reqs.length).toBe(5);
reqs.forEach(req => req.flush());
```

Internationalization (i18n)

Application internationalization is a many-faceted area of development, focused on making applications available and user-friendly to a worldwide audience. This page describes Angular's internationalization (i18n) tools, which can help you make your app available in multiple languages.

See the i18n Example for a simple example of an AOT-compiled app, translated into French.

{@a angular-i18n}

Angular and i18n

Angular simplifies the following aspects of internationalization: * Displaying dates, number, percentages, and currencies in a local format. * Translating text in component templates. * Handling plural forms of words. * Handling alternative text.

This document focuses on **Angular CLI** projects, in which the Angular CLI generates most of the boilerplate necessary to write your app in multiple languages.

{@a setting-up-locale}

Setting up the locale of your app

A locale is an identifier (id) that refers to a set of user preferences that tend to be shared within a region of the world, such as country. This document refers to a locale identifier as a "locale" or "locale id".

A Unicode locale identifier is composed of a Unicode language identifier and (optionally) the character `-` followed by a locale extension. (For historical reasons the character `_` is supported as an alternative to `-`.) For example, in the locale id `fr-CA` the `fr` refers to the French language identifier, and the `CA` refers to the locale extension Canada.

Angular follows the Unicode LDML convention that uses stable identifiers (Unicode locale identifiers) based on the norm [BCP47](http://www.rfc-editor.org/rfc/bcp/bcp47.txt). It is very important that you follow this convention when you define your locale, because the Angular i18n tools use this locale id to find the correct corresponding locale data.

By default, Angular uses the locale `en-US`, which is English as spoken in the United States of America.

To set your app's locale to another value, use the CLI parameter `--locale` with the value of the locale id that you want to use:

```
ng serve --aot --locale fr
```

If you use JIT, you also need to define the `LOCALE_ID` provider in your main module:

For more information about Unicode locale identifiers, see the [CLDR core spec](#).

For a complete list of locales supported by Angular, see [the Angular repository](#).

The locale identifiers used by CLDR and Angular are based on [BCP47](#). These specifications change over time; the following table maps previous identifiers to current ones at time of writing:

Locale name	Old locale id	New locale id
Indonesian	in	id
Hebrew	iw	he
Romanian Moldova	mo	ro-MD
Norwegian Bokmål	no, no-NO	nb
Serbian Latin	sh	sr-Latn
Filipino	tl	fil
Portuguese Brazil	pt-BR	pt
Chinese Simplified	zh-cn, zh-Hans-CN	zh-Hans
Chinese Traditional	zh-tw, zh-Hant-TW	zh-Hant
Chinese Traditional Hong Kong	zh-hk	zh-Hant-HK

i18n pipes

Angular pipes can help you with internationalization: the `DatePipe` , `CurrencyPipe` , `DecimalPipe` and `PercentPipe` use locale data to format data based on the `LOCALE_ID` .

By default, Angular only contains locale data for `en-US` . If you set the value of `LOCALE_ID` to another locale, you must import locale data for that new locale. The CLI imports the locale data for you when you use the parameter `--locale` with `ng serve` and `ng build` .

If you want to import locale data for other languages, you can do it manually:

The first parameter is an object containing the locale data imported from `@angular/common/locales`. By default, the imported locale data is registered with the locale id that is defined in the Angular locale data itself. If you want to register the imported locale data with another locale id, use the second parameter to specify a custom locale id. For example, Angular's locale data defines the locale id for French as "fr". You can use the second parameter to associate the imported French locale data with the custom locale id "fr-FR" instead of "fr".

The files in `@angular/common/locales` contain most of the locale data that you need, but some advanced formatting options might only be available in the extra dataset that you can import from `@angular/common/locales/extra`. An error message informs you when this is the case.

All locale data used by Angular are extracted from the Unicode Consortium's [Common Locale Data Repository (CLDR)](#).

Template translations

This document refers to a unit of translatable text as "text," a "message", or a "text message."

The i18n template translation process has four phases:

1. Mark static text messages in your component templates for translation.
2. An Angular i18n tool extracts the marked text into an industry standard translation source file.
3. A translator edits that file, translating the extracted text into the target language, and returns the file to you.
4. The Angular compiler imports the completed translation files, replaces the original messages with translated text, and generates a new version of the app in the target language.

You need to build and deploy a separate version of the app for each supported language.

{@a i18n-attribute}

Mark text with the i18n attribute

The Angular `i18n` attribute marks translatable content. Place it on every element tag whose fixed text is to be translated.

In the example below, an `<h1>` tag displays a simple English language greeting, "Hello i18n!"

To mark the greeting for translation, add the `i18n` attribute to the `<h1>` tag.

`i18n` is a custom attribute, recognized by Angular tools and compilers. After translation, the compiler removes it. It is not an Angular directive.

{@a help-translator}

Help the translator with a description and meaning

To translate a text message accurately, the translator may need additional information or context.

You can add a description of the text message as the value of the `i18n` attribute, as shown in the example below:

The translator may also need to know the meaning or intent of the text message within this particular app context.

You add context by beginning the `i18n` attribute value with the *meaning* and separating it from the *description* with the `|` character: `<meaning>|<description>`

All occurrences of a text message that have the same meaning will have the same translation. A text message that is associated with different meanings can have different translations.

The Angular extraction tool preserves both the meaning and the description in the translation source file to facilitate contextually-specific translations, but only the combination of meaning and text message are used to generate the specific id of a translation. If you have two similar text messages with different meanings, they are extracted separately. If you have two similar text messages with different descriptions (not different meanings), then they are extracted only once.

{@a custom-id}

Set a custom id for persistence and maintenance

The angular i18n extractor tool generates a file with a translation unit entry for each `i18n` attribute in a template. By default, it assigns each translation unit a unique id such as this one:

When you change the translatable text, the extractor tool generates a new id for that translation unit. You must then update the translation file with the new id.

Alternatively, you can specify a custom id in the `i18n` attribute by using the prefix `@@`. The example below defines the custom id `introductionHeader`:

When you specify a custom id, the extractor tool and compiler generate a translation unit with that custom id.

The custom id is persistent. The extractor tool does not change it when the translatable text changes.

Therefore, you do not need to update the translation. This approach makes maintenance easier.

Use a custom id with a description

You can use a custom id in combination with a description by including both in the value of the `i18n` attribute. In the example below, the `i18n` attribute value includes a description, followed by the custom `id`:

You also can add a meaning, as shown in this example:

Define unique custom ids

Be sure to define custom ids that are unique. If you use the same id for two different text messages, only the first one is extracted, and its translation is used in place of both original text messages.

In the example below the custom id `myId` is used for two different messages:

```
<h3 i18n="@@myId">Hello</h3>
<!-- ... -->
<p i18n="@@myId">Good bye</p>
```

Consider this translation to French:

```
<trans-unit id="myId" datatype="html">
  <source>Hello</source>
  <target state="new">Bonjour</target>
</trans-unit>
```

Because the custom id is the same, both of the elements in the resulting translation contain the same text, `Bonjour`:

```
<h3>Bonjour</h3>
<!-- ... -->
<p>Bonjour</p>
```

{@a no-element}

Translate text without creating an element

If there is a section of text that you would like to translate, you can wrap it in a `<span>` tag. However, if you don't want to create a new DOM element merely to facilitate translation, you can wrap the text in an `<ng-container>` element. The `<ng-container>` is transformed into an html comment:

{@a translate-attributes}

Add i18n translation attributes

You also can translate attributes. For example, assume that your template has an image with a `title` attribute:

This `title` attribute needs to be translated.

To mark an attribute for translation, add an attribute in the form of `i18n-x`, where `x` is the name of the attribute to translate. The following example shows how to mark the `title` attribute for translation by adding the `i18n-title` attribute on the `img` tag:

This technique works for any attribute of any element.

You also can assign a meaning, description, and id with the `i18n-x="<meaning>|<description>@@<id>"` syntax.

{@a plural-ICU}

Translate singular and plural

Different languages have different pluralization rules.

Suppose that you want to say that something was "updated x minutes ago". In English, depending upon the number of minutes, you could display "just now", "one minute ago", or "x minutes ago" (with x being the actual number). Other languages might express the cardinality differently.

The example below shows how to use a `plural` ICU expression to display one of those three options based on when the update occurred:

- The first parameter is the key. It is bound to the component property (`minutes`), which determines the number of minutes.
- The second parameter identifies this as a `plural` translation type.
- The third parameter defines a pluralization pattern consisting of pluralization categories and their matching values.

This syntax conforms to the [ICU Message Format] as specified in the [CLDR pluralization rules].

Pluralization categories include (depending on the language):

- =0 (or any other number)
- zero
- one
- two
- few
- many
- other

After the pluralization category, put the default English text in braces (`{}`).

In the example above, the three options are specified according to that pluralization pattern. For talking about about zero minutes, you use `=0 {just now}`. For one minute, you use `=1 {one minute}`. Any unmatched cardinality uses `other {{{minutes}} minutes ago}`. You could choose to add patterns for two, three, or any other number if the pluralization rules were different. For the example of "minute", only these three patterns are necessary in English.

You can use interpolations and html markup inside of your translations.

{@a select-ICU}

Select among alternative text messages

If your template needs to display different text messages depending on the value of a variable, you need to translate all of those alternative text messages.

You can handle this with a `select` ICU expression. It is similar to the `plural` ICU expressions except that you choose among alternative translations based on a string value instead of a number, and you define those string values.

The following format message in the component template binds to the component's `gender` property, which outputs one of the following string values: "m", "f" or "o". The message maps those values to the appropriate translations:

{@a nesting-ICUS}

Nesting plural and select ICU expressions

You can also nest different ICU expressions together, as shown in this example:

{@a ng-xi18n}

Create a translation source file with *ng xi18n*

Use the `ng xi18n` command provided by the CLI to extract the text messages marked with `i18n` into a translation source file.

Open a terminal window at the root of the app project and enter the `ng xi18n` command:

```
ng xi18n
```

By default, the tool generates a translation file named `messages.xlf` in the XML Localization Interchange File Format (XLIFF, version 1.2).

If you don't use the CLI, you can use the `ng-xi18n` tool directly from the `@angular/compiler-cli` package, or you can manually use the CLI Webpack plugin `ExtractI18nPlugin` from the `@ngtools/webpack` package.

{@a other-formats}

Other translation formats

Angular i18n tooling supports three translation formats: * XLIFF 1.2 (default) * XLIFF 2 * XML Message Bundle (XMB)

You can specify the translation format explicitly with the `--i18nFormat` flag as illustrated in these example commands:

```
ng xi18n --i18nFormat=xlf ng xi18n --i18nFormat=xlf2 ng xi18n --i18nFormat=xmb
```

The sample in this guide uses the default XLIFF 1.2 format.

XLIFF files have the extension .xlf. The XMB format generates .xmb source files but uses .xtb (XML Translation Bundle: XTB) translation files.

{@a ng-xi18n-options}

Other options

You can specify the output path used by the CLI to extract your translation source file with the parameter `--outputPath`:

```
ng xi18n --outputPath src/locale
```

You can change the name of the translation source file that is generated by the extraction tool with the parameter `--outFile` :

ng xi18n --outFile source.xlf

You can specify the base locale of your app with the parameter `--locale` :

ng xi18n --locale fr

The extraction tool uses the locale to add the app locale information into your translation source file. This information is not used by Angular, but external translation tools may need it.

{@a translate}

Translate text messages

The `ng xi18n` command generates a translation source file named `messages.xlf` in the project `src` folder.

The next step is to translate this source file into the specific language translation files. The example in this guide creates a French translation file.

{@a localization-folder}

Create a localization folder

Most apps are translated into more than one other language. For this reason, it is standard practice for the project structure to reflect the entire internationalization effort.

One approach is to dedicate a folder to localization and store related assets, such as internationalization files, there.

Localization and internationalization are [different but closely related terms](https://en.wikipedia.org/wiki/Internationalization_and_localization).

This guide follows that approach. It has a `locale` folder under `src/` . Assets within that folder have a filename extension that matches their associated locale.

Create the translation files

For each translation source file, there must be at least one language translation file for the resulting translation.

For this example:

1. Make a copy of the `messages.xlf` file.

2. Put the copy in the `locale` folder.
3. Rename the copy to `messages.fr.xlf` for the French language translation.

If you were translating to other languages, you would repeat these steps for each target language.

{@a translate-text-nodes}

Translate text nodes

In a large translation project, you would send the `messages.fr.xlf` file to a French translator who would enter the translations using an XLIFF file editor.

This sample file is easy to translate without a special editor or knowledge of French.

1. Open `messages.fr.xlf` and find the first `<trans-unit>` section:

> This XML element represents the translation of the `<h1>` greeting tag that you marked with the `i18n` attribute earlier in this guide.
>
> Note that the translation unit `id=introductionHeader` is derived from the custom `id` that you set earlier, but without the `@@` prefix required in the source HTML.

1. Duplicate the `<source/>` tag, rename it `target`, and then replace its content with the French greeting. If you were working with a more complex translation, you could use the the information and context provided by the source, description, and meaning elements to guide your selection of the appropriate French translation.

1. Translate the other text nodes the same way:

The Angular i18n tools generated the ids for these translation units. Don't change them. Each `id` depends upon the content of the template text and its assigned meaning. If you change either the text or the meaning, then the `id` changes. For more information, see the **[translation file maintenance discussion](#custom-id)**.

{@a translate-plural-select}

Translate plural and select expressions

Plural and *select* ICU expressions are extracted separately, so they require special attention when preparing for translation.

Look for these expressions in relation to other translation units that you recognize from elsewhere in the source template. In this example, you know the translation unit for the `select` must be just below the translation unit for the logo.

{@a translate-plural}

Translate *plural*

To translate a `plural`, translate its ICU format match values:

You can add or remove plural cases, with each language having its own cardinality. (See CLDR plural rules.)

{@a translate-select}

Translate *select*

Below is the content of our example `select` ICU expression in the component template:

The extraction tool broke that into two translation units because ICU expressions are extracted separately.

The first unit contains the text that was outside of the `select`. In place of the `select` is a placeholder, `<x id="ICU">`, that represents the `select` message. Translate the text and move around the placeholder if necessary, but don't remove it. If you remove the placeholder, the ICU expression will not be present in your translated app.

The second translation unit, immediately below the first one, contains the `select` message. Translate that as well.

Here they are together, after translation:

{@a translate-nested}

Translate a nested expression

A nested expression is similar to the previous examples. As in the previous example, there are two translation units. The first one contains the text outside of the nested expression:

The second unit contains the complete nested expression:

And both together:

The entire template translation is complete. The next section describes how to load that translation into the app.

{@a app-pre-translation}

The app and its translation file

The sample app and its translation file are now as follows:

{@a merge}

Merge the completed translation file into the app

To merge the translated text into component templates, compile the app with the completed translation file. Provide the Angular compiler with three translation-specific pieces of information:

- The translation file.
- The translation file format.
- The locale (`fr` or `en-US` for instance).

The compilation process is the same whether the translation file is in `.xlf` format or in another format that Angular understands, such as `.xtb` .

How you provide this information depends upon whether you compile with the JIT compiler or the AOT compiler.

- With AOT, you pass the information as a CLI parameter.
- With JIT, you provide the information at bootstrap time.

{@a merge-aot}

Merge with the AOT compiler

The AOT (*Ahead-of-Time*) compiler is part of a build process that produces a small, fast, ready-to-run application package.

When you internationalize with the AOT compiler, you must pre-build a separate application package for each language and serve the appropriate package based on either server-side language detection or url parameters.

You also need to instruct the AOT compiler to use your translation file. To do so, you use three options with the `ng serve` or `ng build` commands:

- `--i18nFile` : the path to the translation file.
- `--i18nFormat` : the format of the translation file.
- `--locale` : the locale id.

The example below shows how to serve the French language file created in previous sections of this guide:

```
ng serve --aot --i18nFile=src/locale/messages.fr.xlf --i18nFormat=xlf --locale=fr
```

{@a merge-jit}

Merge with the JIT compiler

The JIT compiler compiles the app in the browser as the app loads. Translation with the JIT compiler is a dynamic process of:

1. Importing the appropriate language translation file as a string constant.
2. Creating corresponding translation providers for the JIT compiler.
3. Bootstrapping the app with those providers.

Three providers tell the JIT compiler how to translate the template texts for a particular language while compiling the app:

- `TRANSLATIONS` is a string containing the content of the translation file.
- `TRANSLATIONS_FORMAT` is the format of the file: `xlf`, `xlf2`, or `xtb`.
- `LOCALE_ID` is the locale of the target language.

The Angular `bootstrapModule` method has a second `compilerOptions` parameter that can influence the behavior of the compiler. You can use it to provide the translation providers:

Then provide the `LOCALE_ID` in the main module:

{@a missing-translation}

Report missing translations

By default, when a translation is missing, the build succeeds but generates a warning such as `Missing translation for message "foo"`. You can configure the level of warning that is generated by the Angular compiler:

- Error: throw an error. If you are using AOT compilation, the build will fail. If you are using JIT compilation, the app will fail to load.
- Warning (default): show a 'Missing translation' warning in the console or shell.
- Ignore: do nothing.

If you use the AOT compiler, specify the warning level by using the CLI parameter `--missingTranslation`. The example below shows how to set the warning level to error:

```
ng serve --aot --missingTranslation=error
```

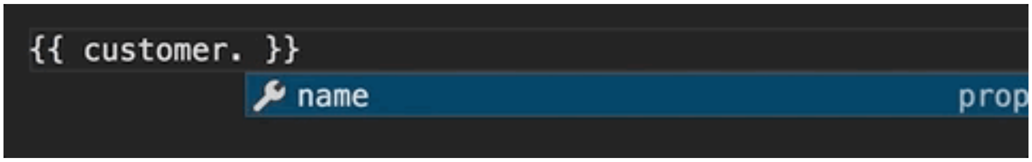
If you use the JIT compiler, specify the warning level in the compiler config at bootstrap by adding the 'MissingTranslationStrategy' property. The example below shows how to set the warning level to error:

Angular Language Service

The Angular Language Service is a way to get completions, errors, hints, and navigation inside your Angular templates whether they are external in an HTML file or embedded in annotations/decorators in a string. The Angular Language Service autodetects that you are opening an Angular file, reads your `tsconfig.json` file, finds all the templates you have in your application, and then provides language services for any templates that you open.

Autocompletion

Autocompletion can speed up your development time by providing you with contextual possibilities and hints as you type. This example shows autocomplete in an interpolation. As you type it out, you can hit tab to complete.

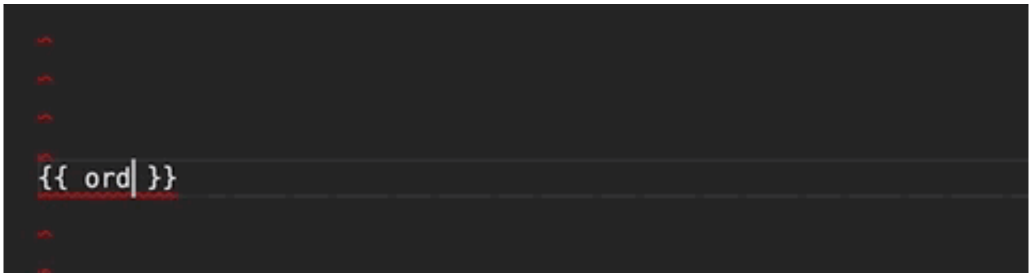


```
{{ customer. }}
```

There are also completions within elements. Any elements you have as a component selector will show up in the completion list.

Error checking

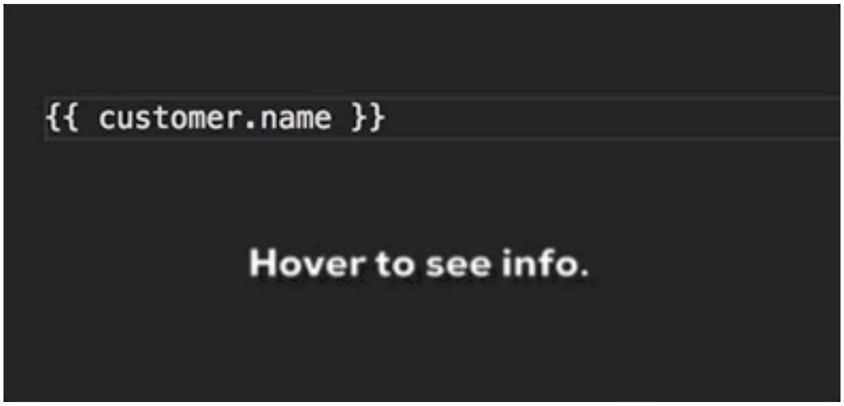
The Angular Language Service can also forewarn you of mistakes in your code. In this example, Angular doesn't know what `orders` is or where it comes from.



```
{{ ord }}
```

Navigation

Navigation allows you to hover to see where a component, directive, module, etc. is from and then click and press F12 to go directly to its definition.



```
{{ customer.name }}
```

Hover to see info.

Angular Language Service in your editor

Angular Language Service is currently available for Visual Studio Code and WebStorm.

Visual Studio Code

In Visual Studio Code, install Angular Language Service from the store, which is accessible from the bottom icon on the left menu pane. You can also use the VS Quick Open (⌘+P) to search for the extension. When you've opened it, enter the following command:

```
ext install ng-template
```

Then click the install button to install the Angular Language Service.

WebStorm

In webstorm, you have to install the language service as a dev dependency. When Angular sees this dev dependency, it provides the language service inside of WebStorm. Webstorm then gives you colorization inside the template and autocomplete in addition to the Angular Language Service.

Here's the dev dependency you need to have in `package.json`:

```
devDependencies {
    "@angular/language-service": "^4.0.0"
}
```

Then in the terminal window at the root of your project, install the `devDependencies` with `npm` or `yarn`:

```
npm install
```

OR

```
yarn
```

OR

```
yarn install
```

Sublime Text

In Sublime Text, you first need an extension to allow Typescript. Install the latest version of typescript in a local `node_modules` directory:

```
npm install --save-dev typescript
```

Then install the Angular Language Service in the same location: `sh npm install --save-dev @angular/language-service`

Starting with TypeScript 2.3, TypeScript has a language service plugin model that the language service can use.

Next, in your user preferences (`Cmd+,` or `Ctrl+,`), add:

```
"typescript-tsdk": "<path to your folder>/node_modules/typescript/lib"
```

Installing in your project

You can also install Angular Language Service in your project with the following `npm` command:

```
npm install --save-dev @angular/language-service
```

Additionally, add the following to the `"compilerOptions"` section of your project's `tsconfig.json` .

```
  "plugins": [
      {"name": "@angular/language-service"}
  ]
```

Note that this only provides diagnostics and completions in `.ts` files. You need a custom sublime plugin (or modifications to the current plugin) for completions in HTML files.

How the Language Service works

When you use an editor with a language service, there's an editor process which starts a separate language process/service to which it speaks through an RPC. Any time you type inside of the editor, it sends information to the other process to track the state of your project. When you trigger a completion list within a template, the editor process first parses the template into an HTML AST, or abstract syntax tree. Then the Angular compiler interprets what module the template is part of, the scope you're in, and the component selector. Then it figures out where in the template AST your cursor is. When it determines the context, it can then determine what the children can be.

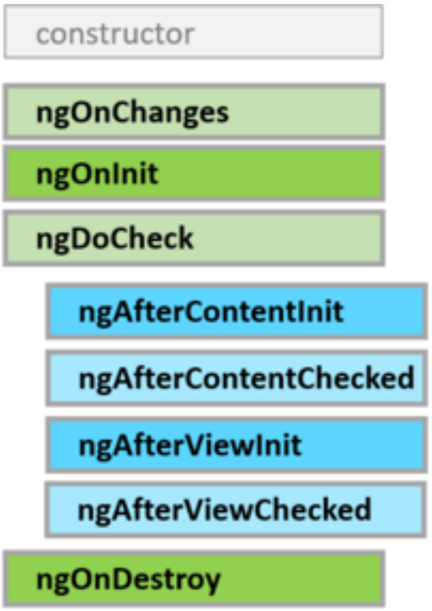
It's a little more involved if you are in an interpolation. If you have an interpolation of `{{data.---}}` inside a `div` and need the completion list after `data.---`, the compiler can't use the HTML AST to find the answer. The HTML AST can only tell the compiler that there is some text with the characters "`{{data.---}}`". That's when the template parser produces an expression AST, which resides within the template AST. The Angular Language Services then looks at `data.---` within its context and asks the TypeScript Language Service what the members of data are. TypeScript then returns the list of possibilities.

For more in-depth information, see the Angular Language Service API

More on Information

For more information, see Chuck Jazdzewski's presentation on the Angular Language Service from ng-conf 2017.

Lifecycle Hooks



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, minus the hooks that are specific to component content and views.

{@a hooks-overview}

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:

No directive or component will implement all of the lifecycle hooks and some of the hooks only make sense for components. Angular only calls a directive/component hook method *if it is defined*.

{@a hooks-purpose-timing}

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 `SimpleChanges` object of current and previous property values. Called before `ngOnInit()` and whenever one or more data-bound input properties change.
`ngOnInit()`	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_ `ngOnChanges()`.
`ngDoCheck()`	Detect and act upon changes that Angular can't or won't detect on its own. Called during every change detection run, immediately after `ngOnChanges()` and `ngOnInit()`.
`ngAfterContentInit()`	Respond after Angular projects external content into the component's view. Called _once_ after the first `ngDoCheck()`. _A component-only hook_.
`ngAfterContentChecked()`	Respond after Angular checks the content projected into the component. Called after the `ngAfterContentInit()` and every subsequent `ngDoCheck()`. _A component-only hook_.
`ngAfterViewInit()`	Respond after Angular initializes the component's views and child views. Called _once_ after the first `ngAfterContentChecked()`. _A component-only hook_.
`ngAfterViewChecked()`	Respond after Angular checks the component's views and child views. Called after the `ngAfterViewInit` and every subsequent `ngAfterContentChecked()`. _A component-only hook_.
`ngOnDestroy`	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.

{@a interface-optional}

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.

{@a other-lifecycle-hooks}

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.

{@a the-sample}

Lifecycle examples

The 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
Peek-a-boo	Demonstrates every lifecycle hook. Each hook method writes to the on-screen log.
Spy	Directives have lifecycle hooks too. A `SpyDirective` can log when the element it spies upon is created or destroyed using the `ngOnInit` and `ngOnDestroy` hooks. This example applies the `SpyDirective` to a ` ` in an `ngFor` *hero* repeater managed by the parent `SpyComponent`.
OnChanges	See how Angular calls the `ngOnChanges()` hook with a `changes` object every time one of the component input properties changes. Shows how to interpret the `changes` object.
DoCheck	Implements an `ngDoCheck()` method with custom change detection. See how often Angular calls this hook and watch it post changes to a log.
AfterView	Shows what Angular means by a *view*. Demonstrates the `ngAfterViewInit` and `ngAfterViewChecked` hooks.
AfterContent	Shows how to project external content into a component and how to distinguish projected content from a component's view children. Demonstrates the `ngAfterContentInit` and `ngAfterContentChecked` hooks.
Counter	Demonstrates a combination of a component and a directive each with its own hooks. 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.

{@a peek-a-boo}

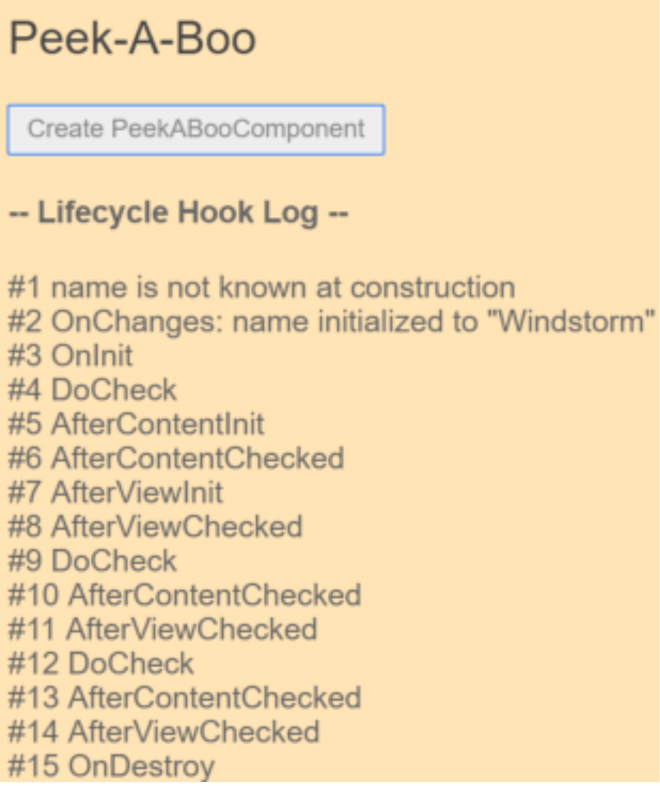
Peek-a-boo: all hooks

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.



Peek-A-Boo

-- 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.

{@a spy}

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.

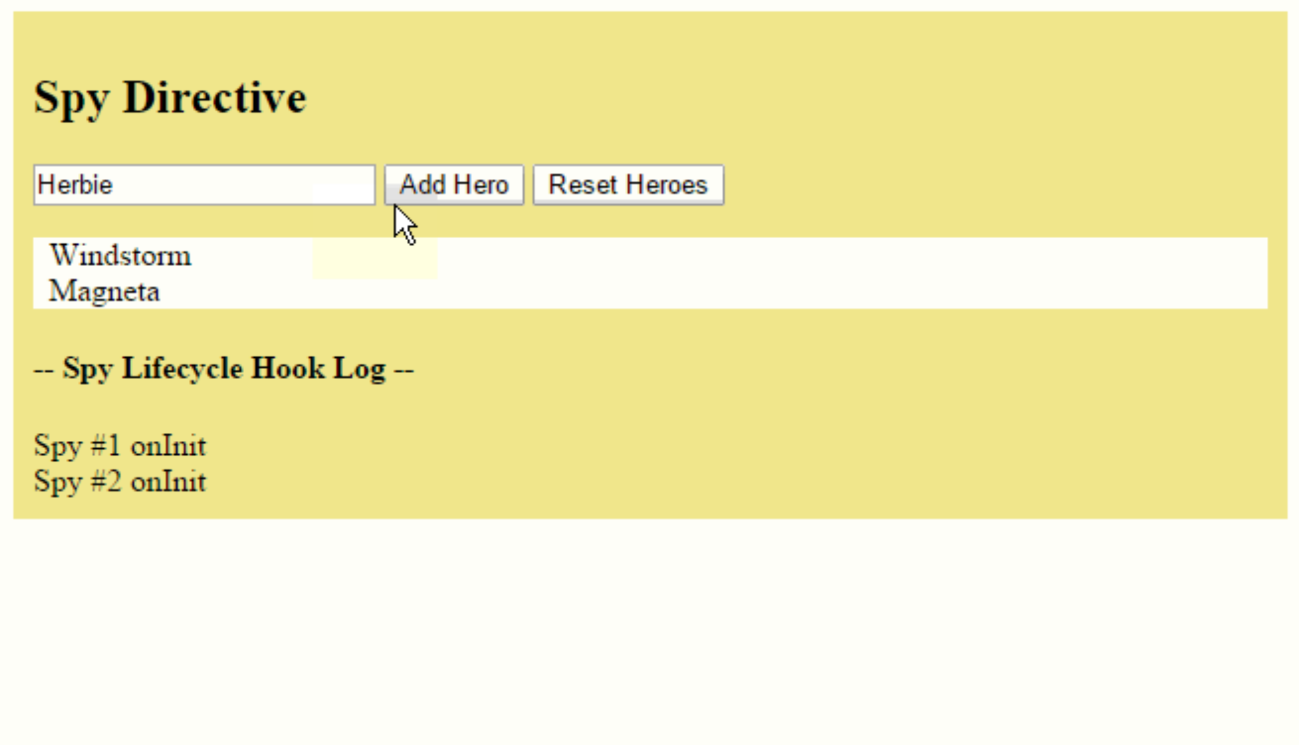
Kidding aside, pay attention to two key points: 1. Angular calls hook methods for *directives* as well as components.

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 `
`. 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`.

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>`:

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:



Windstorm
Magneta

-- Spy Lifecycle Hook Log --

Spy #1 onInit
Spy #2 onInit

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.

{@a oninit}

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](http://misko.hevery.com/code-reviewers-guide/flaw-constructor-does-real-work/) 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.

{@a ondestroy}

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.

{@a onchanges}

OnChanges()

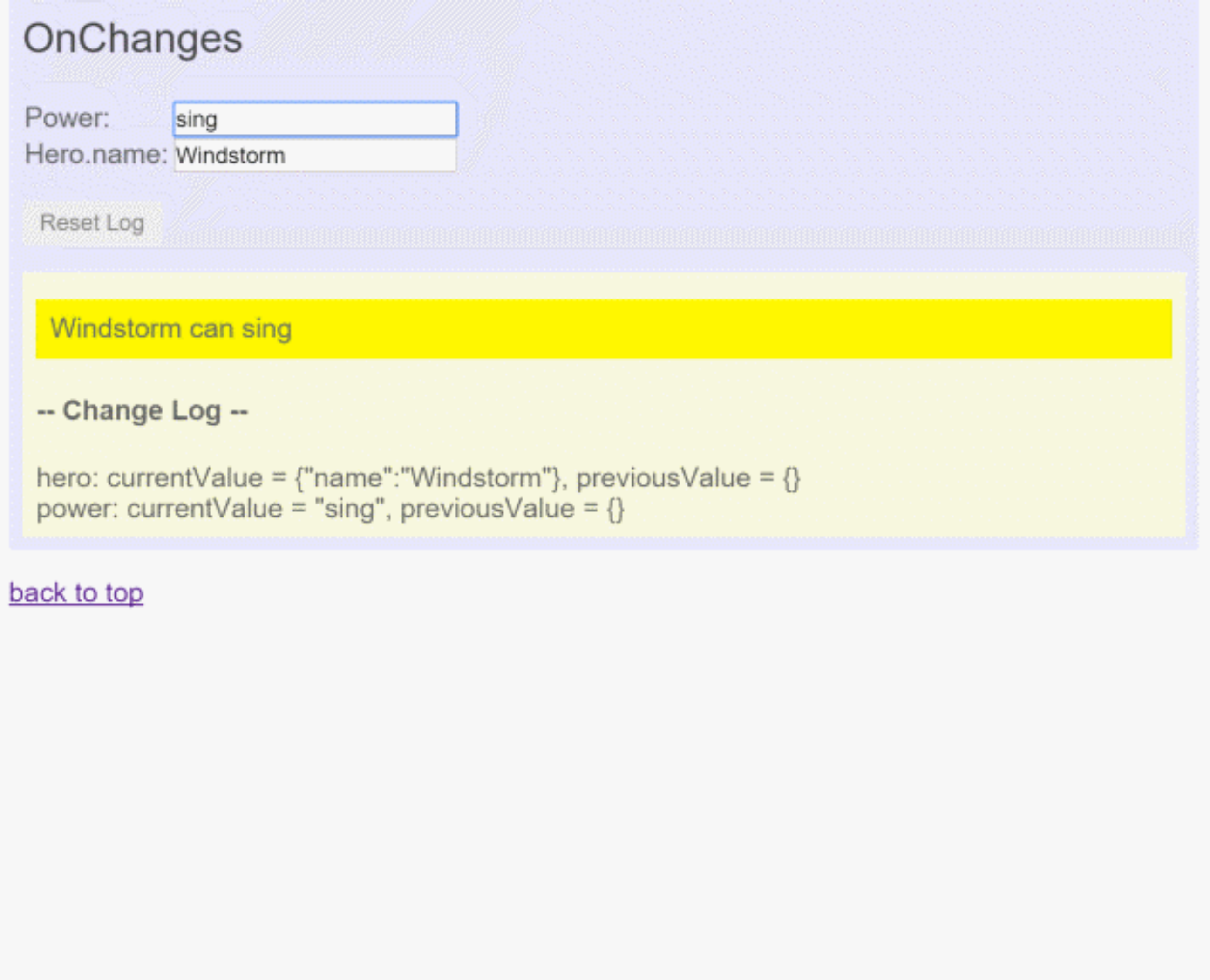
Angular calls its `ngOnChanges()` method whenever it detects changes to ***input properties*** of the component (or directive). This example monitors the `OnChanges` hook.

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` .

The host `OnChangesParentComponent` binds to them like this:

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



Power: sing
Hero.name: Windstorm

Windstorm can sing

-- Change Log --

```
hero: currentValue = {"name":"Windstorm"}, previousValue = {}
power: currentValue = "sing", previousValue = {}
```

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!

{@a docheck}

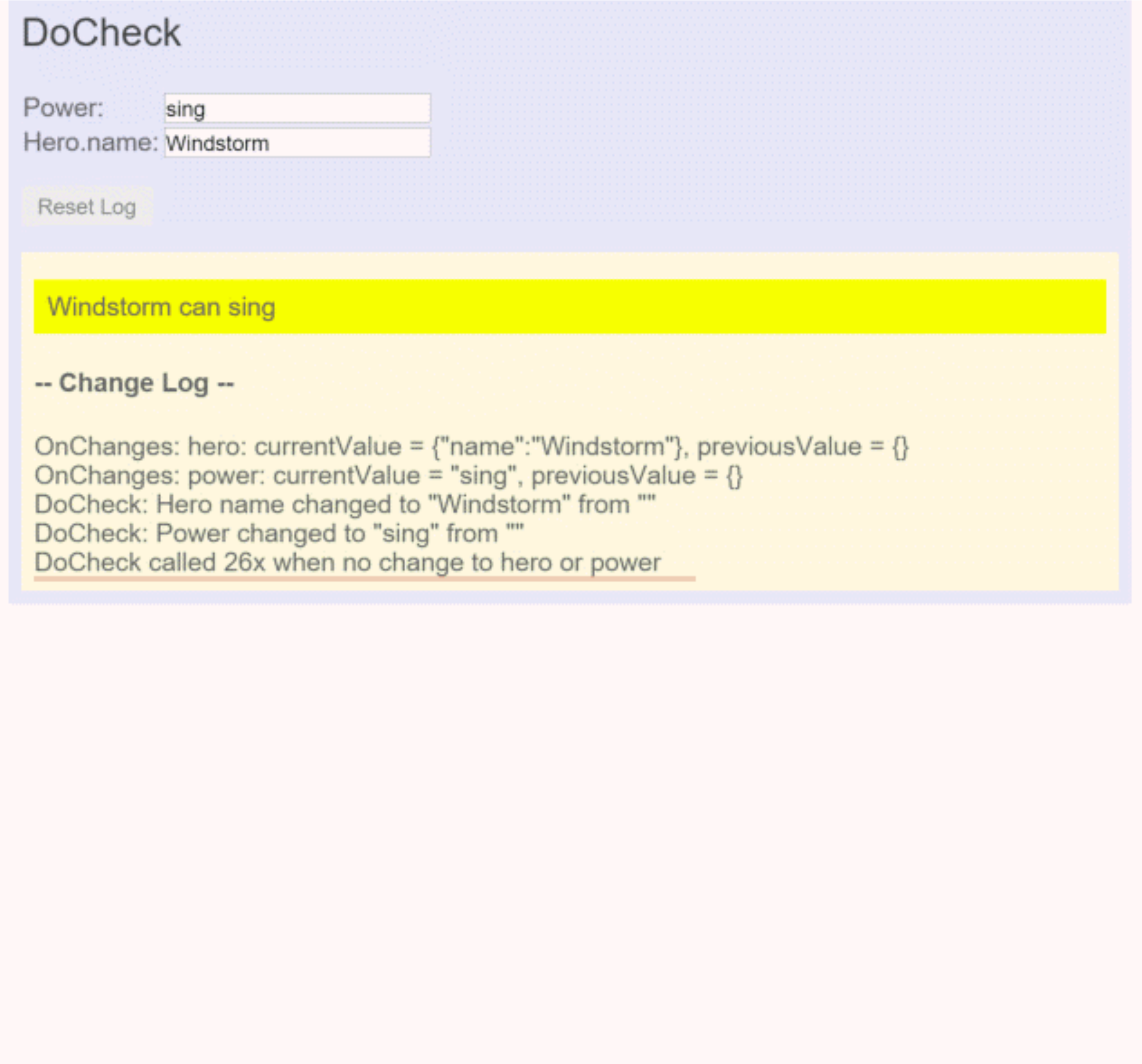
DoCheck()

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:

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

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.

{@a afterview}

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>` :

The `AfterViewComponent` displays this child view *within its template*:

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.

{@a wait-a-tick}

Abide by the unidirectional data flow rule

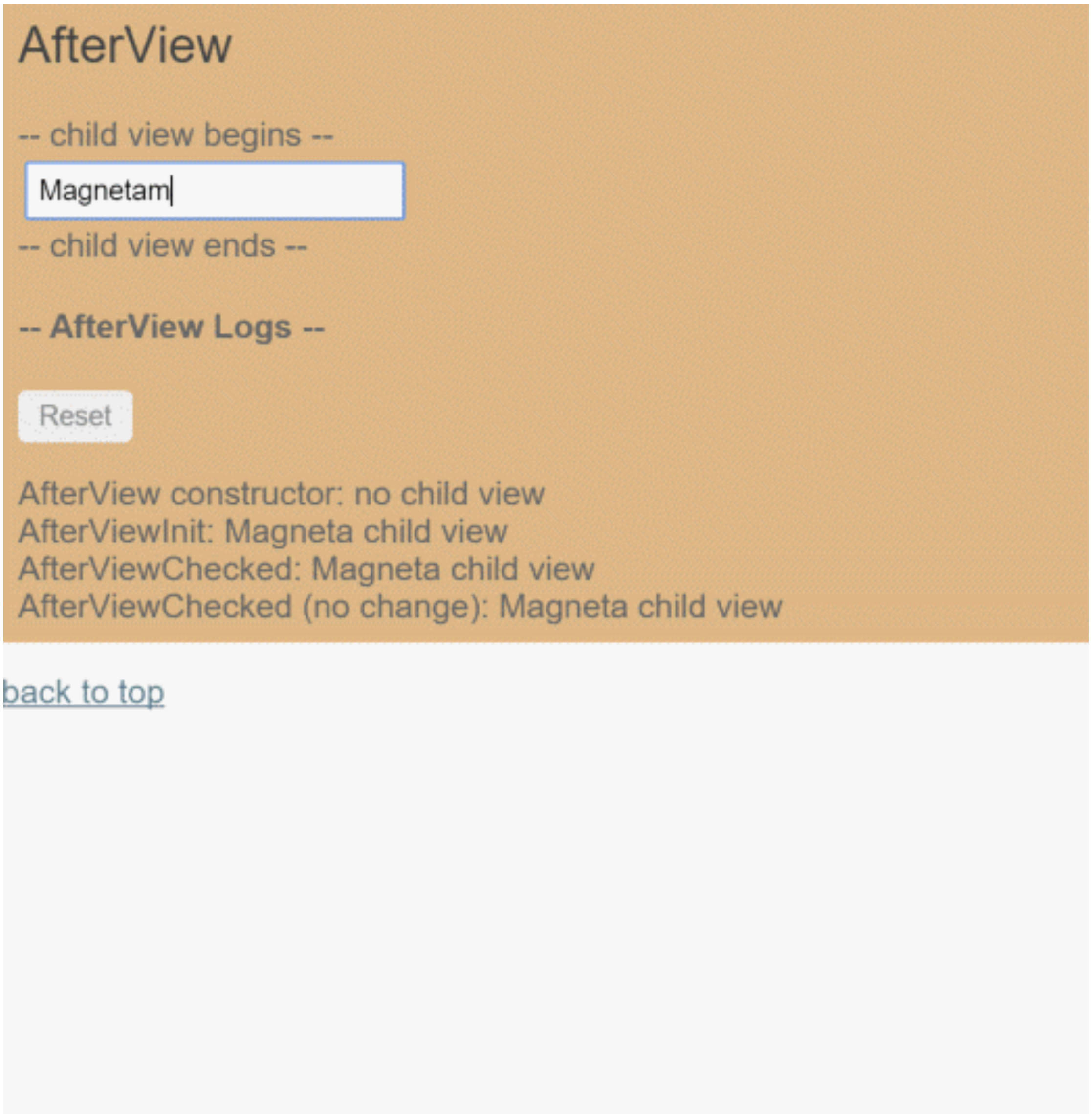
The `doSomething()` method updates the screen when the hero name exceeds 10 characters.

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 --

Magnetam

-- child view ends --

-- AfterView Logs --

```
AfterView constructor: no child view
AfterViewInit: Magneta child view
AfterViewChecked: Magneta child view
AfterViewChecked (no change): Magneta child view
```

Notice that Angular frequently calls `AfterViewChecked()`, often when there are no changes of interest. Write lean hook methods to avoid performance problems.

{@a aftercontent}

AfterContent

The *AfterContent* sample explores the `AfterContentInit()` and `AfterContentChecked()` hooks that Angular calls *after* Angular projects external content into the component.

{@a content-projection}

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.

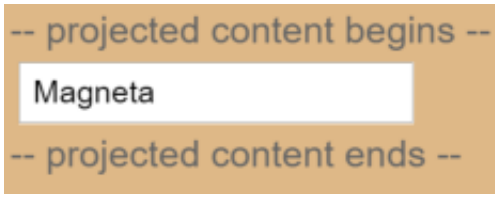
AngularJS developers know this technique as *transclusion*.

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:

Notice that the `<my-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:

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 `<my-child>` from the parent.



-- projected content begins --

Magneta

-- projected content ends --

The telltale signs of *content projection* are twofold: * HTML between component element tags. * The presence of `` tags in the component's template.

{@a aftercontent-hooks}

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.

{@a no-unidirectional-flow-worries}

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.

NgModule FAQs

NgModules help organize an application into cohesive blocks of functionality.

The NgModules guide takes you step-by-step from the most elementary `@NgModule` class to a multi-faceted sample with lazy-loaded modules.

This page answers the questions many developers ask about NgModule design and implementation.

These FAQs assume that you have read the [NgModules](guide/ngmodule) guide.

{@a q-what-to-declare}

What classes should I add to *declarations*?

Add declarable classes—components, directives, and pipes—to a `declarations` list.

Declare these classes in *exactly one* NgModule. Declare them in *this* NgModule if they *belong* to this module.

{@a q-declarable}

What is a *declarable*?

Declarables are the class types—components, directives, and pipes—that you can add to an NgModule's `declarations` list. They're the *only* classes that you can add to `declarations`.

{@a q-what-not-to-declare}

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 NgModule.
- An array of directives imported from another NgModule. For example, don't declare FORMS_DIRECTIVES from `@angular/forms`.
- NgModule classes.

- Service classes.
- Non-Angular classes and objects, such as strings, numbers, functions, entity models, configurations, business logic, and helper classes.

{@a q-why-multiple-mentions}

Why list the same component in multiple *@NgModule* properties?

`AppComponent` is often listed in both `declarations` and `bootstrap`. You might see `HeroComponent` 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.
- `HeroComponent` could be imported from another application module (so you can't declare it) and re-exported by this module.
- `HeroComponent` could be exported for inclusion in an external component's template as well as dynamically loaded in a pop-up dialog.

{@a q-why-cant-bind-to}

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.

You also get this error if "x" really isn't a property or if "x" is a private component property (i.e., lacks the \`@Input\` or \`@Output\` decorator).

For example, if "x" is `ngModel`, you may not have imported the `FormsModule` from `@angular/forms`.

Perhaps you declared "x" in an application feature module but forgot to export it? The "x" class isn't visible to other components of other NgModules until you add it to the `exports` list.

{@a q-what-to-import}

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`.

{@a q-browser-vs-common-module}

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 NgModule. *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.

`BrowserModule` throws an error if you try to lazy load a module that imports it.

Importing `CommonModule` also frees feature modules for use on *any* target platform, not just browsers.

{@a q-reimport}

What if I import the same NgModule twice?

That's not a problem. When three NgModules 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'.

{@a q-what-to-export}

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 class, it stays *private*, visible only to other component 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.

{@a q-what-not-to-export}

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 `HttpModule` because it doesn't export anything. It's only purpose is to add http service providers to the application as a whole.

{@a q-reexport} {@a q-re-export}

Can I re-export classes and NgModules?

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](guide/ngmodule-faq#q-declarable) classes that another NgModule could use. For example, there's no point in re-exporting `HttpModule` because it doesn't export anything. It's only purpose is to add http service providers to the application as a whole.

{@a q-for-root}

What is the *forRoot* method?

The `forRoot` static method is a convention that makes it easy for developers to configure the module's providers.

The `RouterModule.forRoot` method is a good example. 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 NgModule, `AppModule`. Importing it in any other NgModule, particularly in a lazy-loaded NgModule, is contrary to the intent and will likely produce a runtime error.

`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.

{@a q-module-provider-visibility}

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.

This is by design. 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 `HttpModule` 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`.

{@a q-lazy-loaded-module-provider-visibility}

Why is a service provided in a *lazy-loaded* NgModule visible only to that module?

Unlike providers of the NgModules 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.

{@a q-module-provider-duplicates}

What if two NgModules provide the same service?

When two imported NgModules, 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.

{@a q-component-scoped-providers}

How do I restrict service scope to an NgModule?

When an NgModule 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 NgModules 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 an NgModule requires a customized `HttpBackend` that adds a special header for all Http requests. If another NgModule elsewhere in the application also customizes `HttpBackend` or merely imports the `HttpModule`, 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 `HttpModule` 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 NgModules 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.

{@a q-root-component-or-module}

Should I add application-wide providers to the root *AppModule* or the root *AppComponent*?

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 NgModule providers with the application root injector. The services created from root injector providers are available to the entire application. They are *application-scoped*.

Certain services (such as the `Router`) only work when registered 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 are *component-scoped*.

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 for routed applications, "almost" isn't good enough.

`AppComponent` services don't exist at the root level where routing operates. Lazy-loaded modules can't reach them. In the [*NgModules* sample application](), if you had registered `UserService` in the `AppComponent`, the `HeroComponent` couldn't inject it. The application would fail the moment a user navigated to "Heroes".

{@a q-component-or-module}

Should I add other providers to an NgModule or a component?

In general, prefer registering feature-specific providers in NgModules (`@NgModule.providers`) to 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, a hero editing component that needs a private copy of a caching hero service should register the `HeroService` with the `HeroEditorComponent`. Then each new instance of the `HeroEditorComponent` gets its own cached service instance. The changes that editor makes to heroes in its service don't touch the hero instances elsewhere in the application.

[Always register *application-wide* services with the root `AppModule`](), not the root `AppComponent`.

{@a q-why-bad}

Why is it bad if *SharedModule* provides a service to a lazy-loaded NgModule?

This question is addressed in the [Why UserService isn't shared]() section of the [NgModules]() guide, which discusses the importance of keeping providers out of the `SharedModule`.

Suppose the `UserService` was listed in the NgModule's `providers` (which it isn't). Suppose every NgModule imports this `SharedModule` (which they all do).

When the app starts, Angular eagerly loads the `AppModule` and the `ContactModule`.

Both instances of the imported `SharedModule` would provide the `UserService`. Angular registers one of them in the root app injector (see What if I import the same NgModule twice?). Then some component injects `UserService`, Angular finds it in the app root injector, and delivers the app-wide singleton `UserService`. No problem.

Now consider the `HeroModule` *which is lazy-loaded*.

When the router lazy loads the `HeroModule`, 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 `HeroComponent`, it must inject a `UserService`. This time 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.

That's almost certainly a mistake.

To demonstrate, run the live example. Modify the \`SharedModule\` so that it provides the \`UserService\` rather than the \`CoreModule\`. Then toggle between the "Contact" and "Heroes" links a few times. The username flashes irregularly as the Angular creates a new \`UserService\` instance each time.

{@a q-why-child-injector}

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 applications 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.

{@a q-is-it-loaded}

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, such as this `CoreModule` constructor.

{@a q-entry-component-defined}

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.

Most application components are loaded declaratively. Angular uses the component's selector to locate the element in the template. It then creates the HTML representation of the component and inserts it into the DOM at the selected element. These aren't entry components.

A few components are only loaded dynamically and are *never* referenced in a component template.

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.

Angular loads `AppComponent` dynamically because it's either listed *by type* in `@NgModule.bootstrap` or bootstrapped imperatively with the NgModule's `ngDoBootstrap` method.

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`.

The compiler can't discover these *entry components* by looking for them in other component templates. You must tell it about them by adding them to the `entryComponents` list.

Angular automatically adds the following types of components to the NgModule's `entryComponents`:

- The component in the `@NgModule.bootstrap` list.
- Components referenced in router configuration.

You don't have to mention these components explicitly, although doing so is harmless.

{@a q-bootstrap*vs*entry_component}

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 (application launch) process. 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 `entryComponent` lists, although doing so is harmless.

{@a q-when-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.

{@a q-why-entry-components}

Why does Angular need *entryComponents*?

Entry components are also declared. Why doesn't the Angular compiler generate code for every component in `@NgModule.declarations` ? Then you wouldn't need entry components.

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.

{@a q-module-recommendations}

What kinds of NgModules 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. The following suggestions and guidelines have wide appeal.

SharedModule

Create a `SharedModule` with the components, directives, and pipes that you use everywhere in your app. This NgModule 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 NgModules have `providers` . If you deviate from this guideline, know what you're doing and why.

Import the `SharedModule` in your *feature* modules, both those loaded when the app starts and those you lazy load later.

CoreModule

Create a `CoreModule` 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` .

This page sample departs from that advice by declaring and exporting two components that are only used within the root `AppComponent` declared by `AppModule`. Someone following this guideline strictly would have declared these components in the `AppModule` instead.

Feature Modules

Create feature modules around specific application business domains, user workflows, and utility collections.

Feature modules tend to fall into one of the following groups:

- Domain feature modules.
- Routed feature modules.
- Routing modules.
- Service feature modules.
- Widget feature modules.

Real-world NgModules are often hybrids that purposefully deviate from the following guidelines. These guidelines are not laws; follow them unless you have a good reason to do otherwise.

Feature Module	Guidelines

{@a domain-feature-module}Domain	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. Private, supporting sub-components descend from it. Domain feature modules consist mostly of _declarations_. Only the top component is exported. 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. Don't provide application-wide singleton services in a domain feature 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. For an example, see the [Feature Modules](guide/ngmodule#contact-module-v1) section of the [NgModules](guide/ngmodule) guide, before routing is introduced.
{@a routed-feature-module}Routed	_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. This page's `ContactModule`, `HeroModule`, and `CrisisModule` are routed feature modules. Routed feature modules _shouldn't export anything_. They don't have to because their components never appear in the template of an external component. A lazy-loaded routed feature module should _not be imported_ by any NgModule. Doing so would trigger an eager load, defeating the purpose of lazy loading. `HeroModule` and `CrisisModule` are lazy-loaded. They aren't mentioned among the `AppModule` imports. But an eagerly loaded, routed feature module must be imported by another NgModule so that the compiler learns about its components. `ContactModule` is eager loaded and therefore listed among the `AppModule` imports. Routed Feature Modules rarely have _providers_ for reasons [explained earlier](guide/ngmodule-faq#q-why-bad). When they do, the lifetime of the provided services should be the same as the lifetime of the NgModule. Don't provide application-wide singleton services in a routed feature module or in an NgModule that the routed module imports.
{@a routing-module}Routing	A [routing module](guide/router#routing-module) *provides routing configuration* for another NgModule. A routing module separates routing concerns from its companion module. A routing module typically does the following: * Defines routes. * Adds router configuration to the module's `imports`. * Re-exports `RouterModule`. * 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 *should 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. The `AppRoutingModule`, `ContactRoutingModule`, and `HeroRoutingModule` are good examples. See also [Do you need a _Routing Module_?](guide/router#why-routing-module) on the [Routing & Navigation](guide/router) page.
{@a service-feature-module}Service	Service modules *provide utility services* such as data access and messaging. Ideally, they consist entirely of _providers_ and have no _declarations_. The `CoreModule` and Angular's `HttpModule` are good examples. Service Modules should _only_ be imported by the root `AppModule`. Do *not* import service modules in other feature modules. If you deviate from this guideline, know what you're doing and why.
{@a widget-feature-module}Widget	A widget module makes *components, directives, and pipes* available to external NgModules. `CommonModule` and `SharedModule` are widget modules. Many third-party UI component libraries are widget modules. A widget module should consist entirely of _declarations_, most of them exported. A widget module should rarely have _providers_. If you deviate from this guideline, know what you're doing and why. Import widget modules in any module whose component templates need the widgets.

The following table summarizes the key characteristics of each *feature module* group.

Real-world NgModules are often hybrids that knowingly deviate from these guidelines.

Feature Module	Declarations	Providers	Exports	Imported By	Examples
Domain	Yes	Rare	Top component	Feature, `AppModule`	`ContactModule` (before routing)
Routed	Yes	Rare	No	Nobody	`ContactModule`, `HeroModule`, `CrisisModule`
Routing	No	Yes (Guards)	`RouterModule`	Feature (for routing)	`AppRoutingModule`, `ContactRoutingModule`, `HeroRoutingModule`
Service	No	Yes	No	`AppModule`	`HttpModule`, `CoreModule`
Widget	Yes	Rare	Yes	Feature	`CommonModule`, `SharedModule`

{@a q-ng-vs-js-modules}

What's the difference between Angular NgModules and JavaScript Modules?

Angular and JavaScript are different yet complementary module systems.

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:

export class AppComponent { ... }

Then you `import` a part in another module:

import { AppComponent } from './app.component';

This kind of modularity is a feature of the *JavaScript language*.

An *NgModule* is a feature of *Angular* itself.

Angular's `@NgModule` metadata also have `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.

The NgModule classes differ from JavaScript module class 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 NgModule's classes in the `@NgModule.declarations` list.
- An NgModule can only export the declarable classes it owns or imports from other NgModules. It doesn't declare or export any other kind of class.

The NgModule is also special in another way. Unlike JavaScript modules, an NgModule can extend the *entire* application with services by adding providers to the `@NgModule.providers` list.

The provided services don't belong to the NgModule nor are they scoped to the declared classes. They are available _everywhere_.

Here's an *@NgModule* class with imports, exports, and declarations.

Of course you use *JavaScript* modules to write NgModules as seen in the complete `contact.module.ts` file:

{@a q-template-reference}

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 NgModule or exported by an NgModule that this one imports.

{@a q-angular-compiler}

What is the Angular compiler?

The Angular compiler converts the application code you write into highly performant JavaScript code. The `@NgModule` metadata play an important role in guiding the compilation process.

The code you write isn't immediately executable. Consider *components*. 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 NgModules.

{@a q-ngmodule-api}

@NgModule API

The following table summarizes the `@NgModule` metadata properties.

Property	Description
`declarations`	A list of [declarable](guide/ngmodule-faq#q-declarable) classes, the *component*, *directive*, and *pipe* classes that _belong to this NgModule_. These declared classes are visible within the NgModule but invisible to components in a different NgModule unless they are _exported_ from this NgModule and the other NgModule _imports_ this one. Components, directives, and pipes must belong to _exactly_ one NgModule. The compiler emits an error if you try to declare the same class in more than one NgModule. *Do not re-declare a class imported from another NgModule.*
`providers`	A list of dependency-injection providers. Angular registers these providers with the root injector of the NgModule's execution context. That's the application's root injector for all NgModules loaded when the application starts. Angular can inject one of these provider services into any component in the application. If this NgModule or any NgModule loaded at launch provides the `HeroService`, Angular can inject the same `HeroService` intance into any app component. A lazy-loaded NgModule has its own sub-root injector which typically is a direct child of the application root injector. Lazy-loaded services are scoped to the lazy module's injector. If a lazy-loaded NgModule also provides the `HeroService`, 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 NgModules continue to receive the instance created for the application root.
`imports`	A list of supporting NgModules. Specifically, the list of NgModules whose exported components, directives, or pipes are referenced by the component templates declared in this NgModule. A component template can [reference](guide/ngmodule-faq#q-template-reference) another component, directive, or pipe when the referenced class is declared in this module or the class was imported from another module. A component can use the `NgIf` and `NgFor` directives only because its declaring NgModule imported the Angular `CommonModule` (perhaps indirectly by importing `BrowserModule`). You can import many standard directives with the `CommonModule` but some familiar directives belong to other NgModules. A component template can bind with `[(ngModel)]` only after importing the Angular `FormsModule`.

`exports`	A list of declarations—*component*, *directive*, and *pipe* classes—that an importing NgModule can use. Exported declarations are the module's _public API_. A component in another NgModule can [reference](guide/ngmodule-faq#q-template-reference) _this_ NgModule's `HeroComponent` if it imports this module and this module exports `HeroComponent`. Declarations are private by default. If this NgModule does _not_ export `HeroComponent`, no other NgModule can see it. Importing an NgModule does _not_ automatically re-export the imported NgModule's imports. NgModule 'B' can't use `ngIf` just because it imported NgModule `A` which imported `CommonModule`. NgModule 'B' must import `CommonModule` itself. An NgModule can list another NgModule among its `exports`, in which case all of that NgModule's public components, directives, and pipes are exported. [Re-export](guide/ngmodule-faq#q-re-export) makes NgModule transitivity explicit. If NgModule 'A' re-exports `CommonModule` and NgModule 'B' imports NgModule 'A', NgModule 'B' components can use `ngIf` even though 'B' itself didn't import `CommonModule`.
`bootstrap`	A list of components that can be bootstrapped. Usually there's only one component in this list, the _root component_ of the application. Angular can launch with multiple bootstrap components, each with its own location in the host web page. A bootstrap component is automatically an `entryComponent`.
`entryComponents`	A list of components that are _not_ [referenced](guide/ngmodule-faq#q-template-reference) in a reachable component template. Most developers never set this property. The [Angular compiler](guide/ngmodule-faq#q-angular-compiler) must know about every component actually used in the application. The compiler can discover most components by walking the tree of references from one component template to another. But there's always at least one component that's not referenced in any template: the root component, `AppComponent`, that you bootstrap to launch the app. That's why it's called an _entry component_. Routed components are also _entry components_ because they aren't referenced in a template either. The router creates them and drops them into the DOM near a ``. While the bootstrapped and routed components are _entry components_, you usually don't have to add them to a module's `entryComponents` list. Angular automatically adds components in the module's `bootstrap` list to the `entryComponents` list. The `RouterModule` adds routed components to that list. That leaves only the following sources of undiscoverable components: * Components bootstrapped using one of the imperative techniques. * Components dynamically loaded into the DOM by some means other than the router. Both are advanced techniques that few developers ever employ. If you are one of those few, you must add these components to the `entryComponents` list yourself, either programmatically or by hand.

NgModules

NgModules help organize an application into cohesive blocks of functionality.

An NgModule is a class adorned with the **@NgModule** decorator function. `@NgModule` takes a metadata object that tells Angular how to compile and your code. It identifies the module's own components, directives, and pipes, making some of them public so external components can use them. `@NgModule` may add service providers to the application dependency injectors. And there are many more options covered here.

{@a bootstrap}

For a quick overview of NgModules, consider reading the Bootstrapping guide, which introduces NgModules and the essentials of creating and maintaining a single root `AppModule` for the entire application.

This page covers NgModules in greater depth.

Live examples

This page explains NgModules through a progression of improvements to a sample with a "Heroes" theme. Here's an index to live examples at key moments in the evolution of the sample:

- The initial app
- The first contact module
- The revised contact module
- Just before adding SharedModule
- The final version

Frequently asked questions (FAQs)

This page covers NgModule concepts in a tutorial fashion.

The companion NgModule FAQs guide offers answers to specific design and implementation questions. Read this page before reading those FAQs.

{@a angular-modularity}

Angular modularity

NgModules are a great way to organize an application and extend it with capabilities from external libraries.

Many Angular libraries are NgModules (such as `FormsModule`, `HttpModule`, and `RouterModule`). Many third-party libraries are available as NgModules (such as Material Design, Ionic, 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.

NgModules can also add services to the application. Such services might be internally developed, such as the application logger. Services can come from outside sources, such as the Angular router and Http client.

NgModules can be loaded eagerly when the application starts. They can also be *lazy-loaded* asynchronously by the router.

An NgModule is a class decorated with `@NgModule` metadata. By setting metadata properties you tell Angular how your application parts fit together. For example, you can do the following:

- *Declare* which components, directives, and pipes belong to the NgModule.
- *Export* some of those classes so that other component templates can use them.
- *Import* other NgModules with the components, directives, and pipes needed by the components in *this* NgModule.
- *Provide* services at the application level that any application component can use.
- *Bootstrap* the app with one or more top-level, *root* components.

{@a root-module}

The root *AppModule*

Every Angular app has at least one NgModule class, the *root module*. You bootstrap *that* NgModule to launch the application.

By convention, the *root module* class is called `AppModule` and it exists in a file named `app.module.ts`. The **Angular CLI** generates the initial `AppModule` for you when you create a project.

ng new quickstart

The root `AppModule` is all you need in a simple application with a few components.

As the app grows, you may refactor the root `AppModule` into *feature modules* that represent collections of related functionality. For now, stick with the root `AppModule` created by the CLI.

The initial `declarations` array identifies the application's only component, `AppComponent`, the *root* *component* at the top of the app's component tree.

Soon you'll declare more components (and directives and pipes too).

The `@NgModule` metadata `imports` a single helper module, `BrowserModule`, which every browser app must import. `BrowserModule` registers critical application service providers. It also includes common directives like `NgIf` and `NgFor`, which become immediately visible and usable in any of this NgModule's component templates.

The `providers` array registers services with the top-level *dependency injector*. There are no services to register ... yet.

Lastly, the `bootstrap` list identifies the `AppComponent` as the *bootstrap component*. When Angular launches the app, it renders the `AppComponent` inside the `<app-root>` element tag of the `index.html`.

Learn about that in the bootstrapping guide.

The CLI-generated `AppComponent` in this guide's sample has been simplified and consolidated into a single `app.component.ts` file like this:

Run the app and follow along with the steps in this guide:

ng serve

{@a declarations} {@a declare-directive}

Declare directives

{@a declarables}

As the app evolves, you'll add directives, components, and pipes (the *declarables*). You must declare each of these classes in an NgModule.

As an exercise, begin by adding a `highlight.directive.ts` to the `src/app/` folder *by hand*.

The `HighlightDirective` is an attribute directive that sets the background color of its host element. Update the `AppComponent` template to attach this directive to the `<h1>` title element:

The screen of the running app has not changed. The `<h1>` is not highlighted. Angular does not yet recognize the `highlight` attribute and is ignoring it. You must declare the `HighlightDirective` in `AppModule`.

Edit the `app.module.ts` file, import the `HighlightDirective`, and add it to the `AppModule` *declarations* like this:

The Angular CLI would have done all of this for you if you'd created the `HighlightDirective` with the CLI command like this:

ng generate directive highlight

But you didn't. You created the file by hand so you must declare the directive by hand.

{@a declare-component}

Declare components

Now add a `TitleComponent` to the app and this time create it with the CLI.

ng generate component title --flat --no-spec --inline-style

The `--flat` flag tells the CLI to generate all files to the `src/app/` folder.
The `--no-spec` flag skips the test (`.spec`) file.
The `--inline-style` flag prevents generation of the `.css` file (which you won't need).
To see which files would be created or changed by any `ng generate` command, append the `--dryRun` flag (`-d` for short).

Open the `AppModule` and look at the `declarations` where you will see that the CLI added the `TitleComponent` for you.

Now rewrite the `title.component.html` like this.

And move the `title` property from `app.component.ts` into the `title.component.ts`, which looks as follows after a little cleanup.

Rewrite `AppComponent` to display the new `TitleComponent` in the `<app-title>` element and get rid of the `title` property.

Error if component not declared

There was no visible clue when you neglected to declare the `HighlightDirective` attribute directive. The Angular compiler doesn't recognize `highlight` as an `<h1>` attribute but it doesn't complain either. You'd discover it was undeclared only if you were looking for its effect.

Now try removing the declaration of the `TitleComponent` from `AppModule`.

The Angular compiler behaves differently when it encounters an unrecognized HTML element. The app ceases to display the page and the browser console logs the following error

Uncaught Error: Template parse errors: 'app-title' is not a known element: 1. If 'app-title' is an Angular component, then verify that it is part of this NgModule. 2. If 'app-title' is a Web Component then add 'CUSTOM*ELEMENTS*SCHEMA' to the '@NgModule.schemas' of this component to suppress this message.

If you don't get that error, you might get this one: Uncaught Error: Component TitleComponent is not part of any NgModule or the module has not been imported into your module.

Always declare your components, directives, and pipes.

{@a providers}

Service providers

The Dependency Injection page describes the Angular hierarchical dependency-injection system and how to configure that system with providers.

NgModule providers

An NgModule can provide services. A single instance of each provided service becomes available for injection into every class created with that NgModule's injector (or one of its descendant injectors).

When Angular boots the application, it creates the root `AppModule` with a root dependency injector. Angular configures the root injector with the providers specified in the module's `@NgModule.providers`.

Later, when Angular creates a new instance of a class— be it a component, directive, service, or module— that new class can be injected with an instance of a service provided to the root injector by the `AppModule`.

Angular also configures the root injector with the providers specified by [imported NgModules](#imports). An NgModule's own providers are registered _after_ imported NgModule providers. When there are multiple providers for the same injection token, the last registration wins.

Compared to Component providers

Providing a service in `@Component.providers` metadata means that a new service instance will be created for each new instance of *that* component and will be available for injection into *all of that component instance's descendant sub-components*.

The service instance won't be injected into any other component instances. Other instances of the same component class cannot see it. Sibling and ancestor component instances cannot see it.

Component providers always supersede NgModule providers. A component provider for injection token `X` creates a new service instance that "shadows" an NgModule provider for injection token `X`. When the component or any of its sub-components inject `X`, they get the *component* service instance, not the *NgModule* service instance.

Should you provide a service in an *NgModule* or a *component*? The answer depends on how you want to scope the service. If the service should be widely available, provide it in an NgModule. If it should be visible only within a component tree, provide it in the component at the root of that tree.

NgModule provider example

Many applications capture information about the currently logged-in user and make that information accessible through a user service.

Use the CLI to create a `UserService` and provide it in the root `AppModule`.

ng generate service user --module=app

This command creates a skeleton `UserService` in `src/app/user.service.ts` and a companion test file, `src/app/user.service.spec.ts`.

The `--module=app` flag tells the CLI to provide the service class in the NgModule defined in the `src/app/app.module.ts` file.

If you omit the `--module` flag, the CLI still creates the service but *does not provide it* anywhere. You have to do that yourself.

Confirm that the `--module=app` flag did provide the service in the root `AppModule` by inspecting the `@NgModule.providers` array in `src/app/app.module.ts`

Replace the generated contents of `src/app/user.service.ts` with the following dummy implementation.

Update the `TitleComponent` class with a constructor that injects the `UserService` and sets the component's `user` property from the service.

Update the `TitleComponent` template to show the welcome message below the application title.

{@a imports}

NgModule imports

In the revised `TitleComponent`, an `*ngIf` directive guards the message. There is no message if there is no user.

Although `AppModule` doesn't declare the `NgIf` directive, the application still compiles and runs. How can that be? The Angular compiler should either ignore or complain about unrecognized HTML.

Importing *BrowserModule*

Angular does recognize `NgIf` because the `AppModule` imports it indirectly when it imports `BrowserModule`.

Importing `BrowserModule` made all of its public components, directives, and pipes visible to the templates of components declared in `AppModule`, which include `TitleComponent`.

{@a reexport}

Re-exported NgModules

The `NgIf` directive isn't declared in `BrowserModule`. It's declared in `CommonModule` from `@angular/common`.

`CommonModule` contributes many of the common directives that applications need, including `ngIf` and `ngFor`.

`AppModule` doesn't import `CommonModule` directly. But it benefits from the fact that `BrowserModule` imports `CommonModule` **and re-exports it.**

The net effect is that an importer of `BrowserModule` gets `CommonModule` directives automatically as if it had declared them itself.

Many familiar Angular directives don't belong to `CommonModule`. For example, `NgModel` and `RouterLink` belong to Angular's `FormsModule` and `RouterModule` respectively. You must import those NgModules before you can use their directives.

To illustrate this point, you'll extend the sample app with *contact editor* whose `ContactComponent` is a form component. You'll have to import form support from the Angular `FormsModule`.

{@a add-contact-editor}

Add a *contact editor*

Imagine that you added the following *contact editor* files to the project by hand *without the help of the CLI*.

Form components are often complex and this is one is no exception. To make it manageable, all contact-related files are in an `src/app/contact` folder. The `ContactComponent` implementation is spread over three constituent HTML, TypeScript, and css files. There's a [custom pipe](guide/pipes#custom-pipes) (called `Awesome`), a `ContactHighlightDirective`, and a `ContactService` for fetching contacts. The `ContactService` was added to the `AppModule` providers. Now any class can inject the application-wide instances of the `ContactService` and `UserService`.

Import supporting *FormsModule*

The `ContactComponent` is written with Angular forms in the template-driven style.

Notice the `[(ngModel)]` binding in the middle of the component template, `contact.component.html`.

Two-way data binding `[(ngModel)]` is typical of the *template-driven* style. The `ngModel` is the selector for the `NgModel` directive. Although `NgModel` is an Angular directive, the *Angular compiler* won't recognize it for two reasons:

1. `AppModule` doesn't declare `NgModel` (and shouldn't).
2. `NgModel` wasn't imported via `BrowserModule`.

`ContactComponent` wouldn't behave like an Angular form anyway because form features such as validation aren't part of the Angular core.

To correct these problems, the `AppModule` must import *both* the `BrowserModule` *and* the **FormsModule from '@angular/forms'** like this.

You can write Angular form components in template-driven or [reactive](guide/reactive-forms) style. NgModules with components written in the _reactive_ style import the `ReactiveFormsModule`.

Now `[(ngModel)]` binding will work and the user input will be validated by Angular forms, once you declare the new component, pipe, and directive.

Never re-declare

Importing the `FormsModule` makes the `NgModelDirective` (and all of the other `FORMS_DIRECTIVES`) available to components declared in `AppModule`.

Do not also add these directives to the `AppModule` metadata's declarations.

Never re-declare classes that belong to another NgModule. Components, directives, and pipes should be declared in _exactly one NgModule_.

{@a declare-pipe}

Declare pipes

The revised application still won't compile until you declare the contact component, directive, and pipe.

Components and directives are *declarables*. So are **pipes**.

You learned earlier to generate and declare both components and directives with the CLI `ng generate` commands.

There's also a CLI command to generate and declare the `AwesomePipe`:

ng generate pipe awesome

However, if you write these class files by hand or opt-out of declaration with the `--skip-import` flag, you'll have to add the declarations yourself.

You were told to add the *contact editor* files by hand, so you must manually update the `declarations` in the `AppModule`:

Display the *ContactComponent*

Update the `AppComponent` template to display the `ContactComponent` by placing an element with its selector (`<app-contact>`) just below the title.

Run the app

Everything is in place to run the application with its contact editor. Try the example: ## Selector conflicts Look closely at the screen. Notice that the background of the application title text is _blue_. It should be _gold_ (see `src/app/app.component.html`). Only the contact name should be blue (see `src/app/contact/contact.component.html`). What went wrong? This application defines two highlight directives that set the background color of their host elements with a different color (gold and blue). One is defined at the root level (`src/app/highlight.directive.ts`); the other is in the contact editor folder (`src/app/contact/contact-highlight.directive.ts`). Their class names are different (`HighlightDirective` and `ContactHighlightDirective`) but their selectors both match any HTML element with a `highlight` attribute. Both directives are declared in the same `AppModule` so both directives are active for all components declared in `AppModule`. There's nothing

intrinsically wrong with multiple directives selecting the same element. Each could modify the element in a different, non-conflicting way. In _this case_, both directives compete to set the background color of the same element. The directive that's declared later (`ContactHighlightDirective`) always wins because its DOM changes overwrite the changes by the earlier `HighlightDirective`. The `ContactHighlightDirective` will make the application title text blue when it should be gold. Only the contact name should be blue (see `src/app/contact/contact.component.html`). If you cannot rename the selectors, you can resolve the conflicts by creating [feature modules](#feature-modules) that insulate the declarations in one NgModule from the declarations in another.
While it is legal to declare two _directives_ with the same selector in the same NgModule, the compiler will not let you declare two _components_ with the same selector in the same NgModule because it **cannot insert multiple components in the same DOM location**. Nor can you _import_ an NgModule that declares the same selector as another component in this NgModule. The reason is the same: an HTML element may be controlled by at most one Angular component. Either rename the selectors or use [feature modules](#feature-modules) to eliminate the conflict.
## Feature modules This tiny app is already experiencing structural issues. * The root `AppModule` grows larger with each new application class. * There are conflicting directives. The `ContactHighlightDirective` in the contact re-colors the work done by the `HighlightDirective` declared in `AppModule` and colors the application title text when it should color only the `ContactComponent`. * The app lacks clear boundaries between contact functionality and other application features. That lack of clarity makes it harder to assign development responsibilities to different teams. _Feature modules_ can help resolve these issues. Architecturally, a feature module is an NgModule class that is dedicated to an application feature or workflow. Technically, it's another class adorned by the `@NgModule` decorator, just like a root `AppModule`. Feature module metadata have the same properties as root module metadata. When loaded together, the root module and the feature module share the same dependency injector, which means the services provided in a feature module are available to all. These two module types have the following significant technical differences: * You _boot_ the root module to _launch_ the app; you _import_ a feature module to _extend_ the app. * A feature module can expose or hide its [declarables](#declarables) from other NgModules. Otherwise, a feature module is distinguished primarily by its intent. A feature module delivers a cohesive set of functionality focused on an application business domain, user workflow, facility (forms, http, routing), or collection of related utilities. Feature modules help you partition the app into areas of specific interest and purpose. A feature module collaborates with the root module and with other NgModules through the services it provides and the components, directives, and pipes that it shares. {@a contact-module-v1}

Make *contact editor* a feature

In this section, you refactor the *contact editor* functionality out of the root `AppModule` and into a dedicated feature module by following these steps.

1. Create the `ContactModule` feature module in its own folder.

2. Copy the *contact editor* declarations and providers from `AppModule` to `ContactModule`.
3. Export the `ContactComponent`.
4. Import the `ContactModule` into the `AppModule`.
5. Cleanup the `AppModule`.

You'll create one new `ContactModule` class and change one existing `AppModule` class. All other files are untouched.

Create the feature module

Generate the *ContactModule* and its folder with an Angular CLI command.

```
ng generate module contact
```

Here's the generated `ContactModule`.

After modifying the initial `ContactsModule` as outlined above, it looks like this.

The following sections discuss the important changes.

Import *CommonModule*

Notice that `ContactModule` imports `CommonModule`, not `BrowserModule`. The CLI module generation took care of this for you.

Feature module components need the common Angular directives but not the services and bootstrapping logic in `BrowserModule`. See the NgModule FAQs for more details.

Import *FormsModule*

The `ContactModule` imports the `FormsModule` because its `ContactComponent` uses `NgModel`, one of the `FormsModule` directives.

NgModules don't inherit access to the declarations of the root `AppModule` or any other NgModule. Each NgModule must import what it needs. Because `ContactComponent` needs the form directives, its `ContactModule` must import `FormsModule`.

Copy declarations

The `ContactModule` declares the *contact editor* components, directives and pipes.

The app fails to compile at this point, in part because `ContactComponent` is currently declared in both the

`AppModule` and the `ContactModule`. A component may only be declared in one NgModule. You'll fix this problem shortly.

{@a root-scoped-providers}

Providers are root-scoped

The `ContactModule` provides the `ContactService` and the `AppModule` will stop providing it after refactoring.

Architecturally, the `ContactService` belongs to the *contact editor* domain. Classes in the rest of the app do not need the `ContactService` and shouldn't inject it. So it makes sense for the `ContactModule` to provide the `ContactService` as it does.

You might expect that the `ContactService` would only be injectable in classes declared or provided in the `ContactModule`.

That's not the case. *Any* class *anywhere* can inject the `ContactService` because `ContactModule` providers are *root*-scoped.

To be precise, all _eagerly loaded_ modules— modules loaded when the application starts — are root-scoped. This `ContactModule` is eagerly loaded. You will learn that services provided in [_lazy-loaded_ modules](#lazy-loaded-modules) have their own scope.

Angular does not have *module*-scoping mechanism. Unlike components, NgModule instances do not have their own injectors so they can't have their own provider scopes.

`ContactService` remains an *application*-scoped service because Angular registers all NgModule `providers` with the application's *root injector*. This is true whether the service is provided directly in the root `AppModule` or in an imported feature module like `ContactModule`.

In practice, service scoping is rarely an issue. Components don't accidentally inject a service. To inject the `ContactService`, you'd have to import its *type* and explicitly inject the service into a class constructor. Only *contact editor* components should import the `ContactService` type.

If it's really important to you to restrict the scope of a service, provide it in the feature's top-level component (`ContactComponent` in this case).

For more on this topic, see "How do I restrict service scope to a module?" in the NgModule FAQs.

Export public-facing components

The `ContactModule` makes the `ContactComponent` *public* by *exporting* it.

Declared classes are *private* by default. Private declarables may only appear in the templates of components declared by the *same* NgModule. They are invisible to components in *other* NgModules.

That's a problem for the `AppComponent`. Both components *used to be* declared in `AppModule` so Angular could display the `ContactComponent` within the `AppComponent`. Now that the `ContactComponent` is declared in its own feature module. The `AppComponent` cannot see it unless it is public.

The first step toward a solution is to *export* the `ContactComponent`. The second step is to *import* the `ContactModule` in the `AppModule`, which you'll do when you refactor the *AppModule*.

The `AwesomePipe` and `ContactHighlightDirective` remain private and are hidden from the rest of the application.

The `ContactHighlightDirective`, being private, no longer overrides the `HighlightDirective` in the `AppComponent`. The background of the title text is gold as intended.

{@a refactor-appmodule}

Refactor the *AppModule*

Return to the `AppModule` and remove everything specific to the *contact editor* feature set. Leave only the classes required at the application root level.

- Delete the *contact editor* import statements.
- Delete the *contact editor* declarations and providers.
- Delete the `FormsModule` from the `imports` list (the `AppComponent` doesn't need it).
- Import the `ContactModule` so the app can continue to display the exported `ContactComponent`.

Here's the refactored `AppModule`, presented side-by-side with the previous version.

Improvements

There's a lot to like in the revised `AppModule`.

- It does not change as the *Contact* domain grows.
- It only changes when you add new NgModules.
- It's simpler:
  - Fewer import statements.

    - No `FormsModule` import.
    - No *contact editor* declarations.
    - No `ContactService` provider.
    - No *highlight directive* conflicts.

Try this `ContactModule` version of the sample.

Try the live example.

{@a routing-modules} {@a lazy-loaded-modules}

Routing modules

Navigating the app with the Angular Router reveals new dimensions of the NgModule.

In this segment, you'll learn to write *routing modules* that configure the router. You'll discover the implications of *lazy loading* a feature module with the router's `loadChildren` method.

Imagine that the sample app has evolved substantially along the lines of the Tour of Heroes tutorial.

- The app has three feature modules: Contact, Hero (new), and Crisis (new).
- The Angular router helps users navigate among these modules.
- The `ContactComponent` is the default destination when the app starts.
- The `ContactModule` continues to be *eagerly loaded* when the application starts.
- `HeroModule` and the `CrisisModule` are *lazy-loaded*.

There's too much code behind this sample app to review every line. Instead, the guide explores just those parts necessary to understand new aspects of NgModules.

You can examine the complete source for this version of the app in the live example.

{@a app-component-template}

The root *AppComponent*

The revised `AppComponent` template has a title, three links, and a `<router-outlet>`.

The `<app-contact>` element that displayed the `ContactComponent` is gone; you're routing to the *Contact* page now.

The root *AppModule*

The `AppModule` is slimmer now.

The `AppModule` is no longer aware of the application domains such as contacts, heroes, and crises. Those concerns are pushed down to `ContactModule`, `HeroesModule`, and `CrisisModule` respectively and only the routing configuration knows about them.

The significant change from version 2 is the addition of the *AppRoutingModule* to the NgModule `imports`. The `AppRoutingModule` is a routing module that handles the app's routing concerns.

AppRoutingModule

The router is the subject of the Routing & Navigation guide, so this section skips many routing details and concentrates on the *intersection* of NgModules and routing.

You can specify router configuration directly within the root `AppModule` or within a feature module.

The *Router guide* recommends instead that you locate router configuration in separate, dedicated NgModules, called *routing modules*. You then import those routing modules into their corresponding root or feature modules.

The goal is to separate the normal declarative concerns of an NgModule from the often complex router configuration logic.

By convention, a routing module's name ends in `...RoutingModule`. The top-level root module is `AppModule` and it imports its companion *routing module* called `AppRoutingModule`.

Here is this app's `AppRoutingModule`, followed by a discussion.

The `AppRoutingModule` defines three routes:

The first route redirects the empty URL (such as `http://host.com/`) to another route whose path is `contact` (such as `http://host.com/contact`).

The `contact` route isn't defined within the `AppRoutingModule`. It's defined in the *Contact* feature's *own* routing module, `ContactRoutingModule`.

It's standard practice for feature modules with routing components to define their own routes. You'll get to [`ContactRoutingModule`](#contact-routing-module) in a moment.

The remaining two routes use lazy loading syntax to tell the router where to find the modules for the hero and crisis features:

A lazy-loaded NgModule location is a _string_, not a _type_. In this app, the string identifies both the NgModule _file_ and the NgModule _class_, the latter separated from the former by a `#`.

Routing module imports

A *routing module* typically imports the Angular `RouterModule` so it can register routes.

It may also import a *feature module* which registers routes (either directly or through its companion *routing module*).

This `AppRoutingModule` does both.

It first imports the `ContactModule`, which [as you'll see](), imports its own `ContactRoutingModule`.

Import order matters! Because "contacts" is the first defined route and the default route for the app, you must import it *before* all other routing-related modules.

The second import registers the routes defined in this module by calling the `RouterModule.forRoot` class method.

The `forRoot` method does two things:

1. Configures the router with the supplied *routes*.
2. Initializes the Angular router itself.

Call `RouterModule.forRoot` exactly once for the entire app. Calling it in the `AppRoutingModule`, the companion to the root `AppModule`, is a good way to ensure that this method is called exactly once. Never call `RouterModule.forRoot` in a feature's _routing module_.

Re-export *RouterModule*

All *routing modules* should re-export the `RouterModule`.

Re-exporting `RouterModule` makes the router directives available to the companion module that imports it. This is a considerable convenience for the importing module.

For example, the `AppComponent` template relies on the [`routerLink`]() directive to turn the user's clicks into navigations. The Angular compiler only recognizes `routerLink` because

- `AppComponent` is declared by `AppModule`,
- `AppModule` imports `AppRoutingModule`,
- `AppRoutingModule` exports `RouterModule`, and
- `RouterModule` exports the `RouterLink` directive.

If `AppRoutingModule` didn't re-export `RouterModule`, the `AppModule` would have to import the `RouterModule` itself.

{@a contact-routing-module}

Routing to a feature module

The three feature modules (`ContactModule`, `HeroModule`, `CrisisModule`) have corresponding routing modules (`ContactRoutingModule`, `HeroRoutingModule`, `CrisisRoutingModule`).

They follow the same pattern as the `AppRoutingModule`. * define routes * register the routes with Angular's `RouterModule` * export the `RouterModule`.

The `ContactRoutingModule` is the simplest of the three. It defines and registers a single route to the `ContactComponent`.

There is **one critical difference** from `AppRoutingModule`: you pass the routes to `RouterModule.forChild`, not `forRoot`.

Always call \`RouterModule.forChild\` in a feature-routing module. Never call \`RouterModule.forRoot\`.

ContactModule changes

Because the app navigates to the `ContactComponent` instead of simply displaying it in the `AppComponent` template, the `ContactModule` has changed.

- It imports the `ContactRoutingModule`.

- It no longer exports `ContactComponent`.

The `ContactComponent` is only displayed by the router, No template references its `<app-contact>` selector. There's no reason to make it public via the `exports` array.

Here is the latest version, side-by-side with the previous version.

{@a hero-module}

Lazy-loaded routing

The `HeroModule` and `CrisisModule` have corresponding *routing modules*, `HeroRoutingModule` and `CrisisRoutingModule`.

The app *lazy loads* the `HeroModule` and the `CrisisModule`. That means the `HeroModule` and the

`CrisisModule` are not loaded into the browser until the user navigates to their components.

Do not import the `HeroModule` or `CrisisModule` or any of their classes outside of their respective file folders. If you do, you will unintentionally load those modules and all of their code when the application starts, defeating the purpose of lazy loading. For example, if you import the `HeroService` in `AppModule`, the `HeroService` class and all related hero classes will be loaded when the application starts.

Lazy loading can improve the app's perceived performance because the browser doesn't have to process lazy-loaded code when the app starts. It may *never* process that code.

You cannot tell that these modules are lazy-loaded by looking at their *routing modules*. They happen to be a little more complex than `ContactRoutingModule`. For example, The `HeroRoutingModule` has child routes. But the added complexity springs from intrinsic hero and crisis functionality, not from lazy loading. Fundamentally, these *routing modules* are just like `ContactRoutingModule` and you write them the same way.

{@a lazy-load-DI}

Lazy-loaded NgModule providers

There is a **runtime difference** that can be significant. Services provided by lazy-loaded NgModules are only available to classes instantiated within the lazy-loaded context. The reason has to do with dependency injection.

When an NgModule is *eagerly loaded* as the application starts, its providers are added to the application's *root injector*. Any class in the application can inject a service from the *root injector*.

When the router *lazy loads* an NgModule, Angular instantiates the module with a *child injector* (a descendant of the *root injector*) and adds the module's providers to this *child injector*. Classes created with the *child injector* can inject one of its provided services. Classes created with *root injector* cannot.

Each of the three feature modules has its own data access service. Because the `ContactModule` is *eagerly loaded* when the application starts, its `ContactService` is provided by the application's *root dependency injector*. That means the `ContactService` can be injected into any application class, including hero and crisis components.

Because `CrisisModule` is *lazy-loaded*, its `CrisisService` is provided by the `CrisisModule` *child injector*. It can only be injected into one of the crisis components. No other kind of component can inject the `CrisisService` because no other kind of component can be reached along a route that lazy loads the `CrisisModule`.

Lazy-loaded NgModule lifetime

Both eager and lazy-loaded NgModules are created *once* and never destroyed. This means that their provided service instances are created *once* and never destroyed.

As you navigate among the application components, the router creates and destroys instances of the contact, hero, and crisis components. When these components inject data services provided by their modules, they get the same data service instance each time.

If the `HeroService` kept a cache of unsaved changes and the user navigated to the `ContactComponent` or the `CrisisListComponent`, the pending hero changes would remain in the one `HeroService` instance, waiting to be saved.

But if you provided the `HeroService` in the `HeroComponent` instead of the `HeroModule`, new `HeroService` instances would be created each time the user navigated to a hero component. Previously pending hero changes would be lost.

To illustrate this point, the sample app provides the `HeroService` in the `HeroComponent` rather than the `HeroModule`.

Run the app, open the browser development tools, and look at the console as you navigate among the feature pages.

// App starts ContactService instance created. ... // Navigate to Crisis Center CrisisService instance created. ... // Navigate to Heroes HeroService instance created. ... // Navigate to Contact HeroService instance destroyed. ... // Navigate back to Heroes HeroService instance created.

The console log shows the `HeroService` repeatedly created and destroyed. The `ContactService` and `CrisisService` are created but never destroyed, no matter where you navigate.

Run it

Try this routed version of the sample.

Try the live example.

{@a shared-module}

Shared modules

The app is shaping up. But there are a few annoying problems. There are three unnecessarily different *highlight directives* and the many files cluttering the app folder level could be better organized.

You can eliminate the duplication and tidy-up by writing a `SharedModule` to hold the common components, directives, and pipes. Then share this NgModule with the other NgModules that need these declarables.

Use the CLI to create the `SharedModule` class in its `src/app/shared` folder.

ng generate module shared

Now refactor as follows:

- Move the `AwesomePipe` from `src/app/contact` to `src/app/shared`.
- Move the `HighlightDirective` from `src/app/hero` to `src/app/shared`.
- Delete the *highlight directive* classes from `src/app/` and `src/app/contact`.
- Update the `SharedModule` as follows:

Note the following:

- It declares and exports the shared pipe and directive.
- It imports and re-exports the `CommonModule` and `FormsModule`
- It can re-export `FormsModule` without importing it.

Re-exporting NgModules

Technically, there is no need for `SharedModule` to import `CommonModule` or `FormsModule`. `SharedModule` doesn't declare anything that needs material from `CommonModule` or `FormsModule`.

But NgModules that would like to import `SharedModule` for its pipe and highlight directive happen also to declare components that need `NgIf` and `NgFor` from `CommonModule` and do two-way binding with `[(ngModel)]` from the `FormsModule`.

Normally, they'd have to import `CommonModule` and `FormsModule` as well as `SharedModule`. Now they can just import `SharedModule`. By exporting `CommonModule` and `FormsModule`, `SharedModule` makes them available to its importers *for free*.

A trimmer *ContactModule*

See how `ContactModule` became more concise, compared to its previous version:

Notice the following:

- The `AwesomePipe` and `ContactHighlightDirective` are gone.
- The imports include `SharedModule` instead of `CommonModule` and `FormsModule`.
- The new version is leaner and cleaner.

Why *TitleComponent* isn't shared

`SharedModule` exists to make commonly used components, directives, and pipes available for use in the templates of components in many other NgModules.

The `TitleComponent` is used only once by the `AppComponent`. There's no point in sharing it.

{@a no-shared-module-providers}

Why *UserService* isn't shared

While many components share the same service instances, they rely on Angular dependency injection to do this kind of sharing, not the NgModule system.

Several components of the sample inject the `UserService`. There should be only one instance of the `UserService` in the entire application and only one provider of it.

`UserService` is an application-wide singleton. You don't want each NgModule to have its own separate instance. Yet there is a real danger of that happening if the `SharedModule` provides the `UserService`.

Do *not* specify app-wide singleton `providers` in a shared module. A lazy-loaded NgModule that imports that shared module makes its own copy of the service.

{@a core-module}

The Core module

At the moment, the root folder is cluttered with the `UserService` and `TitleComponent` that only appear in the root `AppComponent`. You didn't include them in the `SharedModule` for reasons just explained.

Instead, gather them in a single `CoreModule` that you import once when the app starts and never import anywhere else.

Perform the following steps:

1. Create a `CoreModule` class in an `src/app/core` folder.
2. Move the `TitleComponent` and `UserService` from `src/app/` to `src/app/core`.
3. Declare and export the `TitleComponent`.
4. Provide the `UserService`.
5. Update the root `AppModule` to import `CoreModule`.

Most of this work is familiar. The interesting part is the `CoreModule`.

You're importing some extra symbols from the Angular core library that you're not using yet. They'll become relevant later in this page.

The `@NgModule` metadata should be familiar. You declare the `TitleComponent` because this NgModule owns it. You export it because `AppComponent` (which is in `AppModule`) displays the title in its template. `TitleComponent` needs the Angular `NgIf` directive that you import from `CommonModule`.

`CoreModule` provides the `UserService`. Angular registers that provider with the app root injector, making a singleton instance of the `UserService` available to any component that needs it, whether that component is eagerly or lazily loaded.

Why bother?

This scenario is clearly contrived. The app is too small to worry about a single service file and a tiny, one-time component. A \`TitleComponent\` sitting in the root folder isn't bothering anyone. The root \`AppModule\` can register the \`UserService\` itself, as it does currently, even if you decide to relocate the \`UserService\` file to the \`src/app/core\` folder. Real-world apps have more to worry about. They can have several single-use components (such as spinners, message toasts, and modal dialogs) that appear only in the \`AppComponent\` template. You don't import them elsewhere so they're not shared in that sense. Yet they're too big and messy to leave loose in the root folder. Apps often have many singleton services like this sample's \`UserService\`. Each must be registered exactly once, in the app root injector, when the application starts. While many components inject such services in their constructors—and therefore require JavaScript \`import\` statements to import their symbols—no other component or NgModule should define or re-create the services themselves. Their _providers_ aren't shared. We recommend collecting such single-use classes and hiding their details inside a \`CoreModule\`. A simplified root \`AppModule\` imports \`CoreModule\` in its capacity as orchestrator of the application as a whole.

A trimmer *AppModule*

Here is the updated `AppModule` paired with version 3 for comparison:

`AppModule` now has the following qualities:

- A little smaller because many `src/app/root` classes have moved to other NgModules.
- Stable because you'll add future components and providers to other NgModules, not this one.
- Delegated to imported NgModules rather than doing work.
- Focused on its main task, orchestrating the app as a whole.

{@a core-for-root}

Configure core services with *CoreModule.forRoot*

An NgModule that adds providers to the application can offer a facility for configuring those providers as well.

By convention, the `forRoot` static method both provides and configures services at the same time. It takes a service configuration object and returns a ModuleWithProviders, which is a simple object with the following properties:

- `ngModule` : the `CoreModule` class
- `providers` : the configured providers

The root `AppModule` imports the `CoreModule` and adds the `providers` to the `AppModule` providers.

More precisely, 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 NgModules.

Add a `CoreModule.forRoot` method that configures the core `UserService` .

You've extended the core `UserService` with an optional, injected `UserServiceConfig` . If a `UserServiceConfig` exists, the `UserService` sets the user name from that config.

Here's `CoreModule.forRoot` that takes a `UserServiceConfig` object:

Lastly, call it within the `imports` list of the `AppModule` .

The app displays "Miss Marple" as the user instead of the default "Sherlock Holmes".

Call `forRoot` only in the root module, `AppModule`. Calling it in any other NgModule, particularly in a lazy-loaded NgModule, is contrary to the intent and can produce a runtime error. Remember to _import_ the result; don't add it to any other `@NgModule` list.

{@a prevent-reimport}

Prevent reimport of the *CoreModule*

Only the root `AppModule` should import the `CoreModule` . Bad things happen if a lazy-loaded NgModule imports it.

You could hope that no developer makes that mistake. Or you can guard against it and fail fast by adding the following `CoreModule` constructor.

The constructor tells Angular to inject the `CoreModule` into itself. That seems dangerously circular.

The injection would be circular if Angular looked for `CoreModule` 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 is no ancestor injector that could provide an instance of `CoreModule`. 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 NgModule such as `HeroModule` (try it).

Angular creates a lazy-loaded NgModule 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.

Conclusion

You made it! You can examine and download the complete source for this final version from the live example.

Frequently asked questions

Now that you understand NgModules, you may be interested in the companion NgModule FAQs page with its ready answers to specific design and implementation questions.

Npm Packages

The Angular CLI, Angular applications, and Angular itself depend upon features and functionality provided by libraries that are available as npm packages.

You can download and install these npm packages with the npm client, which runs as a node.js application.

The yarn client is a popular alternative for downloading and installing npm packages. The Angular CLI uses `yarn` by default to install npm packages when you create a new project.

Node.js and npm are essential to Angular development. [Get them now](https://docs.npmjs.com/getting-started/installing-node "Installing Node.js and updating npm") if they're not already installed on your machine. **Verify that you are running node `v4.x.x` or higher and npm `3.x.x` or higher** by running the commands `node -v` and `npm -v` in a terminal/console window. Older versions produce errors. Consider using [nvm](https://github.com/creationix/nvm) for managing multiple versions of node and npm. You may need [nvm](https://github.com/creationix/nvm) if you already have projects running on your machine that use other versions of node and npm.

package.json

Both `npm` and `yarn` install packages identified in a package.json file.

The CLI `ng new` command creates a default `package.json` file for your project. This `package.json` specifies *a starter set of packages* that work well together and jointly support many common application scenarios.

You will add packages to `package.json` as your application evolves. You may even remove some.

This guide focuses on the most important packages in the starter set.

dependencies and *devDependencies*

The `package.json` includes two sets of packages, dependencies and devDependencies.

The *dependencies* are essential to *running* the application. The *devDependencies* are only necessary to *develop* the application.

{@a dependencies}

Dependencies

The `dependencies` section of `package.json` contains:

- **Angular packages**: Angular core and optional modules; their package names begin `@angular/`.
- **Support packages**: 3rd party libraries that must be present for Angular apps to run.
- **Polyfill packages**: Polyfills plug gaps in a browser's JavaScript implementation.

Angular Packages

@angular/animations: Angular's animations library makes it easy to define and apply animation effects such as page and list transitions. Read about it in the Animations guide.

@angular/common: The commonly needed services, pipes, and directives provided by the Angular team. The `HttpClientModule` is also here, in the '@angular/common/http' subfolder.

@angular/core: Critical runtime parts of the framework needed by every application. Includes all metadata decorators, `Component`, `Directive`, dependency injection, and the component lifecycle hooks.

@angular/compiler: Angular's *Template Compiler*. It understands templates and can convert them to code that makes the application run and render. Typically you don’t interact with the compiler directly; rather, you use it indirectly via `platform-browser-dynamic` when JIT compiling in the browser.

@angular/forms: support for both template-driven and reactive forms.

@angular/http: Angular's old, soon-to-be-deprecated, HTTP client.

@angular/platform-browser: Everything DOM and browser related, especially the pieces that help render into the DOM. This package also includes the `bootstrapStatic()` method for bootstrapping applications for production builds that pre-compile with AOT.

@angular/platform-browser-dynamic: Includes Providers and methods to compile and run the app on the client using the JIT compiler.

@angular/router: The router module navigates among your app pages when the browser URL changes.

@angular/upgrade: Set of utilities for upgrading AngularJS applications to Angular.

{@a polyfills}

Polyfill packages

Many browsers lack native support for some features in the latest HTML standards, features that Angular requires. "Polyfills" can emulate the missing features. The Browser Support guide explains which browsers need polyfills and how you can add them.

The default `package.json` installs the **core-js** package which polyfills missing features for several popular browser.

Support packages

rxjs: Many Angular APIs return *observables*. RxJS is an implementation of the proposed Observables specification currently before the TC39 committee that determines standards for the JavaScript language.

zone.js: Angular relies on zone.js to run Angular's change detection processes when native JavaScript operations raise events. Zone.js is an implementation of a specification currently before the TC39 committee that determines standards for the JavaScript language.

{@a dev-dependencies}

DevDependencies

The packages listed in the *devDependencies* section of the `package.json` help you develop the application on your local machine.

You don't deploy them with the production application although there is no harm in doing so.

@angular/cli: The Angular CLI tools.

@angular/compiler-cli: The Angular compiler, which is invoked by the Angular CLI's `build` and `serve` commands.

@angular/language-service: The Angular language service analyzes component templates and provides type and error information that TypeScript-aware editors can use to improve the developer's experience. For example, see the Angular language service extension for VS Code

**@types/... **: TypeScript definition files for 3rd party libraries such as Jasmine and node.

codelyzer: A linter for Angular apps whose rules conform to the Angular style guide.

**jasmine/... **: packages to support the Jasmine test library.

**karma/... **: packages to support the karma test runner.

protractor: an end-to-end (e2e) framework for Angular apps. Built on top of WebDriverJS.

ts-node: TypeScript execution environment and REPL for node.

tslint: a static analysis tool that checks TypeScript code for readability, maintainability, and functionality errors.

typescript: the TypeScript language server, including the *tsc* TypeScript compiler.

So many packages! So many files!

The default `package.json` installs more packages than you'll need for your project.

A given package may contain tens, hundreds, even thousands of files, all of them in your local machine's `node_modules` directory. The sheer volume of files is intimidating,

You can remove packages that you don't need but how can you be sure that you won't need it? As a practical matter, it's better to install a package you don't need than worry about it. Extra packages and package files on your local development machine are harmless.

By default the Angular CLI build process bundles into a single file just the few "vendor" library files that your application actually needs. The browser downloads this bundle, not the original package files.

See the Deployment to learn more.

Pipes

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 arrive, 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 in Plunker 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.

Focus on the component's template.

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](api?type=pipe) of the [API Reference](api); filter for entries that include the word "pipe". Angular doesn't have a `FilterPipe` or an `OrderByPipe` for reasons explained in the [Appendix](guide/pipes#no-filter-pipe) 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`)

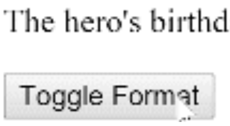
Modify the birthday template to give the date pipe a format parameter. After formatting the hero's April 15th birthday, it renders as **`04/15/88`**:

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:

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'`).

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

Read more about the `DatePipe` format options in the [Date Pipe](api/common/DatePipe) 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`**.

This example—which displays **`FRIDAY, APRIL 15, 1988`**—chains the same pipes as above, but passes in a parameter to `date` as well.

Custom pipes

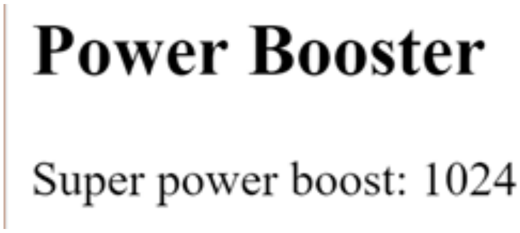
You can write your own custom pipes. Here's a custom pipe named `ExponentialStrengthPipe` that can boost a hero's powers:

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.



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`.

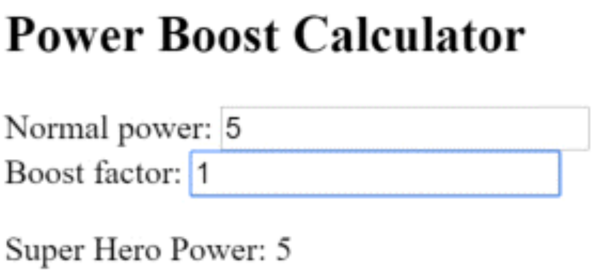
Remember the declarations array
You must register custom pipes. If you don't, Angular reports an error. Angular CLI's generator registers the pipe automatically.

To probe the behavior in the , 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`.



Power Boost Calculator

Normal power: 5
Boost factor: 1

Super Hero Power: 5

{@a change-detection}

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:

The companion component class provides heroes, adds heroes into the array, and can reset the array.

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.

Here's the `FlyingHeroesPipe` implementation, which follows the pattern for custom pipes described earlier.

Notice the odd behavior in the : when you add flying heroes, none of them are displayed under "Heroes who fly."

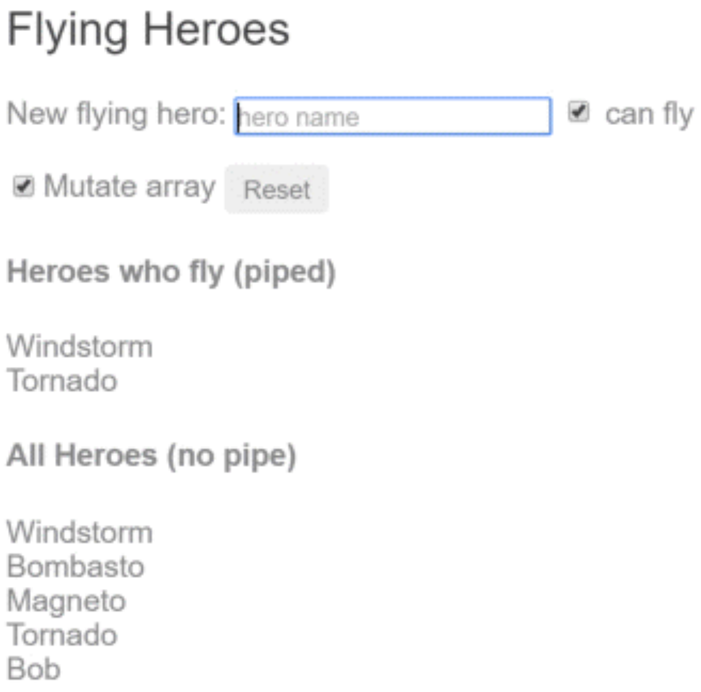
Although you're not getting the behavior you want, Angular isn't broken. It's just using a different change-detection algorithm that ignores changes to the list or any of its items.

Notice how a hero is added:

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: hero name ☑ can fly

☑ Mutate array Reset

Heroes who fly (piped)

Windstorm
Tornado

All Heroes (no pipe)

Windstorm
Bombasto
Magneto
Tornado
Bob

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:

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.

{@a impure-flying-heroes}

An impure *FlyingHeroesPipe*

A flip of the switch turns the `FlyingHeroesPipe` into a `FlyingHeroesImpurePipe`. The complete implementation is as follows:

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.

You can derive a `FlyingHeroesImpureComponent` from `FlyingHeroesComponent`.

The only substantive change is the pipe in the template. You can confirm in the that the *flying heroes* display updates as you add heroes, even when you mutate the `heroes` array.

{@a async-pipe}

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.

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

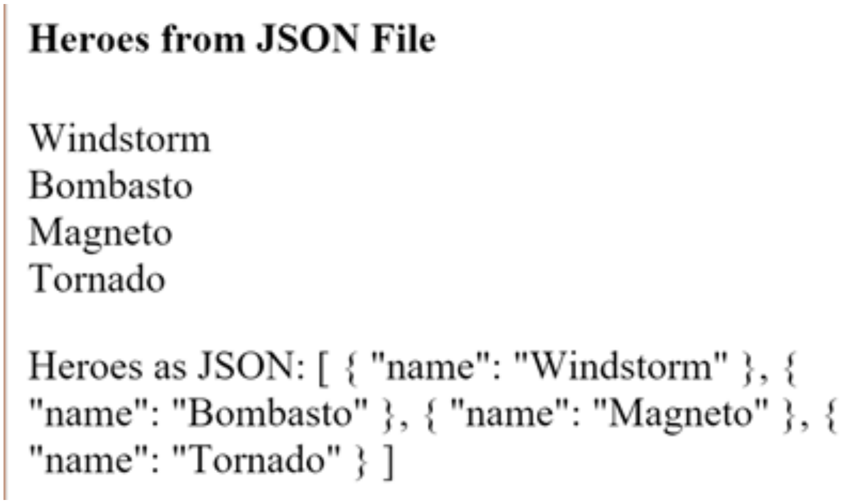
Write one more impure pipe, a pipe that makes an HTTP request.

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:

Now demonstrate it in a harness component whose template defines two bindings to this pipe, both requesting the heroes from the `heroes.json` file.

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](api/common/JsonPipe) provides an easy way to diagnosis a mysteriously failing data binding or inspect an object for future binding.

{@a pure-pipe-pure-fn}

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.

{@a no-filter-pipe}

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.

QuickStart

Good tools make application development quicker and easier to maintain than if you did everything by hand.

The **Angular CLI** is a ***command line interface*** tool that can create a project, add files, and perform a variety of ongoing development tasks such as testing, bundling, and deployment.

The goal in this guide is to build and run a simple Angular application in TypeScript, using the Angular CLI while adhering to the Style Guide recommendations that benefit *every* Angular project.

By the end of the chapter, you'll have a basic understanding of development with the CLI and a foundation for both these documentation samples and for real world applications.

And you can also download the example.

Step 1. Set up the Development Environment

You need to set up your development environment before you can do anything.

Install **Node.js® and npm** if they are not already on your machine.

Verify that you are running at least node `6.9.x` and npm `3.x.x` by running `node -v` and `npm -v` in a terminal/console window. Older versions produce errors, but newer versions are fine.

Then **install the Angular CLI** globally.

```
npm install -g @angular/cli
```

Step 2. Create a new project

Open a terminal window.

Generate a new project and skeleton application by running the following commands:

```
ng new my-app
```

Patience, please. It takes time to set up a new project; most of it is spent installing npm packages.

Step 3: Serve the application

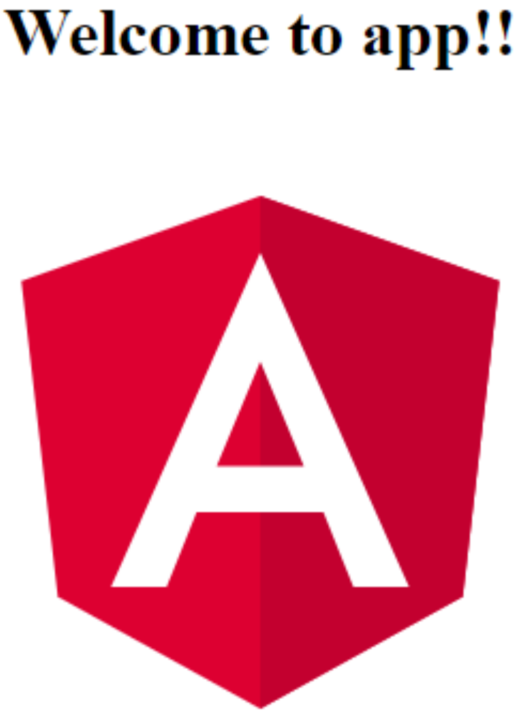
Go to the project directory and launch the server.

```
cd my-app ng serve --open
```

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

Using the `--open` (or just `-o`) option will automatically open your browser on `http://localhost:4200/`.

Your app greets you with a message:



Welcome to app!!

Step 4: Edit your first Angular component

The CLI created the first Angular component for you. This is the *root component* and it is named `app-root`. You can find it in `./src/app/app.component.ts`.

Open the component file and change the `title` property from *Welcome to app!!* to *Welcome to My First Angular App!!*:

The browser reloads automatically with the revised title. That's nice, but it could look better.

Open `src/app/app.component.css` and give the component some style.

Welcome to My First Angular App!!

Looking good!

What's next?

That's about all you'd expect to do in a "Hello, World" app.

You're ready to take the Tour of Heroes Tutorial and build a small application that demonstrates the great things you can build with Angular.

Or you can stick around a bit longer to learn about the files in your brand new project.

Project file review

An Angular CLI project is the foundation for both quick experiments and enterprise solutions.

The first file you should check out is `README.md` . It has some basic information on how to use CLI commands. Whenever you want to know more about how Angular CLI works make sure to visit the Angular CLI repository and Wiki.

Some of the generated files might be unfamiliar to you.

The `src` folder

Your app lives in the `src` folder. All Angular components, templates, styles, images, and anything else your app needs go here. Any files outside of this folder are meant to support building your app.

src
app
app.component.css
app.component.html
app.component.spec.ts
app.component.ts
app.module.ts
assets
.gitkeep
environments

environment.prod.ts
environment.ts
favicon.ico
index.html
main.ts
polyfills.ts
styles.css
test.ts
tsconfig.app.json
tsconfig.spec.json

File	Purpose
`app/app.component.{ts,html,css,spec.ts}`	Defines the `AppComponent` along with an HTML template, CSS stylesheet, and a unit test. It is the **root** component of what will become a tree of nested components as the application evolves.
`app/app.module.ts`	Defines `AppModule`, the [root module](guide/bootstrapping "AppModule: the root module") that tells Angular how to assemble the application. Right now it declares only the `AppComponent`. Soon there will be more components to declare.
`assets/*`	A folder where you can put images and anything else to be copied wholesale when you build your application.
`environments/*`	This folder contains one file for each of your destination environments, each exporting simple configuration variables to use in your application. The files are replaced on-the-fly when you build your app. You might use a different API endpoint for development than you do for production or maybe different analytics tokens. You might even use some mock services. Either way, the CLI has you covered.
`favicon.ico`	Every site wants to look good on the bookmark bar. Get started with your very own Angular icon.
`index.html`	The main HTML page that is served when someone visits your site. Most of the time you'll never need to edit it. The CLI automatically adds all `js` and `css` files when building your app so you never need to add any `

Reactive Forms

Reactive forms is an Angular technique for creating forms in a *reactive* style. This guide explains reactive forms as you follow the steps to build a "Hero Detail Editor" form.

{@a toc}

Try the Reactive Forms live-example.

You can also run the Reactive Forms Demo version and choose one of the intermediate steps from the "demo picker" at the top.

{@a intro}

Introduction to Reactive Forms

Angular offers two form-building technologies: *reactive* forms and *template-driven* forms. The two technologies belong to the `@angular/forms` library and share a common set of form control classes.

But they diverge markedly in philosophy, programming style, and technique. They even have their own modules: the `ReactiveFormsModule` and the `FormsModule`.

Reactive forms

Angular *reactive* forms facilitate a *reactive style* of programming that favors explicit management of the data flowing between a non-UI *data model* (typically retrieved from a server) and a UI-oriented *form model* that retains the states and values of the HTML controls on screen. Reactive forms offer the ease of using reactive patterns, testing, and validation.

With *reactive* forms, you create a tree of Angular form control objects in the component class and bind them to native form control elements in the component template, using techniques described in this guide.

You create and manipulate form control objects directly in the component class. As the component class has immediate access to both the data model and the form control structure, you can push data model values into the form controls and pull user-changed values back out. The component can observe changes in form control state and react to those changes.

One advantage of working with form control objects directly is that value and validity updates are always synchronous and under your control. You won't encounter the timing issues that sometimes plague a template-driven form and reactive forms can be easier to unit test.

In keeping with the reactive paradigm, the component preserves the immutability of the *data model*, treating it as a pure source of original values. Rather than update the data model directly, the component extracts user changes and forwards them to an external component or service, which does something with them (such as saving them) and returns a new *data model* to the component that reflects the updated model state.

Using reactive form directives does not require you to follow all reactive priniciples, but it does facilitate the reactive programming approach should you choose to use it.

Template-driven forms

Template-driven forms, introduced in the Template guide, take a completely different approach.

You place HTML form controls (such as `<input>` and `<select>`) in the component template and bind them to *data model* properties in the component, using directives like `ngModel`.

You don't create Angular form control objects. Angular directives create them for you, using the information in your data bindings. You don't push and pull data values. Angular handles that for you with `ngModel`. Angular updates the mutable *data model* with user changes as they happen.

For this reason, the `ngModel` directive is not part of the ReactiveFormsModule.

While this means less code in the component class, template-driven forms are asynchronous which may complicate development in more advanced scenarios.

{@a async-vs-sync}

Async vs. sync

Reactive forms are synchronous. Template-driven forms are asynchronous. It's a difference that matters.

In reactive forms, you create the entire form control tree in code. You can immediately update a value or drill down through the descendents of the parent form because all controls are always available.

Template-driven forms delegate creation of their form controls to directives. To avoid "*changed after checked*" errors, these directives take more than one cycle to build the entire control tree. That means you must wait a tick before manipulating any of the controls from within the component class.

For example, if you inject the form control with a `@ViewChild(NgForm)` query and examine it in the `ngAfterViewInit` lifecycle hook, you'll discover that it has no children. You must wait a tick, using `setTimeout`, before you can extract a value from a control, test its validity, or set it to a new value.

The asynchrony of template-driven forms also complicates unit testing. You must wrap your test block in `async()` or `fakeAsync()` to avoid looking for values in the form that aren't there yet. With reactive forms, everything is available when you expect it to be.

Which is better, reactive or template-driven?

Neither is "better". They're two different architectural paradigms, with their own strengths and weaknesses. Choose the approach that works best for you. You may decide to use both in the same application.

The balance of this *reactive forms* guide explores the *reactive* paradigm and concentrates exclusively on reactive forms techniques. For information on *template-driven forms*, see the *Forms* guide.

In the next section, you'll set up your project for the reactive form demo. Then you'll learn about the Angular form classes and how to use them in a reactive form.

{@a setup}

Setup

Create a new project named `angular-reactive-forms`:

```
ng new angular-reactive-forms
```

{@a data-model}

Create a data model

The focus of this guide is a reactive forms component that edits a hero. You'll need a `hero` class and some hero data.

Using the CLI, generate a new class named `data-model`:

```
ng generate class data-model
```

And copy the content below:

The file exports two classes and two constants. The `Address` and `Hero` classes define the application *data model*. The `heroes` and `states` constants supply the test data.

{@a create-component}

Create a *reactive forms* component

Generate a new component named `HeroDetail` :

ng generate component HeroDetail

And import:

Next, update the `HeroDetailComponent` class with a `FormControl` . `FormControl` is a directive that allows you to create and manage a `FormControl` instance directly.

Here you are creating a `FormControl` called `name` . It will be bound in the template to an HTML `input` box for the hero name.

A `FormControl` constructor accepts three, optional arguments: the initial data value, an array of validators, and an array of async validators.

This simple control doesn't have data or validators. In real apps, most form controls have both.

This guide touches only briefly on `Validators`. For an in-depth look at them, read the [Form Validation](guide/form-validation) guide.

{@a create-template}

Create the template

Now update the component's template, with the following markup.

To let Angular know that this is the input that you want to associate to the `name` `FormControl` in the class, you need `[formControl]="name"` in the template on the `<input>` .

Disregard the `form-control` _CSS_ class. It belongs to the Bootstrap CSS library, not Angular. It _styles_ the form but in no way impacts the logic of the form.

{@a import}

Import the *ReactiveFormsModule*

The HeroDetailComponent template uses `formControlName` directive from the `ReactiveFormsModule` .

Do the following two things in `app.module.ts`:

1. Use a JavaScript `import` statement to access the `ReactiveFormsModule`.
2. Add `ReactiveFormsModule` to the `AppModule`'s `imports` list.

{@a update}

Display the *HeroDetailComponent*

Revise the `AppComponent` template so it displays the `HeroDetailComponent`.

{@a essentials}

Essential form classes

It may be helpful to read a brief description of the core form classes.

- *AbstractControl* is the abstract base class for the three concrete form control classes: `FormControl`, `FormGroup`, and `FormArray`. It provides their common behaviors and properties, some of which are *observable*.

- *FormControl* tracks the value and validity status of an *individual* form control. It corresponds to an HTML form control such as an input box or selector.

- *FormGroup* tracks the value and validity state of a *group* of `AbstractControl` instances. The group's properties include its child controls. The top-level form in your component is a `FormGroup`.

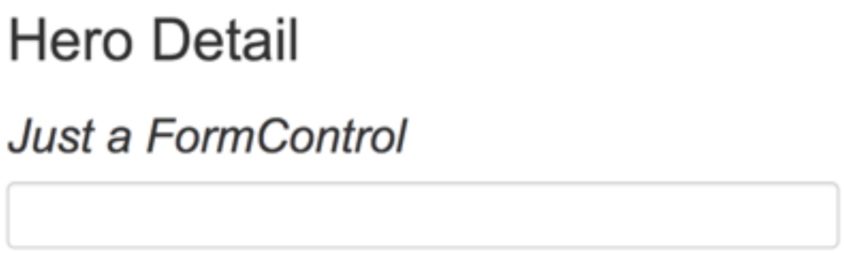
- *FormArray* tracks the value and validity state of a numerically indexed *array* of `AbstractControl` instances.

You'll learn more about these classes as you work through this guide.

Style the app

You used bootstrap CSS classes in the template HTML of both the `AppComponent` and the `HeroDetailComponent`. Add the `bootstrap` *CSS stylesheet* to the head of `styles.css`:

Now that everything is wired up, the browser should display something like this:



Hero Detail

Just a FormControl

{@a formgroup}

Add a FormGroup

Usually, if you have multiple *FormControls*, you'll want to register them within a parent `FormGroup`. This is simple to do. To add a `FormGroup`, add it to the imports section of `hero-detail.component.ts`:

In the class, wrap the `FormControl` in a `FormGroup` called `heroForm` as follows:

Now that you've made changes in the class, they need to be reflected in the template. Update `hero-detail.component.html` by replacing it with the following.

Notice that now the single input is in a `form` element. The `novalidate` attribute in the `<form>` element prevents the browser from attempting native HTML validations.

`formGroup` is a reactive form directive that takes an existing `FormGroup` instance and associates it with an HTML element. In this case, it associates the `FormGroup` you saved as `heroForm` with the form element.

Because the class now has a `FormGroup`, you must update the template syntax for associating the input with the corresponding `FormControl` in the component class. Without a parent `FormGroup`, `[formControl]="name"` worked earlier because that directive can stand alone, that is, it works without being in a `FormGroup`. With a parent `FormGroup`, the `name` input needs the syntax `formControlName=name` in order to be associated with the correct `FormControl` in the class. This syntax tells Angular to look for the parent `FormGroup`, in this case `heroForm`, and then *inside* that group to look for a `FormControl` called `name`.

Disregard the \`form-group\` _CSS_ class. It belongs to the [Bootstrap CSS library](), not Angular. Like the \`form-control\` class, it _styles_ the form but in no way impacts its logic.

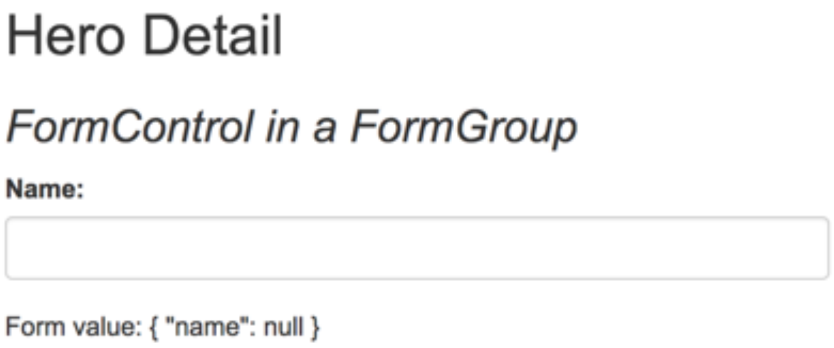
The form looks great. But does it work? When the user enters a name, where does the value go?

{@a json}

Taking a look at the form model

The value goes into the ***form model*** that backs the group's `FormControls`. To see the form model, add the following line after the closing `form` tag in the `hero-detail.component.html`:

The `heroForm.value` returns the *form model*. Piping it through the `JsonPipe` renders the model as JSON in the browser:


Hero Detail

FormControl in a FormGroup

Name:

Form value: { "name": null }

The initial `name` property value is the empty string. Type into the *name* input box and watch the keystokes appear in the JSON.

Great! You have the basics of a form.

In real life apps, forms get big fast. `FormBuilder` makes form development and maintenance easier.

{@a formbuilder}

Introduction to *FormBuilder*

The `FormBuilder` class helps reduce repetition and clutter by handling details of control creation for you.

To use `FormBuilder`, you need to import it into `hero-detail.component.ts`:

Use it now to refactor the `HeroDetailComponent` into something that's a little easier to read and write, by following this plan:

- Explicitly declare the type of the `heroForm` property to be `FormGroup`; you'll initialize it later.
- Inject a `FormBuilder` into the constructor.
- Add a new method that uses the `FormBuilder` to define the `heroForm`; call it `createForm`.
- Call `createForm` in the constructor.

The revised `HeroDetailComponent` looks like this:

`FormBuilder.group` is a factory method that creates a `FormGroup`. `FormBuilder.group` takes an object whose keys and values are `FormControl` names and their definitions. In this example, the `name` control is defined by its initial data value, an empty string.

Defining a group of controls in a single object makes for a compact, readable style. It beats writing an equivalent series of `new FormControl(...)` statements.

{@a validators}

Validators.required

Though this guide doesn't go deeply into validations, here is one example that demonstrates the simplicity of using `Validators.required` in reactive forms.

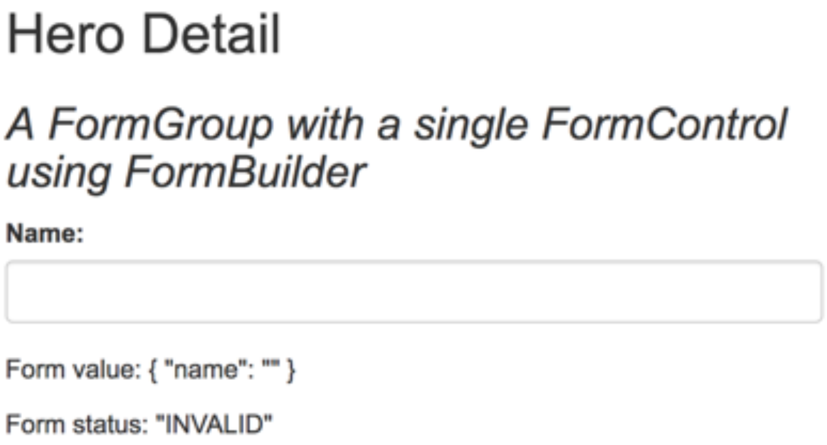
First, import the `Validators` symbol.

To make the `name` `FormControl` required, replace the `name` property in the `FormGroup` with an array. The first item is the initial value for `name`; the second is the required validator, `Validators.required`.

Reactive validators are simple, composable functions. Configuring validation is harder in template-driven forms where you must wrap validators in a directive.

Update the diagnostic message at the bottom of the template to display the form's validity status.

The browser displays the following:



Hero Detail

A FormGroup with a single FormControl using FormBuilder

Name:

Form value: { "name": "" }

Form status: "INVALID"

`Validators.required` is working. The status is `INVALID` because the input box has no value. Type into the input box to see the status change from `INVALID` to `VALID`.

In a real app, you'd replace the diagnosic message with a user-friendly experience.

Using `Validators.required` is optional for the rest of the guide. It remains in each of the following examples with the same configuration.

For more on validating Angular forms, see the Form Validation guide.

More FormControls

A hero has more than a name. A hero has an address, a super power and sometimes a sidekick too.

The address has a state property. The user will select a state with a `<select>` box and you'll populate the `<option>` elements with states. So import `states` from `data-model.ts`.

Declare the `states` property and add some address `FormControls` to the `heroForm` as follows.

Then add corresponding markup in `hero-detail.component.html` within the `form` element.

Reminder: Ignore the many mentions of `form-group`, `form-control`, `center-block`, and `checkbox` in this markup. Those are _bootstrap_ CSS classes that Angular itself ignores. Pay attention to the `formGroupName` and `formControlName` attributes. They are the Angular directives that bind the HTML controls to the Angular `FormGroup` and `FormControl` properties in the component class.

The revised template includes more text inputs, a select box for the `state`, radio buttons for the `power`, and a checkbox for the `sidekick`.

You must bind the option's value property with `[value]="state"`. If you do not bind the value, the select shows the first option from the data model.

The component *class* defines control properties without regard for their representation in the template. You define the `state`, `power`, and `sidekick` controls the same way you defined the `name` control. You tie these controls to the template HTML elements in the same way, specifying the `FormControl` name with the `formControlName` directive.

See the API reference for more information about radio buttons, selects, and checkboxes.

{@a grouping}

Nested FormGroups

This form is getting big and unwieldy. You can group some of the related `FormControls` into a nested `FormGroup`. The `street`, `city`, `state`, and `zip` are properties that would make a good *address* `FormGroup`. Nesting groups and controls in this way allows you to mirror the hierarchical structure of the data model and helps track validation and state for related sets of controls.

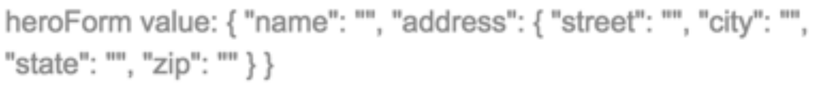
You used the `FormBuilder` to create one `FormGroup` in this component called `heroForm`. Let that be the parent `FormGroup`. Use `FormBuilder` again to create a child `FormGroup` that encapsulates the address controls; assign the result to a new `address` property of the parent `FormGroup`.

You’ve changed the structure of the form controls in the component class; you must make corresponding adjustments to the component template.

In `hero-detail.component.html`, wrap the address-related `FormControls` in a `div`. Add a `formGroupName` directive to the `div` and bind it to `"address"`. That's the property of the *address* child `FormGroup` within the parent `FormGroup` called `heroForm`.

To make this change visually obvious, slip in an `<h4>` header near the top with the text, *Secret Lair*. The new *address* HTML looks like this:

After these changes, the JSON output in the browser shows the revised *form model* with the nested address `FormGroup`:



```
heroForm value: { "name": "", "address": { "street": "", "city": "",
"state": "", "zip": "" } }
```

Great! You’ve made a group and you can see that the template and the form model are talking to one another.

{@a properties}

Inspect *FormControl* Properties

At the moment, you're dumping the entire form model onto the page. Sometimes you're interested only in the state of one particular `FormControl`.

You can inspect an individual `FormControl` within a form by extracting it with the `.get()` method. You can do this *within* the component class or display it on the page by adding the following to the template, immediately after the `{{form.value | json}}` interpolation as follows:

To get the state of a `FormControl` that’s inside a `FormGroup`, use dot notation to path to the control.

You can use this technique to display *any* property of a `FormControl` such as one of the following:

Property	Description
`myControl.value`	the value of a `FormControl`.
`myControl.status`	the validity of a `FormControl`. Possible values: `VALID`, `INVALID`, `PENDING`, or `DISABLED`.
`myControl.pristine`	`true` if the user has _not_ changed the value in the UI. Its opposite is `myControl.dirty`.
`myControl.untouched`	`true` if the control user has not yet entered the HTML control and triggered its blur event. Its opposite is `myControl.touched`.

Learn about other `FormControl` properties in the *AbstractControl* API reference.

One common reason for inspecting `FormControl` properties is to make sure the user entered valid values. Read more about validating Angular forms in the Form Validation guide.

{@a data-model-form-model}

The *data model* and the *form model*

At the moment, the form is displaying empty values. The `HeroDetailComponent` should display values of a hero, possibly a hero retrieved from a remote server.

In this app, the `HeroDetailComponent` gets its hero from a parent `HeroListComponent`

The `hero` from the server is the ***data model***. The `FormControl` structure is the ***form model***.

The component must copy the hero values in the *data model* into the *form model*. There are two important implications:

1. The developer must understand how the properties of the *data model* map to the properties of the *form model*.

2. User changes flow from the DOM elements to the *form model*, not to the *data model*. The form controls never update the *data model*.

The *form* and *data* model structures need not match exactly. You often present a subset of the *data model* on a particular screen. But it makes things easier if the shape of the *form model* is close to the shape of the *data model*.

In this `HeroDetailComponent`, the two models are quite close.

Recall the definition of `Hero` in `data-model.ts`:

Here, again, is the component's `FormGroup` definition.

There are two significant differences between these models:

1. The `Hero` has an `id`. The form model does not because you generally don't show primary keys to users.

2. The `Hero` has an array of addresses. This form model presents only one address, a choice revisited below.

Nonetheless, the two models are pretty close in shape and you'll see in a moment how this alignment facilitates copying the *data model* properties to the *form model* with the `patchValue` and `setValue` methods.

Take a moment to refactor the *address* `FormGroup` definition for brevity and clarity as follows:

Also be sure to update the import from `data-model` so you can reference the `Hero` and `Address` classes:

{@a set-data}

Populate the form model with *setValue* and *patchValue*

Previously you created a control and initialized its value at the same time. You can also initialize or reset the values *later* with the `setValue` and `patchValue` methods.

setValue

With **`setValue`**, you assign *every* form control value *at once* by passing in a data object whose properties exactly match the *form model* behind the `FormGroup`.

The `setValue` method checks the data object thoroughly before assigning any form control values.

It will not accept a data object that doesn't match the FormGroup structure or is missing values for any control in the group. This way, it can return helpful error messages if you have a typo or if you've nested controls incorrectly. `patchValue` will fail silently.

On the other hand, `setValue` will catch the error and report it clearly.

Notice that you can *almost* use the entire `hero` as the argument to `setValue` because its shape is similar to the component's `FormGroup` structure.

You can only show the hero's first address and you must account for the possibility that the `hero` has no addresses at all. This explains the conditional setting of the `address` property in the data object argument:

patchValue

With **`patchValue`**, you can assign values to specific controls in a `FormGroup` by supplying an object of key/value pairs for just the controls of interest.

This example sets only the form's `name` control.

With **`patchValue`** you have more flexibility to cope with wildly divergent data and form models. But unlike `setValue`, `patchValue` cannot check for missing control values and does not throw helpful errors.

When to set form model values (*ngOnChanges*)

Now you know *how* to set the *form model* values. But *when* do you set them? The answer depends upon when the component gets the *data model* values.

The `HeroDetailComponent` in this reactive forms sample is nested within a *master/detail* `HeroListComponent` (discussed below). The `HeroListComponent` displays hero names to the user. When the user clicks on a hero, the list component passes the selected hero into the `HeroDetailComponent` by binding to its `hero` input property.

In this approach, the value of `hero` in the `HeroDetailComponent` changes every time the user selects a new hero. You should call *setValue* in the ngOnChanges hook, which Angular calls whenever the input `hero` property changes as the following steps demonstrate.

First, import the `OnChanges` and `Input` symbols in `hero-detail.component.ts`.

Add the `hero` input property.

Add the `ngOnChanges` method to the class as follows:

reset the form flags

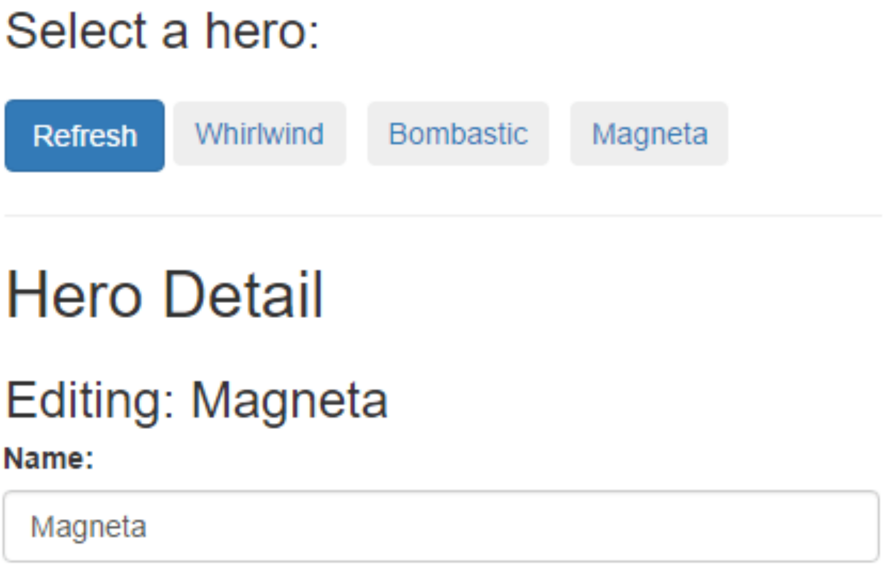
You should reset the form when the hero changes so that control values from the previous hero are cleared and status flags are restored to the *pristine* state. You could call `reset` at the top of `ngOnChanges` like this.

The `reset` method has an optional `state` value so you can reset the flags *and* the control values at the same time. Internally, `reset` passes the argument to `setValue`. A little refactoring and `ngOnChanges` becomes this:

{@a hero-list}

Create the *HeroListComponent* and *HeroService*

The `HeroDetailComponent` is a nested sub-component of the `HeroListComponent` in a *master/detail* view. Together they look a bit like this:



The `HeroListComponent` uses an injected `HeroService` to retrieve heroes from the server and then presents those heroes to the user as a series of buttons. The `HeroService` emulates an HTTP service. It returns an `Observable` of heroes that resolves after a short delay, both to simulate network latency and to indicate visually the necessarily asynchronous nature of the application.

When the user clicks on a hero, the component sets its `selectedHero` property which is bound to the `hero` input property of the `HeroDetailComponent`. The `HeroDetailComponent` detects the changed hero and re-sets its form with that hero's data values.

A "Refresh" button clears the hero list and the current selected hero before refetching the heroes.

The remaining `HeroListComponent` and `HeroService` implementation details are not relevant to understanding reactive forms. The techniques involved are covered elsewhere in the documentation, including the *Tour of Heroes* here and here.

If you're coding along with the steps in this reactive forms tutorial, generate the pertinent files based on the source code displayed below. Notice that `hero-list.component.ts` imports `Observable` and `finally` while `hero.service.ts` imports `Observable`, `of`, and `delay` from `rxjs`. Then return here to learn about *form array* properties.

{@a form-array}

Use *FormArray* to present an array of *FormGroups*

So far, you've seen `FormControls` and `FormGroups`. A `FormGroup` is a named object whose property values are `FormControls` and other `FormGroups`.

Sometimes you need to present an arbitrary number of controls or groups. For example, a hero may have zero, one, or any number of addresses.

The `Hero.addresses` property is an array of `Address` instances. An *address* `FormGroup` can display one `Address`. An Angular `FormArray` can display an array of *address* `FormGroups`.

To get access to the `FormArray` class, import it into `hero-detail.component.ts`:

To *work* with a `FormArray` you do the following:

1. Define the items (`FormControls` or `FormGroups`) in the array.

2. Initialize the array with items created from data in the *data model*.

3. Add and remove items as the user requires.

In this guide, you define a `FormArray` for `Hero.addresses` and let the user add or modify addresses (removing addresses is your homework).

You'll need to redefine the form model in the `HeroDetailComponent` constructor, which currently only displays the first hero address in an *address* `FormGroup`.

From *address* to *secret lairs*

From the user's point of view, heroes don't have *addresses*. *Addresses* are for mere mortals. Heroes have *secret lairs*! Replace the *address* `FormGroup` definition with a *secretLairs* `FormArray` definition:

Changing the form control name from `address` to `secretLairs` drives home an important point: the _form model_ doesn't have to match the _data model_. Obviously there has to be a relationship between the two. But it can be anything that makes sense within the application domain. _Presentation_ requirements often differ from _data_ requirements. The reactive forms approach both emphasizes and facilitates this distinction.

Initialize the "secretLairs" *FormArray*

The default form displays a nameless hero with no addresses.

You need a method to populate (or repopulate) the *secretLairs* with actual hero addresses whenever the parent

`HeroListComponent` sets the `HeroDetailComponent.hero` input property to a new `Hero`.

The following `setAddresses` method replaces the *secretLairs* `FormArray` with a new `FormArray`, initialized by an array of hero address `FormGroups`.

Notice that you replace the previous `FormArray` with the **`FormGroup.setControl` method**, not with `setValue`. You're replacing a *control*, not the *value* of a control.

Notice also that the *secretLairs* `FormArray` contains **`FormGroups`**, not `Addresses`.

Get the *FormArray*

The `HeroDetailComponent` should be able to display, add, and remove items from the *secretLairs* `FormArray`.

Use the `FormGroup.get` method to acquire a reference to that `FormArray`. Wrap the expression in a `secretLairs` convenience property for clarity and re-use.

Display the *FormArray*

The current HTML template displays a single *address* `FormGroup`. Revise it to display zero, one, or more of the hero's *address* `FormGroups`.

This is mostly a matter of wrapping the previous template HTML for an address in a `<div>` and repeating that `<div>` with `*ngFor`.

The trick lies in knowing how to write the `*ngFor`. There are three key points:

1. Add another wrapping `<div>`, around the `<div>` with `*ngFor`, and set its `formArrayName` directive to `"secretLairs"`. This step establishes the *secretLairs* `FormArray` as the context for form controls in the inner, repeated HTML template.

2. The source of the repeated items is the `FormArray.controls`, not the `FormArray` itself. Each control is an *address* `FormGroup`, exactly what the previous (now repeated) template HTML expected.

3. Each repeated `FormGroup` needs a unique `formGroupName` which must be the index of the `FormGroup` in the `FormArray`. You'll re-use that index to compose a unique label for each address.

Here's the skeleton for the *secret lairs* section of the HTML template:

Here's the complete template for the *secret lairs* section:

Add a new lair to the *FormArray*

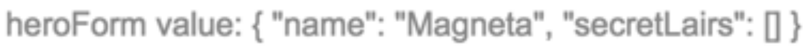
Add an `addLair` method that gets the *secretLairs* `FormArray` and appends a new *address* `FormGroup` to it.

Place a button on the form so the user can add a new *secret lair* and wire it to the component's `addLair` method.

Be sure to **add the `type="button"` attribute**. In fact, you should always specify a button's `type`. Without an explicit type, the button type defaults to "submit". When you later add a _form submit_ action, every "submit" button triggers the submit action which might do something like save the current changes. You do not want to save changes when the user clicks the _Add a Secret Lair_ button.

Try it!

Back in the browser, select the hero named "Magneta". "Magneta" doesn't have an address, as you can see in the diagnostic JSON at the bottom of the form.



```
heroForm value: { "name": "Magneta", "secretLairs": [] }
```

Click the "*Add a Secret Lair*" button. A new address section appears. Well done!

Remove a lair

This example can *add* addresses but it can't *remove* them. For extra credit, write a `removeLair` method and wire it to a button on the repeating address HTML.

{@a observe-control}

Observe control changes

Angular calls `ngOnChanges` when the user picks a hero in the parent `HeroListComponent`. Picking a hero changes the `HeroDetailComponent.hero` input property.

Angular does *not* call `ngOnChanges` when the user modifies the hero's *name* or *secret lairs*. Fortunately, you can learn about such changes by subscribing to one of the form control properties that raises a change event.

These are properties, such as `valueChanges`, that return an RxJS `Observable`. You don't need to know much about RxJS `Observable` to monitor form control values.

Add the following method to log changes to the value of the *name* `FormControl` .

Call it in the constructor, after creating the form.

The `logNameChange` method pushes name-change values into a `nameChangeLog` array. Display that array at the bottom of the component template with this `*ngFor` binding:

Return to the browser, select a hero (e.g, "Magneta"), and start typing in the *name* input box. You should see a new name in the log after each keystroke.

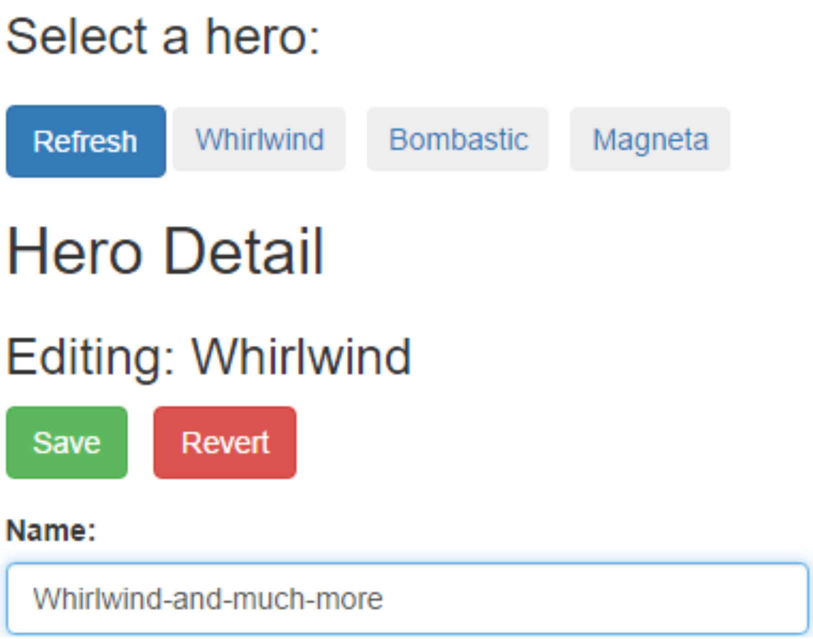
When to use it

An interpolation binding is the easier way to *display* a name change. Subscribing to an observable form control property is handy for triggering application logic *within* the component class.

{@a save}

Save form data

The `HeroDetailComponent` captures user input but it doesn't do anything with it. In a real app, you'd probably save those hero changes. In a real app, you'd also be able to revert unsaved changes and resume editing. After you implement both features in this section, the form will look like this:



Select a hero:

Hero Detail

Editing: Whirlwind

Name:

Whirlwind-and-much-more

Save

In this sample application, when the user submits the form, the `HeroDetailComponent` will pass an instance of the hero *data model* to a save method on the injected `HeroService`.

This original `hero` had the pre-save values. The user's changes are still in the *form model*. So you create a new `hero` from a combination of original hero values (the `hero.id`) and deep copies of the changed form model values, using the `prepareSaveHero` helper.

Address deep copy Had you assigned the `formModel.secretLairs` to `saveHero.addresses` (see line commented out), the addresses in `saveHero.addresses` array would be the same objects as the lairs in the `formModel.secretLairs`. A user's subsequent changes to a lair street would mutate an address street in the `saveHero`. The `prepareSaveHero` method makes copies of the form model's `secretLairs` objects so that can't happen.

Revert (cancel changes)

The user cancels changes and reverts the form to the original state by pressing the *Revert* button.

Reverting is easy. Simply re-execute the `ngOnChanges` method that built the *form model* from the original, unchanged `hero` *data model*.

Buttons

Add the "Save" and "Revert" buttons near the top of the component's template:

The buttons are disabled until the user "dirties" the form by changing a value in any of its form controls (`heroForm.dirty`).

Clicking a button of type `"submit"` triggers the `ngSubmit` event which calls the component's `onSubmit` method. Clicking the revert button triggers a call to the component's `revert` method. Users now can save or revert changes.

This is the final step in the demo. Try the .

Summary

- How to create a reactive form component and its corresponding template.
- How to use `FormBuilder` to simplify coding a reactive form.
- Grouping `FormControls`.
- Inspecting `FormControl` properties.
- Setting data with `patchValue` and `setValue`.
- Adding groups dynamically with `FormArray`.
- Observing changes to the value of a `FormControl`.

- Saving form changes.

{@a source-code}

The key files of the final version are as follows:

You can download the complete source for all steps in this guide from the Reactive Forms Demo live example.

Routing & Navigation

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.

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.

{@a basics}

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.

{@a basics-base-href}

<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.

{@a basics-router-imports}

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.

You'll learn about more options in the [details below](#browser-url-styles).

{@a basics-config}

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 four route definitions, configures the router via the `RouterModule.forRoot` method, and adds the result to the `AppModule`'s `imports` array.

{@a example-config}

The `appRoutes` array of *routes* describes how to navigate. Pass it to the `RouterModule.forRoot` method in the module `imports` to 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, *static* data. 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.

{@a basics-router-outlet}

Router outlet

Given this configuration, when the browser URL for this application becomes `/heroes`, the router matches that URL to the route path `/heroes` and displays the `HeroListComponent` *after* a `RouterOutlet` that you've placed in the host view's HTML.

<router-outlet></router-outlet> <!-- Routed views go here -->

{@a basics-router-links}

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:

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.

The **`RouterLinkActive`** directive on each anchor tag helps visually distinguish the anchor for the currently selected "active" route. The router adds the `active` CSS class to the element when the associated *RouterLink* becomes active. You can add this directive to the anchor or to its parent element.

{@a basics-router-state}

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.

{@a activated-route}

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 [resolve guard](#resolve-guard).
`paramMap`	An `Observable` that contains a [map](api/router/ParamMap) of the required and [optional parameters](#optional-route-parameters) specific to the route. The map supports retrieving single and multiple values from the same parameter.
`queryParamMap`	An `Observable` that contains a [map](api/router/ParamMap) of the [query parameters](#query-parameters) available to all routes. The map supports retrieving single and multiple values from the query parameter.
`fragment`	An `Observable` of the URL [fragment](#fragment) available to all routes.
`outlet`	The name of the `RouterOutlet` used to render the route. For an unnamed outlet, the outlet name is _primary_.
`routeConfig`	The route configuration used for the route that contains the origin path.
`parent`	The route's parent `ActivatedRoute` when this route is a [child route](#child-routing-component).
`firstChild`	Contains the first `ActivatedRoute` in the list of this route's child routes.
`children`	Contains all the [child routes](#child-routing-component) 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](#optional-route-parameters) specific to the route. Use `paramMap` instead. **`queryParams`** — An `Observable` that contains the [query parameters](#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
`NavigationStart`	An [event](api/router/NavigationStart) triggered when navigation starts.
`RoutesRecognized`	An [event](api/router/RoutesRecognized) triggered when the Router parses the URL and the routes are recognized.
`RouteConfigLoadStart`	An [event](api/router/RouteConfigLoadStart) triggered before the `Router` [lazy loads](#asynchronous-routing) a route configuration.
`RouteConfigLoadEnd`	An [event](api/router/RouteConfigLoadEnd) triggered after a route has been lazy loaded.
`NavigationEnd`	An [event](api/router/NavigationEnd) triggered when navigation ends successfully.
`NavigationCancel`	An [event](api/router/NavigationCancel) triggered when navigation is canceled. This is due to a [Route Guard](#guards) returning false during navigation.
`NavigationError`	An [event](api/router/NavigationError) triggered when navigation fails due to an unexpected error.

These events are logged to the console when the `enableTracing` option is enabled also. Since the events are provided as an `Observable`, you can `filter()` for events of interest and `subscribe()` to them to make decisions based on the sequence of events in the navigation process.

{@a basics-summary}

Summary

The application has a configured router. The shell component has a `RouterOutlet` where it can display views produced by the router. It has `RouterLink`s that users can click to navigate via the router.

Here are the key `Router` terms and their meanings:

Router Part	Meaning
`Router`	Displays the application component for the active URL. Manages navigation from one component to the next.
`RouterModule`	A separate NgModule that provides the necessary service providers and directives for navigating through application views.
`Routes`	Defines an array of Routes, each mapping a URL path to a component.
`Route`	Defines how the router should navigate to a component based on a URL pattern. Most routes consist of a path and a component type.
`RouterOutlet`	The directive (`<router-outlet>`) that marks where the router displays a view.
`RouterLink`	The directive for binding a clickable HTML element to a route. Clicking an element with a `routerLink` directive that is bound to a *string* or a *link parameters array* triggers a navigation.
`RouterLinkActive`	The directive for adding/removing classes from an HTML element when an associated `routerLink` contained on or inside the element becomes active/inactive.
`ActivatedRoute`	A service that is provided to each route component that contains route specific information such as route parameters, static data, resolve data, global query params, and the global fragment.
`RouterState`	The current state of the router including a tree of the currently activated routes together with convenience methods for traversing the route tree.
Link parameters array	An array that the router interprets as a routing instruction. You can bind that array to a `RouterLink` or pass the array as an argument to the `Router.navigate` method.
Routing component	An Angular component with a `RouterOutlet` that displays views based on router navigations.

{@a sample-app-intro}

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 .

The sample application in action

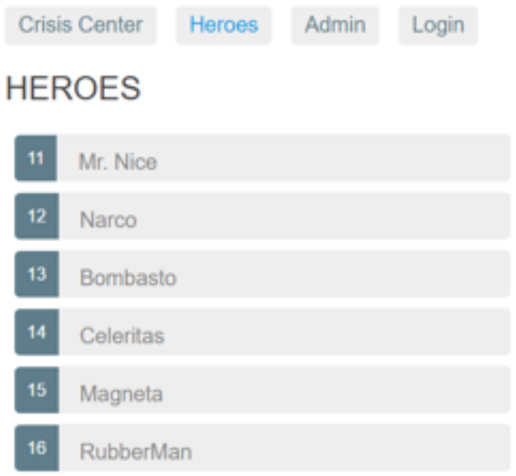
Imagine an application that helps the *Hero Employment Agency* run its business. Heroes need work and the agency finds crises for them to solve.

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.

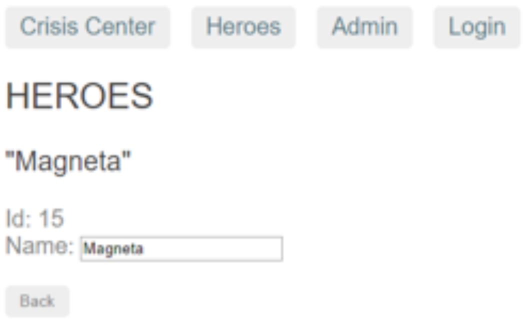
Once the app warms up, you'll see a row of navigation buttons and the *Heroes* view with its list of heroes.



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

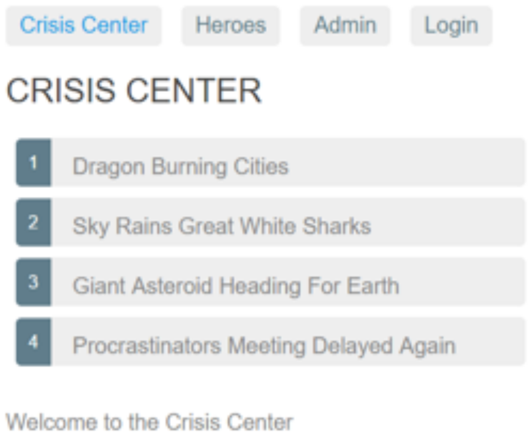
"Magneta"

Id: 15
Name: Magneta

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.



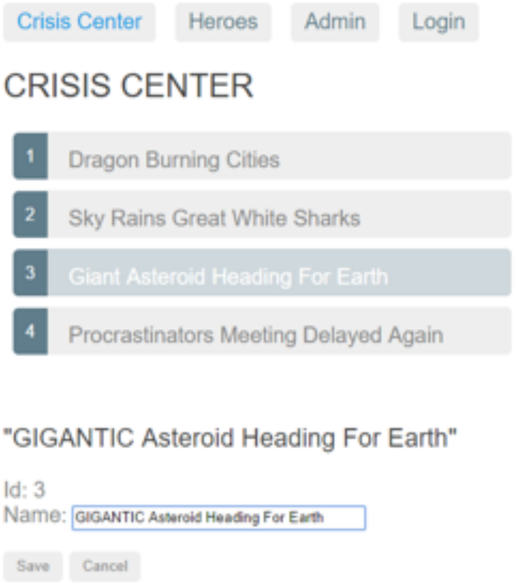
CRISIS CENTER

- 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 view 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

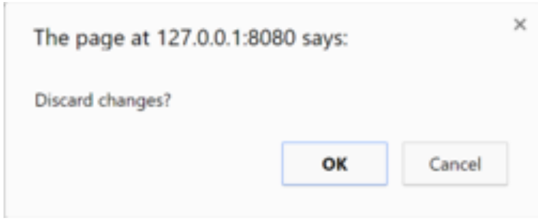
"GIGANTIC Asteroid Heading For Earth"

Id: 3
Name: GIGANTIC Asteroid Heading For Earth

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.



The page at 127.0.0.1:8080 says:

Discard changes?

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.

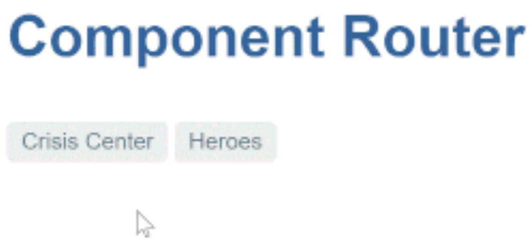
The `Admin` and `Login` buttons illustrate other router capabilities to be covered later in the guide. This short introduction will do for now.

Proceed to the first application milestone.

{@a getting-started}

Milestone 1: Getting started with the router

Begin with a simple version of the app that navigates between two empty views.



{@a base-href}

Set 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:3000/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](#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.

Live example note
A live coding environment like Plunker sets the application base address dynamically so you can't specify a fixed address. That's why the example code replaces the `` with a script that writes the `` tag on the fly. <script>document.write('<base href="' + document.location + '" />');</script> You only need this trick for the live example, not production code.

{@a import}

Importing from the router library

Begin by importing some symbols from the router library. The Router is in its own `@angular/router` package. It's not part of the Angular core. The router is an optional service because not all applications need routing and, depending on your requirements, you may need a different routing library.

You teach the router how to navigate by configuring it with routes.

{@a route-config}

Define routes

A router must be configured with a list of route definitions.

The first configuration defines an array of two routes with simple paths leading to the `CrisisListComponent` and `HeroListComponent`.

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.

Here is the first configuration. Pass the array of routes, `appRoutes`, 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.

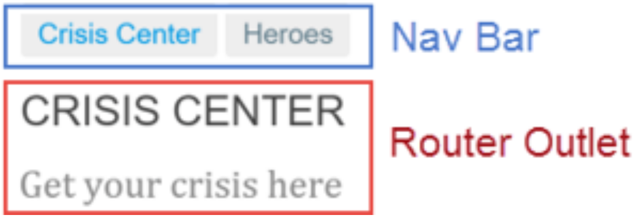
Adding the configured `RouterModule` to the `AppModule` is sufficient for simple route configurations. As the application grows, you'll want to refactor the routing configuration into a separate file and create a **[Routing Module](#routing-module)**, a special type of `Service Module` dedicated to the purpose of routing in feature modules.

Providing the `RouterModule` in the `AppModule` makes the Router available everywhere in the application.

{@a shell}

The *AppComponent* shell

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 views on and off the page. Here's what you get:



{@a shell-template}

The corresponding component template looks like this:

{@a router-outlet}

RouterOutlet

The `RouterOutlet` is a directive from the router library that marks the spot in the template where the router should display the views for that outlet.

The router adds the `` element to the DOM and subsequently inserts the navigated view element immediately _after_ the ``.

{@a router-link}

RouterLink binding

Above the outlet, within the anchor tags, you see [attribute bindings](#) to the `RouterLink` directive that look like `routerLink="..."` .

The links in this example each have a string path, the path of a route that you configured earlier. There are no route parameters yet.

You can also add more contextual information to the `RouterLink` by providing query string parameters or a

URL fragment for jumping to different areas on the page. Query string parameters are provided through the `[queryParams]` binding which takes an object (e.g. `{ name: 'value' }`), while the URL fragment takes a single value bound to the `[fragment]` input binding.

Learn about the how you can also use the _link parameters array_ in the [appendix below](#link-parameters-array).

{@a router-link-active}

RouterLinkActive binding

On each anchor tag, you also see property bindings 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 can also 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.

The `RouterLinkActive` directive toggles css classes for active `RouterLink`s based on the current `RouterState`. This cascades 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.

{@a router-directives}

Router directives

`RouterLink`, `RouterLinkActive` and `RouterOutlet` are directives provided by the Angular `RouterModule` package. They are readily available for you to use in the template.

The current state of `app.component.ts` looks like this:

{@a wildcard}

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_](#example-config) 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"`.

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`.

Create the `PageNotFoundComponent` to display when users visit invalid URLs.

As with the other components, add the `PageNotFoundComponent` to the `AppModule` declarations.

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.

{@a default-route}

The *default* route to heroes

When the application launches, the initial URL in the browser bar is something like:

localhost:3000

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:3000/heroes` into the address bar.

{@a redirect}

Redirecting routes

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.

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] (http://victorsavkin.com/post/146722301646/angular-router-empty-paths-componentless-routes).

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 rest of the starter app is mundane, with little interest from a router perspective. Here are the details for readers inclined to build the sample through to this milestone.

The starter app's structure looks like this:

router-sample
src
app
app.component.ts
app.module.ts
crisis-list.component.ts
hero-list.component.ts
not-found.component.ts
main.ts
index.html
styles.css
tsconfig.json
node_modules ...
package.json

Here are the files discussed in this milestone.

{@a routing-module}

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.

{@a routing-refactor}

Refactor the routing configuration into a *routing module*

Create a file named `app-routing.module.ts` in the `/app` folder to contain the routing module.

Import the `CrisisListComponent` and the `HeroListComponent` components just like you did in the `app.module.ts`. Then move the `Router` imports and routing configuration, including `RouterModule.forRoot`, into this routing module.

Following convention, add a class name `AppRoutingModule` and export it so you can import it later in `AppModule`.

Finally, re-export the Angular `RouterModule` by adding it to the module `exports` array. By re-exporting the `RouterModule` here and importing `AppRoutingModule` in `AppModule`, 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.

Next, update the `app.module.ts` file, first importing the newly created `AppRoutingModule` from `app-routing.module.ts`, then replacing `RouterModule.forRoot` in the `imports` array with the `AppRoutingModule`.

Later in this guide you will create [multiple routing modules](#hero-routing-module) and discover that you must import those routing modules [in the correct order](#routing-module-order).

The application continues to work just the same, and you can use `AppRoutingModule` as the central place to maintain future routing configuration.

{@a why-routing-module}

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.

{@a heroes-feature}

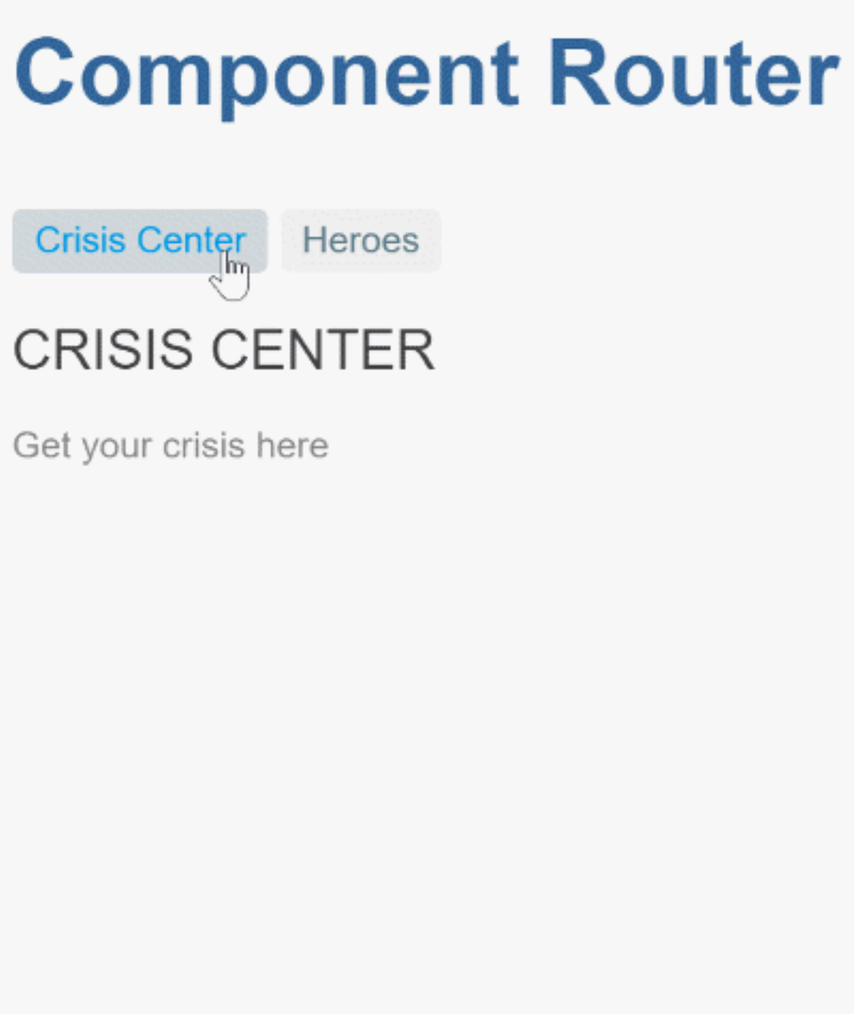
Milestone 3: Heroes feature

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 .

Here's how the user will experience this version of the app:



Component Router

CRISIS CENTER

Get your crisis here

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.

{@a heroes-functionality}

Add heroes functionality

Follow these steps:

- Create the `src/app/heroes` folder; you'll be adding files implementing *hero management* there.
- Delete the placeholder `hero-list.component.ts` that's in the `app` folder.
- Create a new `hero-list.component.ts` under `src/app/heroes`.
- Copy into it the contents of the `app.component.ts` from the "Services" tutorial.
- Make a few minor but necessary changes:

    - Delete the `selector` (routed components don't need them).
    - Delete the `<h1>`.
    - Relabel the `<h2>` to `<h2>HEROES</h2>`.
    - Delete the `<hero-detail>` at the bottom of the template.
    - Rename the `AppComponent` class to `HeroListComponent`.

- Copy the `hero-detail.component.ts` and the `hero.service.ts` files into the `heroes` subfolder.

- Create a (pre-routing) `heroes.module.ts` in the heroes folder that looks like this:

When you're done, you'll have these *hero management* files:

src/app/heroes
hero-detail.component.ts
hero-list.component.ts
hero.service.ts
heroes.module.ts

{@a hero-routing-requirements}

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.

{@a hero-routing-module}

Hero feature route configuration

Create a new `heroes-routing.module.ts` in the `heroes` folder using the same techniques you learned while creating the `AppRoutingModule`.

Put the routing module file in the same folder as its companion module file. Here both `heroes-routing.module.ts` and `heroes.module.ts` are in the same `src/app/heroes` folder. 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.

Import the hero components from their new locations in the `src/app/heroes/` folder, define the two hero routes, and export the `HeroRoutingModule` class.

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.

{@a adding-routing-module}

Add the routing module to the *HeroesModule*

Add the `HeroRoutingModule` to the `HeroModule` just as you added `AppRoutingModule` to the `AppModule`.

Open `heroes.module.ts`. Import the `HeroRoutingModule` token from `heroes-routing.module.ts` and add it to the `imports` array of the `HeroesModule`. The finished `HeroesModule` looks like this:

{@a remove-duplicate-hero-routes}

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.

But you don't want to define the same routes twice. 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.

{@a merge-hero-routes}

Import hero module into AppModule

The heroes feature module is ready, but the application doesn't know about the `HeroesModule` yet. Open `app.module.ts` and revise it as follows.

Import the `HeroesModule` and add it to the `imports` array in the `@NgModule` metadata of the `AppModule`.

Remove the `HeroListComponent` from the `AppModule`'s `declarations` because it's now provided by the `HeroesModule`. This is important. There can be only *one* owner for a declared component. In this case, the `Heroes` module is the owner of the `Heroes` components and is making them available to components in the `AppModule` via the `HeroesModule`.

As a result, the `AppModule` no longer has specific knowledge of the hero feature, its components, or its route details. 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:

{@a routing-module-order}

Module import order matters

Look at the module `imports` array. Notice that the `AppRoutingModule` is *last*. Most importantly, it comes *after* the `HeroesModule`.

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](#inspect-config "Inspect the router config").

{@a route-def-with-parameter}

Route definition with a parameter

Return to the `HeroesRoutingModule` and look at the route definitions again. The route to `HeroDetailComponent` has a twist.

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:3000/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.

{@a route-parameters}

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.

The router composes the destination URL from the array like this: `localhost:3000/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.

{@a activated-route}

Activated Route in action

Import the `Router`, `ActivatedRoute`, and `ParamMap` tokens from the router package.

Import the `switchMap` operator because you need it later to process the `Observable` route parameters.

{@a hero-detail-ctor}

As usual, you write a constructor that asks Angular to inject services that the component requires and reference them as private variables.

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.

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
`has(name)`	Returns \`true\` if the parameter name is in the map of parameters.
`get(name)`	Returns the parameter name value (a \`string\`) if present, or \`null\` if the parameter name is not in the map. Returns the _first_ element if the parameter value is actually an array of values.
`getAll(name)`	Returns a \`string array\` of the parameter name value if found, or an empty \`array\` if the parameter name value is not in the map. Use \`getAll\` when a single parameter could have multiple values.
`keys`	Returns a \`string array\` of all parameter names in the map.

{@a reuse}

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.

{@a snapshot}

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:

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.

{@a nav-to-list}

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`:

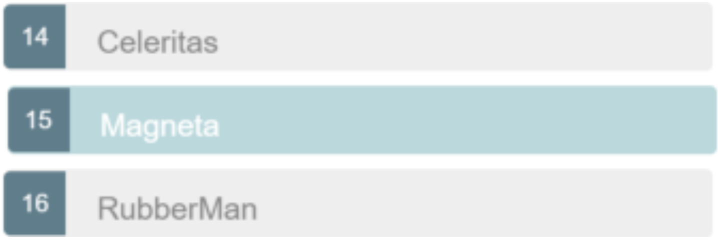
{@a optional-route-parameters}

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:3000/hero/15

You can also add *optional* information to a route request. For example, when returning to the heroes 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.

{@a optionally-selecting}

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](#link-parameters-array)*.

The router embedded the `id` value in the navigation URL because you had defined it as a route parameter with an `:id` placeholder token in the route `path` :

When the user clicks the back button, the `HeroDetailComponent` constructs another *link parameters array* which it uses to navigate back to the `HeroListComponent` .

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 the `HeroListComponent` 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:

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:3000/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](http://www.w3.org/DesignIssues/MatrixURIs.html) by the founder of the web, Tim Berners-Lee. Although matrix notation never made it into the HTML standard, it is legal and it became popular among browser routing systems as a way to isolate parameters belonging to parent and child routes. 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.

{@a route-parameters-activated-route}

Route parameters in the *ActivatedRoute* service

The list of heroes is unchanged. No hero row is highlighted.

The *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:

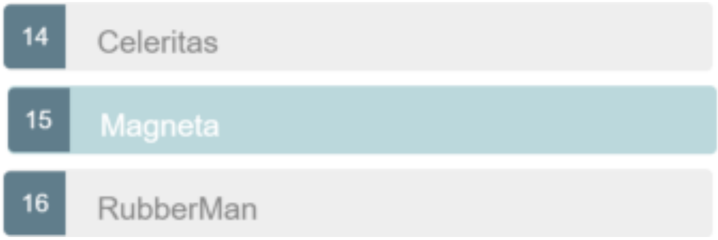
Import the `switchMap` operator to perform an operation on the `Observable` of route parameter map.

Then you inject the `ActivatedRoute` in the `HeroListComponent` constructor.

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:

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.

{@a route-animation}

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 `BrowserAnimationsModule`:

Create an `animations.ts` file in the root `src/app/` folder. The contents look like this:

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 `slideInDownAnimation` set to an animation trigger named *`routeAnimation`*; animated components will refer to this name.

- Specifies the *wildcard state* , `*`, that matches any animation state that the route component is in.

- Defines two *transitions*, one to ease the component in from the left of the screen as it enters the application view (`:enter`), the other to animate the component down as it leaves the application view (`:leave`).

You could create more triggers with different transitions for other route components. This trigger is sufficient for the current milestone.

Back in the `HeroDetailComponent`, import the `slideInDownAnimation` from `'./animations.ts`. Add the `HostBinding` decorator to the imports from `@angular/core`; you'll need it in a moment.

Add an `animations` array to the `@Component` metadata's that contains the `slideInDownAnimation`.

Then add three `@HostBinding` properties to the class to set the animation and styles for the route component's element.

The `'@routeAnimation'` passed to the first `@HostBinding` matches the name of the `slideInDownAnimation` *trigger*, `routeAnimation`. Set the `routeAnimation` property to `true` because you only care about the `:enter` and `:leave` states.

The other two `@HostBinding` properties style the display and position of the component.

The `HeroDetailComponent` will ease in from the left when routed to and will slide down when navigating away.

Applying route animations to individual components works for a simple demo, but in a real life app, it is better to animate routes based on _route paths_.

{@a milestone-3-wrap-up}

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`.
- Apply animations to the route component.

After these changes, the folder structure looks like this:

router-sample
src
app
heroes
hero-detail.component.ts
hero-list.component.ts
hero.service.ts
heroes.module.ts
heroes-routing.module.ts
app.component.ts
app.module.ts
app-routing.module.ts
crisis-list.component.ts
main.ts
index.html
styles.css
tsconfig.json
node_modules ...
package.json

Here are the relevant files for this version of the sample application.

{@a milestone-4}

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:

- Delete the placeholder crisis center file.
- Create an `app/crisis-center` folder.
- Copy the files from `app/heroes` into the new crisis center folder.
- In the new files, change every mention of "hero" to "crisis", and "heroes" to "crises".

You'll turn the `CrisisService` into a purveyor of mock crises instead of mock heroes:

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.

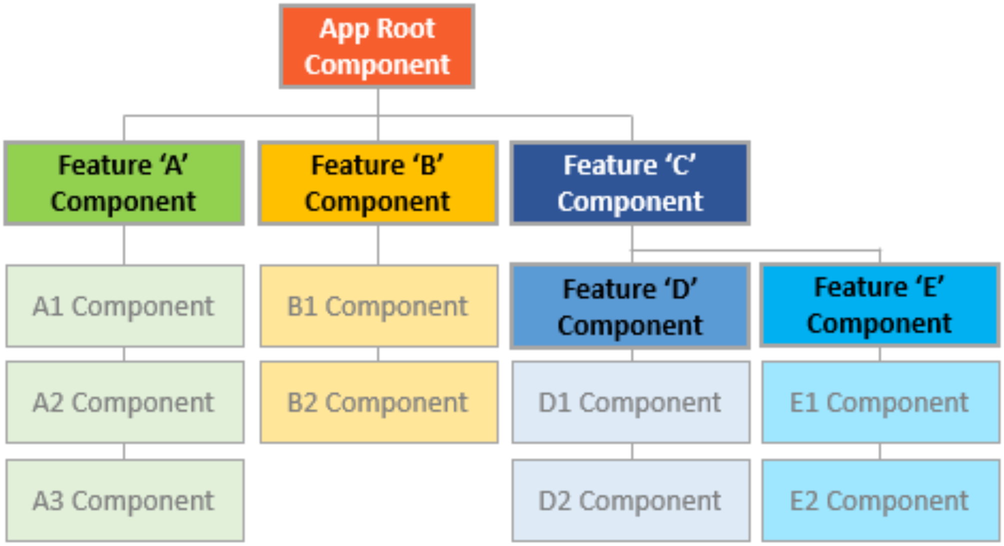
{@a crisis-child-routes}

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:



{@a child-routing-component}

Child routing component

Add the following `crisis-center.component.ts` to the `crisis-center` folder:

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 views.

Unlike `AppComponent` , and most other components, it *lacks a selector*. It doesn't *need* one since you don't *embed* this component in a parent template, instead you use the router to *navigate* to it.

{@a child-route-config}

Child route configuration

As a host page for the "Crisis Center" feature, add the following `crisis-center-home.component.ts` to the `crisis-center` folder.

Create a `crisis-center-routing.module.ts` file as you did the `heroes-routing.module.ts` file. This time, you define **child routes** *within* the parent `crisis-center` route.

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 Detail` route components.

The `Crisis Detail` route is a child of the `Crisis List`. Since the router reuses components by default, the `Crisis Detail` component will be re-used as you select different crises. In contrast, back in the `Hero Detail` route, the component was recreated 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 `CrisisCenterHomeComponent`, the full URL is `/crisis-center` (`/crisis-center` + `''` + `''`).

- To navigate to the `CrisisDetailComponent` for a crisis with `id=2`, the full URL is `/crisis-center/2` (`/crisis-center` + `''` + `'/2'`).

The absolute URL for the latter example, including the `localhost` origin, is

localhost:3000/crisis-center/2

Here's the complete `crisis-center-routing.module.ts` file with its imports.

{@a import-crisis-module}

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`:

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.

{@a relative-navigation}

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 `../` 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.

{@a nav-to-crisis}

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.

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`.

{@a named-outlets}

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.

That's where a popup will go, once you learn how to route a popup component to it.

{@a secondary-routes}

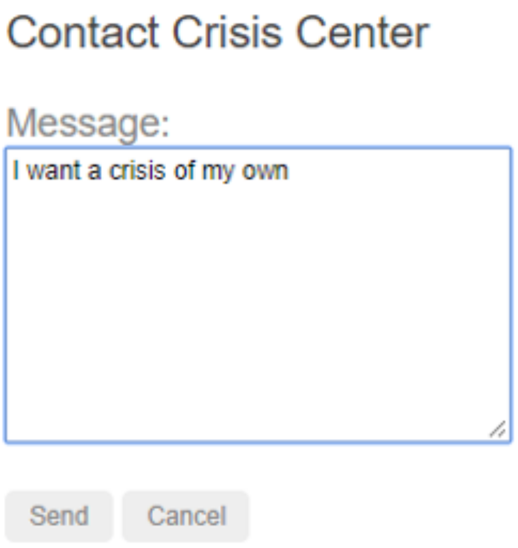
Secondary routes

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.

Create a new component named `ComposeMessageComponent` in `src/app/compose-message.component.ts`. 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

Here's the component and its template:

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.

As with other application components, you add the `ComposeMessageComponent` to the `declarations` of an `NgModule`. Do so in the `AppModule`.

{@a add-secondary-route}

Add a secondary route

Open the `AppRoutingModule` and add a new `compose` route to the `appRoutes`.

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.

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* `popup` *outlet*.

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.

{@a secondary-route-navigation}

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)

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)

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.

{@a clear-secondary-routes}

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:

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.

{@a guards}

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 can 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.

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`.

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.

{@a can-activate-guard}

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`.

Create an `admin` folder with a feature module file, a routing configuration file, and supporting components.

The admin feature file structure looks like this:

src/app/admin
admin-dashboard.component.ts
admin.component.ts
admin.module.ts
admin-routing.module.ts
manage-crises.component.ts
manage-heroes.component.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.

Since the admin dashboard \`RouterLink\` is an empty path route in the \`AdminComponent\`, 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.

The initial admin routing configuration:

{@a component-less-route}

Component-less route: grouping routes without a component

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.

Add an "Admin" link to the `AppComponent` shell so that users can get to this feature.

{@a guard-admin-feature}

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 create an `auth-guard.service.ts` in the application root folder.

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:

Next, open `admin-routing.module.ts`, import the `AuthGuard` class, and update the admin route with a `canActivate` guard property that references it:

The admin feature is now protected by the guard, albeit protected poorly.

{@a teach-auth}

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. Here's a demo `AuthService`:

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.

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.

{@a add-login-component}

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.

Register a `/login` route in the `login-routing.module.ts` and add the necessary providers to the `providers` array. In `app.module.ts`, import the `LoginComponent` and add it to the `AppModule` `declarations`. Import and add the `LoginRoutingModule` to the `AppModule` imports as well.

Guards and the service providers they require _must_ be provided at the module-level. This allows the Router access to retrieve these services from the `Injector` during the navigation process. The same rule applies for feature modules loaded [asynchronously](#asynchronous-routing).

{@a can-activate-child-guard}

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.service.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`:

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.

{@a can-deactivate-guard}

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 users 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.

{@a cancel-save}

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.

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.

The `DialogService`, provided in the `AppModule` for app-wide use, does the asking.

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`).

{@a CanDeactivate}

Create a *guard* that checks for the presence of a `canDeactivate()` method in a component—any component. 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.

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.

Looking back at the `CrisisDetailComponent`, it implements the confirmation workflow for unsaved 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.

Add the `Guard` to the main `AppRoutingModule` `providers` array so the `Router` can inject it during the navigation process.

Now you have given the user a safeguard against unsaved changes. {@a Resolve}

{@a resolve-guard}

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*.

{@a fetch-before-navigating}

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`.

Create the `crisis-detail-resolver.service.ts` file within the `Crisis Center` feature area.

Take the relevant parts of the crisis retrieval logic in `CrisisDetailComponent.ngOnInit` and move them into the `CrisisDetailResolver`. 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. Return that observable 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 `take` operator with an argument of `1` to ensure that the Observable completes after retrieving the first value from the Observable returned by the `getCrisis` method. If it doesn't return a valid `Crisis`, navigate the user back to the `CrisisListComponent`, canceling the previous in-flight navigation to the `CrisisDetailComponent`.

Import this resolver in the `crisis-center-routing.module.ts` and add a `resolve` object to the `CrisisDetailComponent` route configuration.

Remember to add the `CrisisDetailResolver` service to the `CrisisCenterRoutingModule`'s `providers` array.

The `CrisisDetailComponent` should no longer fetch the crisis. Update the `CrisisDetailComponent` to get the crisis from the `ActivatedRoute.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.

Three critical points

1. The router's `Resolve` interface is optional. The `CrisisDetailResolver` 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.

3. The Observable provided to the Router *must* complete. If the Observable does not complete, the navigation will not continue.

The relevant *Crisis Center* code for this milestone follows.

{@a query-parameters}

{@a fragment}

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.

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.

The `queryParamsHandling` feature also provides a `merge` option, which will preserve and combine the current query parameters with any provided query parameters when navigating.

Since you'll be navigating to the *Admin Dashboard* route after logging in, you'll update it to handle the query parameters and fragment.

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.

{@a asynchronous-routing}

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 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 made part 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.

{@a lazy-loading-route-config}

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 (not 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`.

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.

{@a can-load-guard}

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.service.ts`. Import the `CanLoad` interface from `@angular/router`. Add it to the `AuthGuard` class's `implements` list. Then implement `canLoad()` as follows:

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:

{@a preloading}

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*.

{@a how-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.

{@a lazy-load-crisis-center}

Lazy load the *crisis center*

Update the route configuration to lazy load the `CrisisCenterModule`. Take the same steps you used to configure `AdminModule` for lazy 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*:

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:

This tells the `Router` preloader to immediately load *all* lazy loaded routes (routes with a `loadChildren` property).

When you visit `http://localhost:3000`, 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.

{@a preload-canload}

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.

{@a custom-preloading}

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`.

Add a new file to the project called `selective-preloading-strategy.ts` and define a `SelectivePreloadingStrategy` service class as follows:

`SelectivePreloadingStrategy` 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. `SelectivePreloadingStrategy` 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 `SelectivePreloadingStrategy` into `AppRoutingModule`.
2. Replace the `PreloadAllModules` strategy in the call to `forRoot` with this `SelectivePreloadingStrategy`.
3. Add the `SelectivePreloadingStrategy` 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 `SelectivePreloadingStrategy` (it's a service).
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.

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.

{@a redirect-advanced}

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.

{@a url-refactor}

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`.

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](#query-parameters) and the [fragment](#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`.

Since `RouterLink` s aren't tied to route configuration, 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.

With the redirects setup, all previous routes now point to their new destinations and both URLs still function as intended.

{@a inspect-config}

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.

{@a final-app}

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 where you can download the final source code.

{@a appendices}

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.

{@a link-parameters-array}

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:

You've written a two element array when specifying a route parameter like this:

You can provide optional route parameters in an object like this:

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.

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*:

- 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:

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.

{@a browser-url-styles}

{@a location-strategy}

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](guide/dependency-injection#bootstrap).

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.

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`.

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.

{@a hashlocationstrategy}

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`.

Security

This page describes Angular's built-in protections against common web-application vulnerabilities and attacks such as cross-site scripting attacks. It doesn't cover application-level security, such as authentication (*Who is this user?*) and authorization (*What can this user do?*).

For more information about the attacks and mitigations described below, see OWASP Guide Project.

You can run the in Plunker and download the code from there.

Reporting vulnerabilities

To report vulnerabilities in Angular itself, email us at security@angular.io.

For more information about how Google handles security issues, see Google's security philosophy.

Best practices

- **Keep current with the latest Angular library releases.** We regularly update the Angular libraries, and these updates may fix security defects discovered in previous versions. Check the Angular change log for security-related updates.

- **Don't modify your copy of Angular.** Private, customized versions of Angular tend to fall behind the current version and may not include important security fixes and enhancements. Instead, share your Angular improvements with the community and make a pull request.

- **Avoid Angular APIs marked in the documentation as "*Security Risk.*"** For more information, see the Trusting safe values section of this page.

Preventing cross-site scripting (XSS)

Cross-site scripting (XSS) enables attackers to inject malicious code into web pages. Such code can then, for example, steal user data (in particular, login data) or perform actions to impersonate the user. This is one of the most common attacks on the web.

To block XSS attacks, you must prevent malicious code from entering the DOM (Document Object Model). For example, if attackers can trick you into inserting a `<script>` tag in the DOM, they can run arbitrary code on your website. The attack isn't limited to `<script>` tags—many elements and properties in the DOM allow code execution, for example, `<img onerror="...">` and `<a href="javascript:...">`. If attacker-controlled data enters the DOM, expect security vulnerabilities.

Angular’s cross-site scripting security model

To systematically block XSS bugs, Angular treats all values as untrusted by default. When a value is inserted into the DOM from a template, via property, attribute, style, class binding, or interpolation, Angular sanitizes and escapes untrusted values.

Angular templates are the same as executable code: HTML, attributes, and binding expressions (but not the values bound) in templates are trusted to be safe. This means that applications must prevent values that an attacker can control from ever making it into the source code of a template. Never generate template source code by concatenating user input and templates. To prevent these vulnerabilities, use the offline template compiler, also known as *template injection*.

Sanitization and security contexts

Sanitization is the inspection of an untrusted value, turning it into a value that's safe to insert into the DOM. In many cases, sanitization doesn't change a value at all. Sanitization depends on context: a value that's harmless in CSS is potentially dangerous in a URL.

Angular defines the following security contexts:

- **HTML** is used when interpreting a value as HTML, for example, when binding to `innerHtml`.
- **Style** is used when binding CSS into the `style` property.
- **URL** is used for URL properties, such as `<a href>`.
- **Resource URL** is a URL that will be loaded and executed as code, for example, in `<script src>`.

Angular sanitizes untrusted values for HTML, styles, and URLs; sanitizing resource URLs isn't possible because they contain arbitrary code. In development mode, Angular prints a console warning when it has to change a value during sanitization.

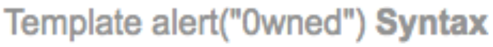
Sanitization example

The following template binds the value of `htmlSnippet`, once by interpolating it into an element's content, and once by binding it to the `innerHTML` property of an element:

Interpolated content is always escaped—the HTML isn't interpreted and the browser displays angle brackets in the element's text content.

For the HTML to be interpreted, bind it to an HTML property such as `innerHTML`. But binding a value that an attacker might control into `innerHTML` normally causes an XSS vulnerability. For example, code contained in a `<script>` tag is executed:

Angular recognizes the value as unsafe and automatically sanitizes it, which removes the `<script>` tag but keeps safe content such as the text content of the `<script>` tag and the `<b>` element.



Template alert("0wned") **Syntax**

Avoid direct use of the DOM APIs

The built-in browser DOM APIs don't automatically protect you from security vulnerabilities. For example, `document`, the node available through `ElementRef`, and many third-party APIs contain unsafe methods. Avoid directly interacting with the DOM and instead use Angular templates where possible.

Content security policy

Content Security Policy (CSP) is a defense-in-depth technique to prevent XSS. To enable CSP, configure your web server to return an appropriate `Content-Security-Policy` HTTP header. Read more about content security policy at An Introduction to Content Security Policy on the HTML5Rocks website.

{@a offline-template-compiler}

Use the offline template compiler

The offline template compiler prevents a whole class of vulnerabilities called template injection, and greatly improves application performance. Use the offline template compiler in production deployments; don't dynamically generate templates. Angular trusts template code, so generating templates, in particular templates containing user data, circumvents Angular's built-in protections. For information about dynamically constructing forms in a safe way, see the Dynamic Forms guide page.

Server-side XSS protection

HTML constructed on the server is vulnerable to injection attacks. Injecting template code into an Angular application is the same as injecting executable code into the application: it gives the attacker full control over the application. To prevent this, use a templating language that automatically escapes values to prevent XSS vulnerabilities on the server. Don't generate Angular templates on the server side using a templating language; doing this carries a high risk of introducing template-injection vulnerabilities.

Trusting safe values

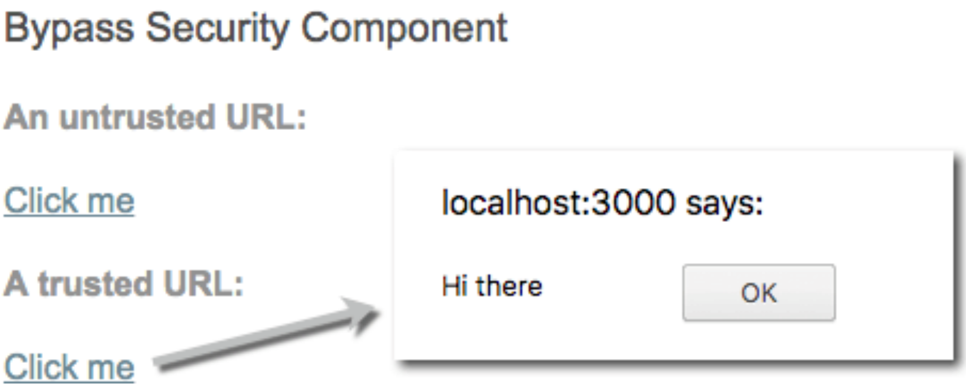
Sometimes applications genuinely need to include executable code, display an `<iframe>` from some URL, or construct potentially dangerous URLs. To prevent automatic sanitization in any of these situations, you can tell Angular that you inspected a value, checked how it was generated, and made sure it will always be secure. But *be careful*. If you trust a value that might be malicious, you are introducing a security vulnerability into your application. If in doubt, find a professional security reviewer.

To mark a value as trusted, inject `DomSanitizer` and call one of the following methods:

- `bypassSecurityTrustHtml`
- `bypassSecurityTrustScript`
- `bypassSecurityTrustStyle`
- `bypassSecurityTrustUrl`
- `bypassSecurityTrustResourceUrl`

Remember, whether a value is safe depends on context, so choose the right context for your intended use of the value. Imagine that the following template needs to bind a URL to a `javascript:alert(...)` call:

Normally, Angular automatically sanitizes the URL, disables the dangerous code, and in development mode, logs this action to the console. To prevent this, mark the URL value as a trusted URL using the `bypassSecurityTrustUrl` call:



Bypass Security Component

An untrusted URL:

Click me

A trusted URL:

Click me

localhost:3000 says:

Hi there

If you need to convert user input into a trusted value, use a controller method. The following template allows users to enter a YouTube video ID and load the corresponding video in an `<iframe>`. The `<iframe src>` attribute is a resource URL security context, because an untrusted source can, for example, smuggle in file downloads that unsuspecting users could execute. So call a method on the controller to construct a trusted video URL, which causes Angular to allow binding into `<iframe src>`:

HTTP-level vulnerabilities

Angular has built-in support to help prevent two common HTTP vulnerabilities, cross-site request forgery (CSRF or XSRF) and cross-site script inclusion (XSSI). Both of these must be mitigated primarily on the server side, but Angular provides helpers to make integration on the client side easier.

Cross-site request forgery

In a cross-site request forgery (CSRF or XSRF), an attacker tricks the user into visiting a different web page (such as `evil.com`) with malignant code that secretly sends a malicious request to the application's web server (such as `example-bank.com`).

Assume the user is logged into the application at `example-bank.com`. The user opens an email and clicks a link to `evil.com`, which opens in a new tab.

The `evil.com` page immediately sends a malicious request to `example-bank.com`. Perhaps it's a request to transfer money from the user's account to the attacker's account. The browser automatically sends the `example-bank.com` cookies (including the authentication cookie) with this request.

If the `example-bank.com` server lacks XSRF protection, it can't tell the difference between a legitimate request from the application and the forged request from `evil.com`.

To prevent this, the application must ensure that a user request originates from the real application, not from a different site. The server and client must cooperate to thwart this attack.

In a common anti-XSRF technique, the application server sends a randomly generated authentication token in a cookie. The client code reads the cookie and adds a custom request header with the token in all subsequent requests. The server compares the received cookie value to the request header value and rejects the request if the values are missing or don't match.

This technique is effective because all browsers implement the *same origin policy*. Only code from the website on which cookies are set can read the cookies from that site and set custom headers on requests to that site. That means only your application can read this cookie token and set the custom header. The malicious code on `evil.com` can't.

Angular's `HttpClient` has built-in support for the client-side half of this technique. Read about it more in the HttpClient guide.

For information about CSRF at the Open Web Application Security Project (OWASP), see Cross-Site Request Forgery (CSRF) and Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet. The Stanford University paper Robust Defenses for Cross-Site Request Forgery is a rich source of detail.

See also Dave Smith's easy-to-understand talk on XSRF at AngularConnect 2016.

Cross-site script inclusion (XSSI)

Cross-site script inclusion, also known as JSON vulnerability, can allow an attacker's website to read data from a JSON API. The attack works on older browsers by overriding native JavaScript object constructors, and then including an API URL using a `<script>` tag.

This attack is only successful if the returned JSON is executable as JavaScript. Servers can prevent an attack by prefixing all JSON responses to make them non-executable, by convention, using the well-known string `")]}',\n"`.

Angular's `HttpClient` library recognizes this convention and automatically strips the string `")]}',\n"` from all responses before further parsing.

For more information, see the XSSI section of this Google web security blog post.

Auditing Angular applications

Angular applications must follow the same security principles as regular web applications, and must be audited as such. Angular-specific APIs that should be audited in a security review, such as the *bypassSecurityTrust* methods, are marked in the documentation as security sensitive.

Communicating with service workers

Importing `ServiceWorkerModule` into your `AppModule` doesn't just register the service worker, it also provides a few services you can use to interact with the service worker and control the caching of your app.

`SwUpdate` service

The `SwUpdate` service gives you access to events that indicate when the service worker has discovered an available update for your app or when it has activated such an update—meaning it is now serving content from that update to your app.

The `SwUpdate` service supports four separate operations: * Getting notified of *available* updates. These are new versions of the app to be loaded if the page is refreshed. * Getting notified of update *activation*. This is when the service worker starts serving a new version of the app immediately. * Asking the service worker to check the server for new updates. * Asking the service worker to activate the latest version of the app for the current tab.

Available and activated updates

The two update events, `available` and `activated`, are `Observable` properties of `SwUpdate`:

You can use these events to notify the user of a pending update or to refresh their pages when the code they are running is out of date.

Checking for updates

It's possible to ask the service worker to check if any updates have been deployed to the server. You might choose to do this if you have a site that changes frequently or want updates to happen on a schedule.

Do this with the `checkForUpdate()` method:

This method returns a `Promise` which indicates that the update check has completed successfully, though it does not indicate whether an update was discovered as a result of the check. Even if one is found, the service worker must still successfully download the changed files, which can fail. If successful, the `available` event will indicate availability of a new version of the app.

Forcing update activation

If the current tab needs to be updated to the latest app version immediately, it can ask to do so with the `activateUpdate()` method:

Doing this could break lazy-loading into currently running apps, especially if the lazy-loaded chunks use filenames with hashes, which change every version.

{@a glob}

Reference: Configuration file

The `src/ngsw-config.json` configuration file specifies which files and data URLs the Angular service worker should cache and how it should update the cached files and data. The CLI processes the configuration file during `ng build --prod`. Manually, you can process it with the `ngsw-config` tool:

```
ngsw-config dist src/ngswn-config.json /base/href
```

The configuration file uses the JSON format. All file paths must begin with `/`, which is the deployment directory—usually `dist` in CLI projects.

Patterns use a limited glob format:

- `**` matches 0 or more path segments.
- `*` matches exactly one path segment or filename segment.
- The `!` prefix marks the pattern as being negative, meaning that only files that don't match the pattern will be included.

Example patterns:

- `/**/*.html` specifies all HTML files.
- `/*.html` specifies only HTML files in the root.
- `!/**/*.map` exclude all sourcemaps.

Each section of the configuration file is described below.

`appData`

This section enables you to pass any data you want that describes this particular version of the app. The `SwUpdate` service includes that data in the update notifications. Many apps use this section to provide additional information for the display of UI popups, notifying users of the available update.

`index`

Specifies the file that serves as the index page to satisfy navigation requests. Usually this is `/index.html`.

`assetGroups`

Assets are resources that are part of the app version that update along with the app. They can include resources loaded from the page's origin as well as third-party resources loaded from CDNs and other external URLs. As not all such external URLs may be known at build time, URL patterns can be matched.

This field contains an array of asset groups, each of which defines a set of asset resources and the policy by which they are cached.

```
{
  "assetGroups": [{
    ...
  }, {
    ...
  }]
}
```

Each asset group specifies both a group of resources and a policy that governs them. This policy determines when the resources are fetched and what happens when changes are detected.

Asset groups follow the Typescript interface shown here:

```
interface AssetGroup {
  name: string;
  installMode?: 'prefetch' | 'lazy';
  updateMode?: 'prefetch' | 'lazy';
  resources: {
    files?: string[];
    versionedFiles?: string[];
    urls?: string[];
  };
}
```

`name`

A `name` is mandatory. It identifies this particular group of assets between versions of the configuration.

`installMode`

The `installMode` determines how these resources are initially cached. The `installMode` can be either of two values:

- `prefetch` tells the Angular service worker to fetch every single listed resource while it's caching the current version of the app. This is bandwidth-intensive but ensures resources are available whenever they're requested, even if the browser is currently offline.

- `lazy` does not cache any of the resources up front. Instead, the Angular service worker only caches resources for which it receives requests. This is an on-demand caching mode. Resources that are never requested will not be cached. This is useful for things like images at different resolutions, so the service worker only caches the correct assets for the particular screen and orientation.

`updateMode`

For resources already in the cache, the `updateMode` determines the caching behavior when a new version of the app is discovered. Any resources in the group that have changed since the previous version are updated in accordance with `updateMode`.

- `prefetch` tells the service worker to download and cache the changed resources immediately.

- `lazy` tells the service worker to not cache those resources. Instead, it treats them as unrequested and waits until they're requested again before updating them. An `updateMode` of `lazy` is only valid if the `installMode` is also `lazy`.

`resources`

This section describes the resources to cache, broken up into three groups.

- `files` lists patterns that match files in the distribution directory. These can be single files or glob-like patterns that match a number of files.

- `versionedFiles` is like `files` but should be used for build artifacts that already include a hash in the filename, which is used for cache busting. The Angular service worker can optimize some aspects of its operation if it can assume file contents are immutable.

- `urls` includes both URLs and URL patterns that will be matched at runtime. These resources are not fetched directly and do not have content hashes, but they will be cached according to their HTTP headers. This is most useful for CDNs such as the Google Fonts service.

`dataGroups`

Unlike asset resources, data requests are not versioned along with the app. They're cached according to manually-configured policies that are more useful for situations such as API requests and other data dependencies.

Data groups follow this Typescript interface:

```
export interface DataGroup {
  name: string;
  urls: string[];
  version?: number;
  cacheConfig: {
    maxSize: number;
    maxAge: string;
    timeout?: string;
    strategy?: 'freshness' | 'performance';
  };
}
```

`name`

Similar to `assetGroups`, every data group has a `name` which uniquely identifies it.

`urls`

A list of URL patterns. URLs that match these patterns will be cached according to this data group's policy.

`version`

Occasionally APIs change formats in a way that is not backward-compatible. A new version of the app may not be compatible with the old API format and thus may not be compatible with existing cached resources from that API.

`version` provides a mechanism to indicate that the resources being cached have been updated in a backwards-incompatible way, and that the old cache entries—those from previous versions—should be discarded.

`version` is an integer field and defaults to `0`.

`cacheConfig`

This section defines the policy by which matching requests will be cached.

`maxSize`

(required) The maximum number of entries, or responses, in the cache. Open-ended caches can grow in unbounded ways and eventually exceed storage quotas, calling for eviction.

`maxAge`

(required) The `maxAge` parameter indicates how long responses are allowed to remain in the cache before being considered invalid and evicted. `maxAge` is a duration string, using the following unit suffixes:

- `d` : days
- `h` : hours
- `m` : minutes
- `s` : seconds
- `u` : milliseconds

For example, the string `3d12h` will cache content for up to three and a half days.

`timeout`

This duration string specifies the network timeout. The network timeout is how long the Angular service worker will wait for the network to respond before using a cached response, if configured to do so.

`strategy`

The Angular service worker can use either of two caching strategies for data resources.

- `performance` , the default, optimizes for responses that are as fast as possible. If a resource exists in the cache, the cached version is used. This allows for some staleness, depending on the `maxAge` , in exchange for better performance. This is suitable for resources that don't change often; for example, user avatar images.

- `freshness` optimizes for currency of data, preferentially fetching requested data from the network. Only if the network times out, according to `timeout` , does the request fall back to the cache. This is useful for resources that change frequently; for example, account balances.

DevOps: Angular service worker in production

This page is a reference for deploying and supporting production apps that use the Angular service worker. It explains how the Angular service worker fits into the larger production environment, the service worker's behavior under various conditions, and available recourses and fail-safes.

Service worker and caching of app resources

Conceptually, you can imagine the Angular service worker as a forward cache or a CDN edge that is installed in the end user's web browser. The service worker's job is to satisfy requests made by the Angular app for resources or data from a local cache, without needing to wait for the network. Like any cache, it has rules for how content is expired and updated.

{@a versions}

App versions

In the context of an Angular service worker, a "version" is a collection of resources that represent a specific build of the Angular app. Whenever a new build of the app is deployed, the service worker treats that build as a new version of the app. This is true even if only a single file is updated. At any given time, the service worker may have multiple versions of the app in its cache and it may be serving them simultaneously. For more information, see the App tabs section below.

To preserve app integrity, the Angular service worker groups all files into a version together. The files grouped into a version usually include HTML, JS, and CSS files. Grouping of these files is essential for integrity because HTML, JS, and CSS files frequently refer to each other and depend on specific content. For example, an `index.html` file might have a `<script>` tag that references `bundle.js` and it might attempt to call a function `startApp()` from within that script. Any time this version of `index.html` is served, the corresponding `bundle.js` must be served with it. For example, assume that the `startApp()` function is renamed to `runApp()` in both files. In this scenario, it is not valid to serve the old `index.html`, which calls `startApp()`, along with the new bundle, which defines `runApp()`.

This file integrity is especially important when lazy loading modules. A JS bundle may reference many lazy chunks, and the filenames of the lazy chunks are unique to the particular build of the app. If a running app at version `X` attempts to load a lazy chunk, but the server has updated to version `X + 1` already, the lazy loading operation will fail.

The version identifier of the app is determined by the contents of all resources, and it changes if any of them change. In practice, the version is determined by the contents of the `ngsw.json` file, which includes hashes for all known content. If any of the cached files change, the file's hash will change in `ngsw.json`, causing the Angular service worker to treat the active set of files as a new version.

With the versioning behavior of the Angular service worker, an application server can ensure that the Angular app always has a consistent set of files.

Update checks

Every time the Angular service worker starts, it checks for updates to the app by looking for updates to the `ngsw.json` manifest.

Note that the service worker starts periodically throughout the usage of the app because the web browser terminates the service worker if the page is idle beyond a given timeout.

Resource integrity

One of the potential side effects of long caching is inadvertently caching an invalid resource. In a normal HTTP cache, a hard refresh or cache expiration limits the negative effects of caching an invalid file. A service worker ignores such constraints and effectively long caches the entire app. Consequently, it is essential that the service worker get the correct content.

To ensure resource integrity, the Angular service worker validates the hashes of all resources for which it has a hash. Typically for a CLI app, this is everything in the `dist` directory covered by the user's `src/ngsw-config.json` configuration.

If a particular file fails validation, the Angular service worker attempts to re-fetch the content using a "cache-busting" URL parameter to eliminate the effects of browser or intermediate caching. If that content also fails validation, the service worker considers the entire version of the app to be invalid and it stops serving the app. If necessary, the service worker enters a safe mode where requests fall back on the network, opting not to use its cache if the risk of serving invalid, broken, or outdated content is high.

Hash mismatches can occur for a variety of reasons:

- Caching layers in between the origin server and the end user could serve stale content.
- A non-atomic deployment could result in the Angular service worker having visibility of partially updated content.
- Errors during the build process could result in updated resources without `ngsw.json` being updated. The reverse could also happen resulting in an updated `ngsw.json` without updated resources.

Unhashed content

The only resources that have hashes in the `ngsw.json` manifest are resources that were present in the `dist` directory at the time the manifest was built. Other resources, especially those loaded from CDNs, have content that is unknown at build time or are updated more frequently than the app is deployed.

If the Angular service worker does not have a hash to validate a given resource, it still caches its contents but it honors the HTTP caching headers by using a policy of "stale while revalidate." That is, when HTTP caching headers for a cached resource indicate that the resource has expired, the Angular service worker continues to serve the content and it attempts to refresh the resource in the background. This way, broken unhashed resources do not remain in the cache beyond their configured lifetimes.

{@a tabs}

App tabs

It can be problematic for an app if the version of resources it's receiving changes suddenly or without warning. See the Versions section above for a description of such issues.

The Angular service worker provides a guarantee: a running app will continue to run the same version of the app. If another instance of the app is opened in a new web browser tab, then the most current version of the app is served. As a result, that new tab can be running a different version of the app than the original tab.

It's important to note that this guarantee is **stronger** than that provided by the normal web deployment model. Without a service worker, there is no guarantee that code lazily loaded later in a running app is from the same version as the initial code for the app.

There are a few limited reasons why the Angular service worker might change the version of a running app. Some of them are error conditions:

- The current version becomes invalid due to a failed hash.
- An unrelated error causes the service worker to enter safe mode; that is, temporary deactivation.

The Angular service worker is aware of which versions are in use at any given moment and it cleans up versions when no tab is using them.

Other reasons the Angular service worker might change the version of a running app are normal events:

- The page is reloaded/refreshed.
- The page requests an update be immediately activated via the `SwUpdate` service.

Service worker updates

The Angular service worker is a small script that runs in web browsers. From time to time, the service worker will be updated with bug fixes and feature improvements.

The Angular service worker is downloaded when the app is first opened and when the app is accessed after a period of inactivity. If the service worker has changed, the service worker will be updated in the background.

Most updates to the Angular service worker are transparent to the app—the old caches are still valid and content is still served normally. However, occasionally a bugfix or feature in the Angular service worker requires the invalidation of old caches. In this case, the app will be refreshed transparently from the network.

Debugging the Angular service worker

Occasionally, it may be necessary to examine the Angular service worker in a running state to investigate issues or to ensure that it is operating as designed. Browsers provide built-in tools for debugging service workers and the Angular service worker itself includes useful debugging features.

Locating and analyzing debugging information

The Angular service worker exposes debugging information under the `ngsw/` virtual directory. Currently, the single exposed URL is `ngsw/state`. Here is an example of this debug page's contents:

```
NGSW Debug Info:

Driver state: NORMAL ((nominal))
Latest manifest hash: eea7f5f464f90789b621170af5a569d6be077e5c
Last update check: never

=== Version eea7f5f464f90789b621170af5a569d6be077e5c ===

Clients: 7b79a015-69af-4d3d-9ae6-95ba90c79486, 5bc08295-aaf2-42f3-a4cc-9e4ef9100f65

=== Idle Task Queue ===
Last update tick: 1s496u
Last update run: never
Task queue:
 * init post-load (update, cleanup)

Debug log:
```

Driver state

The first line indicates the driver state:

```
Driver state: NORMAL ((nominal))
```

`NORMAL` indicates that the service worker is operating normally and is not in a degraded state.

There are two possible degraded states:

- `EXISTING_CLIENTS_ONLY` : the service worker does not have a clean copy of the latest known version of the app. Older cached versions are safe to use, so existing tabs continue to run from cache, but new loads of the app will be served from the network.

- `SAFE_MODE` : the service worker cannot guarantee the safety of using cached data. Either an unexpected error occurred or all c ached versions are invalid. All traffic will be served from the network, running as little service worker code as possible.

In both cases, the parenthetical annotation provides the error that caused the service worker to enter the degraded state.

Latest manifest hash

```
Latest manifest hash: eea7f5f464f90789b621170af5a569d6be077e5c
```

This is the SHA1 hash of the most up-to-date version of the app that the service worker knows about.

Last update check

```
Last update check: never
```

This indicates the last time the service worker checked for a new version, or update, of the app. `never` indicates that the service worker has never checked for an update.

In this example debug file, the update check is currently scheduled, as explained the next section.

Version

```
=== Version eea7f5f464f90789b621170af5a569d6be077e5c ===

Clients: 7b79a015-69af-4d3d-9ae6-95ba90c79486, 5bc08295-aaf2-42f3-a4cc-9e4ef9100f65
```

In this example, the service worker has one version of the app cached and being used to serve two different tabs. Note that this version hash is the "latest manifest hash" listed above. Both clients are on the latest version. Each client is listed by its ID from the `Clients` API in the browser.

Idle task queue

```
=== Idle Task Queue ===
Last update tick: 1s496u
Last update run: never
Task queue:
 * init post-load (update, cleanup)
```

The Idle Task Queue is the queue of all pending tasks that happen in the background in the service worker. If there are any tasks in the queue, they are listed with a description. In this example, the service worker has one such task scheduled, a post-initialization operation involving an update check and cleanup of stale caches.

The last update tick/run counters give the time since specific events happened related to the idle queue. The "Last update run" counter shows the last time idle tasks were actually executed. "Last update tick" shows the time since the last event after which the queue might be processed.

Debug log

```
Debug log:
```

Errors that occur within the service worker will be logged here.

Developer Tools

Browsers such as Chrome provide developer tools for interacting with service workers. Such tools can be powerful when used properly, but there are a few things to keep in mind.

- When using developer tools, the service worker is kept running in the background and never restarts. For the Angular service worker, this means that update checks to the app will generally not happen.
- If you look in the Cache Storage viewer, the cache is frequently out of date. Right click the Cache Storage title and refresh the caches.

Stopping and starting the service worker in the Service Worker pane triggers a check for updates.

Fail-safe

Like any complex system, bugs or broken configurations can cause the Angular service worker to act in unforeseen ways. While its design attempts to minimize the impact of such problems, the Angular service worker contains a failsafe mechanism in case an administrator ever needs to deactivate the service worker quickly.

To deactivate the service worker, remove or rename the `ngsw-config.json` file. When the service worker's request for `ngsw.json` returns a `404`, then the service worker removes all of its caches and de-registers itself, essentially self-destructing.

Getting started

Beginning in Angular 5.0.0, you can easily enable Angular service worker support in any CLI project. This document explains how to enable Angular service worker support in new and existing projects. It then uses a simple example to show you a service worker in action, demonstrating loading and basic caching.

See the .

Adding a service worker to a new application

If you're generating a new CLI project, you can use the CLI to set up the Angular service worker as part of creating the project. To do so, add the `--service-worker` flag to the `ng new` command:

```
ng new my-project --service-worker
```

The `--service-worker` flag takes care of configuring your app to use service workers by adding the `service-worker` package along with setting up the necessary files to support service workers. For information on the details, see the following section which covers the process in detail as it shows you how to add a service worker manually to an existing app.

Adding a service worker to an existing app

To add a service worker to an existing app:

1. Add the service worker package.
2. Enable service worker build support in the CLI.
3. Import and register the service worker.
4. Create the service worker configuration file, which specifies the caching behaviors and other settings.
5. Build the project.

Step 1: Add the service worker package

Add the package `@angular/service-worker`, using the yarn utility as shown here:

```
yarn add @angular/service-worker
```

Step 2: Enable service worker build support in the CLI

To enable the Angular service worker, the CLI must generate an Angular service worker manifest at build time. To cause the CLI to generate the manifest for an existing project, set the `serviceWorker` flag to `true` in the project's `.angular-cli.json` file as shown here:

```
ng set apps.0.serviceWorker=true
```

Step 3: Import and register the service worker

To import and register the Angular service worker:

At the top of the root module, `src/app/app.module.ts`, import `ServiceWorkerModule` and `environment`.

Add `ServiceWorkerModule` to the `@NgModule` `imports` array. Use the `register()` helper to take care of registering the service worker, taking care to disable the service worker when not running in production mode.

The file `ngsw-worker.js` is the name of the prebuilt service worker script, which the CLI copies into `dist/` to deploy along with your server.

Step 4: Create the configuration file, `ngsw-config.json`

The Angular CLI needs a service worker configuration file, called `ngsw-config.json`. The configuration file controls how the service worker caches files and data resources.

You can begin with the boilerplate version from the CLI, which configures sensible defaults for most applications.

Alternately, save the following as `src/ngsw-config.json`:

Step 5: Build the project

Finally, build the project:

```
ng build --prod
```

The CLI project is now set up to use the Angular service worker.

Service worker in action: a tour

This section demonstrates a service worker in action, using an example application.

Serving with `http-server`

As `ng serve` does not work with service workers, you must use a real HTTP server to test your project locally. It's a good idea to test on a dedicated port.

```
cd dist
http-server -p 8080
```

Initial load

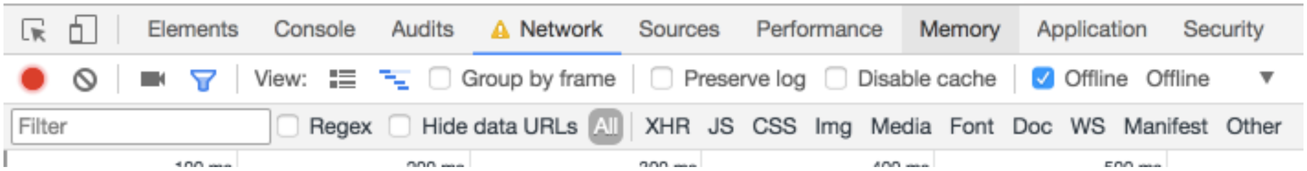
With the server running, you can point your browser at http://localhost:8080/. Your application should load normally.

Tip: When testing Angular service workers, it's a good idea to use an incognito or private window in your browser to ensure the service worker doesn't end up reading from a previous leftover state, which can cause unexpected behavior.

Simulating a network issue

To simulate a network issue, disable network interaction for your application. In Chrome:

1. Select **Tools** > **Developer Tools** (from the Chrome menu located at the top right corner).
2. Go to the **Network tab**.
3. Check the **Offline box**.

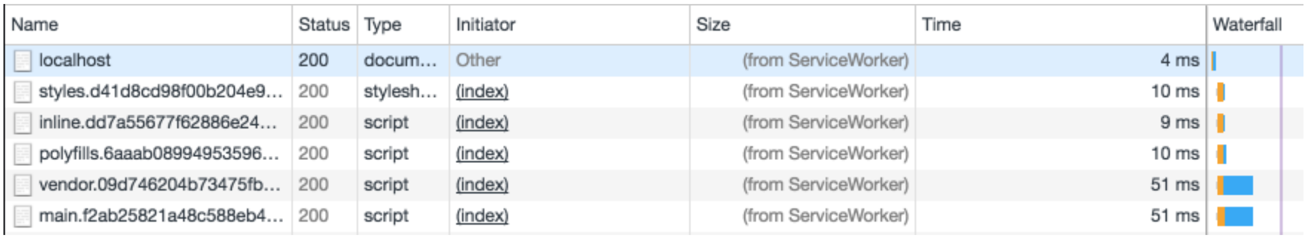

Elements Console Audits Network Sources Performance Memory Application Security
View: □ Group by frame □ Preserve log □ Disable cache ☑ Offline Offline ▼
Filter □ Regex □ Hide data URLs All XHR JS CSS Img Media Font Doc WS Manifest Other

Now the app has no access to network interaction.

For applications that do not use the Angular service worker, refreshing now would display Chrome's Internet disconnected page that says "There is no Internet connection".

With the addition of an Angular service worker, the application behavior changes. On a refresh, the page loads normally.

If you look at the Network tab, you can verify that the service worker is active.



Name	Status	Type	Initiator	Size	Time	Waterfall
localhost	200	docum…	Other	(from ServiceWorker)	4 ms	
styles.d41d8cd98f00b204e9…	200	stylesh…	(index)	(from ServiceWorker)	10 ms	
inline.dd7a55677f62886e24…	200	script	(index)	(from ServiceWorker)	9 ms	
polyfills.6aaab08994953596…	200	script	(index)	(from ServiceWorker)	10 ms	
vendor.09d746204b73475fb…	200	script	(index)	(from ServiceWorker)	51 ms	
main.f2ab25821a48c588eb4…	200	script	(index)	(from ServiceWorker)	51 ms	

Notice that under the "Size" column, the requests state is `(from ServiceWorker)`. This means that the resources are not being loaded from the network. Instead, they are being loaded from the service worker's cache.

What's being cached?

Notice that all of the files the browser needs to render this application are cached. The `ngsw-config.json` boilerplate configuration is set up to cache the specific resources used by the CLI:

- `index.html`.
- `favicon.ico`.
- Build artifacts (JS and CSS bundles).
- Anything under `assets`.

Making changes to your application

Now that you've seen how service workers cache your application, the next step is understanding how updates work.

1. If you're testing in an incognito window, open a second blank tab. This will keep the incognito and the cache state alive during your test.

2. Close the application tab, but not the window. This should also close the Developer Tools.

3. Shut down `http-server`.

4. Next, make a change to the application, and watch the service worker install the update.

5. Open `src/app/app.component.html` for editing.

6. Change the text `Welcome to {{title}}!` to `Bienvenue à {{title}}!`.

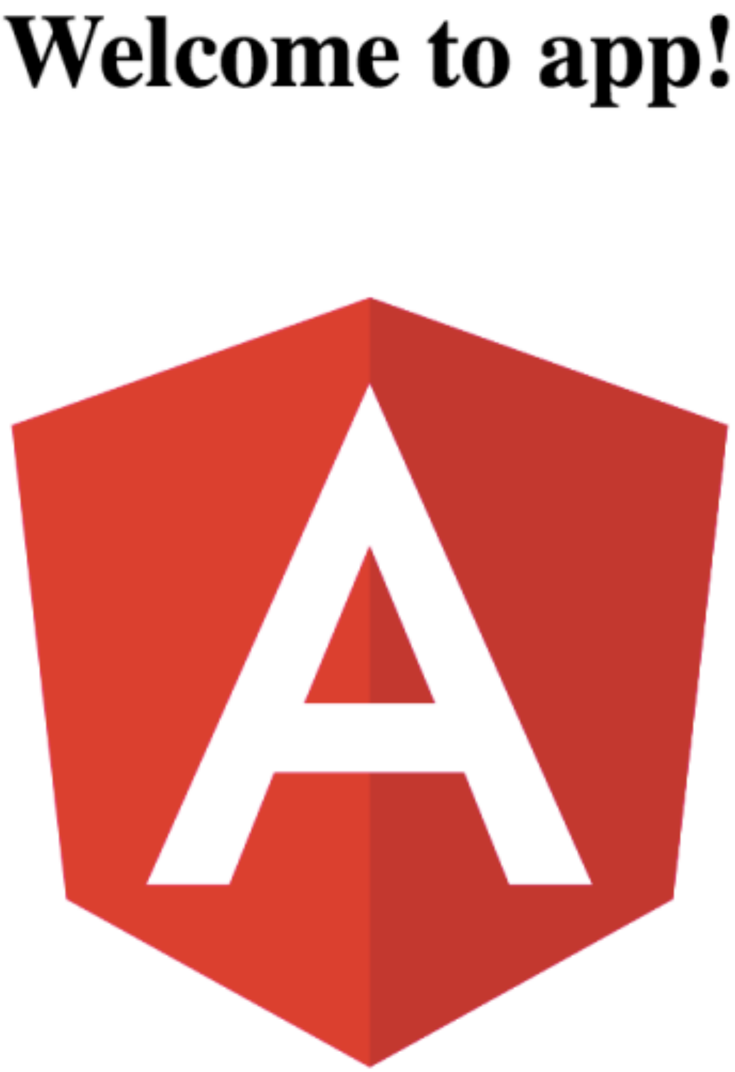
7. Build and run the server again:

```
ng build --prod
cd dist
http-server -p 8080
```

Updating your application in the browser

Now look at how the browser and service worker handle the updated application.

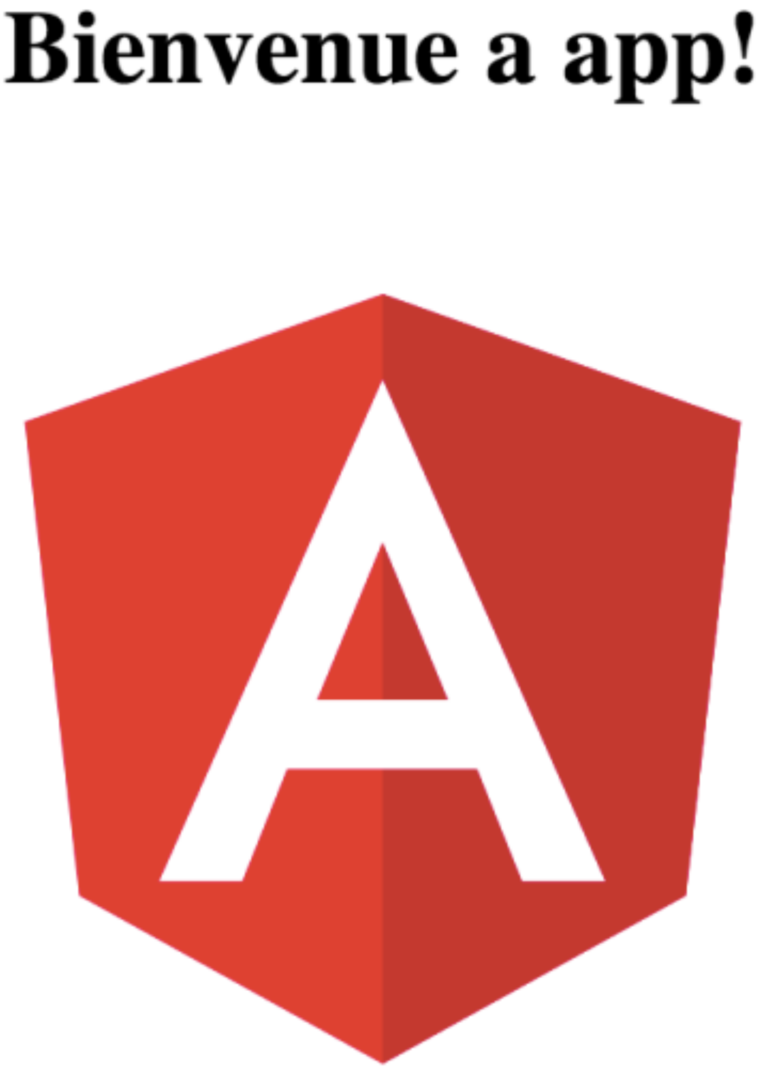
1. Open http://localhost:8080 again in the same window. What happens?



Welcome to app!

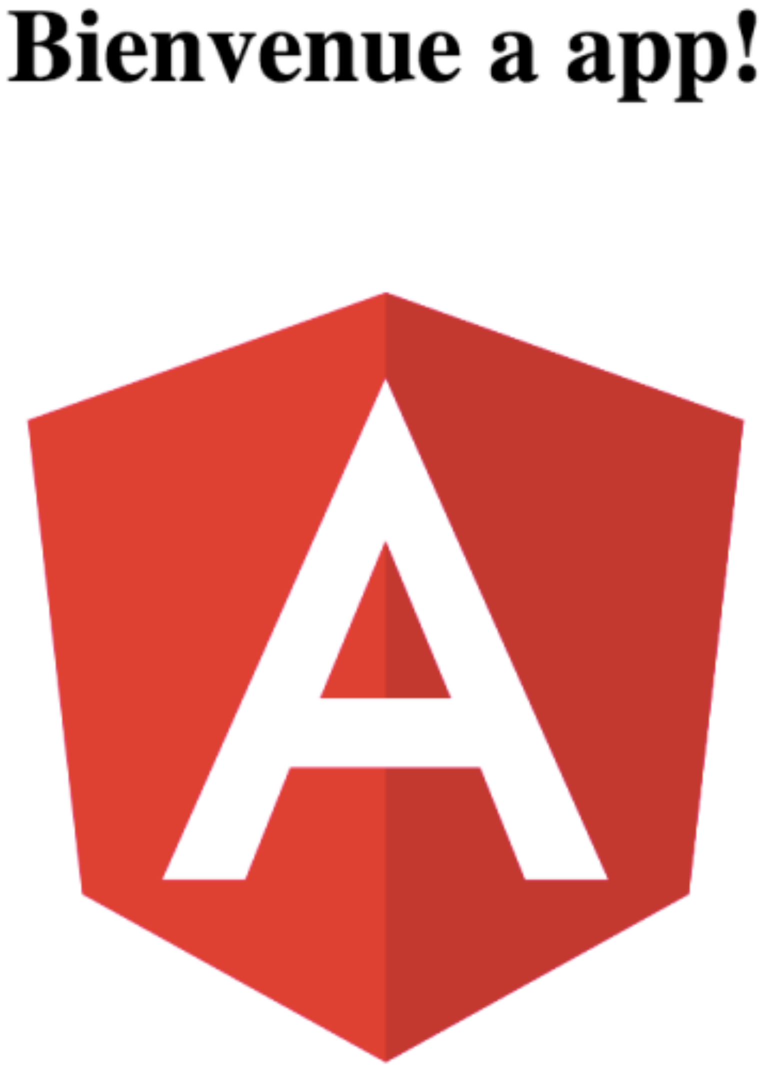
What went wrong? Nothing, actually. The Angular service worker is doing its job and serving the version of the application that it has **installed**, even though there is an update available. In the interest of speed, the service worker doesn't wait to check for updates before it serves the application that it has cached.

If you look at the `http-server` logs, you can see the service worker requesting `/ngsw.json`. This is how the service worker checks for updates.



Bienvenue a app!

1. Refresh the page.



Bienvenue a app!

The service worker installed the updated version of your app *in the background*, and the next time the page is loaded or reloaded, the service worker switches to the latest version.

Introduction to Angular service workers

Service workers augment the traditional web deployment model and empower applications to deliver a user experience with the reliability and performance on par with natively-installed code.

At its simplest, a service worker is a script that runs in the web browser and manages caching for an application.

Service workers function as a network proxy. They intercept all outgoing HTTP requests made by the application and can choose how to respond to them. For example, they can query a local cache and deliver a cached response if one is available. Proxying isn't limited to requests made through programmatic APIs, such as `fetch`; it also includes resources referenced in HTML and even the initial request to `index.html`. Service worker-based caching is thus completely programmable and doesn't rely on server-specified caching headers.

Unlike the other scripts that make up an application, such as the Angular app bundle, the service worker is preserved after the user closes the tab. The next time that browser loads the application, the service worker loads first, and can intercept every request for resources to load the application. If the service worker is designed to do so, it can *completely satisfy the loading of the application, without the need for the network*.

Even across a fast reliable network, round-trip delays can introduce significant latency when loading the application. Using a service worker to reduce dependency on the network can significantly improve the user experience.

Service workers in Angular

Angular applications, as single-page applications, are in a prime position to benefit from the advantages of service workers. Starting with version 5.0.0, Angular ships with a service worker implementation. Angular developers can take advantage of this service worker and benefit from the increased reliability and performance it provides, without needing to code against low-level APIs.

Angular's service worker is designed to optimize the end user experience of using an application over a slow or unreliable network connection, while also minimizing the risks of serving outdated content.

The Angular service worker's behavior follows that design goal:

- Caching an application is like installing a native application. The application is cached as one unit, and all files update together.

- A running application continues to run with the same version of all files. It does not suddenly start receiving cached files from a newer version, which are likely incompatible.
- When users refresh the application, they see the latest fully cached version. New tabs load the latest cached code.
- Updates happen in the background, relatively quickly after changes are published. The previous version of the application is served until an update is installed and ready.
- The service worker conserves bandwidth when possible. Resources are only downloaded if they've changed.

To support these behaviors, the Angular service worker loads a *manifest* file from the server. The manifest describes the resources to cache and includes hashes of every file's contents. When an update to the application is deployed, the contents of the manifest change, informing the service worker that a new version of the application should be downloaded and cached. This manifest is generated from a user-provided configuration file called `ngsw-config.json`, by using a build tool such as the Angular CLI.

Installing the Angular service worker is as simple as including an `NgModule`. In addition to registering the Angular service worker with the browser, this also makes a few services available for injection which interact with the service worker and can be used to control it. For example, an application can ask to be notified when a new update becomes available, or an application can ask the service worker to check the server for available updates.

Prerequisites

To use Angular service workers, you must have the following Angular and CLI versions:

- Angular 5.0.0 or later.
- Angular CLI 1.6.0 or later.

Your application must run in a web browser that supports service workers. Currently, the latest versions of Chrome and Firefox are supported. To learn about other browsers that are service worker ready, see the Can I Use page.

Related resources

For more information about service workers in general, see Service Workers: an Introduction.

For more information about browser support, see the browser support section of Service Workers: an Introduction, Jake Archibald's Is Serviceworker ready?, and Can I Use.

The remainder of this Angular documentation specifically addresses the Angular implementation of service workers.

{@a top}

Set the Document Title

Your app should be able to make the browser title bar say whatever you want it to say. This cookbook explains how to do it.

See the .

To see the browser title bar change in the live example, open it again in the Plunker editor by clicking the icon in the upper right, then pop out the preview window by clicking the blue 'X' button in the upper right corner.

The problem with *<title>*

The obvious approach is to bind a property of the component to the HTML `<title>` like this:

<title>{{This*Does*Not_Work}}</title>

Sorry but that won't work. The root component of the application is an element contained within the `<body>` tag. The HTML `<title>` is in the document `<head>`, outside the body, making it inaccessible to Angular data binding.

You could grab the browser `document` object and set the title manually. That's dirty and undermines your chances of running the app outside of a browser someday.

Running your app outside a browser means that you can take advantage of server-side pre-rendering for near-instant first app render times and for SEO. It means you could run from inside a Web Worker to improve your app's responsiveness by using multiple threads. And it means that you could run your app inside Electron.js or Windows Universal to deliver it to the desktop.

Use the `Title` service

Fortunately, Angular bridges the gap by providing a `Title` service as part of the *Browser platform*. The Title service is a simple class that provides an API for getting and setting the current HTML document title:

- `getTitle() : string` —Gets the title of the current HTML document.
- `setTitle( newTitle : string )` —Sets the title of the current HTML document.

You can inject the `Title` service into the root `AppComponent` and expose a bindable `setTitle` method that calls it:

Bind that method to three anchor tags and voilà!

Here's the complete solution:

Why provide the `Title` service in `bootstrap`

Generally you want to provide application-wide services in the root application component, `AppComponent`.

This cookbook recommends registering the title service during bootstrapping, a location you reserve for configuring the runtime Angular environment.

That's exactly what you're doing. The `Title` service is part of the Angular *browser platform*. If you bootstrap your application into a different platform, you'll have to provide a different `Title` service that understands the concept of a "document title" for that specific platform. Ideally, the application itself neither knows nor cares about the runtime environment.

Anatomy of the Setup Project

The documentation setup procedures install a *lot* of files. Most of them can be safely ignored.

Application files *inside the* `src/` and `e2e/` folders matter most to developers.

Files *outside* those folders condition the development environment. They rarely change and you may never view or modify them. If you do, this page can help you understand their purpose.

File	Purpose
`src/app/`	Angular application files go here. Ships with the "Hello Angular" sample's `AppComponent`, `AppModule`, a component unit test (`app.component.spec.ts`), and the bootstrap file, `main.ts`. Try the sample application and the unit test as _live examples_.
`e2e/`	_End-to-end_ (e2e) tests of the application, written in Jasmine and run by the protractor e2e test runner. Initialized with an e2e test for the "Hello Angular" sample.
`node_modules/`	The _npm_ packages installed with the `npm install` command.

`.editorconfig`
`.git/`
`.gitignore`
`.travis.yml`
	Tooling configuration files and folders. Ignore them until you have a compelling reason to do otherwise.

`CHANGELOG.md`	The history of changes to the _QuickStart_ repository. Delete or ignore.
`favicon.ico`	The application icon that appears in the browser tab.
`index.html`	The application host page. It loads a few essential scripts in a prescribed order. Then it boots the application, placing the root `AppComponent` in the custom `` body tag. The same `index.html` satisfies all documentation application samples.
`karma.conf.js`	Configuration for the karma test runner described in the [Testing](guide/testing) guide.
`karma-test-shim.js`	Script to run karma with SystemJS as described in the [Testing]

	(guide/testing) guide.
`non-essential-files.txt`	A list of files that you can delete if you want to purge your setup of the original QuickStart Seed testing and git maintenance artifacts. See instructions in the optional [_Deleting non-essential files_](guide/setup#non-essential "Setup: Deleting non-essential files") section. *Do this only in the beginning to avoid accidentally deleting your own tests and git setup!*
`LICENSE`	The open source MIT license to use this setup code in your application.
`package.json`	Identifies `npm `package dependencies for the project. Contains command scripts for running the application, running tests, and more. Enter `npm run` for a listing. Read more about them.
`protractor.config.js`	Configuration for the protractor _end-to-end_ (e2e) test runner.
`README.md`	Instruction for using this git repository in your project. Worth reading before deleting.
`styles.css`	Global styles for the application. Initialized with an ` **` style for the QuickStart demo.**
`systemjs.config.js`	Tells the **SystemJS** module loader where to find modules referenced in JavaScript `import` statements. For example: import { Component } from '@angular/core; Don't touch this file unless you are fully versed in SystemJS configuration.
`systemjs.config.extras.js`	Optional extra SystemJS configuration. A way to add SystemJS mappings, such as for application _barrels_, without changing the original `system.config.js`.
`tsconfig.json`	Tells the TypeScript compiler how to transpile TypeScript source files into JavaScript files that run in all modern browsers.
`tslint.json`	The `npm` installed TypeScript linter inspects your TypeScript code and complains when you violate one of its rules. This file defines linting rules favored by the [Angular style guide](guide/styleguide) and by the authors of the documentation.

Setup for local development

{@a develop-locally}

The QuickStart live-coding example is an Angular *playground*. It's not where you'd develop a real application. You should develop locally on your own machine ... and that's also how we think you should learn Angular.

Setting up a new project on your machine is quick and easy with the **QuickStart seed**, maintained on github.

Make sure you have node and npm installed.

{@a clone}

Clone

Perform the *clone-to-launch* steps with these terminal commands.

git clone https://github.com/angular/quickstart.git quickstart cd quickstart npm install npm start

`npm start` fails in _Bash for Windows_ in versions earlier than the Creator's Update (April 2017).

{@a download}

Download

Download the QuickStart seed and unzip it into your project folder. Then perform the remaining steps with these terminal commands.

cd quickstart npm install npm start

`npm start` fails in _Bash for Windows_ in versions earlier than the Creator's Update (April 2017).

{@a non-essential}

Delete *non-essential* files (optional)

You can quickly delete the *non-essential* files that concern testing and QuickStart repository maintenance (***including all git-related artifacts*** such as the `.git` folder and `.gitignore` !).

Do this only in the beginning to avoid accidentally deleting your own tests and git setup!

Open a terminal window in the project folder and enter the following commands for your environment:

OS/X (bash)

xargs rm -rf < non-essential-files.osx.txt rm src/app/.*spec*.ts rm non-essential-files.osx.txt

Windows

for /f %i in (non-essential-files.txt) do del %i /F /S /Q rd .git /s /q rd e2e /s /q

{@a seed}

What's in the QuickStart seed?

The **QuickStart seed** contains the same application as the QuickStart playground. But its true purpose is to provide a solid foundation for *local* development. Consequently, there are *many more files* in the project folder on your machine, most of which you can learn about later.

{@a app-files}

Focus on the following three TypeScript (`.ts`) files in the **`/src`** folder.

src
app
app.component.ts
app.module.ts
main.ts

All guides and cookbooks have *at least these core files*. Each file has a distinct purpose and evolves independently as the application grows.

Files outside `src/` concern building, deploying, and testing your app. They include configuration files and external dependencies.

Files inside `src/` "belong" to your app. Add new Typescript, HTML and CSS files inside the `src/` directory, most of them inside `src/app`, unless told to do otherwise.

The following are all in `src/`

File	Purpose
`app/app.component.ts`	Defines the same `AppComponent` as the one in the QuickStart playground. It is the **root** component of what will become a tree of nested components as the application evolves.
`app/app.module.ts`	Defines `AppModule`, the [root module](guide/bootstrapping "AppModule: the root module") that tells Angular how to assemble the application. Right now it declares only the `AppComponent`. Soon there will be more components to declare.
`main.ts`	Compiles the application with the [JIT compiler](guide/glossary#jit) and [bootstraps](guide/bootstrapping#main "bootstrap the application") the application's main module (`AppModule`) to run in the browser. The JIT compiler is a reasonable choice during the development of most projects and it's the only viable choice for a sample running in a _live-coding_ environment like Plunker. You'll learn about alternative compiling and [deployment](guide/deployment) options later in the documentation.

### Next Step If you're new to Angular, we recommend you follow the [tutorial](tutorial "Tour of Heroes tutorial").

{@a install-prerequisites}

Appendix: node and npm

Node.js and npm are essential to modern web development with Angular and other platforms. Node powers client development and build tools. The *npm* package manager, itself a *node* application, installs JavaScript libraries.

Get them now if they're not already installed on your machine.

Verify that you are running node `v4.x.x` or higher and npm `3.x.x` or higher by running the commands `node -v` and `npm -v` in a terminal/console window. Older versions produce errors.

We recommend nvm for managing multiple versions of node and npm. You may need nvm if you already have projects running on your machine that use other versions of node and npm.

{@a why-locally}

Appendix: Why develop locally

Live coding in the browser is a great way to explore Angular.

Links on almost every documentation page open completed samples in the browser. You can play with the sample code, share your changes with friends, and download and run the code on your own machine.

The QuickStart shows just the `AppComponent` file. It creates the equivalent of `app.module.ts` and `main.ts` internally *for the playground only*. so the reader can discover Angular without distraction. The other samples are based on the QuickStart seed.

As much fun as this is ...

- you can't ship your app in plunker
- you aren't always online when writing code
- transpiling TypeScript in the browser is slow
- the type support, refactoring, and code completion only work in your local IDE

Use the live coding environment as a *playground*, a place to try the documentation samples and experiment on your own. It's the perfect place to reproduce a bug when you want to file a documentation issue or file an issue with Angular itself.

For real development, we strongly recommend developing locally.

Structural Directives

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 .

{@a definition}

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.

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 descendents. 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:

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

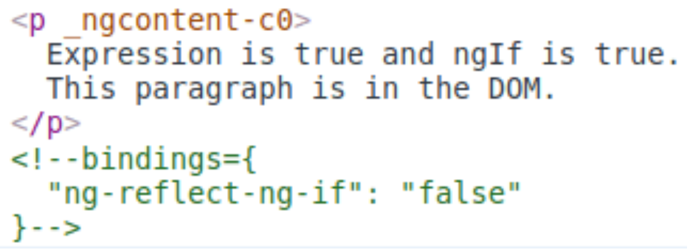
talking about its properties and what the directive does. The guide refers to the _attribute name_ when describing how you apply the directive to an element in the HTML template.
There are two other kinds of Angular directives, described extensively elsewhere: (1) components and (2) attribute directives. 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](guide/attribute-directives) changes the appearance or behavior of an element, component, or another directive. For example, the built-in [`NgStyle`](guide/template-syntax#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](guide/structural-directives#one-per-element) _structural_ directive to a host element.

{@a ngIf}

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.

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"
}-->
```

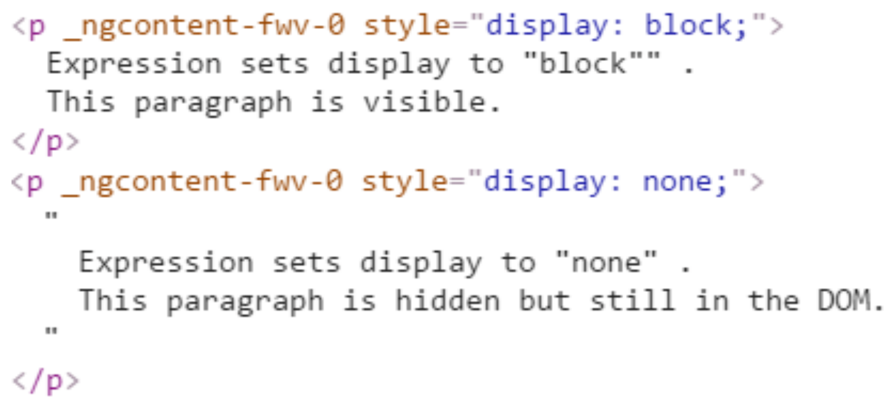
The top paragraph is in the DOM. The bottom, disused paragraph is not; in its place is a comment about "bindings" (more about that later).

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`.

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.

{@a asterisk}

The asterisk (*) prefix

Surely you noticed the asterisk (*) prefix to the directive name and wondered why it is necessary and what it does.

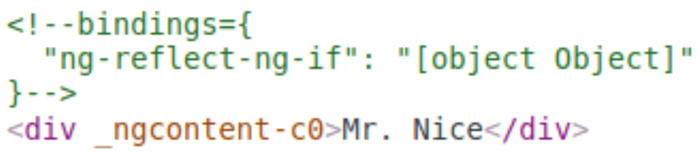
Here is `*ngIf` displaying the hero's name if `hero` exists.

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.

- 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.

{@a ngFor}

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:

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 `
`) as it moves inside the ``. In this example, the `[ngClass]="odd"` stays on the `
`.

{@a microsyntax}

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 `$implicit`.
- The `let-i` and `let-odd` variables were defined as `let i=index` and `let odd=odd`. Angular sets them to the current value of the context's `index` and `odd` properties.
- The context property for `let-hero` wasn't specified. It's intended source is implicit. Angular sets `let-hero` to the value of the context's `$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.

{@a template-input-variable}

{@a template-input-variables}

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`.

{@a one-per-element}

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 descendents. 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.

{@a ngSwitch}

Inside *NgSwitch* directives

The Angular *NgSwitch* is actually a set of cooperating directives: `NgSwitch`, `NgSwitchCase`, and `NgSwitchDefault`.

Here's an example.

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 `` is the host element for the happy ``*ngSwitchCase`. The `` 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.

{@a prefer-asterisk}

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.

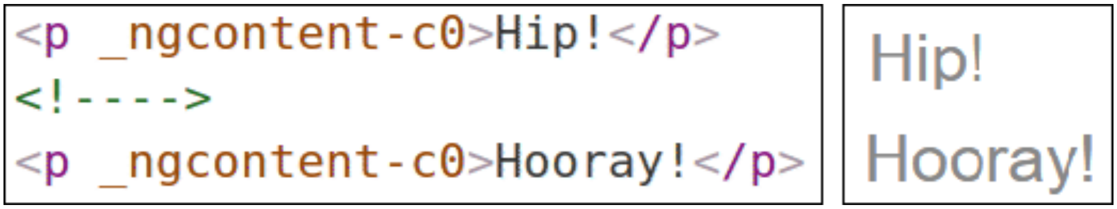
{@a template}

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!".

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.

{@a ngcontainer}

{@a ng-container}

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.

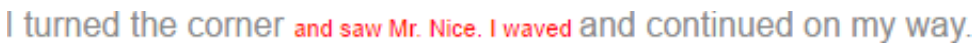
When there isn't a host element, you can usually wrap the content in a native HTML container element, such as a `<div>` , and attach the directive to that wrapper.

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.

You also have a CSS style rule that happens to apply to a `<span>` within a `<p>` aragraph.

The constructed paragraph renders strangely.



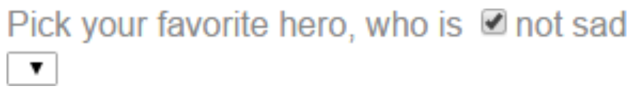
I turned the corner and saw Mr. Nice. I waved and continued on my way.

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,

the drop down is empty.



Pick your favorite hero, who is ☑ not sad

▼

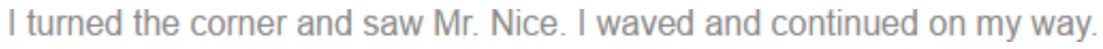
The browser won't display an `<option>` within a `<span>` .

<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*.

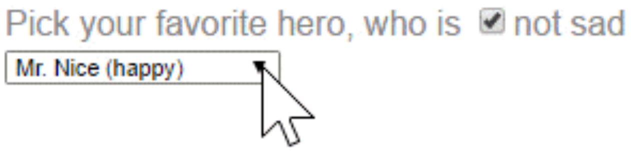
Here's the conditional paragraph again, this time using `<ng-container>` .

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>`.

The drop down works properly.


Pick your favorite hero, who is ☑ not sad

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.

{@a unless}

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***.

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:

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.

The *appUnless* property

The directive consumer expects to bind a true/false condition to `[appUnless]`. That means the directive needs an `appUnless` property, decorated with `@Input`

Read about `@Input` in the [_Template Syntax_](guide/template-syntax#inputs-outputs) guide.

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.

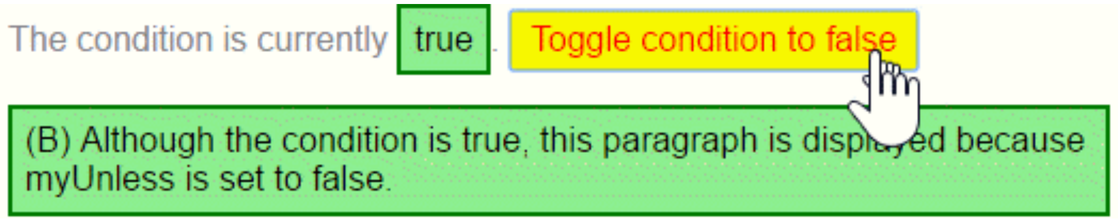
Nobody reads the `appUnless` property so it doesn't need a getter.

The completed directive code looks like this:

Add this directive to the `declarations` array of the AppModule.

Then create some HTML to try it.

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.



The condition is currently true .

(B) Although the condition is true, this paragraph is displayed because myUnless is set to false.

{@a summary}

Summary

You can both try and download the source code for this guide in the .

Here is the source from the `src/app/` folder.

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` .

Style Guide

Looking for an opinionated guide to Angular syntax, conventions, and application structure? Step right in! This style guide presents preferred conventions and, as importantly, explains why.

{@a toc}

Style vocabulary

Each guideline describes either a good or bad practice, and all have a consistent presentation.

The wording of each guideline indicates how strong the recommendation is.

Do is one that should always be followed. _Always_ might be a bit too strong of a word. Guidelines that literally should always be followed are extremely rare. On the other hand, you need a really unusual case for breaking a *Do* guideline.
Consider guidelines should generally be followed. If you fully understand the meaning behind the guideline and have a good reason to deviate, then do so. Please strive to be consistent.
Avoid indicates something you should almost never do. Code examples to *avoid* have an unmistakeable red header.
Why? gives reasons for following the previous recommendations.

File structure conventions

Some code examples display a file that has one or more similarly named companion files. For example, `hero.component.ts` and `hero.component.html`.

The guideline uses the shortcut `hero.component.ts|html|css|spec` to represent those various files. Using this shortcut makes this guide's file structures easier to read and more terse.

{@a single-responsibility}

Single responsibility

Apply the *single responsibility principle* (SRP) to all components, services, and other symbols. This helps make the app cleaner, easier to read and maintain, and more testable.

{@a 01-01}

Rule of One

Style 01-01

Do define one thing, such as a service or component, per file.
Consider limiting files to 400 lines of code.
Why? One component per file makes it far easier to read, maintain, and avoid collisions with teams in source control.
Why? One component per file avoids hidden bugs that often arise when combining components in a file where they may share variables, create unwanted closures, or unwanted coupling with dependencies.
Why? A single component can be the default export for its file which facilitates lazy loading with the router.

The key is to make the code more reusable, easier to read, and less mistake prone.

The following *negative* example defines the `AppComponent` , bootstraps the app, defines the `Hero` model object, and loads heroes from the server all in the same file. *Don't do this*.

It is a better practice to redistribute the component and its supporting classes into their own, dedicated files.

As the app grows, this rule becomes even more important. Back to top

{@a 01-02}

Small functions

Style 01-02

Do define small functions
Consider limiting to no more than 75 lines.
Why? Small functions are easier to test, especially when they do one thing and serve one purpose.
Why? Small functions promote reuse.
Why? Small functions are easier to read.
Why? Small functions are easier to maintain.
Why? Small functions help avoid hidden bugs that come with large functions that share variables with external scope, create unwanted closures, or unwanted coupling with dependencies.

Back to top

Naming

Naming conventions are hugely important to maintainability and readability. This guide recommends naming conventions for the file name and the symbol name.

{@a 02-01}

General Naming Guidelines

Style 02-01

Do use consistent names for all symbols.
Do follow a pattern that describes the symbol's feature then its type. The recommended pattern is `feature.type.ts`.
Why? Naming conventions help provide a consistent way to find content at a glance. Consistency within the project is vital. Consistency with a team is important. Consistency across a company provides tremendous efficiency.
Why? The naming conventions should simply help find desired code faster and make it easier to understand.
Why? Names of folders and files should clearly convey their intent. For example, `app/heroes/hero-list.component.ts` may contain a component that manages a list of heroes.

Back to top

{@a 02-02}

Separate file names with dots and dashes

Style 02-02

Do use dashes to separate words in the descriptive name.
Do use dots to separate the descriptive name from the type.
Do use consistent type names for all components following a pattern that describes the component's feature then its type. A recommended pattern is `feature.type.ts`.
Do use conventional type names including `.service`, `.component`, `.pipe`, `.module`, and `.directive`. Invent additional type names if you must but take care not to create too many.
Why? Type names provide a consistent way to quickly identify what is in the file.
Why? Type names make it easy to find a specific file type using an editor or IDE's fuzzy search techniques.
Why? Unabbreviated type names such as `.service` are descriptive and unambiguous. Abbreviations such as `.srv`, `.svc`, and `.serv` can be confusing.
Why? Type names provide pattern matching for any automated tasks.

Back to top

{@a 02-03}

Symbols and file names

Style 02-03

Do use consistent names for all assets named after what they represent.
Do use upper camel case for class names.
Do match the name of the symbol to the name of the file.
Do append the symbol name with the conventional suffix (such as `Component`, `Directive`, `Module`, `Pipe`, or `Service`) for a thing of that type.
Do give the filename the conventional suffix (such as `.component.ts`, `.directive.ts`, `.module.ts`, `.pipe.ts`, or `.service.ts`) for a file of that type.
Why? Consistent conventions make it easy to quickly identify and reference assets of different types.

Symbol Name	File Name
@Component({ ... }) export class AppComponent { }	app.component.ts
@Component({ ... }) export class HeroesComponent { }	heroes.component.ts
@Component({ ... }) export class HeroListComponent { }	hero-list.component.ts
@Component({ ... }) export class HeroDetailComponent { }	hero-detail.component.ts
@Directive({ ... }) export class ValidationDirective { }	validation.directive.ts
@NgModule({ ... }) export class AppModule	app.module.ts
@Pipe({ name: 'initCaps' }) export class InitCapsPipe implements PipeTransform { }	init-caps.pipe.ts
@Injectable() export class UserProfileService { }	user-profile.service.ts

Back to top

{@a 02-04}

Service names

Style 02-04

Do use consistent names for all services named after their feature.
Do suffix a service class name with `Service`. For example, something that gets data or heroes should be called a `DataService` or a `HeroService`. A few terms are unambiguously services. They typically indicate agency by ending in "-er". You may prefer to name a service that logs messages `Logger` rather than `LoggerService`. Decide if this exception is agreeable in your project. As always, strive for consistency.
Why? Provides a consistent way to quickly identify and reference services.
Why? Clear service names such as `Logger` do not require a suffix.
Why? Service names such as `Credit` are nouns and require a suffix and should be named with a suffix when it is not obvious if it is a service or something else.

Symbol Name	File Name
@Injectable() export class HeroDataService { }	hero-data.service.ts
@Injectable() export class CreditService { }	credit.service.ts
@Injectable() export class Logger { }	logger.service.ts

Back to top

{@a 02-05}

Bootstrapping

Style 02-05

Do put bootstrapping and platform logic for the app in a file named `main.ts`.
Do include error handling in the bootstrapping logic.
Avoid putting app logic in `main.ts`. Instead, consider placing it in a component or service.
Why? Follows a consistent convention for the startup logic of an app.
Why? Follows a familiar convention from other technology platforms.

Back to top

{@a 02-06}

Directive selectors

Style 02-06

Do Use lower camel case for naming the selectors of directives.
Why? Keeps the names of the properties defined in the directives that are bound to the view consistent with the attribute names.
Why? The Angular HTML parser is case sensitive and recognizes lower camel case.

Back to top

{@a 02-07}

Custom prefix for components

Style 02-07

Do use a hyphenated, lowercase element selector value (e.g. `admin-users`).
Do use a custom prefix for a component selector. For example, the prefix `toh` represents from **T**our **o**f **H**eroes and the prefix `admin` represents an admin feature area.
Do use a prefix that identifies the feature area or the app itself.
Why? Prevents element name collisions with components in other apps and with native HTML elements.
Why? Makes it easier to promote and share the component in other apps.
Why? Components are easy to identify in the DOM.

{@a 02-08}

Custom prefix for directives

Style 02-08

Do use a custom prefix for the selector of directives (e.g, the prefix `toh` from **T**our **o**f **H**eroes).
Do spell non-element selectors in lower camel case unless the selector is meant to match a native HTML attribute.
Why? Prevents name collisions.
Why? Directives are easily identified.

Back to top

{@a 02-09}

Pipe names

Style 02-09

Do use consistent names for all pipes, named after their feature.
Why? Provides a consistent way to quickly identify and reference pipes.

Symbol Name	File Name
@Pipe({ name: 'ellipsis' }) export class EllipsisPipe implements PipeTransform { }	ellipsis.pipe.ts
@Pipe({ name: 'initCaps' }) export class InitCapsPipe implements PipeTransform { }	init-caps.pipe.ts

Back to top

{@a 02-10}

Unit test file names

Style 02-10

Do name test specification files the same as the component they test.
Do name test specification files with a suffix of `.spec`.
Why? Provides a consistent way to quickly identify tests.
Why? Provides pattern matching for [karma](http://karma-runner.github.io/) or other test runners.

Test Type	File Names
Components	heroes.component.spec.ts hero-list.component.spec.ts hero-detail.component.spec.ts
Services	logger.service.spec.ts hero.service.spec.ts filter-text.service.spec.ts
Pipes	ellipsis.pipe.spec.ts init-caps.pipe.spec.ts

Back to top

{@a 02-11}

End-to-End (E2E) test file names

Style 02-11

Do name end-to-end test specification files after the feature they test with a suffix of `.e2e-spec`.
Why? Provides a consistent way to quickly identify end-to-end tests.
Why? Provides pattern matching for test runners and build automation.

Test Type	File Names
End-to-End Tests	app.e2e-spec.ts heroes.e2e-spec.ts

Back to top

{@a 02-12}

Angular *NgModule* names

Style 02-12

Do append the symbol name with the suffix `Module`.
Do give the file name the `.module.ts` extension.
Do name the module after the feature and folder it resides in.
Why? Provides a consistent way to quickly identify and reference modules.
Why? Upper camel case is conventional for identifying objects that can be instantiated using a constructor.
Why? Easily identifies the module as the root of the same named feature.
Do suffix a _RoutingModule_ class name with `RoutingModule`.
Do end the filename of a _RoutingModule_ with `-routing.module.ts`.
Why? A `RoutingModule` is a module dedicated exclusively to configuring the Angular router. A consistent class and file name convention make these modules easy to spot and verify.

Symbol Name	File Name
@NgModule({ ... }) export class AppModule { }	app.module.ts
@NgModule({ ... }) export class HeroesModule { }	heroes.module.ts
@NgModule({ ... }) export class VillainsModule { }	villains.module.ts
@NgModule({ ... }) export class AppRoutingModule { }	app-routing.module.ts
@NgModule({ ... }) export class HeroesRoutingModule { }	heroes-routing.module.ts

Back to top

Coding conventions

Have a consistent set of coding, naming, and whitespace conventions.

{@a 03-01}

Classes

Style 03-01

Do use upper camel case when naming classes.
Why? Follows conventional thinking for class names.
Why? Classes can be instantiated and construct an instance. By convention, upper camel case indicates a constructable asset.

Back to top

{@a 03-02}

Constants

Style 03-02

Do declare variables with `const` if their values should not change during the application lifetime.
Why? Conveys to readers that the value is invariant.
Why? TypeScript helps enforce that intent by requiring immediate initialization and by preventing subsequent re-assignment.
Consider spelling `const` variables in lower camel case.
Why? Lower camel case variable names (`heroRoutes`) are easier to read and understand than the traditional UPPER_SNAKE_CASE names (`HERO_ROUTES`).
Why? The tradition of naming constants in UPPER_SNAKE_CASE reflects an era before the modern IDEs that quickly reveal the `const` declaration. TypeScript prevents accidental reassignment.
Do tolerate _existing_ `const` variables that are spelled in UPPER_SNAKE_CASE.
Why? The tradition of UPPER_SNAKE_CASE remains popular and pervasive, especially in third party modules. It is rarely worth the effort to change them at the risk of breaking existing code and documentation.

Back to top

{@a 03-03}

Interfaces

Style 03-03

Do name an interface using upper camel case.
Consider naming an interface without an `I` prefix.
Consider using a class instead of an interface.
Why? TypeScript guidelines discourage the `I` prefix.
Why? A class alone is less code than a _class-plus-interface_.
Why? A class can act as an interface (use `implements` instead of `extends`).
Why? An interface-class can be a provider lookup token in Angular dependency injection.

Back to top

{@a 03-04}

Properties and methods

Style 03-04

Do use lower camel case to name properties and methods.
Avoid prefixing private properties and methods with an underscore.
Why? Follows conventional thinking for properties and methods.
Why? JavaScript lacks a true private property or method.
Why? TypeScript tooling makes it easy to identify private vs. public properties and methods.

Back to top

{@a 03-06}

Import line spacing

Style 03-06

Consider leaving one empty line between third party imports and application imports.
Consider listing import lines alphabetized by the module.
Consider listing destructured imported symbols alphabetically.
Why? The empty line separates _your_ stuff from _their_ stuff.
Why? Alphabetizing makes it easier to read and locate symbols.

Back to top

Application structure and NgModules

Have a near-term view of implementation and a long-term vision. Start small but keep in mind where the app is heading down the road.

All of the app's code goes in a folder named `src`. All feature areas are in their own folder, with their own NgModule.

All content is one asset per file. Each component, service, and pipe is in its own file. All third party vendor scripts are stored in another folder and not in the `src` folder. You didn't write them and you don't want them cluttering `src`. Use the naming conventions for files in this guide. Back to top

{@a 04-01}

LIFT

Style 04-01

Do structure the app such that you can **L**ocate code quickly, **I**dentify the code at a glance, keep the **F**lattest structure you can, and **T**ry to be DRY.
Do define the structure to follow these four basic guidelines, listed in order of importance.
Why? LIFT Provides a consistent structure that scales well, is modular, and makes it easier to increase developer efficiency by finding code quickly. To confirm your intuition about a particular structure, ask: _can I quickly open and start work in all of the related files for this feature_?

Back to top

{@a 04-02}

Locate

Style 04-02

Do make locating code intuitive, simple and fast.
Why? To work efficiently you must be able to find files quickly, especially when you do not know (or do not remember) the file _names_. Keeping related files near each other in an intuitive location saves time. A descriptive folder structure makes a world of difference to you and the people who come after you.

Back to top

{@a 04-03}

Identify

Style 04-03

Do name the file such that you instantly know what it contains and represents.
Do be descriptive with file names and keep the contents of the file to exactly one component.
Avoid files with multiple components, multiple services, or a mixture.
Why? Spend less time hunting and pecking for code, and become more efficient. Longer file names are far better than _short-but-obscure_ abbreviated names.
It may be advantageous to deviate from the _one-thing-per-file_ rule when you have a set of small, closely-related features that are better discovered and understood in a single file than as multiple files. Be wary of this loophole.

Back to top

{@a 04-04}

Flat

Style 04-04

Do keep a flat folder structure as long as possible.
Consider creating sub-folders when a folder reaches seven or more files.
Consider configuring the IDE to hide distracting, irrelevant files such as generated `.js` and `.js.map` files.
Why? No one wants to search for a file through seven levels of folders. A flat structure is easy to scan. On the other hand, psychologists believe that humans start to struggle when the number of adjacent interesting things exceeds nine. So when a folder has ten or more files, it may be time to create subfolders. Base your decision on your comfort level. Use a flatter structure until there is an obvious value to creating a new folder.

Back to top

{@a 04-05}

T-DRY (Try to be *DRY*)

Style 04-05

Do be DRY (Don't Repeat Yourself).
Avoid being so DRY that you sacrifice readability.
Why? Being DRY is important, but not crucial if it sacrifices the other elements of LIFT. That's why it's called _T-DRY_. For example, it's redundant to name a template `hero-view.component.html` because with the `.html`

extension, it is obviously a view. But if something is not obvious or departs from a convention, then spell it out.

Back to top

{@a 04-06}

Overall structural guidelines

Style 04-06

Do start small but keep in mind where the app is heading down the road.
Do have a near term view of implementation and a long term vision.
Do put all of the app's code in a folder named `src`.
Consider creating a folder for a component when it has multiple accompanying files (`.ts`, `.html`, `.css` and `.spec`).
Why? Helps keep the app structure small and easy to maintain in the early stages, while being easy to evolve as the app grows.
Why? Components often have four files (e.g. `*.html`, `*.css`, `*.ts`, and `*.spec.ts`) and can clutter a folder quickly.

{@a file-tree}

Here is a compliant folder and file structure:

<project root>

src

app

core

core.module.ts

exception.service.tslspec.ts

user-profile.service.tslspec.ts

heroes

hero

hero.component.tslhtmllcsslspec.ts

hero-list

hero-list.component.tslhtmllcsslspec.ts

shared

hero-button.component.tslhtmllcsslspec.ts

hero.model.ts

hero.service.tslspec.ts

heroes.component.tslhtmllcsslspec.ts

heroes.module.ts

heroes-routing.module.ts

shared

shared.module.ts

init-caps.pipe.tslspec.ts

text-filter.component.tslspec.ts

text-filter.service.tslspec.ts

villains

villain

...

villain-list

...

shared

...

villains.component.tslhtmllcsslspec.ts

villains.module.ts

villains-routing.module.ts

app.component.tslhtmllcsslspec.ts

app.module.ts

app-routing.module.ts

main.ts

index.html

...

node_modules/...

...

While components in dedicated folders are widely preferred, another option for small apps is to keep components flat (not in a dedicated folder). This adds up to four files to the existing folder, but also reduces the folder nesting. Whatever you choose, be consistent.

Back to top

{@a 04-07}

Folders-by-feature structure

Style 04-07

Do create folders named for the feature area they represent.

Why? A developer can locate the code and identify what each file represents at a glance. The structure is as flat as it can be and there are no repetitive or redundant names.

Why? The LIFT guidelines are all covered.
Why? Helps reduce the app from becoming cluttered through organizing the content and keeping them aligned with the LIFT guidelines.
Why? When there are a lot of files, for example 10+, locating them is easier with a consistent folder structure and more difficult in a flat structure.
Do create an NgModule for each feature area.
Why? NgModules make it easy to lazy load routable features.
Why? NgModules make it easier to isolate, test, and re-use features.
Refer to this _folder and file structure_ example.

Back to top

{@a 04-08}

App *root module*

Style 04-08

Do create an NgModule in the app's root folder, for example, in `/src/app`.
Why? Every app requires at least one root NgModule.
Consider naming the root module `app.module.ts`.
Why? Makes it easier to locate and identify the root module.

Back to top

{@a 04-09}

Feature modules

Style 04-09

Do create an NgModule for all distinct features in an application; for example, a `Heroes` feature.
Do place the feature module in the same named folder as the feature area; for example, in `app/heroes`.
Do name the feature module file reflecting the name of the feature area and folder; for example, `app/heroes/heroes.module.ts`.
Do name the feature module symbol reflecting the name of the feature area, folder, and file; for example, `app/heroes/heroes.module.ts` defines `HeroesModule`.
Why? A feature module can expose or hide its implementation from other modules.
Why? A feature module identifies distinct sets of related components that comprise the feature area.
Why? A feature module can easily be routed to both eagerly and lazily.
Why? A feature module defines clear boundaries between specific functionality and other application

features.
Why? A feature module helps clarify and make it easier to assign development responsibilities to different teams.
Why? A feature module can easily be isolated for testing.

Back to top

{@a 04-10}

Shared feature module

Style 04-10

Do create a feature module named `SharedModule` in a `shared` folder; for example, `app/shared/shared.module.ts` defines `SharedModule`.
Do declare components, directives, and pipes in a shared module when those items will be re-used and referenced by the components declared in other feature modules.
Consider using the name SharedModule when the contents of a shared module are referenced across the entire application.
Avoid providing services in shared modules. Services are usually singletons that are provided once for the entire application or in a particular feature module.
Do import all modules required by the assets in the `SharedModule`; for example, `CommonModule` and `FormsModule`.
Why? `SharedModule` will contain components, directives and pipes that may need features from another common module; for example, `ngFor` in `CommonModule`.
Do declare all components, directives, and pipes in the `SharedModule`.
Do export all symbols from the `SharedModule` that other feature modules need to use.
Why? `SharedModule` exists to make commonly used components, directives and pipes available for use in the templates of components in many other modules.
Avoid specifying app-wide singleton providers in a `SharedModule`. Intentional singletons are OK. Take care.
Why? A lazy loaded feature module that imports that shared module will make its own copy of the service and likely have undesirable results.
Why? You don't want each module to have its own separate instance of singleton services. Yet there is a real danger of that happening if the `SharedModule` provides a service.
src
app
shared
shared.module.ts
init-caps.pipe.ts|spec.ts

text-filter.component.tslspec.ts
text-filter.service.tslspec.ts
app.component.tslhtmllcsslspec.ts
app.module.ts
app-routing.module.ts
main.ts
index.html
...

Back to top

{@a 04-11}

Core feature module

Style 04-11

Consider collecting numerous, auxiliary, single-use classes inside a core module to simplify the apparent structure of a feature module.

Consider calling the application-wide core module, `CoreModule`. Importing `CoreModule` into the root `AppModule` reduces its complexity and emphasizes its role as orchestrator of the application as a whole.

Do create a feature module named `CoreModule` in a `core` folder (e.g. `app/core/core.module.ts` defines `CoreModule`).

Do put a singleton service whose instance will be shared throughout the application in the `CoreModule` (e.g. `ExceptionService` and `LoggerService`).

Do import all modules required by the assets in the `CoreModule` (e.g. `CommonModule` and `FormsModule`).

Why? `CoreModule` provides one or more singleton services. Angular registers the providers with the app root injector, making a singleton instance of each service available to any component that needs them, whether that component is eagerly or lazily loaded.

Why? `CoreModule` will contain singleton services. When a lazy loaded module imports these, it will get a new instance and not the intended app-wide singleton.

Do gather application-wide, single use components in the `CoreModule`. Import it once (in the `AppModule`) when the app starts and never import it anywhere else. (e.g. `NavComponent` and `SpinnerComponent`).

Why? Real world apps can have several single-use components (e.g., spinners, message toasts, and modal dialogs) that appear only in the `AppComponent` template. They are not imported elsewhere so they're not shared in that sense. Yet they're too big and messy to leave loose in the root folder.

Avoid importing the `CoreModule` anywhere except in the `AppModule`.

Why? A lazily loaded feature module that directly imports the `CoreModule` will make its own copy of

services and likely have undesirable results.
Why? An eagerly loaded feature module already has access to the `AppModule`'s injector, and thus the `CoreModule`'s services.
Do export all symbols from the `CoreModule` that the `AppModule` will import and make available for other feature modules to use.
Why? `CoreModule` exists to make commonly used singleton services available for use in the many other modules.
Why? You want the entire app to use the one, singleton instance. You don't want each module to have its own separate instance of singleton services. Yet there is a real danger of that happening accidentally if the `CoreModule` provides a service.
src
app
core
core.module.ts
logger.service.ts|spec.ts
nav
nav.component.ts|html|css|spec.ts
spinner
spinner.component.ts|html|css|spec.ts
spinner.service.ts|spec.ts
app.component.ts|html|css|spec.ts
app.module.ts
app-routing.module.ts
main.ts
index.html
...

`AppModule` is a little smaller because many app/root classes have moved to other modules. `AppModule` is stable because you will add future components and providers to other modules, not this one. `AppModule` delegates to imported modules rather than doing work. `AppModule` is focused on its main task, orchestrating the app as a whole.

Back to top

{@a 04-12}

Prevent re-import of the core module

Style 04-12

Only the root `AppModule` should import the `CoreModule`.

Do guard against reimporting of `CoreModule` and fail fast by adding guard logic.
Why? Guards against reimporting of the `CoreModule`.
Why? Guards against creating multiple instances of assets intended to be singletons.

Back to top

{@a 04-13}

Lazy Loaded folders

Style 04-13

A distinct application feature or workflow may be *lazy loaded* or *loaded on demand* rather than when the application starts.

Do put the contents of lazy loaded features in a *lazy loaded folder*. A typical *lazy loaded folder* contains a *routing component*, its child components, and their related assets and modules.
Why? The folder makes it easy to identify and isolate the feature content.

Back to top

{@a 04-14}

Never directly import lazy loaded folders

Style 04-14

Avoid allowing modules in sibling and parent folders to directly import a module in a *lazy loaded feature*.
Why? Directly importing and using a module will load it immediately when the intention is to load it on demand.

Back to top

Components

{@a 05-02}

Component selector names

Style 05-02

Do use _dashed-case_ or _kebab-case_ for naming the element selectors of components.
Why? Keeps the element names consistent with the specification for [Custom Elements](https://www.w3.org/TR/custom-elements/).

Back to top

{@a 05-03}

Components as elements

Style 05-03

Do give components an _element_ selector, as opposed to _attribute_ or _class_ selectors.
Why? components have templates containing HTML and optional Angular template syntax. They display content. Developers place components on the page as they would native HTML elements and web components.
Why? It is easier to recognize that a symbol is a component by looking at the template's html.

Back to top

{@a 05-04}

Extract templates and styles to their own files

Style 05-04

Do extract templates and styles into a separate file, when more than 3 lines.
Do name the template file `[component-name].component.html`, where [component-name] is the component name.
Do name the style file `[component-name].component.css`, where [component-name] is the component name.
Do specify _component-relative_ URLs, prefixed with `./`.
Why? Large, inline templates and styles obscure the component's purpose and implementation, reducing readability and maintainability.
Why? In most editors, syntax hints and code snippets aren't available when developing inline templates and styles. The Angular TypeScript Language Service (forthcoming) promises to overcome this deficiency for HTML templates in those editors that support it; it won't help with CSS styles.
Why? A _component relative_ URL requires no change when you move the component files, as long as the files stay together.

Why? The `./` prefix is standard syntax for relative URLs; don't depend on Angular's current ability to do without that prefix.

Back to top

{@a 05-12}

Decorate *input* and *output* properties

Style 05-12

Do use the `@Input()` and `@Output()` class decorators instead of the `inputs` and `outputs` properties of the `@Directive` and `@Component` metadata:
Consider placing `@Input()` or `@Output()` on the same line as the property it decorates.
Why? It is easier and more readable to identify which properties in a class are inputs or outputs.
Why? If you ever need to rename the property or event name associated with `@Input` or `@Output`, you can modify it in a single place.
Why? The metadata declaration attached to the directive is shorter and thus more readable.
Why? Placing the decorator on the same line _usually_ makes for shorter code and still easily identifies the property as an input or output. Put it on the line above when doing so is clearly more readable.

Back to top

{@a 05-13}

Avoid aliasing *inputs* and *outputs*

Style 05-13

Avoid _input_ and _output_ aliases except when it serves an important purpose.
Why? Two names for the same property (one private, one public) is inherently confusing.
Why? You should use an alias when the directive name is also an _input_ property, and the directive name doesn't describe the property.

Back to top

{@a 05-14}

Member sequence

Style 05-14

Do place properties up top followed by methods.
Do place private members after public members, alphabetized.
Why? Placing members in a consistent sequence makes it easy to read and helps instantly identify which members of the component serve which purpose.

Back to top

{@a 05-15}

Delegate complex component logic to services

Style 05-15

Do limit logic in a component to only that required for the view. All other logic should be delegated to services.
Do move reusable logic to services and keep components simple and focused on their intended purpose.
Why? Logic may be reused by multiple components when placed within a service and exposed via a function.
Why? Logic in a service can more easily be isolated in a unit test, while the calling logic in the component can be easily mocked.
Why? Removes dependencies and hides implementation details from the component.
Why? Keeps the component slim, trim, and focused.

Back to top

{@a 05-16}

Don't prefix *output* properties

Style 05-16

Do name events without the prefix `on`.
Do name event handler methods with the prefix `on` followed by the event name.
Why? This is consistent with built-in events such as button clicks.
Why? Angular allows for an [alternative syntax](guide/template-syntax#binding-syntax) `on-*`. If the event itself was prefixed with `on` this would result in an `on-onEvent` binding expression.

Back to top

{@a 05-17}

Put presentation logic in the component class

Style 05-17

Do put presentation logic in the component class, and not in the template.
Why? Logic will be contained in one place (the component class) instead of being spread in two places.
Why? Keeping the component's presentation logic in the class instead of the template improves testability, maintainability, and reusability.

Back to top

Directives

{@a 06-01}

Use directives to enhance an element

Style 06-01

Do use attribute directives when you have presentation logic without a template.
Why? Attribute directives don't have an associated template.
Why? An element may have more than one attribute directive applied.

Back to top

{@a 06-03}

HostListener/*HostBinding* decorators versus *host* metadata

Style 06-03

Consider preferring the `@HostListener` and `@HostBinding` to the `host` property of the `@Directive` and `@Component` decorators.
Do be consistent in your choice.
Why? The property associated with `@HostBinding` or the method associated with `@HostListener` can be modified only in a single place—in the directive's class. If you use the `host` metadata property, you must modify both the property/method declaration in the directive's class and the metadata in the decorator associated with the directive.

Compare with the less preferred `host` metadata alternative.

Why? The `host` metadata is only one term to remember and doesn't require extra ES imports.

Back to top

Services

{@a 07-01}

Services are singletons

Style 07-01

Do use services as singletons within the same injector. Use them for sharing data and functionality.
Why? Services are ideal for sharing methods across a feature area or an app.
Why? Services are ideal for sharing stateful in-memory data.

Back to top

{@a 07-02}

Single responsibility

Style 07-02

Do create services with a single responsibility that is encapsulated by its context.
Do create a new service once the service begins to exceed that singular purpose.
Why? When a service has multiple responsibilities, it becomes difficult to test.
Why? When a service has multiple responsibilities, every component or service that injects it now carries the weight of them all.

Back to top

{@a 07-03}

Providing a service

Style 07-03

Do provide services to the Angular injector at the top-most component where they will be shared.
Why? The Angular injector is hierarchical.
Why? When providing the service to a top level component, that instance is shared and available to all child

components of that top level component.
Why? This is ideal when a service is sharing methods or state.
Why? This is not ideal when two different components need different instances of a service. In this scenario it would be better to provide the service at the component level that needs the new and separate instance.

Back to top

{@a 07-04}

Use the @Injectable() class decorator

Style 07-04

Do use the `@Injectable()` class decorator instead of the `@Inject` parameter decorator when using types as tokens for the dependencies of a service.
Why? The Angular Dependency Injection (DI) mechanism resolves a service's own dependencies based on the declared types of that service's constructor parameters.
Why? When a service accepts only dependencies associated with type tokens, the `@Injectable()` syntax is much less verbose compared to using `@Inject()` on each individual constructor parameter.

Back to top

Data Services

{@a 08-01}

Talk to the server through a service

Style 08-01

Do refactor logic for making data operations and interacting with data to a service.
Do make data services responsible for XHR calls, local storage, stashing in memory, or any other data operations.
Why? The component's responsibility is for the presentation and gathering of information for the view. It should not care how it gets the data, just that it knows who to ask for it. Separating the data services moves the logic on how to get it to the data service, and lets the component be simpler and more focused on the view.
Why? This makes it easier to test (mock or real) the data calls when testing a component that uses a data service.
Why? The details of data management, such as headers, HTTP methods, caching, error handling, and retry logic, are irrelevant to components and other data consumers. A data service encapsulates these details. It's

easier to evolve these details inside the service without affecting its consumers. And it's easier to test the consumers with mock service implementations.

Back to top

Lifecycle hooks

Use Lifecycle hooks to tap into important events exposed by Angular.

Back to top

{@a 09-01}

Implement lifecycle hook interfaces

Style 09-01

Do implement the lifecycle hook interfaces.
Why? Lifecycle interfaces prescribe typed method signatures. use those signatures to flag spelling and syntax mistakes.

Back to top

Appendix

Useful tools and tips for Angular.

Back to top

{@a A-01}

Codelyzer

Style A-01

Do use [codelyzer](https://www.npmjs.com/package/codelyzer) to follow this guide.
Consider adjusting the rules in codelyzer to suit your needs.

Back to top

{@a A-02}

File templates and snippets

Style A-02

Do use file templates or snippets to help follow consistent styles and patterns. Here are templates and/or snippets for some of the web development editors and IDEs.
Consider using [snippets](https://marketplace.visualstudio.com/items?itemName=johnpapa.Angular2) for [Visual Studio Code](https://code.visualstudio.com/) that follow these styles and guidelines.

Consider using [snippets](https://atom.io/packages/angular-2-typescript-snippets) for [Atom](https://atom.io/) that follow these styles and guidelines. **Consider** using [snippets](https://github.com/orizens/sublime-angular2-snippets) for [Sublime Text](http://www.sublimetext.com/) that follow these styles and guidelines. **Consider** using [snippets](https://github.com/mhartington/vim-angular2-snippets) for [Vim](http://www.vim.org/) that follow these styles and guidelines.

Back to top

Template Syntax

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

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

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

Many code snippets illustrate the points and concepts, all of them available in the .

{@a html}

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.

{@a interpolation}

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

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

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

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:

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

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.

{@a template-expressions}

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 `!` .

{@a expression-context}

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` .

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`).

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 `hero` template 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.

{@a no-side-effects}

{@a expression-guidelines}

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.

{@a template-statements}

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"`.

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.

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.

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.

{@a binding-syntax}

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
One-way from data source to view target	{{expression}} [target]="expression" bind-target="expression"	Interpolation Property Attribute Class Style
One-way from view target to data source	(target)="statement" on-target="statement"	Event
Two-way	[(target)]="expression" bindon-target="expression"	Two-way

Binding types other than interpolation have a **target name** to the left of the equal sign, either surrounded by punctuation (`[]` , `()`) or 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.

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.

That's HTML Plus.

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

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 `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 ` Still Disabled `. 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
Property	Element property Component property Directive property	
Event	Element event Component event Directive event	
Two-way	Event and property	
Attribute	Attribute (the exception)	
Class	`class` property	
Style	`style` property	

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

{@a property-binding}

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 `heroImageUrl` property:

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

Another is setting a property of a directive:

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

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](guide/template-syntax#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](api/core/ViewChild) and [ContentChild](api/core/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.

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

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:

Technically, Angular is matching the name to a directive [input](guide/template-syntax#inputs-outputs), 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:

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:

{@a one-time-initialization}

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.

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

{@a property-binding-or-interpolation}

Property binding or interpolation?

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

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.

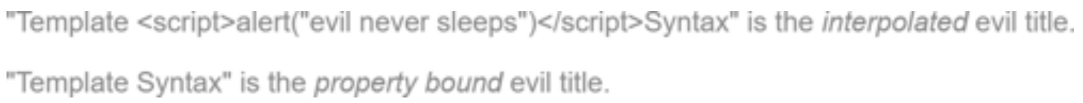
When setting an element property to a non-string data value, you must use *property binding*.

Content security

Imagine the following *malicious content*.

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.

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.

{@a other-bindings}

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:

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:

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:

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

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.

While this is a fine way to toggle a single class name, the [NgClass directive](guide/template-syntax#ngClass) 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]`.

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

While this is a fine way to set a single style, the [NgStyle directive](guide/template-syntax#ngStyle) 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](guide/glossary#dash-case), as shown above, or [camelCase](guide/glossary#camelcase), such as `fontSize`.

{@a event-binding}

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:

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.

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

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:

The `myClick` directive is further described in the section on [aliasing input/output properties](guide/template-syntax#aliasing-io).

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:

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.

{@a eventemitter}

{@a custom-event}

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` :

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.

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.

{@a two-way}

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:

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`:

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 *style* binding, making the displayed text bigger or smaller.

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

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.

{@a directives}

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?

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.

{@a attribute-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

{@a ngClass}

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.

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:

Adding an `ngClass` property binding to `currentClasses` sets the element's classes accordingly:

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

{@a ngStyle}

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.

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:

Adding an `ngStyle` property binding to `currentStyles` sets the element's styles accordingly:

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

{@a ngModel}

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:

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/forms) guide.

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

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.

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.

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](api/forms/ControlValueAccessor) that adapts an element to this protocol. The ` □ ` box is one of those elements. Angular provides *value accessors* for all of the basic HTML form elements and the [_Forms_](guide/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](guide/template-syntax#two-way) and skip `NgModel` altogether. The [`sizer` shown above](guide/template-syntax#two-way) is an example of this technique.

Separate `ngModel` bindings is an improvement over binding to the element's native properties. You can do better.

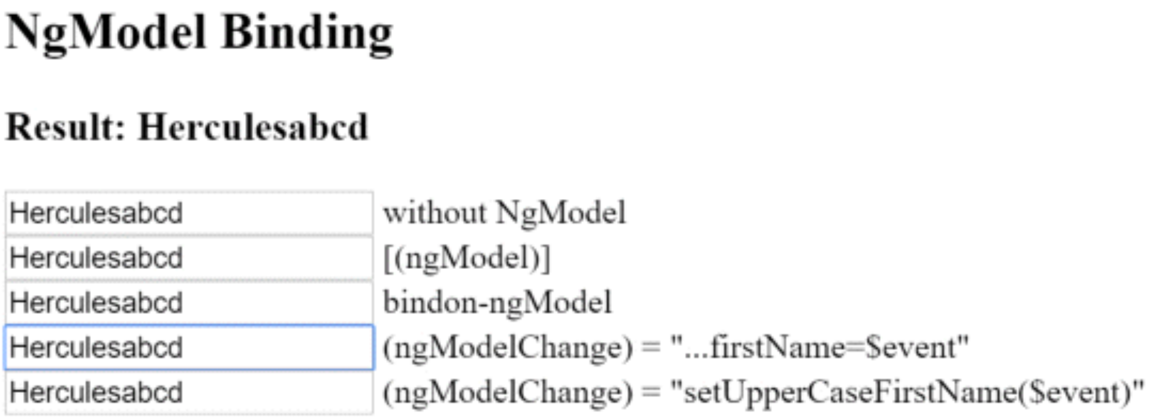
You shouldn't have to mention the data property twice. Angular should be able to capture the component's data property and set it with a single declaration, which it can with the `[(ngModel)]` syntax:

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:

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



NgModel Binding

Result: Herculesabcd

Herculesabcd without NgModel
Herculesabcd [(ngModel)]
Herculesabcd bindon-ngModel
Herculesabcd (ngModelChange) = "...firstName=$event"
Herculesabcd (ngModelChange) = "setUpperCaseFirstName($event)"

{@a structural-directives}

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

{@a ngIf}

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.

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:

Hiding an element is quite different from removing an element with `NgIf`.

When you hide an element, that element and all of its descendents 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 descendents 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.

See also the [_safe navigation operator_](guide/template-syntax#safe-navigation-operator "Safe navigation operator (?.)") described below.

{@a ngFor}

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>`:

You can also apply an `NgForOf` to a component element, as in this example:

Don't forget the asterisk (`*`) in front of `ngFor`.

The text assigned to `*ngFor` is the instruction that guides the repeater process.

{@a microsyntax}

*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.

{@a template-input-variable}

{@a template-input-variables}

Template input variables

The `let` keyword before `hero` creates a *template input variable* called `hero`. The `NgForOf` directive iterates over the `heroes` array 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.

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.

`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](api/common/NgForOf).

{@a trackBy}

*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.

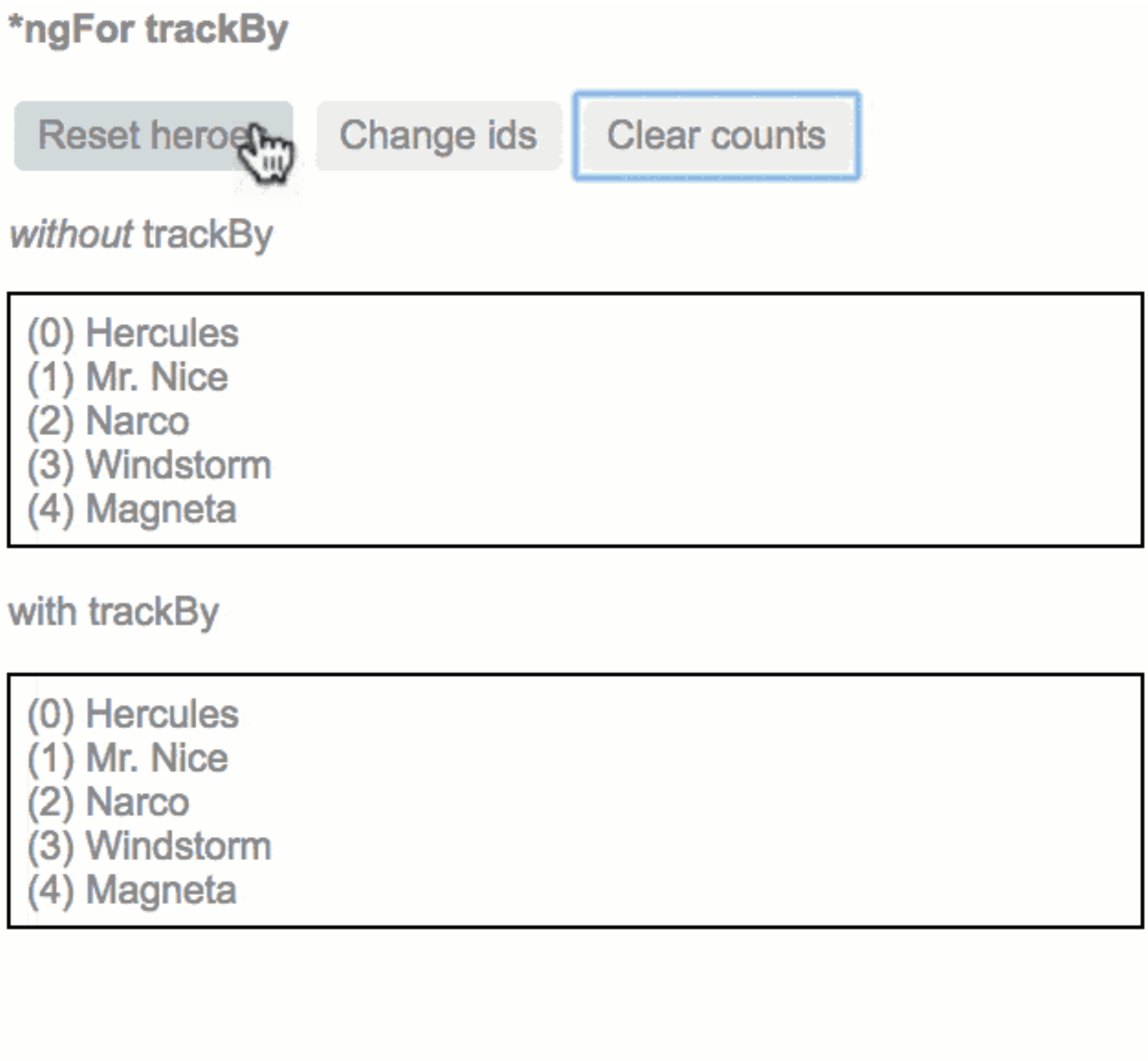
Most, if not all, are previously displayed heroes. *You* know this because the `id` of each hero hasn't changed. But Angular sees only a fresh list of new object references. It has no choice but to tear down the old DOM elements and insert all new DOM elements.

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`.

In the microsyntax expression, set `trackBy` to this method.

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

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



*ngFor trackBy

without trackBy

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

with trackBy

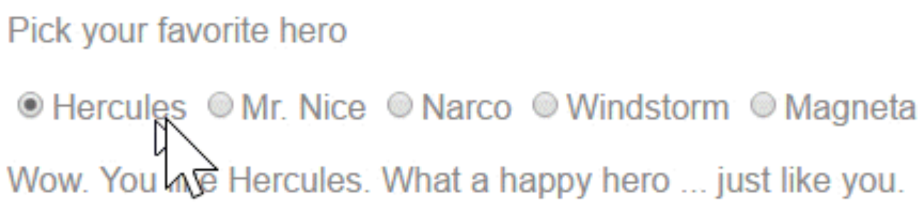
(0) Hercules
(1) Mr. Nice
(2) Narco
(3) Windstorm
(4) Magneta

{@a ngSwitch}

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.



Pick your favorite hero

● Hercules ○ Mr. Nice ○ Narco ○ Windstorm ○ Magneta

Wow. You are 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.

{@a template-reference-variable}

{@a ref-vars}

{@a ref-var}

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.

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

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.

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`.

{@a inputs-outputs}

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 (`=`).

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 *Output* property

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'`.

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 _private_ property.

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

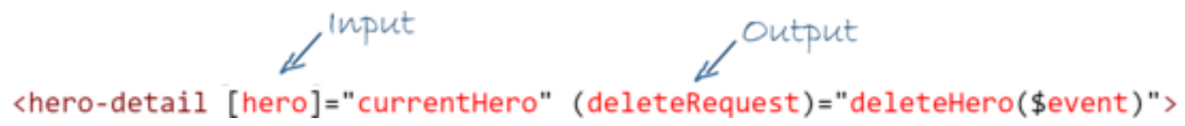
In the sample for this guide, the bindings to `HeroDetailComponent` do not fail because the data bound properties are annotated with `@Input()` and `@Output()` decorators.

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

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`.

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 `clicks` property.

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

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*:

{@a expression-operators}

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*.

{@a pipe}

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 (** `|` **)**:

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:

And you can also [apply parameters](#) to a pipe:

The `json` pipe is particularly helpful for debugging bindings:

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 }

{@a safe-navigation-operator}

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.

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

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.

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

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.

It works perfectly with long property paths such as `a?.b?.c?.d`.

{@a non-null-assertion-operator}

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.

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

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.

Testing

This guide offers tips and techniques for testing Angular applications. Though this page includes some general testing principles and techniques, the focus is on testing applications written with Angular.

{@a top}

Live examples

This guide presents tests of a sample application that is much like the *Tour of Heroes* tutorial. The sample application and all tests in this guide are available as live examples for inspection, experiment, and download:

- A spec to verify the test environment.
- The first component spec with inline template.
- A component spec with external template.
- The QuickStart seed's AppComponent spec.
- The sample application to be tested.
- All specs that test the sample application.
- A grab bag of additional specs.

{@a testing-intro}

Introduction to Angular Testing

This page guides you through writing tests to explore and confirm the behavior of the application. Testing does the following:

1. Guards against changes that break existing code (“regressions”).

2. Clarifies what the code does both when used as intended and when faced with deviant conditions.

3. Reveals mistakes in design and implementation. Tests shine a harsh light on the code from many angles. When a part of the application seems hard to test, the root cause is often a design flaw, something to cure now rather than later when it becomes expensive to fix.

{@a tools-and-tech}

Tools and technologies

You can write and run Angular tests with a variety of tools and technologies. This guide describes specific choices that are known to work well.

Technology	Purpose
Jasmine	The [Jasmine test framework](http://jasmine.github.io/2.4/introduction.html) provides everything needed to write basic tests. It ships with an HTML test runner that executes tests in the browser.
Angular testing utilities	Angular testing utilities create a test environment for the Angular application code under test. Use them to condition and control parts of the application as they interact _within_ the Angular environment.
Karma	The [karma test runner](https://karma-runner.github.io/1.0/index.html) is ideal for writing and running unit tests while developing the application. It can be an integral part of the project's development and continuous integration processes. This guide describes how to set up and run tests with karma.
Protractor	Use protractor to write and run _end-to-end_ (e2e) tests. End-to-end tests explore the application _as users experience it_. In e2e testing, one process runs the real application and a second process runs protractor tests that simulate user behavior and assert that the application respond in the browser as expected.

{@a setup}

Setup

There are two fast paths to getting started with unit testing.

1. Start a new project following the instructions in Setup.

2. Start a new project with the Angular CLI.

Both approaches install npm packages, files, and scripts pre-configured for applications built in their respective modalities. Their artifacts and procedures differ slightly but their essentials are the same and there are no differences in the test code.

In this guide, the application and its tests are based on the setup instructions. For a discussion of the unit testing setup files, see below.

{@a isolated-v-testing-utilities}

Isolated unit tests vs. the Angular testing utilities

Isolated unit tests examine an instance of a class all by itself without any dependence on Angular or any injected values. The tester creates a test instance of the class with `new`, supplying test doubles for the constructor parameters as needed, and then probes the test instance API surface.

You should write isolated unit tests for pipes and services.

You can test components in isolation as well. However, isolated unit tests don't reveal how components interact with Angular. In particular, they can't reveal how a component class interacts with its own template or with other components.

Such tests require the **Angular testing utilities**. The Angular testing utilities include the `TestBed` class and several helper functions from `@angular/core/testing`. They are the main focus of this guide and you'll learn about them when you write your first component test. A comprehensive review of the Angular testing utilities appears later in this guide.

But first you should write a dummy test to verify that your test environment is set up properly and to lock in a few basic testing skills.

{@a 1st-karma-test}

The first karma test

Start with a simple test to make sure that the setup works properly.

Create a new file called `1st.spec.ts` in the application root folder, `src/app/`

Tests written in Jasmine are called _specs_ . **The filename extension must be `.spec.ts`**, the convention adhered to by `karma.conf.js` and other tooling.

Put spec files somewhere within the `src/app/` folder. The `karma.conf.js` tells karma to look for spec files there, for reasons explained below.

Add the following code to `src/app/1st.spec.ts`.

{@a run-karma}

Run with karma

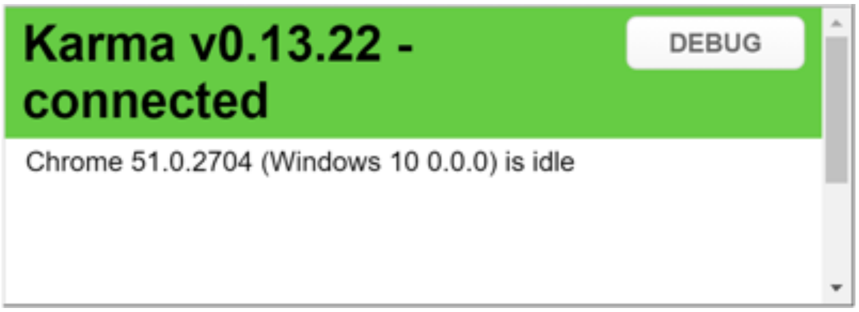
Compile and run it in karma from the command line using the following command:

```
npm test
```

The command compiles the application and test code and starts karma. Both processes watch pertinent files, write messages to the console, and re-run when they detect changes.

The documentation setup defines the `test` command in the `scripts` section of npm's `package.json`. The Angular CLI has different commands to do the same thing. Adjust accordingly.

After a few moments, karma opens a browser and starts writing to the console.



Karma v0.13.22 - connected

Chrome 51.0.2704 (Windows 10 0.0.0) is idle

Hide (don't close!) the browser and focus on the console output, which should look something like this:

> npm test ... [0] 1:37:03 PM - Compilation complete. Watching for file changes. ... [1] Chrome 51.0.2704: Executed 0 of 0 SUCCESS Chrome 51.0.2704: Executed 1 of 1 SUCCESS SUCCESS (0.005 secs / 0.005 secs)

Both the compiler and karma continue to run. The compiler output is preceded by `[0]`; the karma output by `[1]`.

Change the expectation from `true` to `false`.

The *compiler* watcher detects the change and recompiles.

```
[0] 1:49:21 PM - File change detected. Starting incremental compilation... [0] 1:49:25 PM - Compilation complete. Watching for file changes.
```

The *karma* watcher detects the change to the compilation output and re-runs the test.

```
[1] Chrome 51.0.2704 1st tests true is true FAILED [1] Expected false to equal true. [1] Chrome 51.0.2704: Executed 1 of 1 (1 FAILED) (0.005 secs / 0.005 secs)
```

It fails of course.

Restore the expectation from `false` back to `true`. Both processes detect the change, re-run, and karma reports complete success.

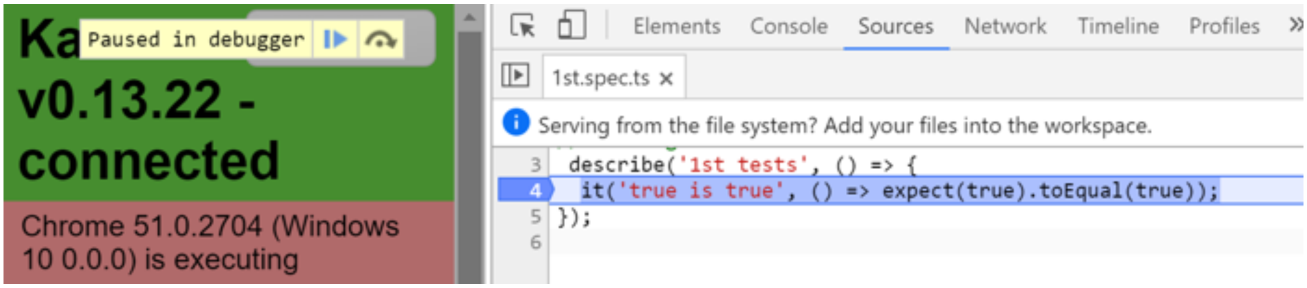
The console log can be quite long. Keep your eye on the last line. When all is well, it reads `SUCCESS`.

{@a test-debugging}

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.



Ka
v0.13.22 -
connected

Chrome 51.0.2704 (Windows 10 0.0.0) is executing

Paused in debugger

Serving from the file system? Add your files into the workspace.

```
 describe('1st tests', () => {
  it('true is true', () => expect(true).toEqual(true));
});
```

{@a live-karma-example}

Try the live example

You can also try this test as a in plunker. All of the tests in this guide are available as live examples.

{@a simple-component-test}

Test a component

An Angular component is the first thing most developers want to test. The `BannerComponent` in `src/app/banner-inline.component.ts` is the simplest component in this application and a good place to start. It presents the application title at the top of the screen within an `<h1>` tag.

This version of the `BannerComponent` has an inline template and an interpolation binding. The component is probably too simple to be worth testing in real life but it's perfect for a first encounter with the Angular testing utilities.

The corresponding `src/app/banner-inline.component.spec.ts` sits in the same folder as the component, for reasons explained in the FAQ answer to "Why put specs next to the things they test?".

Start with ES6 import statements to get access to symbols referenced in the spec.

{@a configure-testing-module}

Here's the `describe` and the `beforeEach` that precedes the tests:

{@a testbed}

TestBed

`TestBed` is the first and most important of the Angular testing utilities. It creates an Angular testing module—an `@NgModule` class—that you configure with the `configureTestingModule` method to produce the module environment for the class you want to test. In effect, you detach the tested component from its own application module and re-attach it to a dynamically-constructed Angular test module tailored specifically for this battery of tests.

The `configureTestingModule` method takes an `@NgModule` -like metadata object. The metadata object can have most of the properties of a normal NgModule.

This metadata object simply declares the component to test, `BannerComponent` . The metadata lack `imports` because (a) the default testing module configuration already has what `BannerComponent` needs and (b) `BannerComponent` doesn't interact with any other components.

Call `configureTestingModule` within a `beforeEach` so that `TestBed` can reset itself to a base state before each test runs.

The base state includes a default testing module configuration consisting of the declarables (components, directives, and pipes) and providers (some of them mocked) that almost everyone needs.

The testing shims mentioned [later](guide/testing#testbed-methods) initialize the testing module configuration to something like the `BrowserModule` from `@angular/platform-browser`.

This default configuration is merely a *foundation* for testing an app. Later you'll call `TestBed.configureTestingModule` with more metadata that define additional imports, declarations, providers, and schemas to fit your application tests. Optional `override` methods can fine-tune aspects of the configuration.

{@a create-component}

createComponent

After configuring `TestBed`, you tell it to create an instance of the *component-under-test*. In this example, `TestBed.createComponent` creates an instance of `BannerComponent` and returns a *component test fixture*.

Do not re-configure `TestBed` after calling `createComponent`.

The `createComponent` method closes the current `TestBed` instance to further configuration. You cannot call any more `TestBed` configuration methods, not `configureTestingModule` nor any of the `override...` methods. If you try, `TestBed` throws an error.

{@a component-fixture}

ComponentFixture, *DebugElement*, and *query(By.css)*

The `createComponent` method returns a **`ComponentFixture`**, a handle on the test environment surrounding the created component. The fixture provides access to the component instance itself and to the **`DebugElement`**, which is a handle on the component's DOM element.

The `title` property value is interpolated into the DOM within `<h1>` tags. Use the fixture's `DebugElement` to `query` for the `<h1>` element by CSS selector.

The **`query`** method takes a predicate function and searches the fixture's entire DOM tree for the *first* element that satisfies the predicate. The result is a *different* `DebugElement`, one associated with the matching DOM element.

The `queryAll` method returns an array of _all_ `DebugElements` that satisfy the predicate. A _predicate_ is a function that returns a boolean. A query predicate receives a `DebugElement` and returns `true` if the element meets the selection criteria.

The **`By`** class is an Angular testing utility that produces useful predicates. Its `By.css` static method produces a standard CSS selector predicate that filters the same way as a jQuery selector.

Finally, the setup assigns the DOM element from the `DebugElement` **`nativeElement`** property to `el`. The tests assert that `el` contains the expected title text.

{@a the-tests}

The tests

Jasmine runs the `beforeEach` function before each of these tests

These tests ask the `DebugElement` for the native HTML element to satisfy their expectations.

{@a detect-changes}

detectChanges: Angular change detection within a test

Each test tells Angular when to perform change detection by calling `fixture.detectChanges()`. The first test does so immediately, triggering data binding and propagation of the `title` property to the DOM element.

The second test changes the component's `title` property *and only then* calls `fixture.detectChanges()`; the new value appears in the DOM element.

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. The fixture does not automatically push the component's `title` property value into the data bound element, a fact demonstrated in the following test:

This behavior (or lack of it) is intentional. It gives the tester an opportunity to inspect or change the state of the component *before Angular initiates data binding or calls lifecycle hooks*.

{@a try-example}

Try the live example

Take a moment to explore this component spec as a and lock in these fundamentals of component unit testing.

{@a auto-detect-changes}

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:

Then add it to the `providers` array of the testing module configuration:

Here are three tests that illustrate how automatic change detection works.

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.

{@a component-with-external-template}

Test a component with an external template

The application's actual `BannerComponent` behaves the same as the version above but is implemented differently. It has *external* template and css files, specified in `templateUrl` and `styleUrls` properties.

That's a problem for the tests. The `TestBed.createComponent` method is synchronous. But the Angular template compiler must read the external files from the file system before it can create a component instance. That's an asynchronous activity. The previous setup for testing the inline component won't work for a component with an external template.

The first asynchronous *beforeEach*

The test setup for `BannerComponent` must give the Angular template compiler time to read the files. The logic in the `beforeEach` of the previous spec is split into two `beforeEach` calls. The first `beforeEach` handles asynchronous compilation.

Notice the `async` function called as the argument to `beforeEach`. The `async` function is one of the Angular testing utilities and has to be imported.

It takes a parameterless function and *returns a function* which becomes the true argument to the `beforeEach`.

The body of the `async` argument looks much like the body of a synchronous `beforeEach`. There is nothing obviously asynchronous about it. For example, it doesn't return a promise and there is no `done` function to call as there would be in standard Jasmine asynchronous tests. Internally, `async` arranges for the body of the `beforeEach` to run in a special *async test zone* that hides the mechanics of asynchronous execution.

All this is necessary in order to call the asynchronous `TestBed.compileComponents` method.

{@a compile-components}

compileComponents

The `TestBed.configureTestingModule` method returns the `TestBed` class so you can chain calls to other `TestBed` static methods such as `compileComponents`.

The `TestBed.compileComponents` method asynchronously compiles all the components configured in the testing module. In this example, the `BannerComponent` is the only component to compile. When `compileComponents` completes, the external templates and css files have been "inlined" and `TestBed.createComponent` can create new instances of `BannerComponent` synchronously.

WebPack developers need not call `compileComponents` because it inlines templates and css as part of the automated build process that precedes running the test.

In this example, `TestBed.compileComponents` only compiles the `BannerComponent`. Tests later in the guide declare multiple components and a few specs import entire application modules that hold yet more components. Any of these components might have external templates and css files. `TestBed.compileComponents` compiles all of the declared components asynchronously at one time.

Do not configure the `TestBed` after calling `compileComponents`. Make `compileComponents` the last step before calling `TestBed.createComponent` to instantiate the _component-under-test_.

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.

{@a second-before-each}

The second synchronous *beforeEach*

A *synchronous* `beforeEach` containing the remaining setup steps follows the asynchronous `beforeEach`.

These are the same steps as in the original `beforeEach`. They include creating an instance of the `BannerComponent` and querying for the elements to inspect.

You can count on the test runner to wait for the first asynchronous `beforeEach` to finish before calling the second.

{@a waiting-compile-components}

Waiting for *compileComponents*

The `compileComponents` method returns a promise so you can perform additional tasks *immediately after* it finishes. For example, you could move the synchronous code in the second `beforeEach` into a `compileComponents().then(...)` callback and write only one `beforeEach`.

Most developers find that hard to read. The two `beforeEach` calls are widely preferred.

{@a live-external-template-example}

Try the live example

Take a moment to explore this component spec as a .

The [Quickstart seed](guide/setup) provides a similar test of its `AppComponent` as you can see in _this_ . It too calls `compileComponents` although it doesn't have to because the `AppComponent`'s template is inline. There's no harm in it and you might call `compileComponents` anyway in case you decide later to re-factor the template into a separate file. The tests in this guide only call `compileComponents` when necessary.

{@a component-with-dependency}

Test a 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` :

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, `src/app/welcome.component.spec.ts` :

This time, in addition to declaring the *component-under-test*, the configuration adds a `UserService` provider to the `providers` list. But not the real `UserService` .

{@a service-test-doubles}

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 `UserService` stub that satisfies the needs of the `WelcomeComponent` and its tests:

{@a get-injected-service}

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`.

{@a testbed-get}

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. Fortunately, in this test suite, the *only* provider of `UserService` is the root testing module, so it is safe to call `TestBed.get` as follows:

The [`inject`](guide/testing#inject) utility function is another way to get one or more services from the test root injector. For a use case in which `inject` and `TestBed.get` do not work, see the section [_Override a component's providers_](guide/testing#component-override), which explains why you must get the service from the component's injector instead.

{@a service-from-injector}

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`.

{@a welcome-spec-setup}

Final setup and tests

Here's the complete `beforeEach` using `TestBed.get`:

And here are some tests:

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 addendum. If the expectation fails, Jasmine displays this addendum after 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.

{@a component-with-async-service}

Test a component with an async service

Many services return values asynchronously. Most data services make an HTTP request to a remote server and the response is necessarily asynchronous.

The "About" view in this sample displays Mark Twain quotes. The `TwainComponent` handles the display, delegating the server request to the `TwainService`.

Both are in the `src/app/shared` folder because the author intends to display Twain quotes on other pages someday. Here is the `TwainComponent`.

The `TwainService` implementation is irrelevant for this particular test. It is sufficient to see within `ngOnInit` that `twainService.getQuote` returns a promise, which means it is asynchronous.

In general, tests should not make calls to remote servers. They should emulate such calls. The setup in this `src/app/shared/twain.component.spec.ts` shows one way to do that:

{@a service-spy}

Spying on the real service

This setup is similar to the `welcome.component.spec` setup. But instead of creating a stubbed service object, it injects the *real* service (see the testing module `providers`) and replaces the critical `getQuote` method with a Jasmine spy.

The spy is designed such that any call to `getQuote` receives an immediately resolved promise with a test quote. The spy bypasses the actual `getQuote` method and therefore does not contact the server.

Faking a service instance and spying on the real service are _both_ great options. Pick the one that seems easiest for the current test suite. Don't be afraid to change your mind. Spying on the real service isn't always easy, especially when the real service has injected dependencies. You can _stub and spy_ at the same time, as shown in [an example below](guide/testing#spy-stub).

Here are the tests with commentary to follow:

{@a sync-tests}

Synchronous tests

The first two tests are synchronous. Thanks to the spy, they verify that `getQuote` is called *after* the first change detection cycle during which Angular calls `ngOnInit`.

Neither test can prove that a value from the service is displayed. The quote itself has not arrived, despite the fact that the spy returns a resolved promise.

This test must wait at least one full turn of the JavaScript engine before the value becomes available. The test must become *asynchronous*.

{@a async}

The *async* function in *it*

Notice the `async` in the third test.

The `async` function is one of the Angular testing utilities. It simplifies coding of asynchronous tests by arranging for the tester's code to run in a special *async test zone* as discussed earlier when it was called in a `beforeEach`.

Although `async` does a great job of hiding asynchronous boilerplate, some functions called within a test (such as `fixture.whenStable`) continue to reveal their asynchronous behavior.

The `fakeAsync` alternative, [covered below](guide/testing#fake-async), removes this artifact and affords a more linear coding experience.

{@a when-stable}

whenStable

The test must wait for the `getQuote` promise to resolve in the next turn of the JavaScript engine.

This test has no direct access to the promise returned by the call to `twainService.getQuote` because it is buried inside `TwainComponent.ngOnInit` and therefore inaccessible to a test that probes only the component API surface.

Fortunately, the `getQuote` promise is accessible to the *async test zone*, which intercepts all promises issued within the *async* method call *no matter where they occur*.

The `ComponentFixture.whenStable` method returns its own promise, which resolves when the `getQuote` promise finishes. In fact, the *whenStable* promise resolves when *all pending asynchronous activities within this test* complete—the definition of "stable."

Then the test resumes and kicks off another round of change detection (`fixture.detectChanges`), which tells Angular to update the DOM with the quote. The `getQuote` helper method extracts the display element text and the expectation confirms that the text matches the test quote.

{@a fakeAsync}

{@a fake-async}

The *fakeAsync* function

The fourth test verifies the same component behavior in a different way.

Notice that `fakeAsync` replaces `async` as the `it` argument. The `fakeAsync` function is another of the Angular testing utilities.

Like async, it *takes* a parameterless function and *returns* a function that becomes the argument to the Jasmine `it` call.

The `fakeAsync` function enables a linear coding style by running the test body in a special *fakeAsync test zone*.

The principle advantage of `fakeAsync` over `async` is that the test appears to be synchronous. There is no `then(...)` to disrupt the visible flow of control. The promise-returning `fixture.whenStable` is gone, replaced by `tick()`.

There _are_ limitations. For example, you cannot make an XHR call from within a `fakeAsync`.

{@a tick}

The *tick* function

The `tick` function is one of the Angular testing utilities and a companion to `fakeAsync`. You can only call it within a `fakeAsync` body.

Calling `tick()` simulates the passage of time until all pending asynchronous activities finish, including the resolution of the `getQuote` promise in this test case.

It returns nothing. There is no promise to wait for. Proceed with the same test code that appeared in the `whenStable.then()` callback.

Even this simple example is easier to read than the third test. To more fully appreciate the improvement, imagine a succession of asynchronous operations, chained in a long sequence of promise callbacks.

{@a jasmine-done}

jasmine.done

While the `async` and `fakeAsync` functions greatly simplify Angular asynchronous testing, you can still fall back to the traditional Jasmine asynchronous testing technique.

You can still pass `it` a function that takes a [`done` callback](). Now you are responsible for chaining promises, handling errors, and calling `done` at the appropriate moment.

Here is a `done` version of the previous two tests:

Although there is no direct access to the `getQuote` promise inside `TwainComponent`, the spy has direct access, which makes it possible to wait for `getQuote` to finish.

Writing test functions with `done`, while more cumbersome than `async` and `fakeAsync`, is a viable and occasionally necessary technique. For example, you can't call `async` or `fakeAsync` when testing code that involves the `intervalTimer`, as is common when testing async `Observable` methods.

{@a component-with-input-output}

Test a 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:

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 definition:

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:

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](guide/testing#routed-component) covers testing components that require the router.

The immediate goal is to test the `DashboardHeroComponent`, not the `DashboardComponent`, so, try the second and third options.

{@a dashboard-standalone}

Test *DashboardHeroComponent* stand-alone

Here's the spec file setup.

The async `beforeEach` was discussed above. Having compiled the components asynchronously with `compileComponents`, the rest of the setup proceeds *synchronously* in a *second* `beforeEach`, using the basic techniques described earlier.

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 first test follows:

It verifies that the hero name is propagated to template with a binding. Because the template passes the hero name through the Angular `UpperCasePipe`, the test must match the element value with the uppercased name:

This small test demonstrates how Angular tests can verify a component's visual representation—something not possible with [isolated unit tests](guide/testing#isolated-component-tests)—at low cost and without resorting to much slower and more complicated end-to-end tests.

The second test verifies click behavior. Clicking the hero should raise a `selected` event that the host component (`DashboardComponent` presumably) can hear:

The component exposes an `EventEmitter` property. The test subscribes to it just as the host component would do.

The `heroEl` is a `DebugElement` that represents the hero `<div>`. The test calls `triggerEventHandler` with the "click" event name. The "click" event binding responds by calling `DashboardHeroComponent.click()`.

If the component behaves as expected, `click()` tells the component's `selected` property to emit the `hero` object, the test detects that value through its subscription to `selected`, and the test should pass.

{@a trigger-event-handler}

triggerEventHandler

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.

In this example, the test triggers a "click" event with a null event object.

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 was pressed. This directive throws an error if the event object doesn't do this correctly.

{@a click-helper}

Clicking a button, an anchor, or an arbitrary HTML element is a common test task.

Make that easy by encapsulating the *click-triggering* process in a helper such as the `click` function below:

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.

click() is not an Angular testing utility
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 this click helper.

{@a component-inside-test-host}

Test a component inside a test host component

In the previous approach, the tests themselves played the role of the host `DashboardComponent`. But does the `DashboardHeroComponent` work correctly when properly data-bound to a host component?

Testing with the actual `DashboardComponent` host is doable but seems more trouble than its worth. It's easier to emulate the `DashboardComponent` host with a *test host* like this one:

The test host binds to `DashboardHeroComponent` as the `DashboardComponent` would but without the distraction of the `Router`, the `HeroService`, or even 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 check that property to verify that the `DashboardHeroComponent.selected` event emitted the right hero.

The setup for the test-host tests is similar to the setup for the stand-alone tests:

This testing module configuration shows two important differences:

1. It *declares* both the `DashboardHeroComponent` and the `TestHostComponent`.

2. It *creates* the `TestHostComponent` instead of the `DashboardHeroComponent`.

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:

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.

{@a routed-component}

Test a routed component

Testing the actual `DashboardComponent` seemed daunting because it injects the `Router`.

It also injects the `HeroService`, but faking that is a familiar story. The `Router` has a complicated API and is entwined with other services and application preconditions.

Fortunately, the `DashboardComponent` isn't doing much with the `Router`

This is often the case. 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. Stubbing the router with a test implementation is an easy option. This should do the trick:

Now set up the testing module with the test stubs for the `Router` and `HeroService`, and create a test instance of the `DashboardComponent` for subsequent testing.

The following test clicks the displayed hero and confirms (with the help of a spy) that `Router.navigateByUrl` is called with the expected url.

{@a inject}

The *inject* function

Notice the `inject` function in the second `it` argument.

The `inject` function is one of the Angular testing utilities. It injects services into the test function where you can alter, spy on, and manipulate them.

The `inject` function has two parameters:

1. An array of Angular dependency injection tokens.
2. A test function whose parameters correspond exactly to each item in the injection token array.

inject uses the TestBed Injector
The `inject` function uses the current `TestBed` injector and can only return services provided at that level. It does not return services from component providers.

This example injects the `Router` from the current `TestBed` injector. That's fine for this test because the `Router` is, and must be, provided by the application root injector.

If you need a service provided by the component's *own* injector, call `fixture.debugElement.injector.get` instead:

Use the component's own injector to get the service actually injected into the component.

The `inject` function 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.

Do not configure the `TestBed` after calling `inject`.

{@a routed-component-w-param}

Test a routed component with parameters

Clicking a *Dashboard* hero triggers navigation to `heroes/:id`, where `:id` is a route parameter whose value is the `id` of the hero to edit. That URL matches a route to the `HeroDetailComponent`.

The router pushes the `:id` token value into the `ActivatedRoute.params` *Observable* property, Angular injects the `ActivatedRoute` into the `HeroDetailComponent`, and the component extracts the `id` so it can fetch the corresponding hero via the `HeroDetailService`. Here's the `HeroDetailComponent` constructor:

`HeroDetailComponent` subscribes to `ActivatedRoute.params` changes in its `ngOnInit` method.

The expression after `route.params` chains an _Observable_ operator that _plucks_ the `id` from the `params` and then chains a `forEach` operator to subscribe to `id`-changing events. The `id` changes every time the user navigates to a different hero. The `forEach` passes the new `id` value to the component's `getHero` method (not shown) which fetches a hero and sets the component's `hero` property. If the`id` parameter is missing, the `pluck` operator fails and the `catch` treats failure as a request to edit a new hero. The [Router](guide/router#route-parameters) guide covers `ActivatedRoute.params` in more detail.

A test can explore how the `HeroDetailComponent` responds to different `id` parameter values by manipulating the `ActivatedRoute` injected into the component's constructor.

By now you know how to stub the `Router` and a data service. Stubbing the `ActivatedRoute` follows the same pattern except for a complication: the `ActivatedRoute.params` is an *Observable*.

{@a stub-observable}

Create an *Observable* test double

The `hero-detail.component.spec.ts` relies on an `ActivatedRouteStub` to set `ActivatedRoute.params` values for each test. This is a cross-application, re-usable *test helper class*. Consider placing such helpers in a `testing` folder sibling to the `app` folder. This sample keeps `ActivatedRouteStub` in `testing/router-stubs.ts`:

Notable features of this stub are:

- The stub implements only two of the `ActivatedRoute` capabilities: `params` and `snapshot.params`.

- *BehaviorSubject* drives the stub's `params` *Observable* and returns the same value to every `params` subscriber until it's given a new value.

- The `HeroDetailComponent` chains its expressions to this stub `params` *Observable* which is now under the tester's control.

- Setting the `testParams` property causes the `subject` to push the assigned value into `params`. That triggers the `HeroDetailComponent` *params* subscription, described above, in the same way that navigation does.

- Setting the `testParams` property also updates the stub's internal value for the `snapshot` property to return.

The [_snapshot_](guide/router#snapshot "Router guide: snapshot") is another popular way for components to consume route parameters.
The router stubs in this guide are meant to inspire you. Create your own stubs to fit your testing needs.

{@a tests-w-observable-double}

Testing with the *Observable* test double

Here's a test demonstrating the component's behavior when the observed `id` refers to an existing hero:

The `createComponent` method and `page` object are discussed [in the next section](guide/testing#page-object). Rely on your intuition for now.

When the `id` cannot be found, the component should re-route to the `HeroListComponent`. The test suite setup provided the same `RouterStub` [described above](#) which spies on the router without actually navigating. This test supplies a "bad" id and expects the component to try to navigate.

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`.

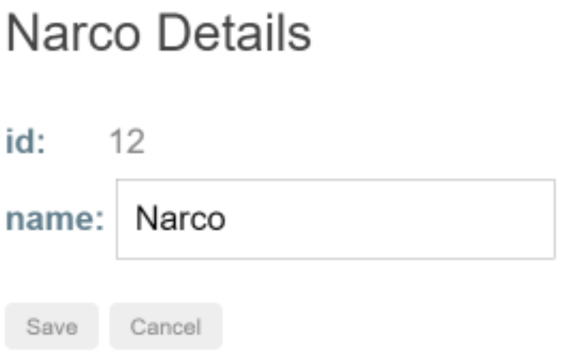
In this implementation, the component should create and display a new hero. New heroes have `id=0` and a blank `name`. This test confirms that the component behaves as expected:

Inspect and download _all_ of the guide's application test code with this live example.

{@a page-object}

Use a *page* object to simplify setup

The `HeroDetailComponent` is a simple view with a title, two hero fields, and two buttons.



Narco Details

id: 12

name: Narco

But there's already plenty of template complexity.

To fully exercise the component, the test needs a lot of setup:

- It must wait until a hero arrives before `*ngIf` allows any element in DOM.
- It needs references to the title `<span>` and the name `<input>` so it can inspect their values.
- It needs references to the two buttons so it can click them.

- It needs 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 madness with a `Page` class that simplifies access to component properties and encapsulates the logic that sets them. Here's the `Page` class for the `hero-detail.component.spec.ts`

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.

The observable tests in the previous 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 drive the point home.

{@a import-module}

Setup with module imports

Earlier component tests configured the testing module with a few `declarations` like this:

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:

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:

It's a bit tighter and smaller, with fewer import statements (not shown).

{@a feature-module-import}

Import the 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:

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 throws an error because `HeroDetailComponent` is declared in both the `HeroModule` and the `DynamicTestModule` (the testing module).
Importing the component's feature module is often the easiest way to configure the tests, especially when the feature module is small and mostly self-contained, as feature modules should be.

{@a component-override}

Override a component's providers

The `HeroDetailComponent` provides its own `HeroDetailService`.

It's not possible to stub the component's `HeroDetailService` in the `providers` of the `TestBed.configureTestingModule`. Those are providers for the *testing module*, not the component. They prepare the dependency injector at the *fixture level*.

Angular creates the component with its *own* injector, which is a *child* of the fixture injector. It registers the component's providers (the `HeroDetailService` in this case) with the child injector. A test cannot get to child injector services from the fixture injector. And `TestBed.configureTestingModule` can't configure them either.

Angular has been creating new instances of the real `HeroDetailService` all along!

These tests could fail or timeout if the `HeroDetailService` made its own XHR calls to a remote server. There might not be a remote server to call. Fortunately, the `HeroDetailService` delegates responsibility for remote data access to an injected `HeroService`. The [previous test configuration](guide/testing#feature-module-import) replaces the real `HeroService` with a `FakeHeroService` 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:

Notice that `TestBed.configureTestingModule` no longer provides a (fake) `HeroService` because it's not needed.

{@a override-component-method}

The *overrideComponent* method

Focus on the `overrideComponent` method.

It takes two arguments: the component type to override (`HeroDetailComponent`) and an override metadata object. The overide metadata object is a generic defined as follows:

type MetadataOverride = { add?: T; remove?: T; set?: T; };

A metadata override object can either add-and-remove elements in metadata properties or completely reset those properties. This example resets the component's `providers` metadata.

The type parameter, `T` , is the kind of metadata you'd pass to the `@Component` decorator:

selector?: string; template?: string; templateUrl?: string; providers?: any[]; ...

{@a spy-stub}

Provide a *spy stub* (*HeroDetailServiceSpy*)

This example completely replaces the component's `providers` array with a new array containing a `HeroDetailServiceSpy` .

The `HeroDetailServiceSpy` is a stubbed version of the real `HeroDetailService` that fakes all necessary features of that service. It neither injects nor delegates to the lower level `HeroService` so there's no need to provide a test double for that.

The related `HeroDetailComponent` tests will assert that methods of the `HeroDetailService` were called by spying on the service methods. Accordingly, the stub implements its methods as spies:

{@a override-tests}

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.

{@a more-overrides}

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.

{@a router-outlet-component}

Test a *RouterOutlet* component

The `AppComponent` displays routed components in a `<router-outlet>`. It also displays a navigation bar with anchors and their `RouterLink` directives.

{@a app-component-html}

The component class does nothing.

Unit tests can confirm that the anchors are wired properly without engaging the router. See why this is worth doing below.

{@a stub-component}

Stubbing unneeded components

The test setup should look familiar.

The `AppComponent` is the declared test subject.

The setup extends the default testing module with one real component (`BannerComponent`) and several stubs.

- `BannerComponent` is simple and harmless to use as is.

- The real `WelcomeComponent` has an injected service. `WelcomeStubComponent` is a placeholder with no service to worry about.

- The real `RouterOutlet` is complex and errors easily. The `RouterOutletStubComponent` (in `testing/router-stubs.ts`) is safely inert.

The component stubs are essential. Without them, the Angular compiler doesn't recognize the `<app-welcome>` and `<router-outlet>` tags and throws an error.

{@a router-link-stub}

Stubbing the *RouterLink*

The `RouterLinkStubDirective` contributes substantively to the test:

The `host` metadata property wires the click event of the host element (the `<a>`) to the directive's `onClick` method. The URL bound to the `[routerLink]` attribute flows to the directive's `linkParams` property. Clicking the anchor should trigger the `onClick` method which sets the telltale `navigatedTo` property. Tests can inspect that property to confirm the expected *click-to-navigation* behavior.

{@a by-directive}

{@a inject-directive}

By.directive and injected directives

A little more setup triggers the initial data binding and gets references to the navigation links:

Two points of special interest:

1. You can locate elements *by directive*, using `By.directive`, not just by css selectors.

2. You can use the component's dependency injector to get an attached directive because Angular always adds attached directives to the component's injector.

{@a app-component-tests}

Here are some tests that leverage this setup:

The "click" test _in this example_ is worthless. It works hard to appear useful when in fact it tests the `RouterLinkStubDirective` 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.

{@a why-stubbed-routerlink-tests}

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`.

{@a shallow-component-test}

"Shallow component tests" with *NO_ERRORS_SCHEMA*

The previous setup declared the `BannerComponent` and stubbed two other components for *no reason other than to avoid a compiler error*.

Without them, the Angular compiler doesn't recognize the `<app-banner>`, `<app-welcome>` and `<router-outlet>` tags in the *app.component.html* template and throws an error.

Add `NO_ERRORS_SCHEMA` to the testing module's `schemas` metadata to tell the compiler to ignore unrecognized elements and attributes. You no longer have to declare irrelevant components and directives.

These tests are ***shallow*** because they only "go deep" into the components you want to test.

Here is a setup, with `import` statements, that demonstrates the improved simplicity of *shallow* tests, relative to the stubbing setup.

The *only* declarations are the *component-under-test* (`AppComponent`) and the `RouterLinkStubDirective` that contributes actively to the tests. The [tests in this example]() are unchanged.

Shallow component tests with `NO_ERRORS_SCHEMA` greatly simplify unit testing of complex templates. However, the compiler no longer alerts you to mistakes such as misspelled or misused components and directives.

{@a attribute-directive}

Test an attribute directive

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.

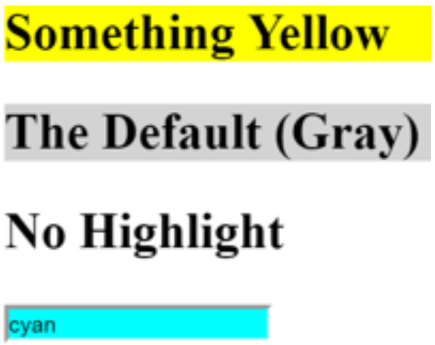
It's used throughout the application, perhaps most simply in the `AboutComponent` :

Testing the specific use of the `HighlightDirective` within the `AboutComponent` requires only the techniques explored above (in particular the ["Shallow test"]() approach).

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.

[Isolated unit 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.



Something Yellow

The Default (Gray)

No Highlight

cyan

The ` ` 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:

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.

{@a isolated-unit-tests}

Isolated Unit Tests

Testing applications with the help of the Angular testing utilities is the main focus of this guide.

However, it's often more productive to explore the inner logic of application classes with *isolated* unit tests that don't depend upon Angular. Such tests are often smaller and easier to read, write, and maintain.

They don't carry extra baggage:

- Import from the Angular test libraries.
- Configure a module.
- Prepare dependency injection `providers`.
- Call `inject` or `async` or `fakeAsync`.

They follow patterns familiar to test developers everywhere:

- Exhibit standard, Angular-agnostic testing techniques.
- Create instances directly with `new`.
- Substitute test doubles (stubs, spys, and mocks) for the real dependencies.

Write both kinds of tests
Good developers write both kinds of tests for the same application part, often in the same spec file. Write simple _isolated_ unit tests to validate the part in isolation. Write _Angular_ tests to validate the part as it interacts with Angular, updates the DOM, and collaborates with the rest of the application.

{@a isolated-service-tests}

Services

Services are good candidates for isolated unit testing. Here are some synchronous and asynchronous unit tests of the `FancyService` written without assistance from Angular testing utilities.

A rough line count suggests that these isolated unit tests are about 25% smaller than equivalent Angular tests. That's telling but not decisive. The benefit comes from reduced setup and code complexity.

Compare these equivalent tests of `FancyService.getTimeoutValue`.

They have about the same line-count, but the Angular-dependent version has more moving parts including a couple of utility functions (`async` and `inject`). Both approaches work and it's not much of an issue if you're using the Angular testing utilities nearby for other reasons. On the other hand, why burden simple service tests with added complexity?

Pick the approach that suits you.

{@a services-with-dependencies}

Services with dependencies

Services often depend on other services that Angular injects into the constructor. You can test these services *without* the `TestBed`. In many cases, it's easier to create and *inject* dependencies by hand.

The `DependentService` is a simple example:

It delegates its only method, `getValue`, to the injected `FancyService`.

Here are several ways to test it.

The first test creates a `FancyService` with `new` and passes it to the `DependentService` constructor.

However, it's rarely that simple. The injected service can be difficult to create or control. You can mock the dependency, use a dummy value, or stub the pertinent service method with a substitute method that's easy to control.

These *isolated* unit testing techniques are great for exploring the inner logic of a service or its simple integration with a component class. Use the Angular testing utilities when writing tests that validate how a service interacts with components *within the Angular runtime environment*.

{@a isolated-pipe-tests}

Pipes

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.

Anything that uses a regular expression is worth testing thoroughly. Use simple Jasmine to explore the expected cases and the edge cases.

{@a write-tests}

Write Angular 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:

{@a isolated-component-tests}

Components

Component tests typically examine how a component class interacts with its own template or with collaborating components. The Angular testing utilities are specifically designed to facilitate such tests.

Consider this `ButtonComp` component.

The following Angular test demonstrates that clicking a button in the template leads to an update of the on-screen message.

The assertions verify that the data values flow from one HTML control (the `<button>`) to the component and from the component back to a *different* HTML control (the `<span>`). A passing test means the component and its template are wired correctly.

Isolated unit tests can more rapidly probe a component at its API boundary, exploring many more conditions with less effort.

Here are a set of unit tests that verify the component's outputs in the face of a variety of component inputs.

Isolated component tests offer a lot of test coverage with less code and almost no setup. This is even more of an advantage with complex components, which may require meticulous preparation with the Angular testing utilities.

On the other hand, isolated unit tests can't confirm that the `ButtonComp` is properly bound to its template or even data bound at all. Use Angular tests for that.

{@a atu-apis}

Angular 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
`async`	Runs the body of a test (\`it\`) or setup (\`beforeEach\`) function within a special _async test zone_. See [discussion above](guide/testing#async).
`fakeAsync`	Runs the body of a test (\`it\`) within a special _fakeAsync test zone_, enabling a linear control flow coding style. See [discussion above](guide/testing#fake-async).
`tick`	Simulates the passage of time and the completion of pending

	asynchronous activities by flushing both _timer_ and _micro-task_ queues within the _fakeAsync test zone_. The curious, dedicated reader might enjoy this lengthy blog post, ["_Tasks, microtasks, queues and schedules_"](https://jakearchibald.com/2015/tasks-microtasks-queues-and-schedules/). Accepts an optional argument that moves the virtual clock forward by the specified number of milliseconds, clearing asynchronous activities scheduled within that timeframe. See [discussion above](guide/testing#tick).
`inject`	Injects one or more services from the current `TestBed` injector into a test function. See [above](guide/testing#inject).
`discardPeriodicTasks`	When a `fakeAsync` test ends with pending timer event _tasks_ (queued `setTimeOut` and `setInterval` callbacks), the test fails with a clear error message. In general, a test should end with no queued tasks. When pending timer tasks are expected, call `discardPeriodicTasks` to flush the _task_ queue and avoid the error.
`flushMicrotasks`	When a `fakeAsync` test ends with pending _micro-tasks_ such as unresolved promises, the test fails with a clear error message. In general, a test should wait for micro-tasks to finish. When pending microtasks are expected, call `flushMicrotasks` to flush the _micro-task_ queue and avoid the error.
`ComponentFixtureAutoDetect`	A provider token for a service that turns on [automatic change detection](guide/testing#automatic-change-detection).
`getTestBed`	Gets the current instance of the `TestBed`. Usually unnecessary because the static class methods of the `TestBed` class are typically sufficient. The `TestBed` instance exposes a few rarely used members that are not available as static methods.

{@a testbed-class-summary}

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[]>; };

{@a metadata-override-object}

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?: T; remove?: T; set?: T; };

{@a testbed-methods} {@a testbed-api-summary}

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 `getTestBed()` 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
`configureTestingModule`	The testing shims (\`karma-test-shim\`, \`browser-test-shim\`) establish the [initial test environment](guide/testing) and a default testing module. The default testing module is configured with basic declaratives and some Angular service substitutes that every tester needs. Call \`configureTestingModule\` to refine the testing module configuration for a particular set of tests by adding and removing imports, declarations (of components, directives, and pipes), and providers.
`compileComponents`	Compile the testing module asynchronously after you've finished configuring it. You **must** call this method if _any_ of the testing module components have a \`templateUrl\` or \`styleUrls\` because fetching component template and style files is necessarily

	asynchronous. See [above](guide/testing#compile-components). 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 `TestBed` configuration. After calling `compileComponent`, the `TestBed` configuration is frozen for the duration of the current spec.
`overrideModule`	Replace metadata for the given `NgModule`. Recall that modules can import other modules. The `overrideModule` method can reach deeply into the current testing module to modify one of these inner modules.
`overrideComponent`	Replace metadata for the given component class, which could be nested deeply within an inner module.
`overrideDirective`	Replace metadata for the given directive class, which could be nested deeply within an inner module.
`overridePipe`	Replace metadata for the given pipe class, which could be nested deeply within an inner module.
{@a testbed-get} `get`	Retrieve a service from the current `TestBed` injector. The `inject` function is often adequate for this purpose. But `inject` throws an error if it can't provide the service. What if the service is optional? The `TestBed.get` method takes an optional second parameter, the object to return if Angular can't find the provider (`null` in this example): After calling `get`, the `TestBed` configuration is frozen for the duration of the current spec.
{@a testbed-initTestEnvironment} `initTestEnvironment`	Initialize the testing environment for the entire test run. The testing shims (`karma-test-shim`, `browser-test-shim`) call it for you so there is rarely a reason for you to call it yourself. You may call this method _exactly once_. If you must change this default in the middle of your test run, call `resetTestEnvironment` first. Specify the Angular compiler factory, a `PlatformRef`, and a default Angular testing module. Alternatives for non-browser platforms are available in the general form `@angular/platform-/testing/`.
`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.

{@a component-fixture-api-summary}

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.

{@a component-fixture-properties}

ComponentFixture properties

Here are the most important properties for testers, in order of likely utility.

Properties	Description
`componentInstance`	The instance of the component class created by \`TestBed.createComponent\`.
`debugElement`	The \`DebugElement\` associated with the root element of the component. The \`debugElement\` provides insight into the component and its DOM element during test and debugging. It's a critical property for testers. The most interesting members are covered [below](guide/testing#debug-element-details).
`nativeElement`	The native DOM element at the root of the component.
`changeDetectorRef`	The \`ChangeDetectorRef\` for the component. The \`ChangeDetectorRef\` is most valuable when testing a component that has the \`ChangeDetectionStrategy.OnPush\` method or the component's change detection is under your programmatic control.

{@a component-fixture-methods}

ComponentFixture methods

The *fixture* methods cause Angular to perform certain tasks on the component tree. Call these method to trigger Angular behavior in response to simulated user action.

Here are the most useful methods for testers.

Methods	Description
`detectChanges`	Trigger a change detection cycle for the component. Call it to initialize the component (it calls `ngOnInit`) and after your test code, change the component's data bound property values. Angular can't see that you've changed `personComponent.name` and won't update the `name` binding until you call `detectChanges`. Runs `checkNoChanges`afterwards to confirm that there are no circular updates unless called as `detectChanges(false)`;
`autoDetectChanges`	Set this to `true` when you want the fixture to detect changes automatically. When autodetect is `true`, the test fixture calls `detectChanges` immediately after creating the component. Then it listens for pertinent zone events and calls `detectChanges` accordingly. When your test code modifies component property values directly, you probably still have to call `fixture.detectChanges` to trigger data binding updates. The default is `false`. Testers who prefer fine control over test behavior tend to keep it `false`.
`checkNoChanges`	Do a change detection run to make sure there are no pending changes. Throws an exceptions if there are.
`isStable`	If the fixture is currently _stable_, returns `true`. If there are async tasks that have not completed, returns `false`.
`whenStable`	Returns a promise that resolves when the fixture is stable. To resume testing after completion of asynchronous activity or asynchronous change detection, hook that promise. See [above](guide/testing#when-stable).
`destroy`	Trigger component destruction.

{@a debug-element-details}

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).
`query`	Calling `query(predicate: Predicate)` returns the first `DebugElement` that matches the [predicate](guide/testing#query-predicate) at any depth in the subtree.
`queryAll`	Calling `queryAll(predicate: Predicate)` returns all `DebugElements` that matches the [predicate](guide/testing#query-predicate) at any depth in subtree.
`injector`	The host dependency injector. For example, the root element's component instance injector.
`componentInstance`	The element's own component instance, if it has one.
`context`	An object that provides parent context for this element. Often an ancestor component instance that governs this element. When an element is repeated within `*ngFor`, the context is an `NgForRow` whose `$implicit` property is the value of the row instance value. For example, the `hero` in `*ngFor="let hero of heroes"`.
`children`	The immediate `DebugElement` children. Walk the tree by descending through `children`. `DebugElement` also has `childNodes`, a list of `DebugNode` objects. `DebugElement` derives from `DebugNode` objects and there are often more nodes than elements. Testers can usually ignore plain nodes.
`parent`	The `DebugElement` parent. Null if this is the root element.
`name`	The element tag name, if it is an element.
`triggerEventHandler`	Triggers the event by its name if there is a corresponding listener in the element's `listeners` collection. The second parameter is the _event object_ expected by the handler. See [above](guide/testing#trigger-event-handler). If the event lacks a listener or there's some other problem, consider calling `nativeElement.dispatchEvent(eventObject)`.
`listeners`	The callbacks attached to the component's `@Output` properties and/or

	the element's event properties.
`providerTokens`	This component's injector lookup tokens. Includes the component itself plus the tokens that the component lists in its `providers` metadata.
`source`	Where to find this element in the source component template.
`references`	Dictionary of objects associated with template local variables (e.g. `#foo`), keyed by the local variable name.

{@a query-predicate}

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":

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.

{@a setup-files}

Test environment setup files

Unit testing requires some configuration and bootstrapping that is captured in *setup files*. The setup files for this guide are provided for you when you follow the Setup instructions. The CLI delivers similar files with the same purpose.

Here's a brief description of this guide's setup files:

The deep details of these files and how to reconfigure them for your needs is a topic beyond the scope of this guide .

File	Description
`karma.conf.js`	The karma configuration file that specifies which plug-ins to use, which application and test files to load, which browser(s) to use, and how to report test results. It loads three other setup files: * `systemjs.config.js` * `systemjs.config.extras.js` * `karma-test-shim.js`
`karma-test-shim.js`	This shim prepares karma specifically for the Angular test environment and launches karma itself. It loads the `systemjs.config.js` file as part of that process.
`systemjs.config.js`	[SystemJS] (https://github.com/systemjs/systemjs/blob/master/README.md) loads the application and test files. This script tells SystemJS where to find those files and how to load them. It's the same version of `systemjs.config.js` you installed during [setup] (guide/testing#setup).
`systemjs.config.extras.js`	An optional file that supplements the SystemJS configuration in `systemjs.config.js` with configuration for the specific needs of the application itself. A stock `systemjs.config.js` can't anticipate those needs. You fill the gaps here. The sample version for this guide adds the **model barrel** to the SystemJs `packages` configuration.

npm packages

The sample tests are written to run in Jasmine and karma. The two "fast path" setups added the appropriate Jasmine and karma npm packages to the `devDependencies` section of the `package.json`. They're installed when you run `npm install`.

## FAQ: Frequently Asked Questions

Why put specs next to the things they test?

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.

TypeScript Configuration

TypeScript is a primary language for Angular application development. It is a superset of JavaScript with design-time support for type safety and tooling.

Browsers can't execute TypeScript directly. Typescript must be "transpiled" into JavaScript using the *tsc* compiler, which requires some configuration.

This page covers some aspects of TypeScript configuration and the TypeScript environment that are important to Angular developers, including details about the following files:

- tsconfig.json—TypeScript compiler configuration.
- typings—TypesScript declaration files.

{@a tsconfig}

tsconfig.json

Typically, you add a TypeScript configuration file called `tsconfig.json` to your project to guide the compiler as it generates JavaScript files.

For details about `tsconfig.json`, see the official [TypeScript wiki] (http://www.typescriptlang.org/docs/handbook/tsconfig-json.html).

The Setup guide uses the following `tsconfig.json` :

This file contains options and flags that are essential for Angular applications.

{@a noImplicitAny}

noImplicitAny and *suppressImplicitAnyIndexErrors*

TypeScript developers disagree about whether the `noImplicitAny` flag should be `true` or `false` . There is no correct answer and you can change the flag later. But your choice now can make a difference in larger projects, so it merits discussion.

When the `noImplicitAny` flag is `false` (the default), and if the compiler cannot infer the variable type based on how it's used, the compiler silently defaults the type to `any` . That's what is meant by *implicit* *`any`* .

The documentation setup sets the `noImplicitAny` flag to `true`. When the `noImplicitAny` flag is `true` and the TypeScript compiler cannot infer the type, it still generates the JavaScript files, but it also **reports an error**. Many seasoned developers prefer this stricter setting because type checking catches more unintentional errors at compile time.

You can set a variable's type to `any` even when the `noImplicitAny` flag is `true`.

When the `noImplicitAny` flag is `true`, you may get *implicit index errors* as well. Most developers feel that *this particular error* is more annoying than helpful. You can suppress them with the following additional flag:

"suppressImplicitAnyIndexErrors":true

The documentation setup sets this flag to `true` as well.

{@a typings}

TypeScript Typings

Many JavaScript libraries, such as jQuery, the Jasmine testing library, and Angular, extend the JavaScript environment with features and syntax that the TypeScript compiler doesn't recognize natively. When the compiler doesn't recognize something, it throws an error.

Use TypeScript type definition files— `d.ts files` —to tell the compiler about the libraries you load.

TypeScript-aware editors leverage these same definition files to display type information about library features.

Many libraries include definition files in their npm packages where both the TypeScript compiler and editors can find them. Angular is one such library. The `node_modules/@angular/core/` folder of any Angular application contains several `d.ts` files that describe parts of Angular.

You need do nothing to get *typings* files for library packages that include `d.ts` files. Angular packages include them already.

lib.d.ts

TypeScript includes a special declaration file called `lib.d.ts`. This file contains the ambient declarations for various common JavaScript constructs present in JavaScript runtimes and the DOM.

Based on the `--target`, TypeScript adds *additional* ambient declarations like `Promise` if the target is `es6`.

Since the QuickStart is targeting `es5`, you can override the list of declaration files to be included:

"lib": ["es2015", "dom"]

Thanks to that, you have all the `es6` typings even when targeting `es5`.

Installable typings files

Many libraries—jQuery, Jasmine, and Lodash among them—do *not* include `d.ts` files in their npm packages. Fortunately, either their authors or community contributors have created separate `d.ts` files for these libraries and published them in well-known locations.

You can install these typings via `npm` using the `@types/*` scoped package and Typescript, starting at 2.0, automatically recognizes them.

For instance, to install typings for `jasmine` you could do `npm install @types/jasmine --save-dev`.

QuickStart identifies two *typings*, or `d.ts`, files:

- jasmine typings for the Jasmine test framework.
- node for code that references objects in the *nodejs* environment; you can view an example in the webpack page.

QuickStart doesn't require these typings but many of the samples do.

Angular Universal: server-side rendering

This guide describes **Angular Universal**, a technology that runs your Angular application on the server.

A normal Angular application executes in the *browser*, rendering pages in the DOM in response to user actions.

Angular Universal generates *static* application pages on the *server* through a process called **server-side rendering (SSR)**.

It can generate and serve those pages in response to requests from browsers. It can also pre-generate pages as HTML files that you serve later.

This guide describes a Universal sample application that launches quickly as a server-rendered page. Meanwhile, the browser downloads the full client version and switches to it automatically after the code loads.

[Download the finished sample code](generated/zips/universal/universal.zip), which runs in a [node express](https://expressjs.com/) server.

{@a why-do-it}

Why Universal

There are three main reasons to create a Universal version of your app.

1. Facilitate web crawlers (SEO)
2. Improve performance on mobile and low-powered devices
3. Show the first page quickly

{@a seo} {@a web-crawlers}

Facilitate web crawlers

Google, Bing, Facebook, Twitter and other social media sites rely on web crawlers to index your application content and make that content searchable on the web.

These web crawlers may be unable to navigate and index your highly-interactive, Angular application as a human user could do.

Angular Universal can generate a static version of your app that is easily searchable, linkable, and navigable without JavaScript. It also makes a site preview available since each URL returns a fully-rendered page.

Enabling web crawlers is often referred to as Search Engine Optimization (SEO).

{@a no-javascript}

Performance on mobile and low performance devices

Some devices don't support JavaScript or execute JavaScript so poorly that the user experience is unacceptable. For these cases, you may require a server-rendered, no-JavaScript version of the app. This version, however limited, may be the only practical alternative for people who otherwise would not be able to use the app at all.

{@a startup-performance}

Show the first page quickly

Displaying the first page quickly can be critical for user engagement.

53% of mobile site visits are abandoned if pages take longer than 3 seconds to load. Your app may have to launch faster to engage these users before they decide to do something else.

With Angular Universal, you can generate landing pages for the app that look like the complete app. The pages are pure HTML, and can display even if JavaScript is disabled. The pages do not handle browser events, but they *do* support navigation through the site using routerLink.

In practice, you'll serve a static version of the landing page to hold the user's attention. At the same time, you'll load the full Angular app behind it in the manner explained below. The user perceives near-instant performance from the landing page and gets the full interactive experience after the full app loads.

{@a how-does-it-work}

How it works

To make a Universal app, you install the `platform-server` package. The `platform-server` package has server implementations of the DOM, `XMLHttpRequest`, and other low-level features that do not rely on a browser.

You compile the client application with the `platform-server` module instead of the `platform-browser` module. and run the resulting Universal app on a web server.

The server (a Node Express server in *this* guide's example) passes client requests for application pages to Universal's `renderModuleFactory` function.

The `renderModuleFactory` function takes as inputs a *template* HTML page (usually `index.html`), an Angular *module* containing components, and a *route* that determines which components to display.

The route comes from the client's request to the server. Each request results in the appropriate view for the requested route.

The `renderModuleFactory` renders that view within the `<app>` tag of the template, creating a finished HTML page for the client.

Finally, the server returns the rendered page to the client.

Working around the browser APIs

Because a Universal `platform-server` app doesn't execute in the browser, you may have to work around some of the browser APIs and capabilities that are missing on the server.

You won't be able reference browser-only native objects such as `window`, `document`, `navigator` or `location`. If you don't need them on the server-rendered page, side-step them with conditional logic.

Alternatively, look for an injectable Angular abstraction over the object you need such as `Location` or `Document`; it may substitute adequately for the specific API that you're calling. If Angular doesn't provide it, you may be able to write your own abstraction that delegates to the browser API while in the browser and to a satisfactory alternative implementation while on the server.

Without mouse or keyboard events, a universal app can't rely on a user clicking a button to show a component. A universal app should determine what to render based solely on the incoming client request. This is a good argument for making the app routeable.

Because the user of a server-rendered page can't do much more than click links, you should swap in the real client app as quickly as possible for a proper interactive experience.

{@a the-example}

The example

The *Tour of Heroes* tutorial is the foundation for the Universal sample described in this guide.

The core application files are mostly untouched, with a few exceptions described below. You'll add more files to support building and serving with Universal.

In this example, the Angular CLI compiles and bundles the Universal version of the app with the AOT (Ahead-of-Time) compiler. A node/express web server turns client requests into the HTML pages rendered by

Universal.

You will create:

- a server-side app module, `app.server.module.ts`
- an entry point for the server-side, `main.server.ts`
- an express web server to handle requests, `server.ts`
- a TypeScript config file, `tsconfig.server.json`
- a Webpack config file for the server, `webpack.server.config.js`

When you're done, the folder structure will look like this:

src/ index.html *app web page* main.ts *bootstrapper for client app* main.server.ts * *bootstrapper for server app* tsconfig.app.json *TypeScript client configuration* tsconfig.server.json * *TypeScript server configuration* tsconfig.spec.json *TypeScript spec configuration* style.css *styles for the app* app/ ... *application code* app.server.module.ts * *server-side application module* server.ts * *express web server* tsconfig.json *TypeScript client configuration* package.json *npm configuration* webpack.server.config.js * *Webpack server configuration*

The files marked with `*` are new and not in the original tutorial sample. This guide covers them in the sections below.

{@a preparation}

Preparation

Download the Tour of Heroes project and install the dependencies from it.

{@a install-the-tools}

Install the tools

To get started, install these packages.

- `@angular/platform-server` - Universal server-side components.
- `@nguniversal/module-map-ngfactory-loader` - For handling lazy-loading in the context of a server-render.
- `@nguniversal/express-engine` - An express engine for Universal applications.
- `ts-loader` - To transpile the server application

Install them with the following commands:

```
npm install --save @angular/platform-server @nguniversal/module-map-ngfactory-loader ts-loader
```

@nguniversal/express-engine

{@a transition}

Modify the client app

A Universal app can act as a dynamic, content-rich "splash screen" that engages the user. It gives the appearance of a near-instant application.

Meanwhile, the browser downloads the client app scripts in background. Once loaded, Angular transitions from the static server-rendered page to the dynamically rendered views of the interactive client app.

You must make a few changes to your application code to support both server-side rendering and the transition to the client app.

The root `AppModule`

Open file `src/app/app.module.ts` and find the `BrowserModule` import in the `NgModule` metadata. Replace that import with this one:

Angular adds the `appId` value (which can be *any* string) to the style-names of the server-rendered pages, so that they can be identified and removed when the client app starts.

You can get runtime information about the current platform and the `appId` by injection.

{@a http-urls}

Absolute HTTP URLs

The tutorial's `HeroService` and `HeroSearchService` delegate to the Angular `Http` module to fetch application data. These services send requests to *relative* URLs such as `api/heroes`.

In a Universal app, `Http` URLs must be *absolute* (e.g., `https://my-server.com/api/heroes`) even when the Universal web server is capable of handling those requests.

You'll have to change the services to make requests with absolute URLs when running on the server and with relative URLs when running in the browser.

One solution is to provide the server's runtime origin under the Angular `APP_BASE_REF` token, inject it into the service, and prepend the origin to the request URL.

Start by changing the `HeroService` constructor to take a second `origin` parameter that is optionally injected via the `APP_BASE_HREF` token.

Note how the constructor prepends the origin (if it exists) to the `heroesUrl`.

You don't provide `APP_BASE_HREF` in the browser version, so the `heroesUrl` remains relative.

You can ignore \`APP_BASE_HREF\` in the browser if you've specified \`\` in the \`index.html\` to satisfy the router's need for a base address, as the tutorial sample does.

{@a server-code}

Server code

To run an Angular Universal application, you'll need a server that accepts client requests and returns rendered pages.

{@a app-server-module}

App server module

The app server module class (conventionally named `AppServerModule`) is an Angular module that wraps the application's root module (`AppModule`) so that Universal can mediate between your application and the server. `AppServerModule` also tells Angular how to bootstrap your application when running as a Universal app.

Create an `app.server.module.ts` file in the `src/app/` directory with the following `AppServerModule` code:

Notice that it imports first the client app's `AppModule`, the Angular Universal's `ServerModule` and the `ModuleMapLoaderModule`.

The `ModuleMapLoaderModule` is a server-side module that allows lazy-loading of routes.

This is also the place to register providers that are specific to running your app under Universal.

{@a web-server}

Universal web server

A *Universal* web server responds to application *page* requests with static HTML rendered by the [Universal template engine](#).

It receives and responds to HTTP requests from clients (usually browsers). It serves static assets such as scripts, css, and images. It may respond to data requests, perhaps directly or as a proxy to a separate data server.

The sample web server for *this* guide is based on the popular Express framework.

Any web server technology can serve a Universal app as long as it can call Universal's `renderModuleFactory`. The principles and decision points discussed below apply to any web server technology that you chose.

Create a `server.ts` file in the root directory and add the following code:

This sample server is not secure! Be sure to add middleware to authenticate and authorize users just as you would for a normal Angular application server.

{@a universal-engine}

Universal template engine

The important bit in this file is the `ngExpressEngine` function:

The `ngExpressEngine` is a wrapper around the universal's `renderModuleFactory` function that turns a client's requests into server-rendered HTML pages. You'll call that function within a *template engine* that's appropriate for your server stack.

The first parameter is the `AppServerModule` that you wrote earlier. It's the bridge between the Universal server-side renderer and your application.

The second parameter is the `extraProviders`. It is an optional Angular dependency injection providers, applicable when running on this server.

{@a provide-origin}

You supply `extraProviders` when your app needs information that can only be determined by the currently running server instance.

The required information in this case is the running server's origin, provided under the `APP_BASE_HREF` token, so that the app can calculate absolute HTTP URLs.

The `ngExpressEngine` function returns a *promise* that resolves to the rendered page.

It's up to your engine to decide what to do with that page. *This engine's* promise callback returns the rendered page to the web server, which then forwards it to the client in the HTTP response.

This wrappers are very useful to hide the complexity of the `renderModuleFactory`. There are more wrappers for different backend technologies at the [Universal repository](https://github.com/angular/universal).

Filter request URLs

The web server must distinguish *app page requests* from other kinds of requests.

It's not as simple as intercepting a request to the root address `/`. The browser could ask for one of the application routes such as `/dashboard`, `/heroes`, or `/detail:12`. In fact, if the app were *only* rendered by the server, *every* app link clicked would arrive at the server as a navigation URL intended for the router.

Fortunately, application routes have something in common: their URLs lack file extensions.

Data requests also lack extensions but they're easy to recognize because they always begin with `/api`.

All static asset requests have a file extension (e.g., `main.js` or `/node_modules/zone.js/dist/zone.js`).

So we can easily recognize the three types of requests and handle them differently.

1. data request - request URL that begins `/api`
2. app navigation - request URL with no file extension
3. static asset - all other requests.

An Express server is a pipeline of middleware that filters and processes URL requests one after the other.

You configure the Express server pipeline with calls to `app.get()` like this one for data requests.

This sample server doesn't handle data requests. The tutorial's "in-memory web api" module, a demo and development tool, intercepts all HTTP calls and simulates the behavior of a remote data server. In practice, you would remove that module and register your web api middleware on the server here.
Universal HTTP requests have different security requirements HTTP requests issued from a browser app are not the same as when issued by the universal app on the server. When a browser makes an HTTP request, the server can make assumptions about cookies, XSRF headers, etc. For example, the browser automatically sends auth cookies for the current user. Angular Universal cannot forward these credentials to a separate data server. If your server handles HTTP requests, you'll have to add your own security plumbing.

The following code filters for request URLs with no extensions and treats them as navigation requests.

Serve static files safely

A single `app.use()` treats all other URLs as requests for static assets such as JavaScript, image, and style files.

To ensure that clients can only download the files that they are *permitted* to see, you will [put all client-facing asset files in the `/dist` folder and will only honor requests for files from the `/dist` folder.

The following express code routes all remaining requests to `/dist`; it returns a `404 - NOT FOUND` if the file is not found.

{@a universal-configuration}

Configure for Universal

The server application requires its own build configuration.

{@a universal-typescript-configuration}

Universal TypeScript configuration

Create a `tsconfig.server.json` file in the project root directory to configure TypeScript and AOT compilation of the universal app.

This config extends from the root's `tsconfig.json` file. Certain settings are noteworthy for their differences.

- The `module` property must be **commonjs** which can be require()'d into our server application.
- The `angularCompilerOptions` section guides the AOT compiler:
    - `entryModule` - the root module of the server application, expressed as `path/to/file#ClassName`.

Universal Webpack configuration

Universal applications doesn't need any extra Webpack configuration, the CLI takes care of that for you, but since the server is a typescript application, you will use Webpack to transpile it.

Create a `webpack.server.config.js` file in the project root directory with the following code.

Webpack configuration is a rich topic beyond the scope of this guide.

Build and run with universal

Now that you've created the TypeScript and Webpack config files, you can build and run the Universal application.

First add the *build* and *serve* commands to the `scripts` section of the `package.json` :

"scripts": { ... "build:universal": "npm run build:client-and-server-bundles && npm run webpack:server", "serve:universal": "node dist/server.js", "build:client-and-server-bundles": "ng build --prod && ng build --prod --app 1 --output-hashing=false", "webpack:server": "webpack --config webpack.server.config.js --progress --colors" ... }

{@a build}

Build

From the command prompt, type

npm run build:universal

The Angular CLI compiles and bundles the universal app into two different folders, `browser` and `server` . Webpack transpiles the `server.ts` file into Javascript.

{@a serve}

Serve

After building the application, start the server.

npm run serve:universal

The console window should say

Node server listening on http://localhost:4000

Universal in action

Open a browser to http://localhost:4000/. You should see the familiar Tour of Heroes dashboard page.

Navigation via `routerLinks` works correctly. You can go from the Dashboard to the Heroes page and back. You can click on a hero on the Dashboard page to display its Details page.

But clicks, mouse-moves, and keyboard entries are inert.

- Clicking a hero on the Heroes page does nothing.
- You can't add or delete a hero.
- The search box on the Dashboard page is ignored.
- The *back* and *save* buttons on the Details page don't work.

User events other than `routerLink` clicks aren't supported. The user must wait for the full client app to arrive.

It will never arrive until you compile the client app and move the output into the `dist/` folder, a step you'll take in just a moment.

Throttling

The transition from the server-rendered app to the client app happens quickly on a development machine. You can simulate a slower network to see the transition more clearly and better appreciate the launch-speed advantage of a universal app running on a low powered, poorly connected device.

Open the Chrome Dev Tools and go to the Network tab. Find the Network Throttling dropdown on the far right of the menu bar.

Try one of the "3G" speeds. The server-rendered app still launches quickly but the full client app may take seconds to load.

{@a summary}

Summary

This guide showed you how to take an existing Angular application and make it into a Universal app that does server-side rendering. It also explained some of the key reasons for doing so.

- Facilitate web crawlers (SEO)
- Support low-bandwidth or low-power devices
- Fast first page load

Angular Universal can greatly improve the perceived startup performance of your app. The slower the network, the more advantageous it becomes to have Universal display the first page to the user.

Upgrading from AngularJS

Angular is the name for the Angular of today and tomorrow. *AngularJS* is the name for all v1.x versions of Angular.

AngularJS apps are great. Always consider the business case before moving to Angular. An important part of that case is the time and effort to get there. This guide describes the built-in tools for efficiently migrating AngularJS projects over to the Angular platform, a piece at a time.

Some applications will be easier to upgrade than others, and there are many ways to make it easier for yourself. It is possible to prepare and align AngularJS applications with Angular even before beginning the upgrade process. These preparation steps are all about making the code more decoupled, more maintainable, and better aligned with modern development tools. That means in addition to making the upgrade easier, you will also improve the existing AngularJS applications.

One of the keys to a successful upgrade is to do it incrementally, by running the two frameworks side by side in the same application, and porting AngularJS components to Angular one by one. This makes it possible to upgrade even large and complex applications without disrupting other business, because the work can be done collaboratively and spread over a period of time. The `upgrade` module in Angular has been designed to make incremental upgrading seamless.

Preparation

There are many ways to structure AngularJS applications. When you begin to upgrade these applications to Angular, some will turn out to be much more easy to work with than others. There are a few key techniques and patterns that you can apply to future proof apps even before you begin the migration.

{@a follow-the-angular-styleguide}

Follow the AngularJS Style Guide

The AngularJS Style Guide collects patterns and practices that have been proven to result in cleaner and more maintainable AngularJS applications. It contains a wealth of information about how to write and organize AngularJS code - and equally importantly - how **not** to write and organize AngularJS code.

Angular is a reimagined version of the best parts of AngularJS. In that sense, its goals are the same as the AngularJS Style Guide's: To preserve the good parts of AngularJS, and to avoid the bad parts. There's a lot more to Angular than just that of course, but this does mean that *following the style guide helps make your*

AngularJS app more closely aligned with Angular.

There are a few rules in particular that will make it much easier to do *an incremental upgrade* using the Angular `upgrade/static` module:

- The Rule of 1 states that there should be one component per file. This not only makes components easy to navigate and find, but will also allow us to migrate them between languages and frameworks one at a time. In this example application, each controller, component, service, and filter is in its own source file.

- The Folders-by-Feature Structure and Modularity rules define similar principles on a higher level of abstraction: Different parts of the application should reside in different directories and NgModules.

When an application is laid out feature per feature in this way, it can also be migrated one feature at a time. For applications that don't already look like this, applying the rules in the AngularJS style guide is a highly recommended preparation step. And this is not just for the sake of the upgrade - it is just solid advice in general!

Using a Module Loader

When you break application code down into one component per file, you often end up with a project structure with a large number of relatively small files. This is a much neater way to organize things than a small number of large files, but it doesn't work that well if you have to load all those files to the HTML page with <script> tags. Especially when you also have to maintain those tags in the correct order. That's why it's a good idea to start using a *module loader*.

Using a module loader such as SystemJS, Webpack, or Browserify allows us to use the built-in module systems of TypeScript or ES2015. You can use the `import` and `export` features that explicitly specify what code can and will be shared between different parts of the application. For ES5 applications you can use CommonJS style `require` and `module.exports` features. In both cases, the module loader will then take care of loading all the code the application needs in the correct order.

When moving applications into production, module loaders also make it easier to package them all up into production bundles with batteries included.

Migrating to TypeScript

If part of the Angular upgrade plan is to also take TypeScript into use, it makes sense to bring in the TypeScript compiler even before the upgrade itself begins. This means there's one less thing to learn and think about during the actual upgrade. It also means you can start using TypeScript features in your AngularJS code.

Since TypeScript is a superset of ECMAScript 2015, which in turn is a superset of ECMAScript 5, "switching" to

TypeScript doesn't necessarily require anything more than installing the TypeScript compiler and renaming files from `*.js` to `*.ts`. But just doing that is not hugely useful or exciting, of course. Additional steps like the following can give us much more bang for the buck:

- For applications that use a module loader, TypeScript imports and exports (which are really ECMAScript 2015 imports and exports) can be used to organize code into modules.
- Type annotations can be gradually added to existing functions and variables to pin down their types and get benefits like build-time error checking, great autocompletion support and inline documentation.
- JavaScript features new to ES2015, like arrow functions, `let`s and `const`s, default function parameters, and destructuring assignments can also be gradually added to make the code more expressive.
- Services and controllers can be turned into *classes*. That way they'll be a step closer to becoming Angular service and component classes, which will make life easier after the upgrade.

Using Component Directives

In Angular, components are the main primitive from which user interfaces are built. You define the different portions of the UI as components and compose them into a full user experience.

You can also do this in AngularJS, using *component directives*. These are directives that define their own templates, controllers, and input/output bindings - the same things that Angular components define. Applications built with component directives are much easier to migrate to Angular than applications built with lower-level features like `ng-controller`, `ng-include`, and scope inheritance.

To be Angular compatible, an AngularJS component directive should configure these attributes:

- `restrict: 'E'`. Components are usually used as elements.
- `scope: {}` - an isolate scope. In Angular, components are always isolated from their surroundings, and you should do this in AngularJS too.
- `bindToController: {}`. Component inputs and outputs should be bound to the controller instead of using the `$scope`.
- `controller` and `controllerAs`. Components have their own controllers.
- `template` or `templateUrl`. Components have their own templates.

Component directives may also use the following attributes:

- `transclude: true/{}`, if the component needs to transclude content from elsewhere.
- `require`, if the component needs to communicate with some parent component's controller.

Component directives **should not** use the following attributes:

- `compile`. This will not be supported in Angular.
- `replace: true`. Angular never replaces a component element with the component template. This attribute is also deprecated in AngularJS.
- `priority` and `terminal`. While AngularJS components may use these, they are not used in Angular and it is better not to write code that relies on them.

An AngularJS component directive that is fully aligned with the Angular architecture may look something like this:

AngularJS 1.5 introduces the component API that makes it easier to define component directives like these. It is a good idea to use this API for component directives for several reasons:

- It requires less boilerplate code.
- It enforces the use of component best practices like `controllerAs`.
- It has good default values for directive attributes like `scope` and `restrict`.

The component directive example from above looks like this when expressed using the component API:

Controller lifecycle hook methods `$onInit()`, `$onDestroy()`, and `$onChanges()` are another convenient feature that AngularJS 1.5 introduces. They all have nearly exact equivalents in Angular, so organizing component lifecycle logic around them will ease the eventual Angular upgrade process.

Upgrading with ngUpgrade

The ngUpgrade library in Angular is a very useful tool for upgrading anything but the smallest of applications. With it you can mix and match AngularJS and Angular components in the same application and have them interoperate seamlessly. That means you don't have to do the upgrade work all at once, since there's a natural coexistence between the two frameworks during the transition period.

How ngUpgrade Works

The primary tool provided by ngUpgrade is called the `UpgradeModule`. This is a module that contains utilities for bootstrapping and managing hybrid applications that support both Angular and AngularJS code.

When you use ngUpgrade, what you're really doing is *running both AngularJS and Angular at the same time*. All Angular code is running in the Angular framework, and AngularJS code in the AngularJS framework. Both of these are the actual, fully featured versions of the frameworks. There is no emulation going on, so you can expect to have all the features and natural behavior of both frameworks.

What happens on top of this is that components and services managed by one framework can interoperate with those from the other framework. This happens in three main areas: Dependency injection, the DOM, and change detection.

Dependency Injection

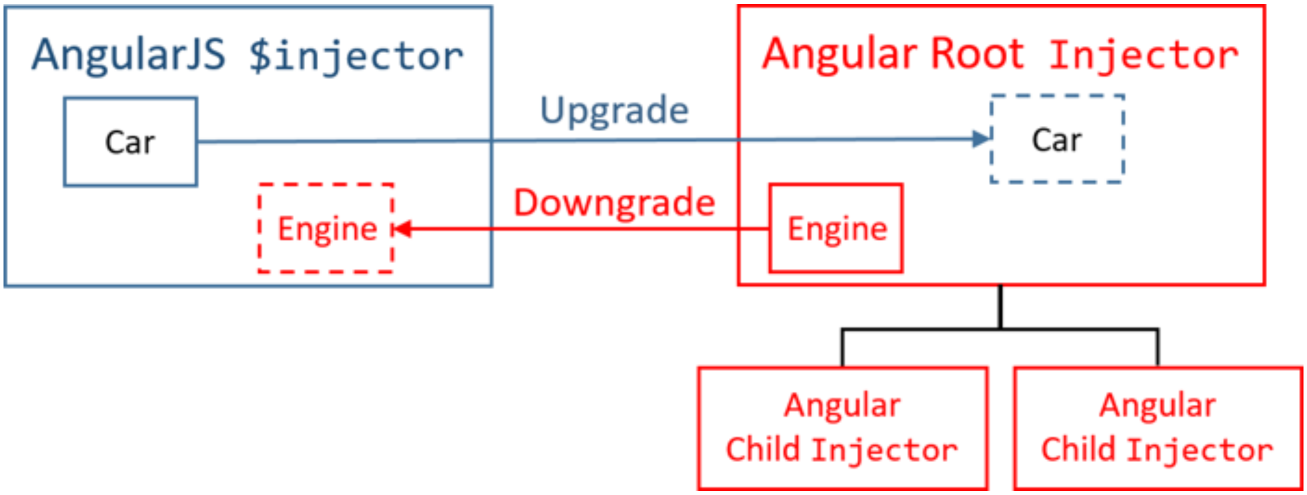
Dependency injection is front and center in both AngularJS and Angular, but there are some key differences between the two frameworks in how it actually works.

AngularJS	Angular
Dependency injection tokens are always strings	Tokens [can have different types](guide/dependency-injection). They are often classes. They may also be strings.
There is exactly one injector. Even in multi-module applications, everything is poured into one big namespace.	There is a [tree hierarchy of injectors](guide/hierarchical-dependency-injection), with a root injector and an additional injector for each component.

Even accounting for these differences you can still have dependency injection interoperability. The `UpgradeModule` resolves the differences and makes everything work seamlessly:

- You can make AngularJS services available for injection to Angular code by *upgrading* them. The same singleton instance of each service is shared between the frameworks. In Angular these services will always be in the *root injector* and available to all components.

- You can also make Angular services available for injection to AngularJS code by *downgrading* them. Only services from the Angular root injector can be downgraded. Again, the same singleton instances are shared between the frameworks. When you register a downgraded service, you must explicitly specify a *string token* that you want to use in AngularJS.



Components and the DOM

In the DOM of a hybrid ngUpgrade application are components and directives from both AngularJS and Angular. These components communicate with each other by using the input and output bindings of their respective frameworks, which ngUpgrade bridges together. They may also communicate through shared injected dependencies, as described above.

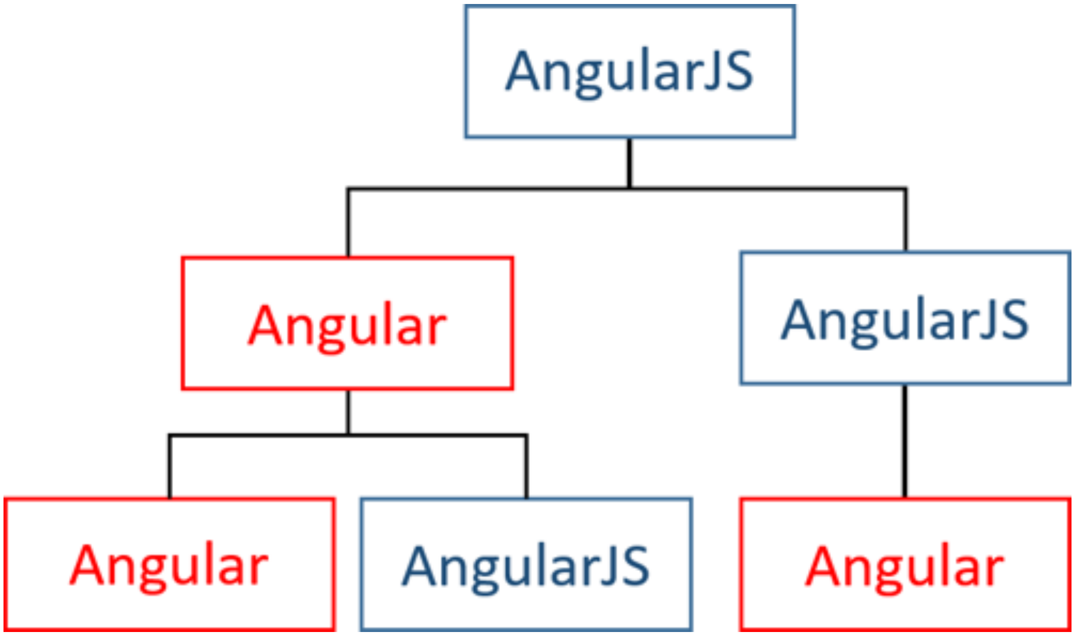
The key thing to understand about a hybrid application is that every element in the DOM is owned by exactly one of the two frameworks. The other framework ignores it. If an element is owned by AngularJS, Angular treats it as if it didn't exist, and vice versa.

So normally a hybrid application begins life as an AngularJS application, and it is AngularJS that processes the root template, e.g. the index.html. Angular then steps into the picture when an Angular component is used somewhere in an AngularJS template. That component's template will then be managed by Angular, and it may contain any number of Angular components and directives.

Beyond that, you may interleave the two frameworks. You always cross the boundary between the two frameworks by one of two ways:

1. By using a component from the other framework: An AngularJS template using an Angular component, or an Angular template using an AngularJS component.

2. By transcluding or projecting content from the other framework. ngUpgrade bridges the related concepts of AngularJS transclusion and Angular content projection together.



Whenever you use a component that belongs to the other framework, a switch between framework boundaries occurs. However, that switch only happens to the elements in the template of that component. Consider a situation where you use an Angular component from AngularJS like this:

<a-component></a-component>

The DOM element `<a-component>` will remain to be an AngularJS managed element, because it's defined in an AngularJS template. That also means you can apply additional AngularJS directives to it, but *not* Angular directives. It is only in the template of the `<a-component>` where Angular steps in. This same rule also applies when you use AngularJS component directives from Angular.

Change Detection

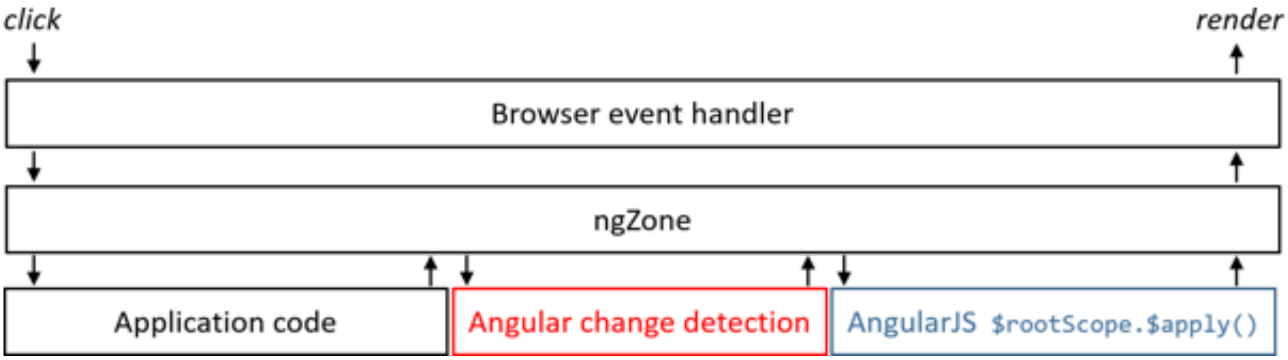
The `scope.$apply()` is how AngularJS detects changes and updates data bindings. After every event that occurs, `scope.$apply()` gets called. This is done either automatically by the framework, or manually by you.

In Angular things are different. While change detection still occurs after every event, no one needs to call `scope.$apply()` for that to happen. This is because all Angular code runs inside something called the Angular zone. Angular always knows when the code finishes, so it also knows when it should kick off change detection. The code itself doesn't have to call `scope.$apply()` or anything like it.

In the case of hybrid applications, the `UpgradeModule` bridges the AngularJS and Angular approaches. Here's what happens:

- Everything that happens in the application runs inside the Angular zone. This is true whether the event originated in AngularJS or Angular code. The zone triggers Angular change detection after every event.

- The `UpgradeModule` will invoke the AngularJS `$rootScope.$apply()` after every turn of the Angular zone. This also triggers AngularJS change detection after every event.



In practice, you do not need to call `$apply()`, regardless of whether it is in AngularJS on Angular. The `UpgradeModule` does it for us. You *can* still call `$apply()` so there is no need to remove such calls from existing code. Those calls just trigger additional AngularJS change detection checks in a hybrid application.

When you downgrade an Angular component and then use it from AngularJS, the component's inputs will be watched using AngularJS change detection. When those inputs change, the corresponding properties in the component are set. You can also hook into the changes by implementing the OnChanges interface in the component, just like you could if it hadn't been downgraded.

Correspondingly, when you upgrade an AngularJS component and use it from Angular, all the bindings defined for the component directive's `scope` (or `bindToController`) will be hooked into Angular change detection. They will be treated as regular Angular inputs. Their values will be written to the upgraded component's scope (or controller) when they change.

Using UpgradeModule with Angular *NgModules*

Both AngularJS and Angular have their own concept of modules to help organize an application into cohesive blocks of functionality.

Their details are quite different in architecture and implementation. In AngularJS, you add Angular assets to the `angular.module` property. In Angular, you create one or more classes adorned with an `NgModule` decorator that describes Angular assets in metadata. The differences blossom from there.

In a hybrid application you run both versions of Angular at the same time. That means that you need at least one module each from both AngularJS and Angular. You will import `UpgradeModule` inside the NgModule, and then use it for bootstrapping the AngularJS module.

Read more about [NgModules](guide/ngmodule).

Bootstrapping hybrid applications

To bootstrap a hybrid application, you must bootstrap each of the Angular and AngularJS parts of the application. You must bootstrap the Angular bits first and then ask the `UpgradeModule` to bootstrap the AngularJS bits next.

In an AngularJS application you have a root AngularJS module, which will also be used to bootstrap the AngularJS application.

Pure AngularJS applications can be automatically bootstrapped by using an `ng-app` directive somewhere on the HTML page. But for hybrid applications, you manually bootstrap via the `UpgradeModule`. Therefore, it is a good preliminary step to switch AngularJS applications to use the manual JavaScript [`angular.bootstrap`] method even before switching them to hybrid mode.

Say you have an `ng-app` driven bootstrap such as this one:

You can remove the `ng-app` and `ng-strict-di` directives from the HTML and instead switch to calling `angular.bootstrap` from JavaScript, which will result in the same thing:

To begin converting your AngularJS application to a hybrid, you need to load the Angular framework. You can see how this can be done with SystemJS by following the instructions in Setup, selectively copying code from the QuickStart github repository.

You also need to install the `@angular/upgrade` package via `npm install @angular/upgrade --save` and add a mapping for the `@angular/upgrade/static` package:

Next, create an `app.module.ts` file and add the following `NgModule` class:

This bare minimum `NgModule` imports `BrowserModule`, the module every Angular browser-based app must have. It also imports `UpgradeModule` from `@angular/upgrade/static`, which exports providers that will be used for upgrading and downgrading services and components.

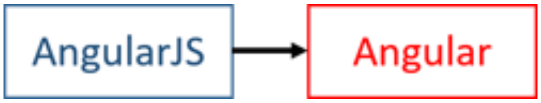
In the constructor of the `AppModule`, use dependency injection to get a hold of the `UpgradeModule` instance, and use it to bootstrap the AngularJS app in the `AppModule.ngDoBootstrap` method. The `upgrade.bootstrap` method takes the exact same arguments as angular.bootstrap:

Note that you do not add a `bootstrap` declaration to the `@NgModule` decorator, since AngularJS will own the root template of the application.

Now you can bootstrap `AppModule` using the `platformBrowserDynamic.bootstrapModule` method.

Congratulations! You're running a hybrid application! The existing AngularJS code works as before *and* you're ready to start adding Angular code.

Using Angular Components from AngularJS Code



Once you're running a hybrid app, you can start the gradual process of upgrading code. One of the more common patterns for doing that is to use an Angular component in an AngularJS context. This could be a completely new component or one that was previously AngularJS but has been rewritten for Angular.

Say you have a simple Angular component that shows information about a hero:

If you want to use this component from AngularJS, you need to *downgrade* it using the `downgradeComponent()` method. The result is an AngularJS *directive*, which you can then register in the AngularJS module:

Because `HeroDetailComponent` is an Angular component, you must also add it to the `declarations` in the `AppModule`.

And because this component is being used from the AngularJS module, and is an entry point into the Angular application, you must add it to the `entryComponents` for the NgModule.

All Angular components, directives and pipes must be declared in an NgModule.

The net result is an AngularJS directive called `heroDetail`, that you can use like any other directive in AngularJS templates.

Note that this AngularJS is an element directive (`restrict: 'E'`) called `heroDetail`. An AngularJS element directive is matched based on its _name_. *The `selector` metadata of the downgraded Angular component is ignored.*

Most components are not quite this simple, of course. Many of them have *inputs and outputs* that connect them to the outside world. An Angular hero detail component with inputs and outputs might look like this:

These inputs and outputs can be supplied from the AngularJS template, and the `downgradeComponent()` method takes care of wiring them up:

Note that even though you are in an AngularJS template, **you're using Angular attribute syntax to bind the**

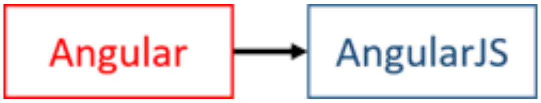
inputs and outputs. This is a requirement for downgraded components. The expressions themselves are still regular AngularJS expressions.

Use kebab-case for downgraded component attributes
There's one notable exception to the rule of using Angular attribute syntax for downgraded components. It has to do with input or output names that consist of multiple words. In Angular, you would bind these attributes using camelCase: [myHero]="hero" But when using them from AngularJS templates, you must use kebab-case: [my-hero]="hero"

The `$event` variable can be used in outputs to gain access to the object that was emitted. In this case it will be the `Hero` object, because that is what was passed to `this.deleted.emit()`.

Since this is an AngularJS template, you can still use other AngularJS directives on the element, even though it has Angular binding attributes on it. For example, you can easily make multiple copies of the component using `ng-repeat`:

Using AngularJS Component Directives from Angular Code



So, you can write an Angular component and then use it from AngularJS code. This is useful when you start to migrate from lower-level components and work your way up. But in some cases it is more convenient to do things in the opposite order: To start with higher-level components and work your way down. This too can be done using the `UpgradeModule`. You can *upgrade* AngularJS component directives and then use them from Angular.

Not all kinds of AngularJS directives can be upgraded. The directive really has to be a *component directive*, with the characteristics described in the preparation guide above. The safest bet for ensuring compatibility is using the component API introduced in AngularJS 1.5.

A simple example of an upgradable component is one that just has a template and a controller:

You can *upgrade* this component to Angular using the `UpgradeComponent` class. By creating a new Angular **directive** that extends `UpgradeComponent` and doing a `super` call inside it's constructor, you have a fully upgraded AngularJS component to be used inside Angular. All that is left is to add it to `AppModule`'s `declarations` array.

Upgraded components are Angular **directives**, instead of **components**, because Angular is unaware that AngularJS will create elements under it. As far as Angular knows, the upgraded component is just a directive - a tag - and Angular doesn't have to concern itself with it's children.

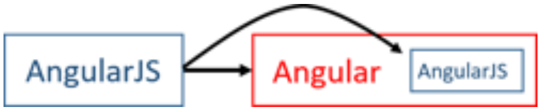
An upgraded component may also have inputs and outputs, as defined by the scope/controller bindings of the original AngularJS component directive. When you use the component from an Angular template, provide the inputs and outputs using **Angular template syntax**, observing the following rules:

	Binding definition	Template syntax
Attribute binding	`myAttribute: '@myAttribute'`	``
Expression binding	`myOutput: '&myOutput'`	``
One-way binding	`myValue: '	``
Two-way binding	`myValue: '=myValue'`	As a two-way binding: ``. Since most AngularJS two-way bindings actually only need a one-way binding in practice, `` is often enough.

For example, imagine a hero detail AngularJS component directive with one input and one output:

You can upgrade this component to Angular, annotate inputs and outputs in the upgrade directive, and then provide the input and output using Angular template syntax:

Projecting AngularJS Content into Angular Components



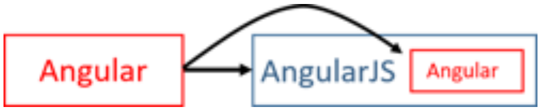
When you are using a downgraded Angular component from an AngularJS template, the need may arise to *transclude* some content into it. This is also possible. While there is no such thing as transclusion in Angular, there is a very similar concept called *content projection*. The `UpgradeModule` is able to make these two features interoperate.

Angular components that support content projection make use of an `<ng-content>` tag within them. Here's an example of such a component:

When using the component from AngularJS, you can supply contents for it. Just like they would be transcluded in AngularJS, they get projected to the location of the `<ng-content>` tag in Angular:

When AngularJS content gets projected inside an Angular component, it still remains in "AngularJS land" and is managed by the AngularJS framework.

Transcluding Angular Content into AngularJS Component Directives



Just as you can project AngularJS content into Angular components, you can *transclude* Angular content into AngularJS components, whenever you are using upgraded versions from them.

When an AngularJS component directive supports transclusion, it may use the `ng-transclude` directive in its template to mark the transclusion point:

If you upgrade this component and use it from Angular, you can populate the component tag with contents that will then get transcluded:

Making AngularJS Dependencies Injectable to Angular

When running a hybrid app, you may encounter situations where you need to inject some AngularJS dependencies into your Angular code. Maybe you have some business logic still in AngularJS services. Maybe you want access to AngularJS's built-in services like `$location` or `$timeout`.

In these situations, it is possible to *upgrade* an AngularJS provider to Angular. This makes it possible to then inject it somewhere in Angular code. For example, you might have a service called `HeroesService` in AngularJS:

You can upgrade the service using a Angular factory provider that requests the service from the AngularJS `$injector`.

Many developers prefer to declare the factory provider in a separate `ajs-upgraded-providers.ts` file so that they are all together, making it easier to reference them, create new ones and delete them once the upgrade is over.

It's also recommended to export the `heroesServiceFactory` function so that Ahead-of-Time compilation can pick it up.

You can then inject it in Angular using it's class as a type annotation:

In this example you upgraded a service class. You can use a TypeScript type annotation when you inject it. While it doesn't affect how the dependency is handled, it enables the benefits of static type checking. This is not required though, and any AngularJS service, factory, or provider can be upgraded.

Making Angular Dependencies Injectable to AngularJS

In addition to upgrading AngularJS dependencies, you can also *downgrade* Angular dependencies, so that you can use them from AngularJS. This can be useful when you start migrating services to Angular or creating new services in Angular while retaining components written in AngularJS.

For example, you might have an Angular service called `Heroes` :

Again, as with Angular components, register the provider with the `NgModule` by adding it to the module's `providers` list.

Now wrap the Angular `Heroes` in an *AngularJS factory function* using `downgradeInjectable()` and plug the factory into an AngularJS module. The name of the AngularJS dependency is up to you:

After this, the service is injectable anywhere in AngularJS code:

Using Ahead-of-time compilation with hybrid apps

You can take advantage of Ahead-of-time (AOT) compilation on hybrid apps just like on any other Angular application. The setup for an hybrid app is mostly the same as described in the Ahead-of-time Compilation chapter save for differences in `index.html` and `main-aot.ts`

The `index.html` will likely have script tags loading AngularJS files, so the `index.html` for AOT must also load those files. An easy way to copy them is by adding each to the `copy-dist-files.js` file.

You'll need to use the generated `AppModuleFactory` , instead of the original `AppModule` to bootstrap the hybrid app:

And that's all you need do to get the full benefit of AOT for Angular apps!

PhoneCat Upgrade Tutorial

In this section, you'll learn to prepare and upgrade an application with `ngUpgrade` . The example app is Angular PhoneCat from the original AngularJS tutorial, which is where many of us began our Angular adventures. Now you'll see how to bring that application to the brave new world of Angular.

During the process you'll learn how to apply the steps outlined in the preparation guide. You'll align the application with Angular and also start writing in TypeScript.

To follow along with the tutorial, clone the angular-phonecat repository and apply the steps as you go.

In terms of project structure, this is where the work begins:

angular-phonecat

bower.json

karma.conf.js

package.json

app

core

checkmark

checkmark.filter.js

checkmark.filter.spec.js

phone

phone.module.js

phone.service.js

phone.service.spec.js

core.module.js

phone-detail

phone-detail.component.js

phone-detail.component.spec.js

phone-detail.module.js

phone-detail.template.html

phone-list

phone-list.component.js

phone-list.component.spec.js

phone-list.module.js

phone-list.template.html

img

...

phones

...

app.animations.js

app.config.js

app.css

app.module.js

index.html

e2e-tests

protractor-conf.js

scenarios.js

This is actually a pretty good starting point. The code uses the AngularJS 1.5 component API and the organization follows the AngularJS Style Guide, which is an important preparation step before a successful upgrade.

- Each component, service, and filter is in its own source file, as per the Rule of 1.
- The `core`, `phone-detail`, and `phone-list` modules are each in their own subdirectory. Those subdirectories contain the JavaScript code as well as the HTML templates that go with each particular feature. This is in line with the Folders-by-Feature Structure and Modularity rules.
- Unit tests are located side-by-side with application code where they are easily found, as described in the rules for Organizing Tests.

Switching to TypeScript

Since you're going to be writing Angular code in TypeScript, it makes sense to bring in the TypeScript compiler even before you begin upgrading.

You'll also start to gradually phase out the Bower package manager in favor of NPM, installing all new dependencies using NPM, and eventually removing Bower from the project.

Begin by installing TypeScript to the project.

```
npm i typescript --save-dev
```

Install type definitions for the existing libraries that you're using but that don't come with prepackaged types: AngularJS and the Jasmine unit test framework.

```
npm install @types/jasmine @types/angular @types/angular-animate @types/angular-cookies @types/angular-mocks @types/angular-resource @types/angular-route @types/angular-sanitize --save-dev
```

You should also configure the TypeScript compiler with a `tsconfig.json` in the project directory as described in the TypeScript Configuration guide. The `tsconfig.json` file tells the TypeScript compiler how to turn your TypeScript files into ES5 code bundled into CommonJS modules.

Finally, you should add some npm scripts in `package.json` to compile the TypeScript files to JavaScript (based on the `tsconfig.json` configuration file):

```
"script": { "tsc": "tsc", "tsc:w": "tsc -w", ...
```

Now launch the TypeScript compiler from the command line in watch mode:

```
npm run tsc:w
```

Keep this process running in the background, watching and recompiling as you make changes.

Next, convert your current JavaScript files into TypeScript. Since TypeScript is a super-set of ECMAScript

2015, which in turn is a super-set of ECMAScript 5, you can simply switch the file extensions from `.js` to `.ts` and everything will work just like it did before. As the TypeScript compiler runs, it emits the corresponding `.js` file for every `.ts` file and the compiled JavaScript is what actually gets executed. If you start the project HTTP server with `npm start`, you should see the fully functional application in your browser.

Now that you have TypeScript though, you can start benefiting from some of its features. There's a lot of value the language can provide to AngularJS applications.

For one thing, TypeScript is a superset of ES2015. Any app that has previously been written in ES5 - like the PhoneCat example has - can with TypeScript start incorporating all of the JavaScript features that are new to ES2015. These include things like `let`s and `const`s, arrow functions, default function parameters, and destructuring assignments.

Another thing you can do is start adding *type safety* to your code. This has actually partially already happened because of the AngularJS typings you installed. TypeScript are checking that you are calling AngularJS APIs correctly when you do things like register components to Angular modules.

But you can also start adding *type annotations* to get even more out of TypeScript's type system. For instance, you can annotate the checkmark filter so that it explicitly expects booleans as arguments. This makes it clearer what the filter is supposed to do.

In the `Phone` service, you can explicitly annotate the `$resource` service dependency as an `angular.resource.IResourceService` - a type defined by the AngularJS typings.

You can apply the same trick to the application's route configuration file in `app.config.ts`, where you are using the location and route services. By annotating them accordingly TypeScript can verify you're calling their APIs with the correct kinds of arguments.

The [AngularJS 1.x type definitions](https://www.npmjs.com/package/@types/angular) you installed are not officially maintained by the Angular team, but are quite comprehensive. It is possible to make an AngularJS 1.x application fully type-annotated with the help of these definitions. If this is something you wanted to do, it would be a good idea to enable the \`noImplicitAny\` configuration option in \`tsconfig.json\`. This would cause the TypeScript compiler to display a warning when there's any code that does not yet have type annotations. You could use it as a guide to inform us about how close you are to having a fully annotated project.

Another TypeScript feature you can make use of is *classes*. In particular, you can turn component controllers into classes. That way they'll be a step closer to becoming Angular component classes, which will make life easier once you upgrade.

AngularJS expects controllers to be constructor functions. That's exactly what ES2015/TypeScript classes are under the hood, so that means you can just plug in a class as a component controller and AngularJS will happily use it.

Here's what the new class for the phone list component controller looks like:

What was previously done in the controller function is now done in the class constructor function. The dependency injection annotations are attached to the class using a static property `$inject`. At runtime this becomes the `PhoneListController.$inject` property.

The class additionally declares three members: The array of phones, the name of the current sort key, and the search query. These are all things you have already been attaching to the controller but that weren't explicitly declared anywhere. The last one of these isn't actually used in the TypeScript code since it's only referred to in the template, but for the sake of clarity you should define all of the controller members.

In the Phone detail controller, you'll have two members: One for the phone that the user is looking at and another for the URL of the currently displayed image:

This makes the controller code look a lot more like Angular already. You're all set to actually introduce Angular into the project.

If you had any AngularJS services in the project, those would also be a good candidate for converting to classes, since like controllers, they're also constructor functions. But you only have the `Phone` factory in this project, and that's a bit special since it's an `ngResource` factory. So you won't be doing anything to it in the preparation stage. You'll instead turn it directly into an Angular service.

Installing Angular

Having completed the preparation work, get going with the Angular upgrade of PhoneCat. You'll do this incrementally with the help of ngUpgrade that comes with Angular. By the time you're done, you'll be able to remove AngularJS from the project completely, but the key is to do this piece by piece without breaking the application.

The project also contains some animations. You won't upgrade them in this version of the guide. Turn to the [Angular animations](guide/animations) guide to learn about that.

Install Angular into the project, along with the SystemJS module loader. Take a look at the results of the Setup instructions and get the following configurations from there:

- Add Angular and the other new dependencies to `package.json`
- The SystemJS configuration file `systemjs.config.js` to the project root directory.

Once these are done, run:

npm install

Soon you can load Angular dependencies into the application via `index.html`, but first you need to do some directory path adjustments. You'll need to load files from `node_modules` and the project root instead of from the `/app` directory as you've been doing to this point.

Move the `app/index.html` file to the project root directory. Then change the development server root path in `package.json` to also point to the project root instead of `app`:

"start": "http-server ./ -a localhost -p 8000 -c-1",

Now you're able to serve everything from the project root to the web browser. But you do *not* want to have to change all the image and data paths used in the application code to match the development setup. For that reason, you'll add a `<base>` tag to `index.html`, which will cause relative URLs to be resolved back to the `/app` directory:

Now you can load Angular via SystemJS. You'll add the Angular polyfills and the SystemJS config to the end of the `<head>` section, and then you'll use `System.import` to load the actual application:

You also need to make a couple of adjustments to the `systemjs.config.js` file installed during setup.

Point the browser to the project root when loading things through SystemJS, instead of using the `<base>` URL.

Install the `upgrade` package via `npm install @angular/upgrade --save` and add a mapping for the `@angular/upgrade/static` package.

Creating the *AppModule*

Now create the root `NgModule` class called `AppModule`. There is already a file named `app.module.ts` that holds the AngularJS module. Rename it to `app.module.ajs.ts` and update the corresponding script name in the `index.html` as well. The file contents remain:

Now create a new `app.module.ts` with the minimum `NgModule` class:

Bootstrapping a hybrid PhoneCat

Next, you'll bootstrap the application as a *hybrid application* that supports both AngularJS and Angular components. After that, you can start converting the individual pieces to Angular.

The application is currently bootstrapped using the AngularJS `ng-app` directive attached to the `<html>` element of the host page. This will no longer work in the hybrid app. Switch to the ngUpgrade bootstrap method instead.

First, remove the `ng-app` attribute from `index.html`. Then import `UpgradeModule` in the `AppModule`, and override it's `ngDoBootstrap` method:

Note that you are bootstrapping the AngularJS module from inside `ngDoBootstrap`. The arguments are the same as you would pass to `angular.bootstrap` if you were manually bootstrapping AngularJS: the root element of the application; and an array of the AngularJS 1.x modules that you want to load.

Finally, bootstrap the `AppModule` in `src/main.ts`. This file has been configured as the application entrypoint in `systemjs.config.js`, so it is already being loaded by the browser.

Now you're running both AngularJS and Angular at the same time. That's pretty exciting! You're not running any actual Angular components yet. That's next.

#### Why declare _angular_ as _angular.IAngularStatic_? `@types/angular` is declared as a UMD module, and due to the way [UMD typings](https://github.com/Microsoft/TypeScript/wiki/What's-new-in-TypeScript#support-for-umd-module-definitions) work, once you have an ES6 `import` statement in a file all UMD typed modules must also be imported via `import` statements instead of being globally available. AngularJS is currently loaded by a script tag in `index.html`, which means that the whole app has access to it as a global and uses the same instance of the `angular` variable. If you used `import * as angular from 'angular'` instead, you'd also have to load every file in the AngularJS app to use ES2015 modules in order to ensure AngularJS was being loaded correctly. This is a considerable effort and it often isn't worth it, especially since you are in the process of moving your code to Angular. Instead, declare `angular` as `angular.IAngularStatic` to indicate it is a global variable and still have full typing support.

Upgrading the Phone service

The first piece you'll port over to Angular is the `Phone` service, which resides in `app/core/phone/phone.service.ts` and makes it possible for components to load phone information from the server. Right now it's implemented with ngResource and you're using it for two things:

- For loading the list of all phones into the phone list component.
- For loading the details of a single phone into the phone detail component.

You can replace this implementation with an Angular service class, while keeping the controllers in AngularJS land.

In the new version, you import the Angular HTTP module and call its `Http` service instead of `ngResource`.

Re-open the `app.module.ts` file, import and add `HttpModule` to the `imports` array of the `AppModule`:

Now you're ready to upgrade the Phone service itself. Replace the ngResource-based service in `phone.service.ts` with a TypeScript class decorated as `@Injectable`:

The `@Injectable` decorator will attach some dependency injection metadata to the class, letting Angular know about its dependencies. As described by the Dependency Injection Guide, this is a marker decorator you need to use for classes that have no other Angular decorators but still need to have their dependencies injected.

In its constructor the class expects to get the `Http` service. It will be injected to it and it is stored as a private field. The service is then used in the two instance methods, one of which loads the list of all phones, and the other loads the details of a specified phone:

The methods now return Observables of type `PhoneData` and `PhoneData[]`. This is a type you don't have yet. Add a simple interface for it:

`@angular/upgrade/static` has a `downgradeInjectable` method for the purpose of making Angular services available to AngularJS code. Use it to plug in the `Phone` service:

Here's the full, final code for the service:

Notice that you're importing the `map` operator of the RxJS `Observable` separately. Do this for every RxJS operator.

The new `Phone` service has the same features as the original, `ngResource`-based service. Because it's an Angular service, you register it with the `NgModule` providers:

Now that you are loading `phone.service.ts` through an import that is resolved by SystemJS, you should **remove the <script> tag** for the service from `index.html`. This is something you'll do to all components as you upgrade them. Simultaneously with the AngularJS to Angular upgrade you're also migrating code from scripts to modules.

At this point, you can switch the two components to use the new service instead of the old one. While you `$inject` it as the downgraded `phone` factory, it's really an instance of the `Phone` class and you annotate its type accordingly:

Now there are two AngularJS components using an Angular service! The components don't need to be aware of this, though the fact that the service returns Observables and not Promises is a bit of a giveaway. In any case, what you've achieved is a migration of a service to Angular without having to yet migrate the components that use it.

You could use the \`toPromise\` method of \`Observable\` to turn those Observables into Promises in the service. In many cases that reduce the number of changes to the component controllers.

Upgrading Components

Upgrade the AngularJS components to Angular components next. Do it one component at a time while still keeping the application in hybrid mode. As you make these conversions, you'll also define your first Angular *pipes*.

Look at the phone list component first. Right now it contains a TypeScript controller class and a component definition object. You can morph this into an Angular component by just renaming the controller class and turning the AngularJS component definition object into an Angular `@Component` decorator. You can then also remove the static `$inject` property from the class:

The `selector` attribute is a CSS selector that defines where on the page the component should go. In AngularJS you do matching based on component names, but in Angular you have these explicit selectors. This one will match elements with the name `phone-list`, just like the AngularJS version did.

Now convert the template of this component into Angular syntax. The search controls replace the AngularJS `$ctrl` expressions with Angular's two-way `[(ngModel)]` binding syntax:

Replace the list's `ng-repeat` with an `*ngFor` as described in the Template Syntax page. Replace the image tag's `ng-src` with a binding to the native `src` property.

No Angular *filter* or *orderBy* filters

The built-in AngularJS `filter` and `orderBy` filters do not exist in Angular, so you need to do the filtering and sorting yourself.

You replaced the `filter` and `orderBy` filters with bindings to the `getPhones()` controller method, which implements the filtering and ordering logic inside the component itself.

Now you need to downgrade the Angular component so you can use it in AngularJS. Instead of registering a component, you register a `phoneList` *directive*, a downgraded version of the Angular component.

The `as angular.IDirectiveFactory` cast tells the TypeScript compiler that the return value of the `downgradeComponent` method is a directive factory.

The new `PhoneListComponent` uses the Angular `ngModel` directive, located in the `FormsModule`. Add the `FormsModule` to `NgModule` imports, declare the new `PhoneListComponent` and finally add it to `entryComponents` since you downgraded it:

Remove the <script> tag for the phone list component from `index.html`.

Now set the remaining `phone-detail.component.ts` as follows:

This is similar to the phone list component. The new wrinkle is the `RouteParams` type annotation that identifies the `routeParams` dependency.

The AngularJS injector has an AngularJS router dependency called `$routeParams`, which was injected into `PhoneDetails` when it was still an AngularJS controller. You intend to inject it into the new `PhoneDetailsComponent`.

Unfortunately, AngularJS dependencies are not automatically available to Angular components. You must upgrade this service via a factory provider to make `$routeParams` an Angular injectable. Do that in a new file called `ajs-upgraded-providers.ts` and import it in `app.module.ts`:

Convert the phone detail component template into Angular syntax as follows:

There are several notable changes here:

- You've removed the `$ctrl.` prefix from all expressions.
- You've replaced `ng-src` with property bindings for the standard `src` property.
- You're using the property binding syntax around `ng-class`. Though Angular does have a very similar `ngClass` as AngularJS does, its value is not magically evaluated as an expression. In Angular, you always specify in the template when an attribute's value is a property expression, as opposed to a literal string.
- You've replaced `ng-repeat`s with `*ngFor`s.
- You've replaced `ng-click` with an event binding for the standard `click`.
- You've wrapped the whole template in an `ngIf` that causes it only to be rendered when there is a phone present. You need this because when the component first loads, you don't have `phone` yet and the expressions will refer to a non-existing value. Unlike in AngularJS, Angular expressions do not fail silently when you try to refer to properties on undefined objects. You need to be explicit about cases where this is expected.

Add `PhoneDetailComponent` component to the `NgModule` *declarations* and *entryComponents*:

You should now also remove the phone detail component <script> tag from `index.html`.

Add the *CheckmarkPipe*

The AngularJS directive had a `checkmark` *filter*. Turn that into an Angular **pipe**.

There is no upgrade method to convert filters into pipes. You won't miss it. It's easy to turn the filter function into an equivalent Pipe class. The implementation is the same as before, repackaged in the `transform` method. Rename the file to `checkmark.pipe.ts` to conform with Angular conventions:

Now import and declare the newly created pipe and remove the filter <script> tag from `index.html`:

AOT compile the hybrid app

To use AOT with a hybrid app, you have to first set it up like any other Angular application, as shown in [the Ahead-of-time Compilation chapter](#).

Then change `main-aot.ts` to bootstrap the `AppComponentFactory` that was generated by the AOT compiler:

You need to load all the AngularJS files you already use in `index.html` in `aot/index.html` as well:

These files need to be copied together with the polyfills. The files the application needs at runtime, like the `.json` phone lists and images, also need to be copied.

Install `fs-extra` via `npm install fs-extra --save-dev` for better file copying, and change `copy-dist-files.js` to the following:

And that's all you need to use AOT while upgrading your app!

Adding The Angular Router And Bootstrap

At this point, you've replaced all AngularJS application components with their Angular counterparts, even though you're still serving them from the AngularJS router.

Add the Angular router

Angular has an [all-new router](#).

Like all routers, it needs a place in the UI to display routed views. For Angular that's the `<router-outlet>` and it belongs in a *root component* at the top of the applications component tree.

You don't yet have such a root component, because the app is still managed as an AngularJS app. Create a new `app.component.ts` file with the following `AppComponent` class:

It has a simple template that only includes the `. This component just renders the contents of the active route and nothing else.

The selector tells Angular to plug this root component into the `<phonecat-app>` element on the host web page when the application launches.

Add this `<phonecat-app>` element to the `index.html`. It replaces the old AngularJS `ng-view` directive:

Create the *Routing Module*

A router needs configuration whether it's the AngularJS or Angular or any other router.

The details of Angular router configuration are best left to the Routing documentation which recommends that you create a `NgModule` dedicated to router configuration (called a *Routing Module*).

This module defines a `routes` object with two routes to the two phone components and a default route for the empty path. It passes the `routes` to the `RouterModule.forRoot` method which does the rest.

A couple of extra providers enable routing with "hash" URLs such as `#!/phones` instead of the default "push state" strategy.

Now update the `AppModule` to import this `AppRoutingModule` and also the declare the root `AppComponent` as the bootstrap component. That tells Angular that it should bootstrap the app with the *root* `AppComponent` and insert it's view into the host web page.

You must also remove the bootstrap of the AngularJS module from `ngDoBootstrap()` in `app.module.ts` and the `UpgradeModule` import.

And since you are routing to `PhoneListComponent` and `PhoneDetailComponent` directly rather than using a route template with a `<phone-list>` or `<phone-detail>` tag, you can do away with their Angular selectors as well.

Generate links for each phone

You no longer have to hardcode the links to phone details in the phone list. You can generate data bindings for each phone's `id` to the `routerLink` directive and let that directive construct the appropriate URL to the `PhoneDetailComponent`:

See the Routing page for details.

Use route parameters

The Angular router passes route parameters differently. Correct the `PhoneDetail` component constructor to expect an injected `ActivatedRoute` object. Extract the `phoneId` from the `ActivatedRoute.snapshot.params` and fetch the phone data as before:

You are now running a pure Angular application!

Say Goodbye to AngularJS

It is time to take off the training wheels and let the application begin its new life as a pure, shiny Angular app. The remaining tasks all have to do with removing code - which of course is every programmer's favorite task!

The application is still bootstrapped as a hybrid app. There's no need for that anymore.

Switch the bootstrap method of the application from the `UpgradeModule` to the Angular way.

If you haven't already, remove all references to the `UpgradeModule` from `app.module.ts`, as well as any factory provider for AngularJS services, and the `app/ajs-upgraded-providers.ts` file.

Also remove any `downgradeInjectable()` or `downgradeComponent()` you find, together with the associated AngularJS factory or directive declarations. Since you no longer have downgraded components, you no longer list them in `entryComponents`.

You may also completely remove the following files. They are AngularJS module configuration files and not needed in Angular:

- `app/app.module.ajs.ts`
- `app/app.config.ts`
- `app/core/core.module.ts`
- `app/core/phone/phone.module.ts`
- `app/phone-detail/phone-detail.module.ts`
- `app/phone-list/phone-list.module.ts`

The external typings for AngularJS may be uninstalled as well. The only ones you still need are for Jasmine and Angular polyfills. The `@angular/upgrade` package and it's mapping in `systemjs.config.js` can also go.

npm uninstall @angular/upgrade --save npm uninstall @types/angular @types/angular-animate @types/angular-cookies @types/angular-mocks @types/angular-resource @types/angular-route @types/angular-sanitize --save-dev

Finally, from `index.html`, remove all references to AngularJS scripts and jQuery. When you're done, this is what it should look like:

That is the last you'll see of AngularJS! It has served us well but now it's time to say goodbye.

Appendix: Upgrading PhoneCat Tests

Tests can not only be retained through an upgrade process, but they can also be used as a valuable safety measure when ensuring that the application does not break during the upgrade. E2E tests are especially useful for this purpose.

E2E Tests

The PhoneCat project has both E2E Protractor tests and some Karma unit tests in it. Of these two, E2E tests can be dealt with much more easily: By definition, E2E tests access the application from the *outside* by interacting with the various UI elements the app puts on the screen. E2E tests aren't really that concerned with the internal structure of the application components. That also means that, although you modify the project quite a bit during the upgrade, the E2E test suite should keep passing with just minor modifications. You didn't change how the application behaves from the user's point of view.

During TypeScript conversion, there is nothing to do to keep E2E tests working. But when you change the bootstrap to that of a Hybrid app, you must make a few changes.

Update the `protractor-conf.js` to sync with hybrid apps:

ng12Hybrid: true

When you start to upgrade components and their templates to Angular, you'll make more changes because the E2E tests have matchers that are specific to AngularJS. For PhoneCat you need to make the following changes in order to make things work with Angular:

Previous code	New code	Notes
\`by.repeater('phone in $ctrl.phones').column('phone.name')\`	\`by.css('.phones .name')\`	The repeater matcher relies on AngularJS \`ng-repeat\`
\`by.repeater('phone in $ctrl.phones')\`	\`by.css('.phones li')\`	The repeater matcher relies on AngularJS \`ng-repeat\`
\`by.model('$ctrl.query')\`	\`by.css('input')\`	The model matcher relies on AngularJS \`ng-model\`
\`by.model('$ctrl.orderProp')\`	\`by.css('select')\`	The model matcher relies on AngularJS \`ng-model\`
\`by.binding('$ctrl.phone.name')\`	\`by.css('h1')\`	The binding matcher relies on AngularJS data binding

When the bootstrap method is switched from that of `UpgradeModule` to pure Angular, AngularJS ceases to exist on the page completely. At this point, you need to tell Protractor that it should not be looking for an AngularJS app anymore, but instead it should find *Angular apps* from the page.

Replace the `ng12Hybrid` previously added with the following in `protractor-conf.js`:

useAllAngular2AppRoots: true,

Also, there are a couple of Protractor API calls in the PhoneCat test code that are using the AngularJS `$location` service under the hood. As that service is no longer present after the upgrade, replace those calls with ones that use WebDriver's generic URL APIs instead. The first of these is the redirection spec:

And the second is the phone links spec:

Unit Tests

For unit tests, on the other hand, more conversion work is needed. Effectively they need to be *upgraded* along with the production code.

During TypeScript conversion no changes are strictly necessary. But it may be a good idea to convert the unit test code into TypeScript as well.

For instance, in the phone detail component spec, you can use ES2015 features like arrow functions and block-scoped variables and benefit from the type definitions of the AngularJS services you're consuming:

Once you start the upgrade process and bring in SystemJS, configuration changes are needed for Karma. You need to let SystemJS load all the new Angular code, which can be done with the following kind of shim file:

The shim first loads the SystemJS configuration, then Angular's test support libraries, and then the application's spec files themselves.

Karma configuration should then be changed so that it uses the application root dir as the base directory, instead of `app`.

Once done, you can load SystemJS and other dependencies, and also switch the configuration for loading application files so that they are *not* included to the page by Karma. You'll let the shim and SystemJS load them.

Since the HTML templates of Angular components will be loaded as well, you must help Karma out a bit so that it can route them to the right paths:

The unit test files themselves also need to be switched to Angular when their production counterparts are switched. The specs for the checkmark pipe are probably the most straightforward, as the pipe has no dependencies:

The unit test for the phone service is a bit more involved. You need to switch from the mocked-out AngularJS `$httpBackend` to a mocked-out Angular Http backend.

For the component specs, you can mock out the `Phone` service itself, and have it provide canned phone data. You use Angular's component unit testing APIs for both components.

Finally, revisit both of the component tests when you switch to the Angular router. For the details component, provide a mock of Angular `ActivatedRoute` object instead of using the AngularJS `$routeParams`.

And for the phone list component, a few adjustments to the router make the `RouteLink` directives work.

User Input

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 .

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:

{@a click}

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:

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.

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.

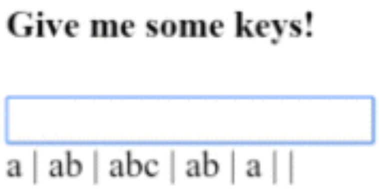
The properties of an `$event` object vary depending on the type of DOM event. For example, a mouse event includes different information than a input box editing event.

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!

a | ab | abc | ab | a | |

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 |

{@a keyup1}

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:

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

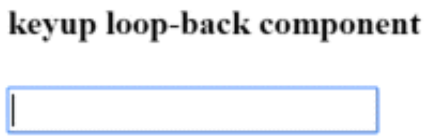
There's another way to get the user data: use Angular **template reference variables**. These variables provide direct access to an element from within the template. To declare a template reference variable, precede an identifier with a hash (or pound) character (#).

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

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.

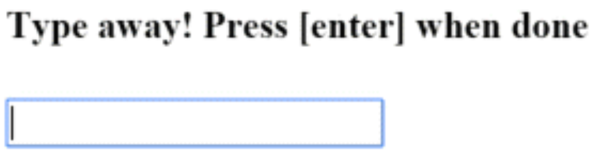
A nice aspect of this approach is that the component gets clean data values from the view. It no longer requires knowledge of the `$event` and its structure. {@a key-event}

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*.

Here's how it works.



Type away! Press [enter] when done

On blur

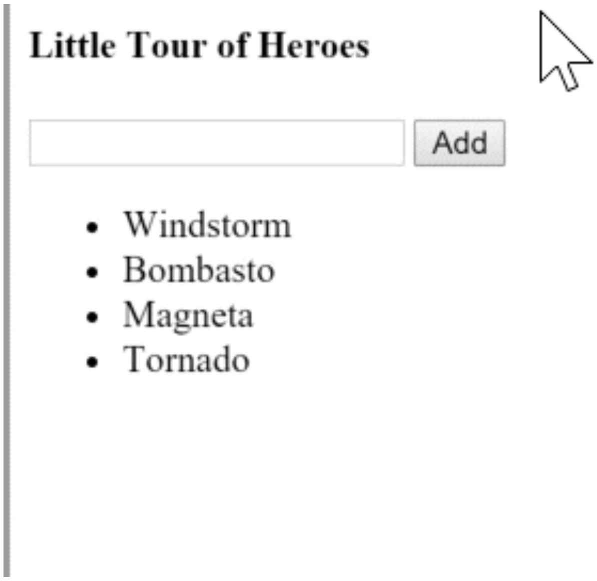
In the previous example, the current state of the input box is lost if the user mouses away and clicks elsewhere on the page without first pressing *Enter*. The component's `value` property is updated only when the user presses *Enter*.

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

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

- Windstorm
- Bombasto
- Magneta
- Tornado

Below is the "Little Tour of Heroes" component.

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.

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` .

Visual Studio 2015 QuickStart

{@a top}

Some developers prefer Visual Studio as their Integrated Development Environment (IDE).

This cookbook describes the steps required to set up and use the Angular QuickStart files in **Visual Studio 2015 within an ASP.NET 4.x project**.

There is no *live example* for this cookbook because it describes Visual Studio, not the QuickStart application itself.

{@a asp-net-4}

ASP.NET 4.x Project

To set up the QuickStart files with an **ASP.NET 4.x project** in Visual Studio 2015, follow these steps:

If you prefer a `File | New Project` experience and are using **ASP.NET Core**, then consider the _experimental_ ASP.NET Core + Angular template for Visual Studio 2015. Note that the resulting code does not map to the docs. Adjust accordingly.

Prerequisite: Node.js

Install **Node.js® and npm** if they are not already on your machine.

Verify that you are running node version `4.6.x` or greater, and npm `3.x.x` or greater by running `node -v` and `npm -v` in a terminal window. Older versions produce errors.

Prerequisite: Visual Studio 2015 Update 3

The minimum requirement for developing Angular applications with Visual Studio is Update 3. Earlier versions do not follow the best practices for developing applications with TypeScript. To view your version of Visual Studio 2015, go to `Help | About Visual Studio`.

If you don't have it, install **Visual Studio 2015 Update 3**. Or use `Tools | Extensions and Updates` to update to Update 3 directly from Visual Studio 2015.

Prerequisite: Configure External Web tools

Configure Visual Studio to use the global external web tools instead of the tools that ship with Visual Studio:

- Open the **Options** dialog with `Tools` | `Options` .
- In the tree on the left, select `Projects and Solutions` | `External Web Tools` .
- On the right, move the `$(PATH)` entry above the `$(DevEnvDir`) entries. This tells Visual Studio to use the external tools (such as npm) found in the global path before using its own version of the external tools.
- Click OK to close the dialog.
- Restart Visual Studio for this change to take effect.

Visual Studio now looks first for external tools in the current workspace and if it doesn't find them, it looks in the global path. If Visual Studio doesn't find them in either location, it will use its own versions of the tools.

Prerequisite: Install TypeScript 2.2 for Visual Studio 2015

While Visual Studio Update 3 ships with TypeScript support out of the box, it currently doesn’t ship with TypeScript 2.2, which you need to develop Angular applications.

To install TypeScript 2.2:

- Download and install **TypeScript 2.2 for Visual Studio 2015**

- OR install it with npm: `npm install -g typescript@2.2` .

You can find out more about TypeScript 2.2 support in Visual studio **here**.

At this point, Visual Studio is ready. It’s a good idea to close Visual Studio and restart it to make sure everything is clean.

Step 1: Download the QuickStart files

Download the QuickStart source from GitHub. If you downloaded as a zip file, extract the files.

Step 2: Create the Visual Studio ASP.NET project

Create the ASP.NET 4.x project in the usual way as follows:

- In Visual Studio, select `File` | `New` | `Project` from the menu.

- In the template tree, select `Templates` | `Visual C#` (or `Visual Basic`) | `Web`.
- Select the `ASP.NET Web Application` template, give the project a name, and click OK.
- Select the desired ASP.NET 4.5.2 template and click OK.

This cookbook uses the `Empty` template with no added folders, no authentication, and no hosting. Pick the template and options appropriate for your project.

Step 3: Copy the QuickStart files into the ASP.NET project folder

Copy the QuickStart files you downloaded from GitHub into the folder containing the `.csproj` file. Include the files in the Visual Studio project as follows:

- Click the `Show All Files` button in Solution Explorer to reveal all of the hidden files in the project.
- Right-click on each folder/file to be included in the project and select `Include in Project`. Minimally, include the following folder/files:
  - src/app folder (answer *No* if asked to search for TypeScript Typings)
  - src/styles.css
  - src/index.html
  - package.json
  - src/tsconfig.json

Step 4: Restore the required packages

Restore the packages required for an Angular application as follows:

- Right-click on the `package.json` file in Solution Explorer and select `Restore Packages`. This uses `npm` to install all of the packages defined in the `package.json` file. It may take some time.
- If desired, open the Output window (`View` | `Output`) to watch the npm commands execute.
- Ignore the warnings.
- When the restore is finished, a message in the bottom message bar of Visual Studio should say: `Installing packages complete`. Be patient. This could take a while.
- Click the `Refresh` icon in Solution Explorer.
- **Do not** include the `node_modules` folder in the project. Let it be a hidden project folder.

Step 5: Build and run the app

First, ensure that `src/index.html` is set as the start page. Right-click `index.html` in Solution Explorer and select option `Set As Start Page` .

To run in VS with F5

Most Visual Studio developers like to press the F5 key and see the IIS server come up. To use the IIS server with the QuickStart app, you must make the following three changes.

1. In `index.html` , change base href from `<base href="/">` to `<base href="/src/">` .
2. Also in `index.html` , change the scripts to use `/node_modules` with a slash instead of `node_modules` without the slash.
3. In `src/systemjs.config.js` , near the top of the file, change the npm `path` to `/node_modules/` with a slash.

After these changes, `npm start` no longer works. You must choose to configure _either_ for F5 with IIS _or_ for `npm start` with the lite-server.

For apps that use routing

If your app uses routing, you need to teach the server to always return `index.html` when the user asks for an HTML page for reasons explained in the Deployment guide.

Everything seems fine while you move about *within* the app. But you'll see the problem right away if you refresh the browser or paste a link to an app page (called a "deep link") into the browser address bar.

You'll most likely get a *404 - Page Not Found* response from the server for any address other than `/` or `/index.html` .

You have to configure the server to return `index.html` for requests to these "unknown" pages. The `lite-server` development server does out-of-the-box. If you've switched over to F5 and IIS, you have to configure IIS to do it. This section walks through the steps to adapt the QuickStart application.

Configure IIS rewrite rules

Visual Studio ships with IIS Express, which has the rewrite module baked in. However, if you're using regular IIS you'll have to install the rewrite module.

Tell Visual Studio how to handle requests for route app pages by adding these rewrite rules near the bottom of the `web.config` :

<system.webServer> <rewrite> <rules> <rule name="Angular Routes" stopProcessing="true"> <match url=".*" /> <conditions logicalGrouping="MatchAll"> <add input="{REQUEST*FILENAME}" matchType="IsFile"*

*negate="true" /> <add input="{REQUEST*FILENAME}" matchType="IsDirectory" negate="true" /> </conditions> <action type="Rewrite" url="/src/" /> </rule> </rules> </rewrite> </system.webServer>

The match url, ``, will rewrite every request. You'll have to adjust this if you want some requests to get through, such as web API requests. The URL in `` should match the base href in `index.html`.

Build and launch the app with debugger by clicking the **Run** button or by pressing `F5`.

It's faster to run without the debugger by pressing `Ctrl-F5`.

The default browser opens and displays the QuickStart sample application.

Try editing any of the project files. Save and refresh the browser to see the changes.

Webpack: An Introduction

Webpack is a popular module bundler, a tool for bundling application source code in convenient *chunks* and for loading that code from a server into a browser.

It's an excellent alternative to the *SystemJS* approach used elsewhere in the documentation. This guide offers a taste of Webpack and explains how to use it with Angular applications.

{@a top}

You can also download the final result.

{@a what-is-webpack}

What is Webpack?

Webpack is a powerful module bundler. A *bundle* is a JavaScript file that incorporates *assets* that *belong* together and should be served to the client in a response to a single file request. A bundle can include JavaScript, CSS styles, HTML, and almost any other kind of file.

Webpack roams over your application source code, looking for `import` statements, building a dependency graph, and emitting one or more *bundles*. With plugins and rules, Webpack can preprocess and minify different non-JavaScript files such as TypeScript, SASS, and LESS files.

You determine what Webpack does and how it does it with a JavaScript configuration file, `webpack.config.js`.

{@a entries-outputs}

Entries and outputs

You supply Webpack with one or more *entry* files and let it find and incorporate the dependencies that radiate from those entries. The one entry point file in this example is the application's root file, `src/main.ts`:

Webpack inspects that file and traverses its `import` dependencies recursively.

It sees that you're importing `@angular/core` so it adds that to its dependency list for potential inclusion in the bundle. It opens the `@angular/core` file and follows *its* network of `import` statements until it has built the complete dependency graph from `main.ts` down.

Then it **outputs** these files to the `app.js` *bundle file* designated in configuration:

output: { filename: 'app.js' }

This `app.js` output bundle is a single JavaScript file that contains the application source and its dependencies. You'll load it later with a `<script>` tag in the `index.html`.

{@a multiple-bundles}

Multiple bundles

You probably don't want one giant bundle of everything. It's preferable to separate the volatile application app code from comparatively stable vendor code modules.

Change the configuration so that it has two entry points, `main.ts` and `vendor.ts`:

entry: { app: 'src/app.ts', vendor: 'src/vendor.ts' },

output: { filename: '[name].js' }

Webpack constructs two separate dependency graphs and emits *two* bundle files, one called `app.js` containing only the application code and another called `vendor.js` with all the vendor dependencies.

The `[name]` in the output name is a *placeholder* that a Webpack plugin replaces with the entry names, `app` and `vendor`. Plugins are [covered later](guide/webpack#commons-chunk-plugin) in the guide.

To tell Webpack what belongs in the vendor bundle, add a `vendor.ts` file that only imports the application's third-party modules:

{@a loaders}

Loaders

Webpack can bundle any kind of file: JavaScript, TypeScript, CSS, SASS, LESS, images, HTML, fonts, whatever. Webpack *itself* only understands JavaScript files. Teach it to transform non-JavaScript file into their JavaScript equivalents with *loaders*. Configure loaders for TypeScript and CSS as follows.

rules: [{ test: /.ts$/, loader: 'awesome-typescript-loader' }, { test: /.css$/, loaders: 'style-loader!css-loader' }]

When Webpack encounters `import` statements like the following, it applies the `test` RegEx patterns.

import { AppComponent } from './app.component.ts';

import 'uiframework/dist/uiframework.css';

When a pattern matches the filename, Webpack processes the file with the associated loader.

The first `import` file matches the `.ts` pattern so Webpack processes it with the `awesome-typescript-loader`. The imported file doesn't match the second pattern so its loader is ignored.

The second `import` matches the second `.css` pattern for which you have *two* loaders chained by the (!) character. Webpack applies chained loaders *right to left*. So it applies the `css` loader first to flatten CSS `@import` and `url(...)` statements. Then it applies the `style` loader to append the css inside `<style>` elements on the page.

{@a plugins}

Plugins

Webpack has a build pipeline with well-defined phases. Tap into that pipeline with plugins such as the `uglify` minification plugin:

plugins: [new webpack.optimize.UglifyJsPlugin()]

{@a configure-webpack}

Configuring Webpack

After that brief orientation, you are ready to build your own Webpack configuration for Angular apps.

Begin by setting up the development environment.

Create a new project folder.

mkdir angular-webpack cd angular-webpack

Add these files:

Many of these files should be familiar from other Angular documentation guides, especially the [Typescript configuration](guide/typescript-configuration) and [npm packages](guide/npm-packages) guides. Webpack, the plugins, and the loaders are also installed as packages. They are listed in the updated `packages.json`.

Open a terminal window and install the npm packages.

npm install

{@a polyfills}

Polyfills

You'll need polyfills to run an Angular application in most browsers as explained in the Browser Support guide.

Polyfills should be bundled separately from the application and vendor bundles. Add a `polyfills.ts` like this one to the `src/` folder.

Loading polyfills
Load `zone.js` early within `polyfills.ts`, immediately after the other ES6 and metadata shims.

Because this bundle file will load first, `polyfills.ts` is also a good place to configure the browser environment for production or development.

{@a common-configuration}

Common configuration

Developers typically have separate configurations for development, production, and test environments. All three have a lot of configuration in common.

Gather the common configuration in a file called `webpack.common.js`.

{@a inside-webpack-commonjs}

Inside *webpack.common.js*

Webpack is a NodeJS-based tool that reads configuration from a JavaScript commonjs module file.

The configuration imports dependencies with `require` statements and exports several objects as properties of a `module.exports` object.

- `entry` —the entry-point files that define the bundles.
- `resolve` —how to resolve file names when they lack extensions.
- `module.rules` — `module` is an object with `rules` for deciding how files are loaded.
- `plugins` —creates instances of the plugins.

{@a common-entries}

entry

The first export is the `entry` object:

This `entry` object defines the three bundles:

- `polyfills` —the polyfills needed to run Angular applications in most modern browsers.
- `vendor` —the third-party dependencies such as Angular, lodash, and bootstrap.css.
- `app` —the application code.

{@a common-resolves}

resolve extension-less imports

The app will `import` dozens if not hundreds of JavaScript and TypeScript files. You could write `import` statements with explicit extensions like this example:

import { AppComponent } from './app.component.ts';

But most `import` statements don't mention the extension at all. Tell Webpack to resolve extension-less file requests by looking for matching files with `.ts` extension or `.js` extension (for regular JavaScript files and pre-compiled TypeScript files).

If Webpack should resolve extension-less files for styles and HTML, add \`.css\` and \`.html\` to the list.

{@a common-rules}

module.rules

Rules tell Webpack which loaders to use for each file, or module:

- `awesome-typescript-loader` —a loader to transpile the Typescript code to ES5, guided by the `tsconfig.json` file.
- `angular2-template-loader` —loads angular components' template and styles.
- `html-loader` —for component templates.
- images/fonts—Images and fonts are bundled as well.
- CSS—the first pattern matches application-wide styles; the second handles component-scoped styles (the ones specified in a component's `styleUrls` metadata property).

The first pattern is for the application-wide styles. It excludes \`.css\` files within the \`src/app\` directory where the component-scoped styles sit. The \`ExtractTextPlugin\` (described below) applies the \`style\` and \`css\` loaders to these files. The second pattern filters for component-scoped styles and loads them as strings via the \`raw-loader\`, which is what Angular expects to do with styles specified in a \`styleUrls\` metadata property.
Multiple loaders can be chained using the array notation.

{@a common-plugins}

plugins

Finally, create instances of three plugins:

{@a commons-chunk-plugin}

CommonsChunkPlugin

The `app.js` bundle should contain only application code. All vendor code belongs in the `vendor.js` bundle.

Of course the application code imports vendor code. On its own, Webpack is not smart enough to keep the vendor code out of the `app.js` bundle. The `CommonsChunkPlugin` does that job.

The \`CommonsChunkPlugin\` identifies the hierarchy among three _chunks_: \`app\` -> \`vendor\` -> \`polyfills\`. Where Webpack finds that \`app\` has shared dependencies with \`vendor\`, it removes them from \`app\`. It would remove \`polyfills\` from \`vendor\` if they shared dependencies, which they don't.

{@a html-webpack-plugin}

HtmlWebpackPlugin

Webpack generates a number of js and CSS files. You *could* insert them into the `index.html` *manually*. That would be tedious and error-prone. Webpack can inject those scripts and links for you with the `HtmlWebpackPlugin`.

{@a environment-configuration}

Environment-specific configuration

The `webpack.common.js` configuration file does most of the heavy lifting. Create separate, environment-specific configuration files that build on `webpack.common` by merging into it the peculiarities particular to the target environments.

These files tend to be short and simple.

{@a development-configuration}

Development configuration

Here is the `webpack.dev.js` development configuration file.

The development build relies on the Webpack development server, configured near the bottom of the file.

Although you tell Webpack to put output bundles in the `dist` folder, the dev server keeps all bundles in memory; it doesn't write them to disk. You won't find any files in the `dist` folder, at least not any generated from *this development build*.

The `HtmlWebpackPlugin`, added in `webpack.common.js`, uses the `publicPath` and the `filename` settings to generate appropriate `<script>` and `<link>` tags into the `index.html`.

The CSS styles are buried inside the Javascript bundles by default. The `ExtractTextPlugin` extracts them into external `.css` files that the `HtmlWebpackPlugin` inscribes as `<link>` tags into the `index.html`.

Refer to the Webpack documentation for details on these and other configuration options in this file.

Grab the app code at the end of this guide and try:

npm start

{@a production-configuration}

Production configuration

Configuration of a *production* build resembles *development* configuration with a few key changes.

You'll deploy the application and its dependencies to a real production server. You won't deploy the artifacts needed only in development.

Put the production output bundle files in the `dist` folder.

Webpack generates file names with cache-busting hash. Thanks to the `HtmlWebpackPlugin`, you don't have to update the `index.html` file when the hash changes.

There are additional plugins:

- * `NoEmitOnErrorsPlugin` —stops the build if there is an error.
- * `UglifyJsPlugin` —minifies the bundles.
- * `ExtractTextPlugin` —extracts embedded css as external files, adding cache-busting hash to the filename.
- * `DefinePlugin` —use to define environment variables that you can reference within the application.
- * `LoaderOptionsPlugins` —to override options of certain loaders.

Thanks to the `DefinePlugin` and the `ENV` variable defined at top, you can enable Angular production mode like this:

Grab the app code at the end of this guide and try:

```
npm run build
```

{@a test-configuration}

Test configuration

You don't need much configuration to run unit tests. You don't need the loaders and plugins that you declared for your development and production builds. You probably don't need to load and process the application-wide styles files for unit tests and doing so would slow you down; you'll use the `null` loader for those CSS files.

You could merge the test configuration into the `webpack.common` configuration and override the parts you don't want or need. But it might be simpler to start over with a completely fresh configuration.

Reconfigure Karma to use Webpack to run the tests:

You don't precompile the TypeScript; Webpack transpiles the Typescript files on the fly, in memory, and feeds the emitted JS directly to Karma. There are no temporary files on disk.

The `karma-test-shim` tells Karma what files to pre-load and primes the Angular test framework with test versions of the providers that every app expects to be pre-loaded.

Notice that you do *not* load the application code explicitly. You tell Webpack to find and load the test files (the files ending in `.spec.ts`). Each spec file imports all—and only—the application source code that it tests. Webpack loads just *those* specific application files and ignores the other files that you aren't testing.

Grab the app code at the end of this guide and try:

```
npm test
```

{@a try}

Trying it out

Here is the source code for a small application that bundles with the Webpack techniques covered in this guide.

The `app.component.html` displays this downloadable Angular logo . Create a folder called `images` under the project's `assets` folder, then right-click (Cmd+click on Mac) on the image and download it to that folder.

{@a bundle-ts}

Here again are the TypeScript entry-point files that define the `polyfills` and `vendor` bundles.

{@a highlights}

Highlights

- There are no `<script>` or `<link>` tags in the `index.html`. The `HtmlWebpackPlugin` inserts them dynamically at runtime.
- The `AppComponent` in `app.component.ts` imports the application-wide css with a simple `import` statement.
- The `AppComponent` itself has its own html template and css file. WebPack loads them with calls to `require()`. Webpack stashes those component-scoped files in the `app.js` bundle too. You don't see those calls in the source code; they're added behind the scenes by the `angular2-template-loader` plug-in.
- The `vendor.ts` consists of vendor dependency `import` statements that drive the `vendor.js` bundle. The application imports these modules too; they'd be duplicated in the `app.js` bundle if the `CommonsChunkPlugin` hadn't detected the overlap and removed them from `app.js`. {@a conclusion}

Conclusion

You've learned just enough Webpack to configurate development, test and production builds for a small Angular application.

You could always do more. Search the web for expert advice and expand your Webpack knowledge.

Back to top