

This is the archived documentation for Angular v4. Please visit [angular.io](#) to see documentation for the current version of Angular.

QuickStart

[Contents >](#)

Step 1. Set up the Development Environment

Step 2. Create a new project

Step 3: Serve the application

...

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 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!!*:

src/app/app.component.ts

```
export class AppComponent {  
  title = '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.

src/app/app.component.css

```
h1 {  
  color: #369;  
  font-family: Arial, Helvetica, sans-serif;  
  font-size: 250%;  
}
```

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 <code>AppComponent</code> 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 <code>AppModule</code> , the root module that tells Angular how to assemble the application. Right now it declares only the <code>AppComponent</code> . 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 <code>js</code> and <code>css</code> files when building your app so you never need to add any <code><script></code> or <code><link></code> tags here manually.

`main.ts`

The main entry point for your app. Compiles the application with the [JIT compiler](#) and bootstraps the application's root module (`AppModule`) to run in the browser. You can also use the [AOT compiler](#) without changing any code by passing in `--aot` to `ng build` or `ng serve`.

`polyfills.ts`

Different browsers have different levels of support of the web standards. Polyfills help normalize those differences. You should be pretty safe with `core-js` and `zone.js`, but be sure to check out the [Browser Support guide](#) for more information.

`styles.css`

Your global styles go here. Most of the time you'll want to have local styles in your components for easier maintenance, but styles that affect all of your app need to be in a central place.

`test.ts`

This is the main entry point for your unit tests. It has some custom configuration that might be unfamiliar, but it's not something you'll need to edit.

`tsconfig.``{app|spec}.json`

TypeScript compiler configuration for the Angular app (`tsconfig.app.json`) and for the unit tests (`tsconfig.spec.json`).

The root folder

The `src/` folder is just one of the items inside the project's root folder. Other files help you build, test, maintain, document, and deploy the app. These files go in the root folder next to `src/`.

`my-app`

```
e2e
  app.e2e-spec.ts
  app.po.ts
  tsconfig.e2e.json
node_modules/...
src/...
.angular-cli.json
.editorconfig
.gitignore
karma.conf.js
package.json
protractor.conf.js
README.md
tsconfig.json
tslint.json
```

File**Purpose**

e2e/

Inside `e2e/` live the end-to-end tests. They shouldn't be inside `src/` because e2e tests are really a separate app that just so happens to test your main app. That's also why they have their own `tsconfig.e2e.json`.

node_modules/

`Node.js` creates this folder and puts all third party modules listed in

`package.json` inside of it.

`.angular-cli.json` Configuration for Angular CLI. In this file you can set several defaults and also configure what files are included when your project is built. Check out the official documentation if you want to know more.

`.editorconfig` Simple configuration for your editor to make sure everyone that uses your project has the same basic configuration. Most editors support an `.editorconfig` file. See <http://editorconfig.org> for more information.

`.gitignore` Git configuration to make sure autogenerated files are not committed to source control.

`karma.conf.js` Unit test configuration for the [Karma test runner](#), used when running `ng test`.

`package.json` `npm` configuration listing the third party packages your project uses. You can also add your own [custom scripts](#) here.

`protractor.conf.js` End-to-end test configuration for [Protractor](#), used when running `ng e2e`.

`README.md` Basic documentation for your project, pre-filled with CLI command information. Make sure to enhance it with project documentation so that anyone checking out the repo can build your app!

`tsconfig.json` TypeScript compiler configuration for your IDE to pick up and give you helpful tooling.

`tslint.json`

Linting configuration for [TSLint](#) together with [Codelyzer](#), used when running `ng lint`. Linting helps keep your code style consistent.

Next Step

If you're new to Angular, continue with the [tutorial](#). You can skip the "Setup" step since you're already using the Angular CLI setup.

This is the archived documentation for Angular v4. Please visit [angular.io](#) to see documentation for the current version of Angular.

Tutorial: Tour of Heroes

Contents

[What you'll build](#)

[Next step](#)

The grand plan for this tutorial is to build an app that helps a staffing agency manage its stable of heroes.

The Tour of Heroes app covers the core fundamentals of Angular. You'll build a basic app that has many of the features you'd expect to find in a full-blown, data-driven app: acquiring and displaying a list of heroes, editing a selected hero's detail, and navigating among different views of heroic data.

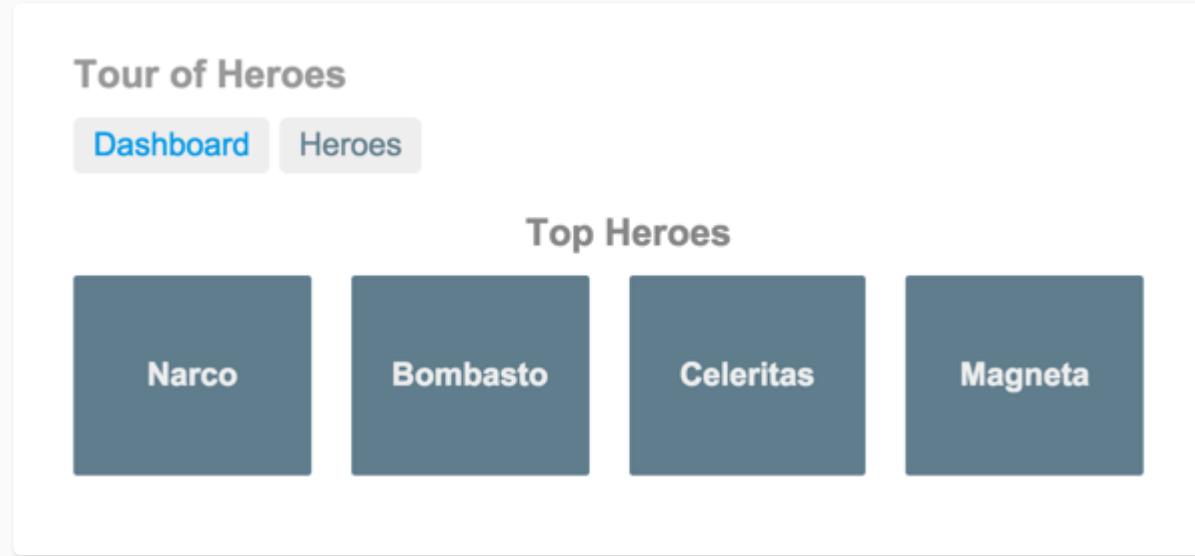
You'll use built-in directives to show and hide elements and display lists of hero data. You'll create components to display hero details and show an array of heroes. You'll use one-way data binding for read-only data. You'll add editable fields to update a model with two-way data binding. You'll bind component methods to user events, like keystrokes and clicks. You'll enable users to select a hero from a master list and edit that hero in the details view. You'll format data with pipes. You'll create a shared service to assemble the heroes. And you'll use routing to navigate among different views and their components.

You'll learn enough core Angular to get started and gain confidence that Angular can do whatever you need it to do. You'll cover a lot of ground at an introductory level, and you'll find many links to pages with greater depth.

When you're done with this tutorial, the app will look like this [live example](#) / [download example](#).

What you'll build

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



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

If you click the dashboard hero "Magneta," the router opens a "Hero Details" view where you can change the hero's name.

Tour of Heroes

[Dashboard](#) [Heroes](#)

Magneta details!

id: 15

name: Magneta

[Back](#)

Clicking the "Back" button returns you to the Dashboard. Links at the top take you to either of the main views. If you click "Heroes," the app displays the "Heroes" master list view.

Tour of Heroes

[Dashboard](#) [Heroes](#)

My Heroes

11 Mr. Nice

12 Narco

13 Bombasto

14 Celeritas

15 Magneta

16 RubberMan

17 Dynama

18 Dr IQ

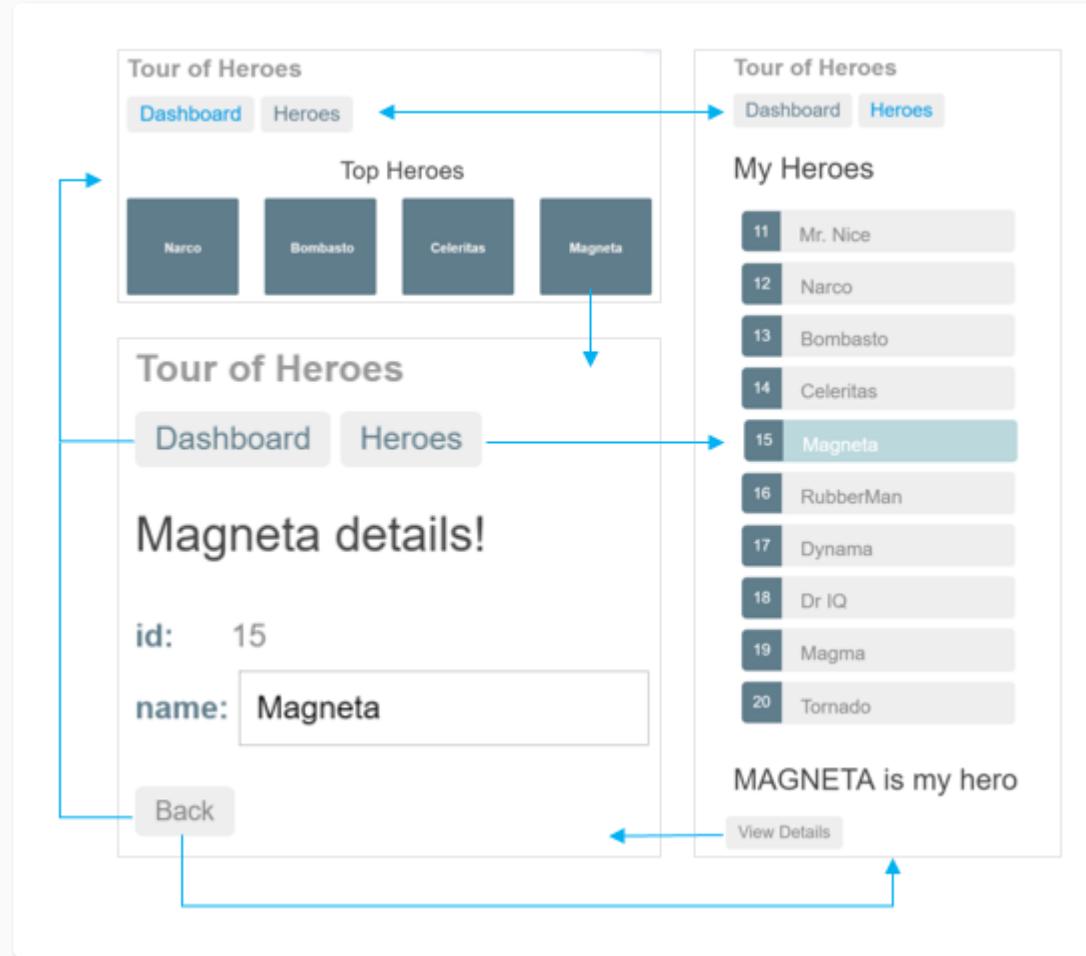
19 Magma

20 Tornado

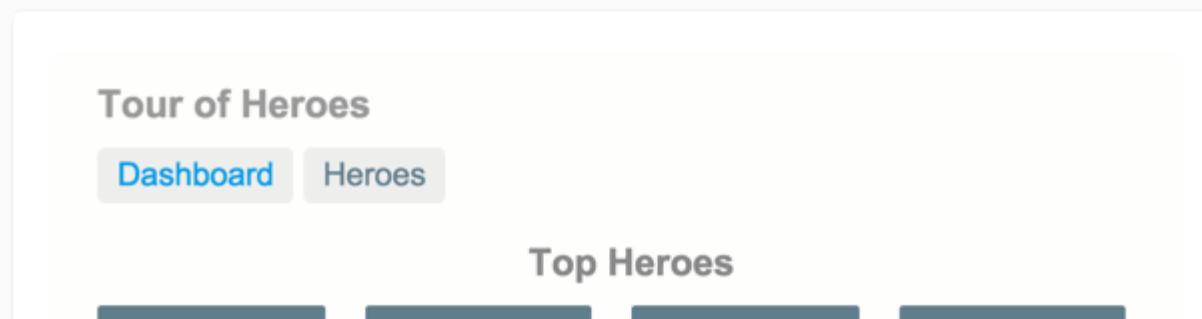
When you click a different hero name, the read-only mini detail beneath the list reflects the new choice.

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

The following diagram captures all of the navigation options.



Here's the app in action:



Narco

Bombasto

Celeritas

Magneta



Next step

You'll build the Tour of Heroes app, step by step. Each step is motivated with a requirement that you've likely met in many applications. Everything has a reason.

Along the way, you'll become familiar with many of the core fundamentals of Angular.

Start now by building a simple [hero editor](#).

This is the archived documentation for Angular v4. Please visit [angular.io](#) to see documentation for the current version of Angular.

The Hero Editor

[Contents >](#)

[Setup to develop locally](#)

[Keep the app transpiling and running](#)

[Show the hero](#)

...

Setup to develop locally

Follow the [setup](#) instructions for creating a new project named `angular-tour-of-heroes`.

The file structure should look like this:

```
angular-tour-of-heroes
  src
    app
      app.component.ts
      app.module.ts
    main.ts
```

```
index.html  
styles.css  
systemjs.config.js  
tsconfig.json  
node_modules ...  
package.json
```

When you're done with this page, the app should look like this [live example / download example](#).

Keep the app transpiling and running

Enter the following command in the terminal window:

```
npm start
```



This command runs the TypeScript compiler in "watch mode", recompiling automatically when the code changes. The command simultaneously launches the app in a browser and refreshes the browser when the code changes.

You can keep building the Tour of Heroes without pausing to recompile or refresh the browser.

Show the hero

Add two properties to the `AppComponent`: a `title` property for the app name and a `hero` property for a hero named "Windstorm."

app.component.ts (AppComponent class)

```
export class AppComponent {  
  title = 'Tour of Heroes';  
  hero = 'Windstorm';  
}
```



Now update the template in the `@Component` decorator with data bindings to these new properties.

app.component.ts (@Component)

```
template: `<h1>{{title}}</h1><h2>{{hero}} details!</h2>`
```



The browser refreshes and displays the title and hero name.

The double curly braces are Angular's *interpolation binding* syntax. These interpolation bindings present the component's `title` and `hero` property values, as strings, inside the HTML header tags.

Read more about interpolation in the [Displaying Data](#) page.

Hero object

The hero needs more properties. Convert the `hero` from a literal string to a class.

Create a `Hero` class with `id` and `name` properties. Add these properties near the top of the `app.component.ts` file, just below the import statement.

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

```
export class Hero {  
  id: number;  
  name: string;  
}
```

In the `AppComponent` class, refactor the component's `hero` property to be of type `Hero`, then initialize it with an `id` of `1` and the name `Windstorm`.

src/app/app.component.ts (hero property)

```
hero: Hero = {  
  id: 1,  
  name: 'Windstorm'  
};
```

Because you changed the hero from a string to an object, update the binding in the template to refer to the hero's `name` property.

src/app/app.component.ts

```
template: `<h1>{{title}}</h1><h2>{{hero.name}} details!</h2>`
```

The browser refreshes and continues to display the hero's name.

Adding HTML with multi-line template strings

To show all of the hero's properties, add a `<div>` for the hero's `id` property and another `<div>` for the hero's `name`. To keep the template readable, place each `<div>` on its own line.

The backticks around the component template let you put the `<h1>`, `<h2>`, and `<div>` elements on their own lines, thanks to the *template literals* feature in ES2015 and TypeScript. For more information, see [Template literals](#).

app.component.ts (AppComponent's template)

```
template: `

<h1>{{title}}</h1>
<h2>{{hero.name}} details!</h2>
<div><label>id: </label>{{hero.id}}</div>
<div><label>name: </label>{{hero.name}}</div>
`
```



Edit the hero name

Users should be able to edit the hero name in an `<input>` textbox. The textbox should both *display* the hero's `name` property and *update* that property as the user types.

You need a two-way binding between the `<input>` form element and the `hero.name` property.

Two-way binding

Refactor the hero name in the template so it looks like this:

src/app/app.component.ts

```
<div>
  <label>name: </label>
```



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

`[(ngModel)]` is the Angular syntax to bind the `hero.name` property to the textbox. Data flows *in both directions*: from the property to the textbox, and from the textbox back to the property.

Unfortunately, immediately after this change, the application breaks. If you looked in the browser console, you'd see Angular complaining that "`ngModel` ... isn't a known property of `input`."

Although `NgModel` is a valid Angular directive, it isn't available by default. It belongs to the optional `FormsModule`. You must opt-in to using that module.

Import the `FormsModule`

Open the `app.module.ts` file and import the `FormsModule` symbol from the `@angular/forms` library. Then add the `FormsModule` to the `@NgModule` metadata's `imports` array, which contains the list of external modules that the app uses.

The updated `AppModule` looks like this:

app.module.ts (FormsModule import)

```
1. import { NgModule }      from '@angular/core';  
2. import { BrowserModule } from '@angular/platform-browser';  
3. import { FormsModule }   from '@angular/forms'; // <-- NgModel lives here  
4.  
5. import { AppComponent }  from './app.component';  
6.  
7. @NgModule({  
8.   imports: [  
9.     BrowserModule,
```

```
10.  FormsModule // <-- import the FormsModule before binding with  
11.  [(ngModel)]  
12.  ],  
13.  declarations: [  
14.    AppComponent  
15.  ],  
16.  bootstrap: [ AppComponent ]  
17. })  
18. export class AppModule { }
```

Read more about `FormsModule` and `ngModel` in the [Two-way data binding with ngModel](#) section of the [Forms](#) guide and the [Two-way binding with NgModel](#) section of the [Template Syntax](#) guide.

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

Summary

Take stock of what you've built.

- The Tour of Heroes app uses the double curly braces of interpolation (a type of one-way data binding) to display the app title and properties of a `Hero` object.
- You wrote a multi-line template using ES2015's template literals to make the template readable.
- You added a two-way data binding to the `<input>` element using the built-in `ngModel` directive. This binding both displays the hero's name and allows users to change it.
- The `ngModel` directive propagates changes to every other binding of the `hero.name`.

Your app should look like this [live example](#) / [download example](#) .

Here's the complete `app.component.ts` as it stands now:

src/app/app.component.ts

```
1. import { Component } from '@angular/core';
2.
3. export class Hero {
4.   id: number;
5.   name: string;
6. }
7.
8. @Component({
9.   selector: 'my-app',
10.  template: `
11.    <h1>{{title}}</h1>
12.    <h2>{{hero.name}} details!</h2>
13.    <div><label>id: </label>{{hero.id}}</div>
14.    <div>
15.      <label>name: </label>
16.      <input [(ngModel)]="hero.name" placeholder="name">
17.    </div>
18.    `
19. })
20. export class AppComponent {
21.   title = 'Tour of Heroes';
22.   hero: Hero = {
23.     id: 1,
24.     name: 'Windstorm'
25.   };
26. }
```

Next step

In the [next tutorial page](#), you'll build on the Tour of Heroes app to display a list of heroes. You'll also allow the user to select heroes and display their details. You'll learn more about how to retrieve lists and bind them to the template.

This is the archived documentation for Angular v4. Please visit [angular.io](#) to see documentation for the current version of Angular.

Master/Detail

[Contents >](#)

[Where you left off](#)

[Keep the app transpiling and running](#)

[Displaying heroes](#)

•••

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

When you're done with this page, the app should look like this [live example](#) / [download example](#).

Where you left off

Before you continue with this page of the Tour of Heroes, verify that you have the following structure after [The Hero Editor](#) page. If your structure doesn't match, go back to that page to figure out what you missed.

```
angular-tour-of-heroes
```

```
  src
```

```
    app
```

```
app.component.ts  
app.module.ts  
main.ts  
index.html  
styles.css  
systemjs.config.js  
tsconfig.json  
node_modules ...  
package.json
```

Keep the app transpiling and running

Enter the following command in the terminal window:

```
npm start
```



This command runs the TypeScript compiler in "watch mode", recompiling automatically when the code changes. The command simultaneously launches the app in a browser and refreshes the browser when the code changes.

You can keep building the Tour of Heroes without pausing to recompile or refresh the browser.

Displaying heroes

To display a list of heroes, you'll add heroes to the view's template.

Create heroes

Create an array of ten heroes.

src/app/app.component.ts (hero array)

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



The `HEROES` array is of type `Hero`, the class defined in the previous page. Eventually this app will fetch the list of heroes from a web service, but for now you can display mock heroes.

Expose heroes

Create a public property in `AppComponent` that exposes the heroes for binding.

app.component.ts (hero array property)

```
heroes = HEROES;
```



The `heroes` type isn't defined because TypeScript infers it from the `HEROES` array.

The hero data is separated from the class implementation because ultimately the hero names will come from a data service.

Display hero names in a template

To display the hero names in an unordered list, insert the following chunk of HTML below the title and above the hero details.

app.component.ts (heroes template)

```
<h2>My Heroes</h2>
<ul class="heroes">
  <li>
    <!-- each hero goes here -->
  </li>
</ul>
```

Now you can fill the template with hero names.

List heroes with ngFor

The goal is to bind the array of heroes in the component to the template, iterate over them, and display them individually.

Modify the `` tag by adding the built-in directive `*ngFor`.

app.component.ts (ngFor)

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



The (*) prefix to `ngFor` is a critical part of this syntax. It indicates that the `` element and its children constitute a master template.

The `ngFor` directive iterates over the component's `heroes` array and renders an instance of this template for each hero in that array.

The `let hero` part of the expression identifies `hero` as the template input variable, which holds the current hero item for each iteration. You can reference this variable within the template to access the current hero's properties.

Read more about `ngFor` and template input variables in the [Showing an array property with *ngFor](#) section of the [Displaying Data](#) page and the `ngFor` section of the [Template Syntax](#) page.

Within the `` tags, add content that uses the `hero` template variable to display the hero's properties.

app.component.ts (ngFor template)

```
<li *ngFor="let hero of heroes">
  <span class="badge">{{hero.id}}</span> {{hero.name}}
</li>
```



When the browser refreshes, a list of heroes appears.

Style the heroes

Users should get a visual cue of which hero they are hovering over and which hero is selected.

To add styles to your component, set the `styles` property on the `@Component` decorator to the following CSS classes:

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

```
styles: [`  
  .selected {  
    background-color: #CFD8DC !important;  
    color: white;  
  }  
  .heroes {  
    margin: 0 0 2em 0;  
    list-style-type: none;  
    padding: 0;  
    width: 15em;  
  }  
  .heroes li {  
    cursor: pointer;  
    position: relative;  
    left: 0;  
    background-color: #EEE;  
    margin: .5em;  
    padding: .3em 0;  
    height: 1.6em;  
    border-radius: 4px;  
  }  
  .heroes li.selected:hover {  
    background-color: #BBB8DC !important;  
    color: white;
```

```
}

.heroes li:hover {
  color: #607D8B;
  background-color: #DDD;
  left: .1em;
}

.heroes .text {
  position: relative;
  top: -3px;
}

.heroes .badge {
  display: inline-block;
  font-size: small;
  color: white;
  padding: 0.8em 0.7em 0 0.7em;
  background-color: #607D8B;
  line-height: 1em;
  position: relative;
  left: -1px;
  top: -4px;
  height: 1.8em;
  margin-right: .8em;
  border-radius: 4px 0 0 4px;
}

`]
```

Remember to use the backtick notation for multi-line strings.

Adding these styles makes the file much longer. In a later page you'll move the styles to a separate file.

When you assign styles to a component, they are scoped to that specific component. These styles apply only to the `AppComponent` and don't affect the outer HTML.

The template for displaying heroes should look like this:

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

```
<h2>My Heroes</h2>
<ul class="heroes">
  <li *ngFor="let hero of heroes">
    <span class="badge">{{hero.id}}</span> {{hero.name}}
  </li>
</ul>
```



Selecting a hero

The app now displays a list of heroes as well as a single hero in the details view. But the list and the details view are not connected. When users select a hero from the list, the selected hero should appear in the details view. This UI pattern is known as "master/detail." In this case, the *master* is the heroes list and the *detail* is the selected hero.

Next you'll connect the master to the detail through a `selectedHero` component property, which is bound to a click event.

Handle click events

Add a click event binding to the `` like this:

app.component.ts (template excerpt)

```
<li *ngFor="let hero of heroes" (click)="onSelect(hero)">  
  ...  
</li>
```



The parentheses identify the `` element's `click` event as the target. The `onSelect(hero)` expression calls the `AppComponent` method, `onSelect()`, passing the template input variable `hero`, as an argument. That's the same `hero` variable you defined previously in the `ngFor` directive.

Learn more about event binding at the [User Input](#) page and the [Event binding](#) section of the [Template Syntax](#) page.

Add a click handler to expose the selected hero

You no longer need the `hero` property because you're no longer displaying a single hero; you're displaying a list of heroes. But the user will be able to select one of the heroes by clicking on it. So replace the `hero` property with this simple `selectedHero` property:

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



```
selectedHero: Hero;
```

The hero names should all be unselected before the user picks a hero, so you won't initialize the `selectedHero` as you did with `hero`.

Add an `onSelect()` method that sets the `selectedHero` property to the `hero` that the user clicks.

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

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

The template still refers to the old `hero` property. Bind to the new `selectedHero` property instead as follows:

app.component.ts (template excerpt)

```
<h2>{{selectedHero.name}} details!</h2>  
<div><label>id: </label>{{selectedHero.id}}</div>  
<div>  
  <label>name: </label>  
  <input [(ngModel)]="selectedHero.name" placeholder="name"/>  
</div>
```

Hide the empty detail with `ngIf`

When the app loads, `selectedHero` is undefined. The selected hero is initialized when the user clicks a hero's name. Angular can't display properties of the undefined `selectedHero` and throws the following error, visible in the browser's console:

```
EXCEPTION: TypeError: Cannot read property 'name' of undefined in [null]
```

Although `selectedHero.name` is displayed in the template, you must keep the hero detail out of the DOM until there is a selected hero.

Wrap the HTML hero detail content of the template with a `<div>`. Then add the `ngIf` built-in directive and set it to the `selectedHero` property of the component.

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

```
<div *ngIf="selectedHero">
  <h2>{{selectedHero.name}} details!</h2>
  <div><label>id: </label>{{selectedHero.id}}</div>
  <div>
    <label>name: </label>
    <input [(ngModel)]="selectedHero.name" placeholder="name"/>
  </div>
</div>
```

Don't forget the asterisk (`*`) in front of `ngIf` .

The app no longer fails and the list of names displays again in the browser.

When there is no selected hero, the `ngIf` directive removes the hero detail HTML from the DOM. There are no hero detail elements or bindings to worry about.

When the user picks a hero, `selectedHero` becomes defined and `ngIf` puts the hero detail content into the DOM and evaluates the nested bindings.

Read more about `ngIf` and `ngFor` in the [Structural Directives](#) page and the [Built-in directives](#) section of the [Template Syntax](#) page.

Style the selected hero

While the selected hero details appear below the list, it's difficult to identify the selected hero within the list itself.

In the `styles` metadata that you added above, there is a custom CSS class named `selected`. To make the selected hero more visible, you'll apply this `selected` class to the `` when the user clicks on a hero name. For example, when the user clicks "Magneta", it should render with a distinctive but subtle background color like this:



In the template, add the following `[class.selected]` binding to the ``:

app.component.ts (setting the CSS class)

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



When the expression (`hero === selectedHero`) is `true`, Angular adds the `selected` CSS class. When the expression is `false`, Angular removes the `selected` class.

Read more about the `[class]` binding in the [Template Syntax](#) guide.

The final version of the `` looks like this:

app.component.ts (styling each hero)

```
<li *ngFor="let hero of heroes"
  [class.selected]="hero === selectedHero"
  (click)="onSelect(hero)">
  <span class="badge">{{hero.id}}</span> {{hero.name}}
</li>
```

After clicking "Magneta", the list should look like this:

Tour of Heroes

My Heroes

11 Mr. Nice

12 Narco

13 Bombasto

14 Celeritas

15 Magneta

16 RubberMan

17 Dynama

18 Dr IQ

19 Magma

20 Tornado

Here's the complete `app.component.ts` as of now:

src/app/app.component.ts

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



```
2.
3. export class Hero {
4.   id: number;
5.   name: string;
6. }
7.
8. const HEROES: Hero[] = [
9.   { id: 11, name: 'Mr. Nice' },
10.  { id: 12, name: 'Narco' },
11.  { id: 13, name: 'Bombasto' },
12.  { id: 14, name: 'Celeritas' },
13.  { id: 15, name: 'Magneta' },
14.  { id: 16, name: 'RubberMan' },
15.  { id: 17, name: 'Dynamite' },
16.  { id: 18, name: 'Dr IQ' },
17.  { id: 19, name: 'Magma' },
18.  { id: 20, name: 'Tornado' }
19. ];
20.
21. @Component({
22.   selector: 'my-app',
23.   template: `
24.     <h1>{{title}}</h1>
25.     <h2>My Heroes</h2>
26.     <ul class="heroes">
27.       <li *ngFor="let hero of heroes"
28.           [class.selected]="hero === selectedHero"
29.           (click)="onSelect(hero)">
30.             <span class="badge">{{hero.id}}</span> {{hero.name}}
31.       </li>
```

```
32.      </ul>
33.      <div *ngIf="selectedHero">
34.        <h2>{{selectedHero.name}} details!</h2>
35.        <div><label>id: </label>{{selectedHero.id}}</div>
36.        <div>
37.          <label>name: </label>
38.          <input [(ngModel)]="selectedHero.name" placeholder="name"/>
39.        </div>
40.      </div>
41.    `,
42.    styles: [`
43.      .selected {
44.        background-color: #CFD8DC !important;
45.        color: white;
46.      }
47.      .heroes {
48.        margin: 0 0 2em 0;
49.        list-style-type: none;
50.        padding: 0;
51.        width: 15em;
52.      }
53.      .heroes li {
54.        cursor: pointer;
55.        position: relative;
56.        left: 0;
57.        background-color: #EEE;
58.        margin: .5em;
59.        padding: .3em 0;
60.        height: 1.6em;
61.        border-radius: 4px;
```

```
62.    }
63.    .heroes li.selected:hover {
64.      background-color: #BBD8DC !important;
65.      color: white;
66.    }
67.    .heroes li:hover {
68.      color: #607D8B;
69.      background-color: #DDD;
70.      left: .1em;
71.    }
72.    .heroes .text {
73.      position: relative;
74.      top: -3px;
75.    }
76.    .heroes .badge {
77.      display: inline-block;
78.      font-size: small;
79.      color: white;
80.      padding: 0.8em 0.7em 0 0.7em;
81.      background-color: #607D8B;
82.      line-height: 1em;
83.      position: relative;
84.      left: -1px;
85.      top: -4px;
86.      height: 1.8em;
87.      margin-right: .8em;
88.      border-radius: 4px 0 0 4px;
89.    }
90.  `]
91. })
```

```
92. export class AppComponent {  
93.   title = 'Tour of Heroes';  
94.   heroes = HEROES;  
95.   selectedHero: Hero;  
96.  
97.   onSelect(hero: Hero): void {  
98.     this.selectedHero = hero;  
99.   }  
100. }
```

Summary

Here's what you achieved in this page:

- The Tour of Heroes app displays a list of selectable heroes.
- You added the ability to select a hero and show the hero's details.
- You learned how to use the built-in directives `ngIf` and `ngFor` in a component's template.

Your app should look like this [live example](#) / [download example](#).

Next step

You've expanded the Tour of Heroes app, but it's far from complete. An app shouldn't be one monolithic component. In the [next page](#), you'll split the app into subcomponents and make them work together.

This is the archived documentation for Angular v4. Please visit [angular.io](#) to see documentation for the current version of Angular.

Multiple Components

[Contents >](#)

[Where you left off](#)

[Make a hero detail component](#)

[Hero detail template](#)

•••

The `AppComponent` is doing *everything* at the moment. In the beginning, it showed details of a single hero. Then it became a master/detail form with both a list of heroes and the hero detail. Soon there will be new requirements and capabilities. You can't keep piling features on top of features in one component; that's not maintainable.

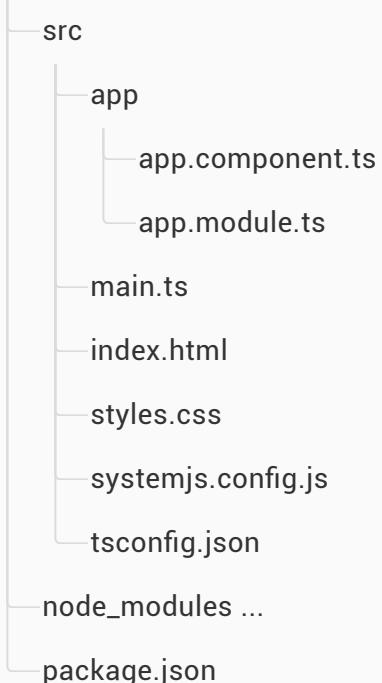
You'll need to break it up into sub-components, each focused on a specific task or workflow. Eventually, the `AppComponent` could become a simple shell that hosts those sub-components.

In this page, you'll take the first step in that direction by carving out the hero details into a separate, reusable component. When you're done, the app should look like this [live example](#) / [download example](#).

Where you left off

Before getting started on this page, verify that you have the following structure from earlier in the Tour of Heroes. If not, go back to the previous pages.

angular-tour-of-heroes



Keep the app transpiling and running while you build the Tour of Heroes by entering the `npm start` command in a terminal window [as you did before](#).

Make a hero detail component

Add a file named `hero-detail.component.ts` to the `app/` folder. This file will hold the new `HeroDetailComponent`.

The file and component names follow the standard described in the Angular [style guide](#).

- The component *class* name should be written in *upper camel case* and end in the word "Component".
The hero detail component class is `HeroDetailComponent`.

- The component *file* name should be spelled in *lower dash case*, each word separated by dashes, and end in `.component.ts`. The `HeroDetailComponent` class goes in the `hero-detail.component.ts` file.

Start writing the `HeroDetailComponent` as follows:

app/hero-detail.component.ts (initial version)

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

@Component({
  selector: 'hero-detail',
})
export class HeroDetailComponent {
```

To define a component, you always import the `Component` symbol.

The `@Component` decorator provides the Angular metadata for the component. The CSS selector name, `hero-detail`, will match the element tag that identifies this component within a parent component's template. [Near the end of this tutorial page](#), you'll add a `<hero-detail>` element to the `AppComponent` template.

Always `export` the component class because you'll always `import` it elsewhere.

Hero detail template

To move the hero detail view to the `HeroDetailComponent`, cut the hero detail *content* from the bottom of the `AppComponent` template and paste it into a new `template` property in the `@Component` metadata.

The `HeroDetailComponent` has a *hero*, not a *selected hero*. Replace the word, "selectedHero", with the word, "hero", everywhere in the template. When you're done, the new template should look like this:

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

```
@Component({
  selector: 'hero-detail',
  template: `
    <div *ngIf="hero">
      <h2>{{hero.name}} details!</h2>
      <div><label>id: </label>{{hero.id}}</div>
      <div>
        <label>name: </label>
        <input [(ngModel)]="hero.name" placeholder="name"/>
      </div>
    </div>
  `
})
```



Add the *hero* property

The `HeroDetailComponent` template binds to the component's `hero` property. Add that property to the `HeroDetailComponent` class like this:

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

```
hero: Hero;
```



The `hero` property is typed as an instance of `Hero`. The `Hero` class is still in the `app.component.ts` file. Now there are two components that need to reference the `Hero` class. The Angular [style guide](#) recommends one class per file anyway.

Move the `Hero` class from `app.component.ts` to its own `hero.ts` file.

src/app/hero.ts

```
export class Hero {  
  id: number;  
  name: string;  
}
```



Now that the `Hero` class is in its own file, the `AppComponent` and the `HeroDetailComponent` have to import it. Add the following `import` statement near the top of *both* the `app.component.ts` and the `hero-detail.component.ts` files.

src/app/hero-detail.component.ts

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



The `hero` property is an *input* property

[Later in this page](#), the parent `AppComponent` will tell the child `HeroDetailComponent` which hero to display by binding its `selectedHero` to the `hero` property of the `HeroDetailComponent`. The binding will look like this:

src/app/app.component.html

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



Putting square brackets around the `hero` property, to the left of the equal sign (=), makes it the *target* of a property binding expression. You must declare a *target* binding property to be an *input* property. Otherwise,

Angular rejects the binding and throws an error.

First, amend the `@angular/core` import statement to include the `Input` symbol.

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

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



Then declare that `hero` is an *input* property by preceding it with the `@Input` decorator that you imported earlier.

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

```
@Input() hero: Hero;
```



Read more about *input* properties in the [Attribute Directives](#) page.

That's it. The `hero` property is the only thing in the `HeroDetailComponent` class.

src/app/hero-detail.component.ts

```
export class HeroDetailComponent {  
  @Input() hero: Hero;  
}
```



All it does is receive a hero object through its `hero` input property and then bind to that property with its template.

Here's the complete `HeroDetailComponent`.

src/app/hero-detail.component.ts

```
1. import { Component, Input } from '@angular/core';
2.
3. import { Hero } from './hero';
4. @Component({
5.   selector: 'hero-detail',
6.   template: `
7.     <div *ngIf="hero">
8.       <h2>{{hero.name}} details!</h2>
9.       <div><label>id: </label>{{hero.id}}</div>
10.      <div>
11.        <label>name: </label>
12.        <input [(ngModel)]="hero.name" placeholder="name"/>
13.      </div>
14.    </div>
15.  `
16. })
17. export class HeroDetailComponent {
18.   @Input() hero: Hero;
19. }
```



Declare `HeroDetailComponent` in the `AppModule`

Every component must be declared in one—and only one—NgModule.

Open `app.module.ts` in your editor and import the `HeroDetailComponent` so you can refer to it.

src/app/app.module.ts

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



Add `HeroDetailComponent` to the module's `declarations` array.

src/app/app.module.ts

```
declarations: [
  AppComponent,
  HeroDetailComponent
],
```



In general, the `declarations` array contains a list of application components, pipes, and directives that belong to the module. A component must be declared in a module before other components can reference it. This module declares only the two application components, `AppComponent` and `HeroDetailComponent`.

Read more about NgModules in the [NgModules](#) guide.

Add the *HeroDetailComponent* to the *AppComponent*

The `AppComponent` is still a master/detail view. It used to display the hero details on its own, before you cut out that portion of the template. Now it will delegate to the `HeroDetailComponent`.

Recall that `hero-detail` is the CSS [selector](#) in the `HeroDetailComponent` metadata. That's the tag name of the element that represents the `HeroDetailComponent`.

Add a `<hero-detail>` element near the bottom of the `AppComponent` template, where the hero detail view used to be.

Coordinate the master `AppComponent` with the `HeroDetailComponent` by binding the `selectedHero` property of the `AppComponent` to the `hero` property of the `HeroDetailComponent`.

app.component.ts (excerpt)

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

Now every time the `selectedHero` changes, the `HeroDetailComponent` gets a new hero to display.

The revised `AppComponent` template should look like this:

app.component.ts (excerpt)

```
template: `

<h1>{{title}}</h1>
<h2>My Heroes</h2>
<ul class="heroes">
  <li *ngFor="let hero of heroes"
    [class.selected]="hero === selectedHero"
    (click)="onSelect(hero)">
    <span class="badge">{{hero.id}}</span> {{hero.name}}
  </li>
</ul>
<hero-detail [hero]="selectedHero"></hero-detail>
`,
```

What changed?

As [before](#), whenever a user clicks on a hero name, the hero detail appears below the hero list. But now the `HeroDetailView` is presenting those details.

Refactoring the original `AppComponent` into two components yields benefits, both now and in the future:

1. You simplified the `AppComponent` by reducing its responsibilities.
2. You can evolve the `HeroDetailComponent` into a rich hero editor without touching the parent `AppComponent`.
3. You can evolve the `AppComponent` without touching the hero detail view.
4. You can re-use the `HeroDetailComponent` in the template of some future parent component.

Review the app structure

Verify that you have the following structure:

```
angular-tour-of-heroes
├── src
│   ├── app
│   │   ├── app.component.ts
│   │   ├── app.module.ts
│   │   ├── hero.ts
│   │   └── hero-detail.component.ts
│   ├── main.ts
│   ├── index.html
│   ├── styles.css
│   └── systemjs.config.js
```

```
tsconfig.json  
node_modules ...  
package.json
```

Here are the code files discussed in this page.

[src/app/hero-detail.component.ts](#)[src/app/app.component.ts](#)[src/app/hero.ts](#)[src/app/app.module.ts](#)

```
1. import { Component, Input } from '@angular/core';  
2.  
3. import { Hero } from './hero';  
4. @Component({  
5.   selector: 'hero-detail',  
6.   template: `  
7.     <div *ngIf="hero">  
8.       <h2>{{hero.name}} details!</h2>  
9.       <div><label>id: </label>{{hero.id}}</div>  
10.      <div>  
11.        <label>name: </label>  
12.        <input [(ngModel)]="hero.name" placeholder="name"/>  
13.      </div>  
14.    </div>  
15.  `  
16. })  
17. export class HeroDetailComponent {  
18.   @Input() hero: Hero;  
19. }
```

Summary

Here's what you achieved in this page:

- You created a reusable component.
- You learned how to make a component accept input.
- You learned to declare the required application directives in an NgModule. You listed the directives in the `@NgModule` decorator's `declarations` array.
- You learned to bind a parent component to a child component.

Your app should look like this [live example](#) / [download example](#).

Next step

The Tour of Heroes app is more reusable with shared components, but its (mock) data is still hard coded within the `AppComponent`. That's not sustainable. Data access should be refactored to a separate service and shared among the components that need data.

You'll learn to create services in the [next tutorial](#) page.

This is the archived documentation for Angular v4. Please visit [angular.io](#) to see documentation for the current version of Angular.

Services

[Contents >](#)

[Where you left off](#)

[Keep the app transpiling and running](#)

[Creating a hero service](#)

•••

As the Tour of Heroes app evolves, you'll add more components that need access to hero data.

Instead of copying and pasting the same code over and over, you'll create a single reusable data service and inject it into the components that need it. Using a separate service keeps components lean and focused on supporting the view, and makes it easy to unit-test components with a mock service.

Because data services are invariably asynchronous, you'll finish the page with a *Promise*-based version of the data service.

When you're done with this page, the app should look like this [live example](#) / [download example](#).

Where you left off

Before continuing with the Tour of Heroes, verify that you have the following structure. If not, go back to the previous pages.

```
angular-tour-of-heroes
```

```
  src
    └── app
        ├── app.component.ts
        ├── app.module.ts
        ├── hero.ts
        └── hero-detail.component.ts
    main.ts
    index.html
    styles.css
    systemjs.config.js
    tsconfig.json
    node_modules ...
    package.json
```

Keep the app transpiling and running

Enter the following command in the terminal window:

```
npm start
```



This command runs the TypeScript compiler in "watch mode", recompiling automatically when the code changes. The command simultaneously launches the app in a browser and refreshes the browser when the code changes.

You can keep building the Tour of Heroes without pausing to recompile or refresh the browser.

Creating a hero service

The stakeholders want to show the heroes in various ways on different pages. Users can already select a hero from a list. Soon you'll add a dashboard with the top performing heroes and create a separate view for editing hero details. All three views need hero data.

At the moment, the `AppComponent` defines mock heroes for display. However, defining heroes is not the component's job, and you can't easily share the list of heroes with other components and views. In this page, you'll move the hero data acquisition business to a single service that provides the data and share that service with all components that need the data.

Create the HeroService

Create a file in the `app` folder called `hero.service.ts`.

The naming convention for service files is the service name in lowercase followed by `.service`. For a multi-word service name, use lower [dash-case](#). For example, the filename for `SpecialSuperHeroService` is `special-super-hero.service.ts`.

Name the class `HeroService` and export it for others to import.

`src/app/hero.service.ts (starting point)`

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

@Injectable()
export class HeroService {
}
```



Injectable services

Notice that you imported the Angular `Injectable` function and applied that function as an `@Injectable()` decorator.

Don't forget the parentheses. Omitting them leads to an error that's difficult to diagnose.

The `@Injectable()` decorator tells TypeScript to emit metadata about the service. The metadata specifies that Angular may need to inject other dependencies into this service.

Although the `HeroService` doesn't have any dependencies at the moment, applying the `@Injectable()` decorator from the start ensures consistency and future-proofing.

Getting hero data

Add a `getHeroes()` method stub.

src/app/hero.service.ts (getHeroes stub)



```
@Injectable()
export class HeroService {
  getHeroes(): void {} // stub
```

```
}
```

The `HeroService` could get `Hero` data from anywhere—a web service, local storage, or a mock data source. Removing data access from the component means you can change your mind about the implementation anytime, without touching the components that need hero data.

Move the mock hero data

Cut the `HEROES` array from `app.component.ts` and paste it to a new file in the `app` folder named `mock-heroes.ts`. Additionally, copy the `import {Hero} ...` statement because the heroes array uses the `Hero` class.

src/app/mock-heroes.ts

```
1. import { Hero } from './hero';
2.
3. export const HEROES: Hero[] = [
4.   { id: 11, name: 'Mr. Nice' },
5.   { id: 12, name: 'Narco' },
6.   { id: 13, name: 'Bombasto' },
7.   { id: 14, name: 'Celeritas' },
8.   { id: 15, name: 'Magneta' },
9.   { id: 16, name: 'RubberMan' },
10.  { id: 17, name: 'Dynamite' },
11.  { id: 18, name: 'Dr IQ' },
12.  { id: 19, name: 'Magma' },
13.  { id: 20, name: 'Tornado' }
14. ];
```

The `HEROES` constant is exported so it can be imported elsewhere, such as the `HeroService`.

In `app.component.ts`, where you cut the `HEROES` array, add an uninitialized `heroes` property:

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

```
heroes: Hero[];
```



Return mocked hero data

Back in the `HeroService`, import the mock `HEROES` and return it from the `getHeroes()` method. The `HeroService` looks like this:

src/app/hero.service.ts

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

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

@Injectable()
export class HeroService {
  getHeroes(): Hero[] {
    return HEROES;
  }
}
```



Import the hero service

You're ready to use the `HeroService` in other components, starting with `AppComponent`.

Import the `HeroService` so that you can reference it in the code.

src/app/app.component.ts (hero-service-import)

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



Don't use `new` with the `HeroService`

How should the `AppComponent` acquire a runtime concrete `HeroService` instance?

You could create a new instance of the `HeroService` with `new` like this:

src/app/app.component.ts

```
heroService = new HeroService(); // don't do this
```



However, this option isn't ideal for the following reasons:

- The component has to know how to create a `HeroService`. If you change the `HeroService` constructor, you must find and update every place you created the service. Patching code in multiple places is error prone and adds to the test burden.
- You create a service each time you use `new`. What if the service caches heroes and shares that cache with others? You couldn't do that.
- With the `AppComponent` locked into a specific implementation of the `HeroService`, switching implementations for different scenarios, such as operating offline or using different mocked versions for testing, would be difficult.

Inject the `HeroService`

Instead of using the `\n` line, you'll add two lines.

- Add a constructor that also defines a private property.
- Add to the component's `providers` metadata.

Add the constructor:

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

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



The constructor itself does nothing. The parameter simultaneously defines a private `heroService` property and identifies it as a `HeroService` injection site.

Now Angular knows to supply an instance of the `HeroService` when it creates an `AppComponent`.

Read more about dependency injection in the [Dependency Injection](#) page.

The `injector` doesn't know yet how to create a `HeroService`. If you ran the code now, Angular would fail with this error:

```
EXCEPTION: No provider for HeroService! (AppComponent -> HeroService)
```



To teach the injector how to make a `HeroService`, add the following `providers` array property to the bottom of the component metadata in the `@Component` call.

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



```
providers: [HeroService]
```

The `providers` array tells Angular to create a fresh instance of the `HeroService` when it creates an `AppComponent`. The `AppComponent`, as well as its child components, can use that service to get hero data.

getHeroes() in the `AppComponent`

The service is in a `heroService` private variable.

You could call the service and get the data in one line.

src/app/app.component.ts

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

You don't really need a dedicated method to wrap one line. Write it anyway:

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

```
getHeroes(): void {
  this.heroes = this.heroService.getHeroes();
}
```

The `ngOnInit` lifecycle hook

`AppComponent` should fetch and display hero data with no issues.

You might be tempted to call the `getHeroes()` method in a constructor, but a constructor should not contain complex logic, especially a constructor that calls a server, such as a data access method. The

constructor is for simple initializations, like wiring constructor parameters to properties.

To have Angular call `getHeroes()`, you can implement the Angular *ngOnInit lifecycle hook*. Angular offers interfaces for tapping into critical moments in the component lifecycle: at creation, after each change, and at its eventual destruction.

Each interface has a single method. When the component implements that method, Angular calls it at the appropriate time.

Read more about lifecycle hooks in the [Lifecycle Hooks](#) page.

Here's the essential outline for the `OnInit` interface (don't copy this into your code):

src/app/app.component.ts

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

export class AppComponent implements OnInit {
  ngOnInit(): void {
  }
}
```

Add the implementation for the `OnInit` interface to your export statement:

```
export class AppComponent implements OnInit {}
```

Write an `ngOnInit` method with the initialization logic inside. Angular will call it at the right time. In this case, initialize by calling `getHeroes()`.

src/app/app.component.ts (ng-on-init)

```
ngOnInit(): void {  
  this.getHeroes();  
}
```



The app should run as expected, showing a list of heroes and a hero detail view when you click on a hero name.

Async services and Promises

The `HeroService` returns a list of mock heroes immediately; its `getHeroes()` signature is synchronous.

src/app/app.component.ts

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

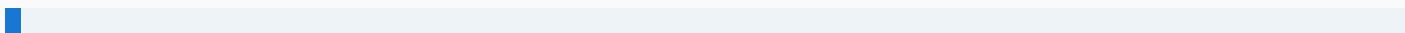


Eventually, the hero data will come from a remote server. When using a remote server, users don't have to wait for the server to respond; additionally, you aren't able to block the UI during the wait.

To coordinate the view with the response, you can use *Promises*, which is an asynchronous technique that changes the signature of the `getHeroes()` method.

The hero service makes a Promise

A *Promise* essentially promises to call back when the results are ready. You ask an asynchronous service to do some work and give it a callback function. The service does that work and eventually calls the function with the results or an error.



This is a simplified explanation. Read more about ES2015 Promises in the [Promises for asynchronous programming](#) page of [Exploring ES6](#).

Update the `HeroService` with this Promise-returning `getHeroes()` method:

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

```
getHeroes(): Promise<Hero[]> {
  return Promise.resolve(HEROES);
}
```



You're still mocking the data. You're simulating the behavior of an ultra-fast, zero-latency server, by returning an *immediately resolved Promise* with the mock heroes as the result.

Act on the Promise

As a result of the change to `HeroService`, `this.heroes` is now set to a `Promise` rather than an array of heroes.

src/app/app.component.ts (getHeroes - old)

```
getHeroes(): void {
  this.heroes = this.heroService.getHeroes();
}
```



You have to change the implementation to *act on the Promise when it resolves*. When the `Promise` resolves successfully, you'll have heroes to display.

Pass the callback function as an argument to the Promise's `then()` method:

src/app/app.component.ts (getHeroes - revised)

```
getHeroes(): void {
  this.heroService.getHeroes().then(heroes => this.heroes = heroes);
}
```



As described in [Arrow functions](#), the ES2015 arrow function in the callback is more succinct than the equivalent function expression and gracefully handles `this`.

The callback sets the component's `heroes` property to the array of heroes returned by the service.

The app is still running, showing a list of heroes, and responding to a name selection with a detail view.

At the end of this page, [Appendix: take it slow](#) describes what the app might be like with a poor connection.

Review the app structure

Verify that you have the following structure after all of your refactoring:

```
angular-tour-of-heroes
```

```
  src
```

```
    app
```

```
      app.component.ts
```

```
  app.module.ts
  hero.ts
  hero-detail.component.ts
  hero.service.ts
  mock-heroes.ts

  main.ts
  index.html
  styles.css
  systemjs.config.js
  tsconfig.json

  node_modules ...
  package.json
```

Here are the code files discussed in this page.

[src/app/hero.service.ts](#) [src/app/app.component.ts](#) [src/app/mock-heroes.ts](#)

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

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

@Injectable()
export class HeroService {
  getHeroes(): Promise<Hero[]> {
```



```
    return Promise.resolve(HEROES);
}
}
```

Summary

Here's what you achieved in this page:

- You created a service class that can be shared by many components.
- You used the `ngOnInit` lifecycle hook to get the hero data when the `AppComponent` activates.
- You defined the `HeroService` as a provider for the `AppComponent`.
- You created mock hero data and imported them into the service.
- You designed the service to return a Promise and the component to get the data from the Promise.

Your app should look like this [live example](#) / [download example](#).

Next step

The Tour of Heroes has become more reusable using shared components and services. The next goal is to create a dashboard, add menu links that route between the views, and format data in a template. As the app evolves, you'll discover how to design it to make it easier to grow and maintain.

Read about the Angular component router and navigation among the views in the [next tutorial](#) page.

Appendix: Take it slow

To simulate a slow connection, import the `Hero` symbol and add the following `getHeroesSlowly()` method to the `HeroService`.

app/hero.service.ts (getHeroesSlowly)

```
getHeroesSlowly(): Promise<Hero[]> {
  return new Promise(resolve => {
    // Simulate server latency with 2 second delay
    setTimeout(() => resolve(this.getHeroes()), 2000);
  });
}
```



Like `getHeroes()`, it also returns a `Promise`. But this `Promise` waits two seconds before resolving the `Promise` with mock heroes.

Back in the `AppComponent`, replace `getHeroes()` with `getHeroesSlowly()` and see how the app behaves.

This is the archived documentation for Angular v4. Please visit [angular.io](#) to see documentation for the current version of Angular.

Routing

[Contents >](#)

[Where you left off](#)

[Keep the app transpiling and running](#)

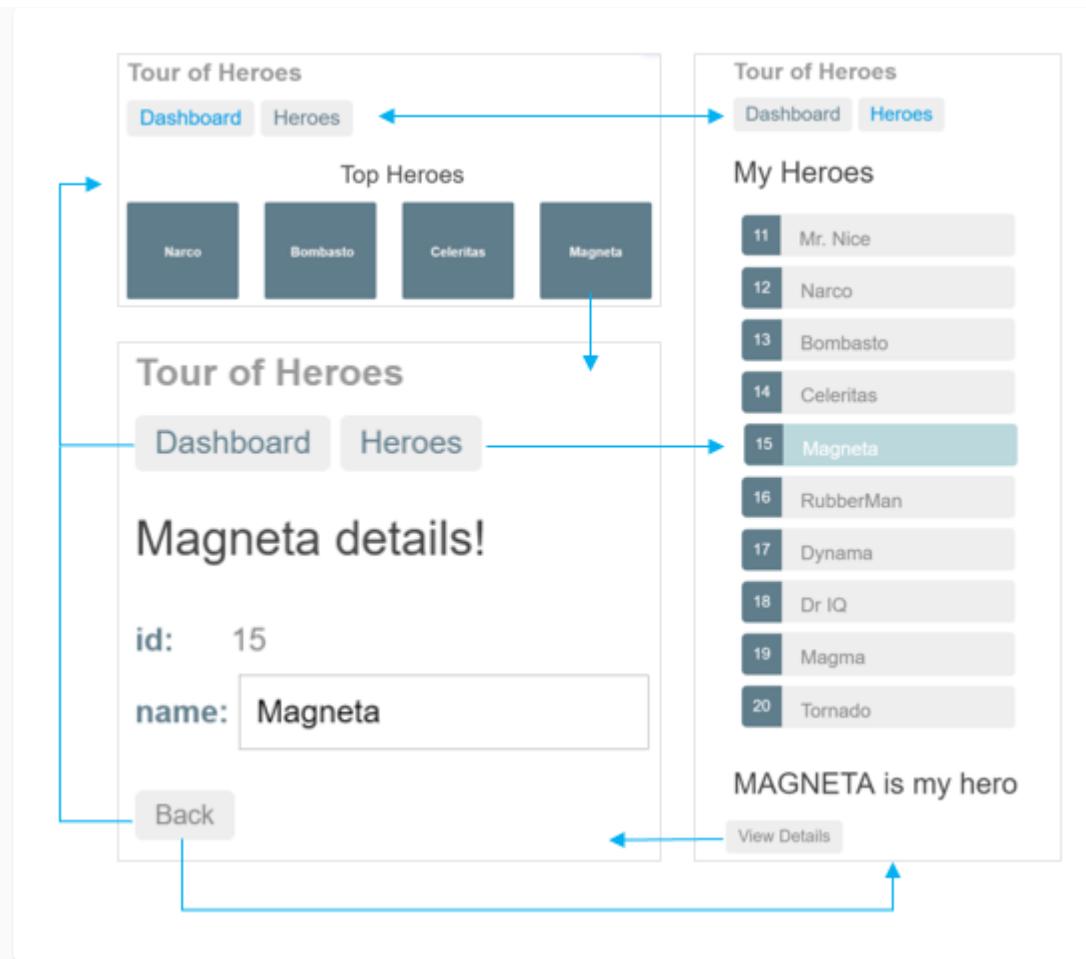
[Action plan](#)

•••

There are new requirements for the Tour of Heroes app:

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

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



To satisfy these requirements, you'll add Angular's router to the app.

For more information about the router, read the [Routing and Navigation](#) page.

When you're done with this page, the app should look like this [live example](#) / [download example](#).

Where you left off

Before continuing with the Tour of Heroes, verify that you have the following structure.

```
angular-tour-of-heroes
├── src
│   ├── app
│   │   ├── app.component.ts
│   │   ├── app.module.ts
│   │   ├── hero.service.ts
│   │   ├── hero.ts
│   │   ├── hero-detail.component.ts
│   │   └── mock-heroes.ts
│   ├── main.ts
│   ├── index.html
│   ├── styles.css
│   ├── systemjs.config.js
│   └── tsconfig.json
└── node_modules ...
└── package.json
```

Keep the app transpiling and running

Enter the following command in the terminal window:

```
npm start
```



This command runs the TypeScript compiler in "watch mode", recompiling automatically when the code changes. The command simultaneously launches the app in a browser and refreshes the browser when the code changes.

You can keep building the Tour of Heroes without pausing to recompile or refresh the browser.

Action plan

Here's the plan:

- Turn `AppComponent` into an application shell that only handles navigation.
- Relocate the *Heroes* concerns within the current `AppComponent` to a separate `HeroesComponent`.
- Add routing.
- Create a new `DashboardComponent`.
- Tie the *Dashboard* into the navigation structure.

Routing is another name for *navigation*. The router is the mechanism for navigating from view to view.

Splitting the *AppComponent*

The current app loads `AppComponent` and immediately displays the list of heroes.

The revised app should present a shell with a choice of views (*Dashboard* and *Heroes*) and then default to one of them.

The `AppComponent` should only handle navigation, so you'll move the display of `Heroes` out of `AppComponent` and into its own `HeroesComponent`.

HeroesComponent

`AppComponent` is already dedicated to `Heroes`. Instead of moving the code out of `AppComponent`, rename it to `HeroesComponent` and create a separate `AppComponent` shell.

Do the following:

- Rename the `app.component.ts` file to `heroes.component.ts`.
- Rename the `AppComponent` class as `HeroesComponent` (rename locally, *only* in this file).
- Rename the selector `my-app` as `my-heroes`.

src/app/heroes.component.ts (showing renamings only)

```
@Component({
  selector: 'my-heroes',
})
export class HeroesComponent implements OnInit {
```

Create `AppComponent`

The new `AppComponent` is the application shell. It will have some navigation links at the top and a display area below.

Perform these steps:

- Create the file `src/app/app.component.ts`.
- Define an exported `AppComponent` class.

- Add an `@Component` decorator above the class with a `my-app` selector.
- Move the following from `HeroesComponent` to `AppComponent` :
 - `title` class property.
 - `@Component` template `<h1>` element, which contains a binding to `title`.
- Add a `<my-heroes>` element to the app template just below the heading so you still see the heroes.
- Add `HeroesComponent` to the `declarations` array of `AppModule` so Angular recognizes the `<my-heroes>` tags.
- Add `HeroService` to the `providers` array of `AppModule` because you'll need it in every other view.
- Remove `HeroService` from the `HeroesComponent` `providers` array since it was promoted.
- Add the supporting `import` statements for `AppComponent`.

The first draft looks like this:

`src/app/app.component.ts (v1)` `src/app/app.module.ts (v1)`

```
1. import { Component } from '@angular/core';
2.
3. @Component({
4.   selector: 'my-app',
5.   template: `
6.     <h1>{{title}}</h1>
7.     <my-heroes></my-heroes>
8.   `,
9. })
10. export class AppComponent {
11.   title = 'Tour of Heroes';
12. }
```



The app still runs and displays heroes.

Add routing

Instead of displaying automatically, heroes should display after users click a button. In other words, users should be able to navigate to the list of heroes.

Use the Angular router to enable navigation.

The Angular router is an external, optional NgModule called `RouterModule`. The router is a combination of multiple provided services (`RouterModule`), multiple directives (`RouterOutlet`, `RouterLink`, `RouterLinkActive`), and a configuration (`Routes`). You'll configure the routes first.

<base href>

Open `index.html` and ensure there is a `<base href="...>` element (or a script that dynamically sets this element) at the top of the `<head>` section.

src/index.html (base-href)

```
<head>
  <base href="/">
```



BASE HREF IS ESSENTIAL

For more information, see the [Set the base href](#) section of the [Routing and Navigation](#) page.

Configure routes

Create a configuration file for the app routes.

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

Define the first route as a route to the heroes component.

src/app/app.module.ts (heroes route)

```
import { RouterModule } from '@angular/router';

RouterModule.forRoot([
  {
    path: 'heroes',
    component: HeroesComponent
  }
])
```



The `Routes` are an array of *route definitions*.

This route definition has the following parts:

- *Path*: The router matches this route's path to the URL in the browser address bar (`heroes`).
- *Component*: The component that the router should create when navigating to this route (`HeroesComponent`).

Read more about defining routes with `Routes` in the [Routing & Navigation](#) page.

Make the router available

Import the `RouterModule` and add it to the `AppModule` imports array.

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

```
1. import { NgModule }      from '@angular/core';
2. import { BrowserModule } from '@angular/platform-browser';
3. import { FormsModule }   from '@angular/forms';
4. import { RouterModule }  from '@angular/router';
5.
6. import { AppComponent }   from './app.component';
7. import { HeroDetailComponent } from './hero-detail.component';
8. import { HeroesComponent } from './heroes.component';
9. import { HeroService }    from './hero.service';
10.
11. @NgModule({
12.   imports: [
13.     BrowserModule,
14.     FormsModule,
15.     RouterModule.forRoot([
16.       {
17.         path: 'heroes',
18.         component: HeroesComponent
19.       }
20.     ])
21.   ],
22.   declarations: [
23.     AppComponent,
24.     HeroDetailComponent,
25.     HeroesComponent
26.   ],
27.   providers: [
28.     HeroService
```

```
29.  ],
30.  bootstrap: [ AppComponent ]
31. })
32. export class AppModule {
33. }
```

The `forRoot()` method is called because a configured router is provided at the app's root. The `forRoot()` method supplies the Router service providers and directives needed for routing, and performs the initial navigation based on the current browser URL.

Router outlet

If you paste the path, `/heroes`, into the browser address bar at the end of the URL, the router should match it to the `heroes` route and display the `HeroesComponent`. However, you have to tell the router where to display the component. To do this, you can add a `<router-outlet>` element at the end of the template.

`RouterOutlet` is one of the directives provided by the `RouterModule`. The router displays each component immediately below the `<router-outlet>` as users navigate through the app.

Router links

Users shouldn't have to paste a route URL into the address bar. Instead, add an anchor tag to the template that, when clicked, triggers navigation to the `HeroesComponent`.

The revised template looks like this:

src/app/app.component.ts (template-v2)



```
template: `

<h1>{{title}}</h1>
<a routerLink="/heroes">Heroes</a>
<router-outlet></router-outlet>

`
```

Note the `routerLink` binding in the anchor tag. The `RouterLink` directive (another of the `RouterModule` directives) is bound to a string that tells the router where to navigate when the user clicks the link.

Since the link is not dynamic, a routing instruction is defined with a one-time binding to the route path. Looking back at the route configuration, you can confirm that `'/heroes'` is the path of the route to the `HeroesComponent`.

Read more about dynamic router links and the link parameters array in the [Appendix: Link Parameters Array](#) section of the [Routing & Navigation](#) page.

Refresh the browser. The browser displays the app title and heroes link, but not the heroes list.

The browser's address bar shows `/`. The route path to `HeroesComponent` is `/heroes`, not `/`. Soon you'll add a route that matches the path `/`.

Click the *Heroes* navigation link. The address bar updates to `/heroes` and the list of heroes displays.

`AppComponent` now looks like this:

src/app/app.component.ts (v2)

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



```
2.  
3. @Component({  
4.   selector: 'my-app',  
5.   template: `  
6.     <h1>{{title}}</h1>  
7.     <a routerLink="/heroes">Heroes</a>  
8.     <router-outlet></router-outlet>  
9.   `,  
10. })  
11. export class AppComponent {  
12.   title = 'Tour of Heroes';  
13. }
```

The *AppComponent* is now attached to a router and displays routed views. For this reason, and to distinguish it from other kinds of components, this component type is called a *router component*.

Add a dashboard

Routing only makes sense when multiple views exist. To add another view, create a placeholder

DashboardComponent , which users can navigate to and from.

src/app/dashboard.component.ts (v1)

```
import { Component } from '@angular/core';  
  
@Component({  
  selector: 'my-dashboard',  
  template: '<h3>My Dashboard</h3>'  
})  
export class DashboardComponent { }
```

You'll make this component more useful later.

Configure the dashboard route

To teach `app.module.ts` to navigate to the dashboard, import the dashboard component and add the following route definition to the `Routes` array of definitions.

src/app/app.module.ts (Dashboard route)

```
{  
  path: 'dashboard',  
  component: DashboardComponent  
},
```

Also import and add `DashboardComponent` to the `AppModule`'s `declarations`.

src/app/app.module.ts (dashboard)

```
declarations: [  
  AppComponent,  
  DashboardComponent,  
  HeroDetailComponent,  
  HeroesComponent  
,
```

Add a redirect route

Currently, the browser launches with `/` in the address bar. When the app starts, it should show the dashboard and display a `/dashboard` URL in the browser address bar.

To make this happen, use a redirect route. Add the following to the array of route definitions:

src/app/app.module.ts (redirect)

```
{  
  path: '',  
  redirectTo: '/dashboard',  
  pathMatch: 'full'  
},
```

Read more about *redirects* in the [Redirecting routes](#) section of the [Routing & Navigation](#) page.

Add navigation to the template

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

src/app/app.component.ts (template-v3)

```
template: `  
  <h1>{{title}}</h1>  
  <nav>  
    <a routerLink="/dashboard">Dashboard</a>  
    <a routerLink="/heroes">Heroes</a>  
  </nav>  
  <router-outlet></router-outlet>
```

The `<nav>` tags don't do anything yet, but they'll be useful later when you style the links.

In your browser, go to the application root (`/`) and reload. The app displays the dashboard and you can navigate between the dashboard and the heroes.

Add heroes to the dashboard

To make the dashboard more interesting, you'll display the top four heroes at a glance.

Replace the `template` metadata with a `templateUrl` property that points to a new template file.

src/app/dashboard.component.ts (metadata)

```
@Component({
  selector: 'my-dashboard',
  templateUrl: './dashboard.component.html',
})
```

Create that file with this content:

src/app/dashboard.component.html

```
<h3>Top Heroes</h3>
<div class="grid grid-pad">
  <div *ngFor="let hero of heroes" class="col-1-4">
    <div class="module hero">
```

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

`*ngFor` is used again to iterate over a list of heroes and display their names. The extra `<div>` elements will help with styling later.

Sharing the `HeroService`

To populate the component's `heroes` array, you can re-use the `HeroService`.

Earlier, you removed the `HeroService` from the `providers` array of `HeroesComponent` and added it to the `providers` array of `AppModule`. That move created a singleton `HeroService` instance, available to all components of the app. Angular injects `HeroService` and you can use it in the `DashboardComponent`.

Get heroes

In `dashboard.component.ts`, add the following `import` statements.

src/app/dashboard.component.ts (imports)

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

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



Now create the `DashboardComponent` class like this:

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

```
export class DashboardComponent implements OnInit {  
  
  heroes: Hero[] = [];  
  
  constructor(private heroService: HeroService) { }  
  
  ngOnInit(): void {  
    this.heroService.getHeroes()  
      .then(heroes => this.heroes = heroes.slice(1, 5));  
  }  
}
```

This kind of logic is also used in the `HeroesComponent` :

- Define a `heroes` array property.
- Inject the `HeroService` in the constructor and hold it in a private `heroService` field.
- Call the service to get heroes inside the Angular `ngOnInit()` lifecycle hook.

In this dashboard you specify four heroes (2nd, 3rd, 4th, and 5th) with the `Array.slice` method.

Refresh the browser to see four hero names in the new dashboard.

Navigating to hero details

While the details of a selected hero displays at the bottom of the `HeroesComponent`, users should be able to navigate to the `HeroDetailComponent` in the following additional ways:

- From the dashboard to a selected hero.
- From the heroes list to a selected hero.
- From a "deep link" URL pasted into the browser address bar.

Routing to a hero detail

You can add a route to the `HeroDetailComponent` in `app.module.ts`, where the other routes are configured.

The new route is unusual in that you must tell the `HeroDetailComponent` which hero to show. You didn't have to tell the `HeroesComponent` or the `DashboardComponent` anything.

Currently, the parent `HeroesComponent` sets the component's `hero` property to a hero object with a binding like this:

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



But this binding won't work in any of the routing scenarios.

Parameterized route

You can add the hero's `id` to the URL. When routing to the hero whose `id` is 11, you could expect to see a URL such as this:

```
/detail/11
```



The `/detail/` part of the URL is constant. The trailing numeric `id` changes from hero to hero. You need to represent the variable part of the route with a *parameter* (or *token*) that stands for the hero's `id`.

Configure a route with a parameter

Use the following *route definition*.

src/app/app.module.ts (hero detail)

```
{  
  path: 'detail/:id',  
  component: HeroDetailComponent  
},
```



The colon (:) in the path indicates that `:id` is a placeholder for a specific hero `id` when navigating to the `HeroDetailComponent`.

Be sure to import the hero detail component before creating this route.

You're finished with the app routes.

You didn't add a 'Hero Detail' link to the template because users don't click a navigation *link* to view a particular hero; they click a *hero name*, whether the name displays on the dashboard or in the heroes list.

You don't need to add the hero clicks until the `HeroDetailComponent` is revised and ready to be navigated to.

Revise the *HeroDetailComponent*

Here's what the `HeroDetailComponent` looks like now:

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

```
1. import { Component, Input } from '@angular/core';  
2. import { Hero } from './hero';  
3.
```



```
4. @Component({
5.   selector: 'hero-detail',
6.   template: `
7.     <div *ngIf="hero">
8.       <h2>{{hero.name}} details!</h2>
9.       <div>
10.         <label>id: </label>{{hero.id}}
11.       </div>
12.       <div>
13.         <label>name: </label>
14.         <input [(ngModel)]="hero.name" placeholder="name"/>
15.       </div>
16.     </div>
17.   `
18. })
19. export class HeroDetailComponent {
20.   @Input() hero: Hero;
21. }
```

The template won't change. Hero names will display the same way. The major changes are driven by how you get hero names.

You'll no longer receive the hero in a parent component property binding. The new `HeroDetailComponent` should take the `id` parameter from the `paramMap` Observable in the `ActivatedRoute` service and use the `HeroService` to fetch the hero with that `id`.

Add the following imports:

src/app/hero-detail.component.ts

// Keep the Input import for now, you'll remove it later:



```
import { Component, Input, OnInit } from '@angular/core';
import { ActivatedRoute, ParamMap } from '@angular/router';
import { Location } from '@angular/common';

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

Inject the `ActivatedRoute`, `HeroService`, and `Location` services into the constructor, saving their values in private fields:

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

```
constructor(
  private heroService: HeroService,
  private route: ActivatedRoute,
  private location: Location
) {}
```

Import the `switchMap` operator to use later with the route parameters `Observable`.

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

```
import 'rxjs/add/operator/switchMap';
```

Tell the class to implement the `OnInit` interface.

src/app/hero-detail.component.ts

```
export class HeroDetailComponent implements OnInit {
```

Inside the `ngOnInit()` lifecycle hook, use the `paramMap` Observable to extract the `id` parameter value from the `ActivatedRoute` service and use the `HeroService` to fetch the hero with that `id`.

src/app/hero-detail.component.ts

```
ngOnInit(): void {
  this.route.paramMap
    .switchMap((params: ParamMap) => this.heroService.getHero(+params.get('id')))
    .subscribe(hero => this.hero = hero);
}
```



The `switchMap` operator maps the `id` in the Observable route parameters to a new `Observable`, the result of the `HeroService.getHero()` method.

If a user re-navigates to this component while a `getHero` request is still processing, `switchMap` cancels the old request and then calls `HeroService.getHero()` again.

The hero `id` is a number. Route parameters are always strings. So the route parameter value is converted to a number with the JavaScript `(+)` operator.

Do you need to unsubscribe?

As described in the [ActivatedRoute: the one-stop-shop for route information](#) section of the [Routing & Navigation](#) page, the `Router` manages the observables it provides and localizes the subscriptions.

The subscriptions are cleaned up when the component is destroyed, protecting against memory leaks, so you don't need to unsubscribe from the route `paramMap` Observable.

Add `HeroService.getHero()`

In the previous code snippet, `HeroService` doesn't have a `getHero()` method. To fix this issue, open `HeroService` and add a `getHero()` method that filters the heroes list from `getHeroes()` by `id`.

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

```
getHero(id: number): Promise<Hero> {
  return this.getHeroes()
    .then(heroes => heroes.find(hero => hero.id === id));
}
```

Find the way back

Users have several ways to navigate *to* the `HeroDetailComponent`.

To navigate somewhere else, users can click one of the two links in the `AppComponent` or click the browser's back button. Now add a third option, a `goBack()` method that navigates backward one step in the browser's history stack using the `Location` service you injected previously.

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

```
goBack(): void {
  this.location.back();
}
```

Going back too far could take users out of the app. In a real app, you can prevent this issue with the `CanDeactivate` guard. Read more on the [CanDeactivate](#) page.

You'll wire this method with an event binding to a *Back* button that you'll add to the component template.

```
<button (click)="goBack()">Back</button>
```



Migrate the template to its own file called `hero-detail.component.html`:

src/app/hero-detail.component.html

```
<div *ngIf="hero">
  <h2>{{hero.name}} details!</h2>
  <div>
    <label>id: </label>{{hero.id}}</div>
    <div>
      <label>name: </label>
      <input [(ngModel)]="hero.name" placeholder="name" />
    </div>
    <button (click)="goBack()">Back</button>
  </div>
```



Update the component metadata with a `templateUrl` pointing to the template file that you just created.

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

```
@Component({
  selector: 'hero-detail',
  templateUrl: './hero-detail.component.html',
})
```



Refresh the browser and see the results.

Select a dashboard hero

When a user selects a hero in the dashboard, the app should navigate to the `HeroDetailComponent` to view and edit the selected hero.

Although the dashboard heroes are presented as button-like blocks, they should behave like anchor tags.

When hovering over a hero block, the target URL should display in the browser status bar and the user should be able to copy the link or open the hero detail view in a new tab.

To achieve this effect, reopen `dashboard.component.html` and replace the repeated `<div *ngFor...>` tags with `<a>` tags. Change the opening `<a>` tag to the following:

src/app/dashboard.component.html (repeated `<a>` tag)

```
<a *ngFor="let hero of heroes" [routerLink]=["/detail", hero.id] class="col-1-4">
```

Notice the `[routerLink]` binding. As described in the [Router links](#) section of this page, top-level navigation in the `AppComponent` template has router links set to fixed paths of the destination routes, `/dashboard` and `/heroes`.

This time, you're binding to an expression containing a *link parameters array*. The array has two elements: the *path* of the destination route and a *route parameter* set to the value of the current hero's `id`.

The two array items align with the *path* and *:id* token in the parameterized hero detail route definition that you added to `app.module.ts` earlier:

src/app/app.module.ts (hero detail)

```
{
  path: 'detail/:id',
  component: HeroDetailComponent
```

},

Refresh the browser and select a hero from the dashboard; the app navigates to that hero's details.

Refactor routes to a *Routing Module*

Almost 20 lines of `AppModule` are devoted to configuring four routes. Most applications have many more routes and they add guard services to protect against unwanted or unauthorized navigations. (Read more about guard services in the [Route Guards](#) section of the [Routing & Navigation](#) page.) Routing considerations could quickly dominate this module and obscure its primary purpose, which is to establish key facts about the entire app for the Angular compiler.

It's a good idea to refactor the routing configuration into its own class. The current `RouterModule.forRoot()` produces an Angular `ModuleWithProviders`, a class dedicated to routing should be a *routing module*. For more information, see the [Milestone #2: The Routing Module](#) section of the [Routing & Navigation](#) page.

By convention, a routing module name contains the word "Routing" and aligns with the name of the module that declares the components navigated to.

Create an `app-routing.module.ts` file as a sibling to `app.module.ts`. Give it the following contents, extracted from the `AppModule` class:

src/app/app-routing.module.ts

```
1. import { NgModule }           from '@angular/core';
2. import { RouterModule, Routes } from '@angular/router';
3.
4. import { DashboardComponent } from './dashboard.component';
5. import { HeroesComponent }    from './heroes.component';
6. import { HeroDetailComponent } from './hero-detail.component';
```

```
7.  
8. const routes: Routes = [  
9.   { path: '', redirectTo: '/dashboard', pathMatch: 'full' },  
10.  { path: 'dashboard', component: DashboardComponent },  
11.  { path: 'detail/:id', component: HeroDetailComponent },  
12.  { path: 'heroes', component: HeroesComponent }  
13.];  
14.  
15. @NgModule({  
16.   imports: [ RouterModule.forRoot(routes) ],  
17.   exports: [ RouterModule ]  
18. })  
19. export class AppRoutingModule {}
```

The following points are typical of routing modules:

- The Routing Module pulls the routes into a variable. The variable clarifies the routing module pattern in case you export the module in the future.
- The Routing Module adds `RouterModule.forRoot(routes)` to `imports`.
- The Routing Module adds `RouterModule` to `exports` so that the components in the companion module have access to Router declarables, such as `RouterLink` and `RouterOutlet`.
- There are no `declarations`. Declarations are the responsibility of the companion module.
- If you have guard services, the Routing Module adds module `providers`. (There are none in this example.)

Update `AppModule`

Delete the routing configuration from `AppModule` and import the `AppRoutingModule`. Use an ES2015 `import` statement and add it to the `NgModule.imports` list.

Here is the revised `AppModule`, compared to its pre-refactor state:

src/app/app.module.ts (after) src/app/app.module.ts (before)

```
1. import { NgModule }      from '@angular/core';
2. import { BrowserModule } from '@angular/platform-browser';
3. import { FormsModule }   from '@angular/forms';
4.
5. import { AppComponent }    from './app.component';
6. import { DashboardComponent } from './dashboard.component';
7. import { HeroDetailComponent } from './hero-detail.component';
8. import { HeroesComponent }  from './heroes.component';
9. import { HeroService }     from './hero.service';
10.
11. import { AppRoutingModule } from './app-routing.module';
12.
13. @NgModule({
14.   imports: [
15.     BrowserModule,
16.     FormsModule,
17.     AppRoutingModule
18.   ],
19.   declarations: [
20.     AppComponent,
21.     DashboardComponent,
22.     HeroDetailComponent,
23.     HeroesComponent
24.   ],
25.   providers: [ HeroService ],
```

```
26.   bootstrap: [ AppComponent ]  
27. })  
28. export class AppModule { }
```

The revised and simplified `AppModule` is focused on identifying the key pieces of the app.

Select a hero in the *HeroesComponent*

In the `HeroesComponent`, the current template exhibits a "master/detail" style with the list of heroes at the top and details of the selected hero below.

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

```
template: `  
  <h1>{{title}}</h1>  
  <h2>My Heroes</h2>  
  <ul class="heroes">  
    <li *ngFor="let hero of heroes"  
        [class.selected]="hero === selectedHero"  
        (click)="onSelect(hero)">  
      <span class="badge">{{hero.id}}</span> {{hero.name}}  
    </li>  
  </ul>  
  <hero-detail [hero]="selectedHero"></hero-detail>  
`,
```

Delete the `<h1>` at the top.

Delete the last line of the template with the `<hero-detail>` tags.

You'll no longer show the full `HeroDetailComponent` here. Instead, you'll display the hero detail on its own page and route to it as you did in the dashboard.

However, when users select a hero from the list, they won't go to the detail page. Instead, they'll see a mini detail on *this* page and have to click a button to navigate to the *full detail* page.

Add the *mini detail*

Add the following HTML fragment at the bottom of the template where the `<hero-detail>` used to be:

src/app/heroes.component.ts

```
<div *ngIf="selectedHero">
  <h2>
    {{selectedHero.name | uppercase}} is my hero
  </h2>
  <button (click)="gotoDetail()">View Details</button>
</div>
```

After clicking a hero, users should see something like this below the hero list:

MR. NICE is my hero

`View Details`

Format with the uppercase pipe

The hero's name is displayed in capital letters because of the `uppercase` pipe that's included in the interpolation binding, right after the pipe operator (`|`).

```
 {{selectedHero.name | uppercase}} is my hero
```



Pipes are a good way to format strings, currency amounts, dates and other display data. Angular ships with several pipes and you can write your own.

Read more about pipes on the [Pipes](#) page.

Move content out of the component file

You still have to update the component class to support navigation to the `HeroDetailComponent` when users click the *View Details* button.

The component file is big. It's difficult to find the component logic amidst the noise of HTML and CSS.

Before making any more changes, migrate the template and styles to their own files.

First, move the template contents from `heroes.component.ts` into a new `heroes.component.html` file. Don't copy the backticks. As for `heroes.component.ts`, you'll come back to it in a minute. Next, move the styles contents into a new `heroes.component.css` file.

The two new files should look like this:

```
src/app/heroes.component.html    src/app/heroes.component.css
```

```
1. <h2>My Heroes</h2>
```



```
2. <ul class="heroes">
3.   <li *ngFor="let hero of heroes"
4.     [class.selected]="hero === selectedHero"
5.     (click)="onSelect(hero)">
6.       <span class="badge">{{hero.id}}</span> {{hero.name}}
7.     </li>
8.   </ul>
9.   <div *ngIf="selectedHero">
10.    <h2>
11.      {{selectedHero.name | uppercase}} is my hero
12.    </h2>
13.    <button (click)="gotoDetail()">View Details</button>
14.  </div>
```

Now, back in the component metadata for `heroes.component.ts`, delete `template` and `styles`, replacing them with `templateUrl` and `styleUrls` respectively. Set their properties to refer to the new files.

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

```
@Component({
  selector: 'my-heroes',
  templateUrl: './heroes.component.html',
  styleUrls: [ './heroes.component.css' ]
})
```

The `styleUrls` property is an array of style file names (with paths). You could list multiple style files from different locations if you needed them.

Update the `HeroesComponent` class

The `HeroesComponent` navigates to the `HeroesDetailComponent` in response to a button click. The button's click event is bound to a `gotoDetail()` method that navigates *imperatively* by telling the router where to go.

This approach requires the following changes to the component class:

1. Import the `Router` from the Angular router library.
2. Inject the `Router` in the constructor, along with the `HeroService`.
3. Implement `gotoDetail()` by calling the router `navigate()` method.

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

```
gotoDetail(): void {
  this.router.navigate(['/detail', this.selectedHero.id]);
}
```

Note that you're passing a two-element *link parameters array*—a path and the route parameter—to the router `navigate()` method, just as you did in the `[routerLink]` binding back in the `DashboardComponent`. Here's the revised `HeroesComponent` class:

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

```
1. export class HeroesComponent implements OnInit {
2.   heroes: Hero[];
3.   selectedHero: Hero;
4.
5.   constructor(
6.     private router: Router,
```

```
7.     private heroService: HeroService) { }
8.
9.     getHeroes(): void {
10.       this.heroService.getHeroes().then(heroes => this.heroes = heroes);
11.     }
12.
13.    ngOnInit(): void {
14.      this.getHeroes();
15.    }
16.
17.    onSelect(hero: Hero): void {
18.      this.selectedHero = hero;
19.    }
20.
21.    gotoDetail(): void {
22.      this.router.navigate(['/detail', this.selectedHero.id]);
23.    }
24. }
```

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

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

Style the app

The app is functional but it needs styling. The dashboard heroes should display in a row of rectangles. You've received around 60 lines of CSS for this purpose, including some simple media queries for responsive design.

As you now know, adding the CSS to the component `styles` metadata would obscure the component logic. Instead, edit the CSS in a separate `*.css` file.

Add a `dashboard.component.css` file to the `app` folder and reference that file in the component metadata's `styleUrls` array property like this:

src/app/dashboard.component.ts (styleUrls)

```
styleUrls: [ './dashboard.component.css' ]
```

Add stylish hero details

You've also been provided with CSS styles specifically for the `HeroDetailComponent`.

Add a `hero-detail.component.css` to the `app` folder and refer to that file inside the `styleUrls` array as you did for `DashboardComponent`. Also, in `hero-detail.component.ts`, remove the `hero` property `@Input` decorator and its import.

Here's the content for the component CSS files.

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

```
1. label {  
2.   display: inline-block;  
3.   width: 3em;  
4.   margin: .5em 0;  
5.   color: #607D8B;  
6.   font-weight: bold;  
7. }
```

```
8. input {  
9.   height: 2em;  
10.  font-size: 1em;  
11.  padding-left: .4em;  
12. }  
13. button {  
14.   margin-top: 20px;  
15.   font-family: Arial;  
16.   background-color: #eee;  
17.   border: none;  
18.   padding: 5px 10px;  
19.   border-radius: 4px;  
20.   cursor: pointer; cursor: hand;  
21. }  
22. button:hover {  
23.   background-color: #cf8dc;  
24. }  
25. button:disabled {  
26.   background-color: #eee;  
27.   color: #ccc;  
28.   cursor: auto;  
29. }
```

Style the navigation links

The provided CSS makes the navigation links in the `AppComponent` look more like selectable buttons. You'll surround those links in `<nav>` tags.

Add an `app.component.css` file to the `app` folder with the following content.

src/app/app.component.css (navigation styles)

```
1. h1 {  
2.   font-size: 1.2em;  
3.   color: #999;  
4.   margin-bottom: 0;  
5. }  
6. h2 {  
7.   font-size: 2em;  
8.   margin-top: 0;  
9.   padding-top: 0;  
10. }  
11. nav a {  
12.   padding: 5px 10px;  
13.   text-decoration: none;  
14.   margin-top: 10px;  
15.   display: inline-block;  
16.   background-color: #eee;  
17.   border-radius: 4px;  
18. }  
19. nav a:visited, a:link {  
20.   color: #607D8B;  
21. }  
22. nav a:hover {  
23.   color: #039be5;  
24.   background-color: #CFD8DC;  
25. }  
26. nav a.active {  
27.   color: #039be5;  
28. }
```

The `routerLinkActive` directive

The Angular router provides a `routerLinkActive` directive you can use to add a class to the HTML navigation element whose route matches the active route. All you have to do is define the style for it.

src/app/app.component.ts (active router links)

```
template: `

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

<nav>

  <a routerLink="/dashboard" routerLinkActive="active">Dashboard</a>
  <a routerLink="/heroes" routerLinkActive="active">Heroes</a>

</nav>

<router-outlet></router-outlet>

`,
```

Add a `styleUrls` property that refers to this CSS file as follows:

src/app/app.component.ts

```
styleUrls: ['./app.component.css'],
```

Global application styles

When you add styles to a component, you keep everything a component needs—HTML, the CSS, the code—together in one convenient place. It's easy to package it all up and re-use the component somewhere else.

You can also create styles at the *application level*/outside of any component.

The designers provided some basic styles to apply to elements across the entire app. These correspond to the full set of master styles that you installed earlier during [setup](#). Here's an excerpt:

src/styles.css (excerpt)

```
1. /* Master Styles */
2. h1 {
3.   color: #369;
4.   font-family: Arial, Helvetica, sans-serif;
5.   font-size: 250%;
6. }
7. h2, h3 {
8.   color: #444;
9.   font-family: Arial, Helvetica, sans-serif;
10.  font-weight: lighter;
11. }
12. body {
13.   margin: 2em;
14. }
15. body, input[text], button {
16.   color: #888;
17.   font-family: Cambria, Georgia;
18. }
19. /* everywhere else */
20. * {
21.   font-family: Arial, Helvetica, sans-serif;
22. }
```

Create the file `styles.css`. Ensure that the file contains the [master styles provided here](#). Also edit `index.html` to refer to this stylesheet.

src/index.html (link ref)

```
<link rel="stylesheet" href="styles.css">
```



Look at the app now. The dashboard, heroes, and navigation links are styled.

The screenshot shows the Angular application's dashboard. At the top, there is a header with the title "Tour of Heroes". Below the header, there are two navigation tabs: "Dashboard" (which is active and highlighted in blue) and "Heroes". The main content area is titled "Top Heroes" and displays four cards, each representing a hero: "Narco", "Bombasto", "Celeritas", and "Magneta". The cards are dark blue rectangular boxes with the hero names centered in white text.

Application structure and code

Review the sample source code in the [live example](#) / [download example](#) for this page. Verify that you have the following structure:

```
angular-tour-of-heroes
```

```
src
  └── app
      ├── app.component.css
      ├── app.component.ts
      ├── app.module.ts
      ├── app-routing.module.ts
      ├── dashboard.component.css
      ├── dashboard.component.html
      ├── dashboard.component.ts
      ├── hero.service.ts
      ├── hero.ts
      ├── hero-detail.component.css
      ├── hero-detail.component.html
      ├── hero-detail.component.ts
      ├── heroes.component.css
      ├── heroes.component.html
      ├── heroes.component.ts
      └── mock-heroes.ts

  └── main.ts
  └── index.html
  └── styles.css
  └── systemjs.config.js
  └── tsconfig.json
```

```
└── node_modules ...
```

```
    └── package.json
```

Summary

Here's what you achieved in this page:

- You added the Angular router to navigate among different components.
- You learned how to create router links to represent navigation menu items.
- You used router link parameters to navigate to the details of the user-selected hero.
- You shared the `HeroService` among multiple components.
- You moved HTML and CSS out of the component file and into their own files.
- You added the `uppercase` pipe to format data.

Your app should look like this [live example](#) / [download example](#).

Next step

You have much of the foundation you need to build an app. You're still missing a key piece: remote data access.

In the [next tutorial page](#) you'll replace the mock data with data retrieved from a server using http.

This is the archived documentation for Angular v4. Please visit [angular.io](#) to see documentation for the current version of Angular.

HTTP

[Contents >](#)

[Where you left off](#)

[Keep the app transpiling and running](#)

[Providing HTTP Services](#)

•••

In this page, you'll make the following improvements.

- Get the hero data from a server.
- Let users add, edit, and delete hero names.
- Save the changes to the server.

You'll teach the app to make corresponding HTTP calls to a remote server's web API.

When you're done with this page, the app should look like this [live example](#) / [download example](#).

Where you left off

In the [previous page](#), you learned to navigate between the dashboard and the fixed heroes list, editing a selected hero along the way. That's the starting point for this page.

Keep the app transpiling and running

Enter the following command in the terminal window:

```
npm start
```



This command runs the TypeScript compiler in "watch mode", recompiling automatically when the code changes. The command simultaneously launches the app in a browser and refreshes the browser when the code changes.

You can keep building the Tour of Heroes without pausing to recompile or refresh the browser.

Providing HTTP Services

The `HttpModule` is not a core NgModule. `HttpModule` is Angular's optional approach to web access. It exists as a separate add-on module called `@angular/http` and is shipped in a separate script file as part of the Angular npm package.

You're ready to import from `@angular/http` because `systemjs.config` configured *SystemJS* to load that library when you need it.

Register for HTTP services

The app will depend on the Angular `http` service, which itself depends on other supporting services. The `HttpModule` from the `@angular/http` library holds providers for a complete set of HTTP services.

To allow access to these services from anywhere in the app, add `HttpModule` to the `imports` list of the `AppModule`.

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

```
1. import { NgModule }      from '@angular/core';
2. import { BrowserModule } from '@angular/platform-browser';
3. import { FormsModule }   from '@angular/forms';
4. import { HttpClientModule } from '@angular/http';
5.
6. import { AppRoutingModule } from './app-routing.module';
7.
8. import { AppComponent }      from './app.component';
9. import { DashboardComponent } from './dashboard.component';
10. import { HeroesComponent }   from './heroes.component';
11. import { HeroDetailComponent } from './hero-detail.component';
12. import { HeroService }       from './hero.service';
13.
14. @NgModule({
15.   imports: [
16.     BrowserModule,
17.     FormsModule,
18.     HttpClientModule,
19.     AppRoutingModule
20.   ],
21.   declarations: [
22.     AppComponent,
23.     DashboardComponent,
24.     HeroDetailComponent,
25.     HeroesComponent,
26.   ],
27.   providers: [ HeroService ],
28.   bootstrap: [ AppComponent ]
```

```
29. })
30. export class AppModule { }
```

Notice that you also supply `HttpModule` as part of the *imports* array in root NgModule `AppModule`.

Simulate the web API

We recommend registering app-wide services in the root `AppModule providers`.

Until you have a web server that can handle requests for hero data, the HTTP client will fetch and save data from a mock service, the *in-memory web API*.

Update `src/app/app.module.ts` with this version, which uses the mock service:

src/app/app.module.ts (v2)

```
1. import { NgModule }      from '@angular/core';
2. import { BrowserModule } from '@angular/platform-browser';
3. import { FormsModule }   from '@angular/forms';
4. import { HttpModule }    from '@angular/http';
5.
6. import { AppRoutingModule } from './app-routing.module';
7.
8. // Imports for loading & configuring the in-memory web api
9. import { InMemoryWebApiModule } from 'angular-in-memory-web-api';
10. import { InMemoryDataService } from './in-memory-data.service';
11.
12. import { AppComponent }     from './app.component';
13. import { DashboardComponent } from './dashboard.component';
14. import { HeroesComponent }  from './heroes.component';
```

```
15. import { HeroDetailComponent } from './hero-detail.component';
16. import { HeroService } from './hero.service';
17.
18. @NgModule({
19.   imports: [
20.     BrowserModule,
21.     FormsModule,
22.     HttpClientModule,
23.     InMemoryWebApiModule.forRoot(InMemoryDataService),
24.     AppRoutingModule
25.   ],
26.   declarations: [
27.     AppComponent,
28.     DashboardComponent,
29.     HeroDetailComponent,
30.     HeroesComponent,
31.   ],
32.   providers: [ HeroService ],
33.   bootstrap: [ AppComponent ]
34. })
35. export class AppModule { }
```

Rather than require a real API server, this example simulates communication with the remote server by adding the `InMemoryWebApiModule` to the module `imports`, effectively replacing the `Http` client's XHR backend service with an in-memory alternative.

`InMemoryWebApiModule.forRoot(InMemoryDataService),`



The `forRoot()` configuration method takes an `InMemoryDataService` class that primes the in-memory database. Add the file `in-memory-data.service.ts` in `app` with the following content:

src/app/in-memory-data.service.ts

```
import { InMemoryDbService } from 'angular-in-memory-web-api';
export class InMemoryDataService implements InMemoryDbService {
  createDb() {
    const heroes = [
      { id: 0, name: 'Zero' },
      { id: 11, name: 'Mr. Nice' },
      { id: 12, name: 'Narco' },
      { id: 13, name: 'Bombasto' },
      { id: 14, name: 'Celeritas' },
      { id: 15, name: 'Magneta' },
      { id: 16, name: 'RubberMan' },
      { id: 17, name: 'Dynamite' },
      { id: 18, name: 'Dr IQ' },
      { id: 19, name: 'Magma' },
      { id: 20, name: 'Tornado' }
    ];
    return {heroes};
  }
}
```

This file replaces `mock-heroes.ts`, which is now safe to delete. Added hero "Zero" to confirm that the data service can handle a hero with `id==0`.

The in-memory web API is only useful in the early stages of development and for demonstrations such as this Tour of Heroes. Don't worry about the details of this backend substitution; you can skip it when you have a real web API server.

Heroes and HTTP

In the current `HeroService` implementation, a Promise resolved with mock heroes is returned.

src/app/hero.service.ts (old getHeroes)

```
getHeroes(): Promise<Hero[]> {
  return Promise.resolve(HEROES);
}
```



This was implemented in anticipation of ultimately fetching heroes with an HTTP client, which must be an asynchronous operation.

Now convert `getHeroes()` to use HTTP.

src/app/hero.service.ts (updated getHeroes and new class members)

```
1. private heroesUrl = 'api/heroes'; // URL to web api
2.
3. constructor(private http: Http) { }
4.
5. getHeroes(): Promise<Hero[]> {
6.   return this.http.get(this.heroesUrl)
7.     .toPromise()
```



```
8.         .then(response => response.json().data as Hero[])
9.         .catch(this.handleError);
10.    }
11.
12.   private handleError(error: any): Promise<any> {
13.     console.error('An error occurred', error); // for demo purposes only
14.     return Promise.reject(error.message || error);
15.   }

```

Update the import statements as follows:

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

```
import { Injectable }      from '@angular/core';
import { Headers, Http }  from '@angular/http';

import 'rxjs/add/operator/toPromise';

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

Refresh the browser. The hero data should successfully load from the mock server.

HTTP Promise

The Angular `http.get` returns an RxJS `Observable`. *Observables* are a powerful way to manage asynchronous data flows. You'll read about *Observables* later in this page.

For now, you've converted the `Observable` to a `Promise` using the `toPromise` operator.

```
.toPromise()
```

The Angular `Observable` doesn't have a `toPromise` operator out of the box.

There are many operators like `toPromise` that extend `Observable` with useful capabilities. To use those capabilities, you have to add the operators themselves. That's as easy as importing them from the RxJS library like this:

```
import 'rxjs/add/operator/toPromise';
```



You'll add more operators, and learn why you must do so, [later in this tutorial](#).

Extracting the data in the `then` callback

In the `Promise`'s `then()` callback, you call the `json` method of the `HTTP | Response` to extract the data within the response.

```
.then(response => response.json().data as Hero[])
```



The response JSON has a single `data` property, which holds the array of heroes that the caller wants. So you grab that array and return it as the resolved Promise value.

Note the shape of the data that the server returns. This particular in-memory web API example returns an object with a `data` property. Your API might return something else. Adjust the code to match your web API.

The caller is unaware that you fetched the heroes from the (mock) server. It receives a Promise of *heroes* just as it did before.

Error Handling

At the end of `getHeroes()`, you `catch` server failures and pass them to an error handler.

```
.catch(this.handleError);
```



This is a critical step. You must anticipate HTTP failures, as they happen frequently for reasons beyond your control.

```
private handleError(error: any): Promise<any> {
  console.error('An error occurred', error); // for demo purposes only
  return Promise.reject(error.message || error);
}
```



This demo service logs the error to the console; in real life, you would handle the error in code. For a demo, this works.

The code also includes an error to the caller in a rejected promise, so that the caller can display a proper error message to the user.

Get hero by id

When the `HeroDetailComponent` asks the `HeroService` to fetch a hero, the `HeroService` currently fetches all heroes and filters for the one with the matching `id`. That's fine for a simulation, but it's wasteful to ask a real server for all heroes when you only want one. Most web APIs support a *get-by-id* request in the form `api/hero/:id` (such as `api/hero/11`).

Update the `HeroService.getHero()` method to make a *get-by-id* request:

src/app/hero.service.ts

```
getHero(id: number): Promise<Hero> {
  const url = `${this.heroesUrl}/${id}`;
  return this.http.get(url)
    .toPromise()
    .then(response => response.json().data as Hero)
    .catch(this.handleError);
}
```

This request is almost the same as `getHeroes()`. The hero id in the URL identifies which hero the server should update.

Also, the `data` in the response is a single hero object rather than an array.

Unchanged `getHeroes` API

Although you made significant internal changes to `getHeroes()` and `getHero()`, the public signatures didn't change. You still return a Promise from both methods. You won't have to update any of the components that call them.

Now it's time to add the ability to create and delete heroes.

Updating hero details

Try editing a hero's name in the hero detail view. As you type, the hero name is updated in the view heading. But if you click the Back button, the changes are lost.

Updates weren't lost before. What changed? When the app used a list of mock heroes, updates were applied directly to the hero objects within the single, app-wide, shared list. Now that you're fetching data from a server, if you want changes to persist, you must write them back to the server.

Add the ability to save hero details

At the end of the hero detail template, add a save button with a `click` event binding that invokes a new component method named `save()`.

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

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

Add the following `save()` method, which persists hero name changes using the hero service `update()` method and then navigates back to the previous view.

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

```
save(): void {
  this.heroService.update(this.hero)
    .then(() => this.goBack());
}
```

Add a hero service `update()` method

The overall structure of the `update()` method is similar to that of `getHeroes()`, but it uses an HTTP `put()` to persist server-side changes.

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

```
private headers = new Headers({'Content-Type': 'application/json'});

update(hero: Hero): Promise<Hero> {
  const url = `${this.heroesUrl}/${hero.id}`;
  return this.http
    .put(url, JSON.stringify(hero), {headers: this.headers})
    .toPromise()
    .then(() => hero)
    .catch(this.handleError);
}
```



To identify which hero the server should update, the hero `id` is encoded in the URL. The `put()` body is the JSON string encoding of the hero, obtained by calling `JSON.stringify`. The body content type (`application/json`) is identified in the request header.

Refresh the browser, change a hero name, save your change, and click the browser Back button. Changes should now persist.

Add the ability to add heroes

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

Insert the following into the heroes component HTML, just after the heading:

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

```
<div>
  <label>Hero name:</label> <input #heroName />
  <button (click)="add(heroName.value); heroName.value=''">
    Add
  </button>
</div>
```

In response to a click event, call the component's click handler and then clear the input field so that it's ready for another name.

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

```
add(name: string): void {
  name = name.trim();
  if (!name) { return; }
  this.heroService.create(name)
    .then(hero => {
      this.heroes.push(hero);
      this.selectedHero = null;
    });
}
```

When the given name is non-blank, the handler delegates creation of the named hero to the hero service, and then adds the new hero to the array.

Implement the `create()` method in the `HeroService` class.

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

```
create(name: string): Promise<Hero> {
```

```
    return this.http
      .post(this.heroesUrl, JSON.stringify({name: name}), {headers: this.headers})
      .toPromise()
      .then(res => res.json().data as Hero)
      .catch(this.handleError);
  }
```

Refresh the browser and create some heroes.

Add the ability to delete a hero

Each hero in the heroes view should have a delete button.

Add the following button element to the heroes component HTML, after the hero name in the repeated

`` element.

```
<button class="delete"
  (click)="delete(hero); $event.stopPropagation()">x</button>
```

The `` element should now look like this:

src/app/heroes.component.html (li-element)

```
<li *ngFor="let hero of heroes" (click)="onSelect(hero)"
  [class.selected]="hero === selectedHero">
  <span class="badge">{{hero.id}}</span>
  <span>{{hero.name}}</span>
  <button class="delete"
```

```
(click)="delete(hero); $event.stopPropagation()">x</button>
</li>
```

In addition to calling the component's `delete()` method, the delete button's click handler code stops the propagation of the click event—you don't want the `` click handler to be triggered because doing so would select the hero that the user will delete.

The logic of the `delete()` handler is a bit trickier:

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

```
delete(hero: Hero): void {
  this.heroService
    .delete(hero.id)
    .then(() => {
      this.heroes = this.heroes.filter(h => h !== hero);
      if (this.selectedHero === hero) { this.selectedHero = null; }
    });
}
```

Of course you delegate hero deletion to the hero service, but the component is still responsible for updating the display: it removes the deleted hero from the array and resets the selected hero, if necessary.

To place the delete button at the far right of the hero entry, add this CSS:

src/app/heroes.component.css (additions)

```
button.delete {
  float:right;
  margin-top: 2px;
  margin-right: .8em;
```

```
background-color: gray !important;  
color:white;  
}
```

Hero service *delete()* method

Add the hero service's `delete()` method, which uses the `delete()` HTTP method to remove the hero from the server:

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

```
delete(id: number): Promise<void> {  
  const url = `${this.heroesUrl}/${id}`;  
  return this.http.delete(url, {headers: this.headers})  
    .toPromise()  
    .then(() => null)  
    .catch(this.handleError);  
}
```

Refresh the browser and try the new delete functionality.

Observables

Each `Http` service method returns an `Observable` of `HTTP Response` objects.

The `HeroService` converts that `Observable` into a `Promise` and returns the promise to the caller. This section shows you how, when, and why to return the `Observable` directly.

Background

An *Observable* is a stream of events that you can process with array-like operators.

Angular core has basic support for observables. Developers augment that support with operators and extensions from the [RxJS library](#). You'll see how shortly.

Recall that the `HeroService` chained the `toPromise` operator to the `Observable` result of `http.get()`. That operator converted the `Observable` into a `Promise` and you passed that promise back to the caller.

Converting to a `Promise` is often a good choice. You typically ask `http.get()` to fetch a single chunk of data. When you receive the data, you're done. The calling component can easily consume a single result in the form of a `Promise`.

But requests aren't always done only once. You may start one request, cancel it, and make a different request before the server has responded to the first request.

A *request-cancel-new-request* sequence is difficult to implement with `Promise`s, but easy with `Observable`s.

Add the ability to search by name

You're going to add a *hero search* feature to the Tour of Heroes. As the user types a name into a search box, you'll make repeated HTTP requests for heroes filtered by that name.

Start by creating `HeroSearchService` that sends search queries to the server's web API.

src/app/hero-search.service.ts

```
1. import { Injectable } from '@angular/core';
2. import { Http }      from '@angular/http';
3.
4. import { Observable }   from 'rxjs/Observable';
5. import 'rxjs/add/operator/map';
6.
```

```
7. import { Hero }           from './hero';
8.
9. @Injectable()
10. export class HeroSearchService {
11.
12.   constructor(private http: Http) {}
13.
14.   search(term: string): Observable<Hero[]> {
15.     return this.http
16.       .get(`api/heroes/?name=${term}`)
17.       .map(response => response.json().data as Hero[]);
18.   }
19. }
```

The `http.get()` call in `HeroSearchService` is similar to the one in the `HeroService`, although the URL now has a query string.

More importantly, you no longer call `toPromise()`. Instead you return the *Observable* from the the `http.get()`, after chaining it to another RxJS operator, `map()`, to extract heroes from the response data. RxJS operator chaining makes response processing easy and readable. See the [discussion below about operators](#).

HeroSearchComponent

Create a `HeroSearchComponent` that calls the new `HeroSearchService`.

The component template is simple—just a text box and a list of matching search results.

src/app/hero-search.component.html

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



```
<h4>Hero Search</h4>
<input #searchBox id="search-box" (keyup)="search(searchBox.value)" />
<div>
  <div *ngFor="let hero of heroes | async"
    (click)="gotoDetail(hero)" class="search-result" >
    {{hero.name}}
  </div>
</div>
</div>
```

Also, add styles for the new component.

src/app/hero-search.component.css

```
1. .search-result{
2.   border-bottom: 1px solid gray;
3.   border-left: 1px solid gray;
4.   border-right: 1px solid gray;
5.   width: 195px;
6.   height: 16px;
7.   padding: 5px;
8.   background-color: white;
9.   cursor: pointer;
10. }
11.
12. .search-result:hover {
13.   color: #eee;
14.   background-color: #607D8B;
15. }
16.
```

```
17. #search-box{  
18.   width: 200px;  
19.   height: 20px;  
20. }
```

As the user types in the search box, a *keyup* event binding calls the component's `search()` method with the new search box value.

As expected, the `*ngFor` repeats hero objects from the component's `heroes` property.

But as you'll soon see, the `heroes` property is now an *Observable* of hero arrays, rather than just a hero array. The `*ngFor` can't do anything with an `Observable` until you route it through the `async` pipe (`AsyncPipe`). The `async` pipe subscribes to the `Observable` and produces the array of heroes to `*ngFor`.

Create the `HeroSearchComponent` class and metadata.

src/app/hero-search.component.ts

```
1. import { Component, OnInit } from '@angular/core';  
2. import { Router }           from '@angular/router';  
3.  
4. import { Observable }       from 'rxjs/Observable';  
5. import { Subject }          from 'rxjs/Subject';  
6.  
7. // Observable class extensions  
8. import 'rxjs/add/observable/of';  
9.  
10. // Observable operators  
11. import 'rxjs/add/operator/catch';  
12. import 'rxjs/add/operator/debounceTime';
```

```
13. import 'rxjs/add/operator/distinctUntilChanged';
14.
15. import { HeroSearchService } from './hero-search.service';
16. import { Hero } from './hero';
17.
18. @Component({
19.   selector: 'hero-search',
20.   templateUrl: './hero-search.component.html',
21.   styleUrls: [ './hero-search.component.css' ],
22.   providers: [HeroSearchService]
23. })
24. export class HeroSearchComponent implements OnInit {
25.   heroes: Observable<Hero[]>;
26.   private searchTerms = new Subject<string>();
27.
28.   constructor(
29.     private heroSearchService: HeroSearchService,
30.     private router: Router) {}
31.
32.   // Push a search term into the observable stream.
33.   search(term: string): void {
34.     this.searchTerms.next(term);
35.   }
36.
37.   ngOnInit(): void {
38.     this.heroes = this.searchTerms
39.       .debounceTime(300)          // wait 300ms after each keystroke before
40.       .distinctUntilChanged()    // ignore if next search term is same as
                                // previous
```

```

41.         .switchMap(term => term   // switch to new observable each time the
   term changes
42.             // return the http search observable
43.             ? this.heroSearchService.search(term)
44.             // or the observable of empty heroes if there was no search term
45.             : Observable.of<Hero>([]))
46.         .catch(error => {
47.             // TODO: add real error handling
48.             console.log(error);
49.             return Observable.of<Hero>([]);
50.         });
51.     }
52.
53.     gotoDetail(hero: Hero): void {
54.         let link = ['/detail', hero.id];
55.         this.router.navigate(link);
56.     }
57. }
```

Search terms

Focus on the `searchTerms` :

```

private searchTerms = new Subject<string>();

// Push a search term into the observable stream.
search(term: string): void {
    this.searchTerms.next(term);
}
```

A `Subject` is a producer of an *observable* event stream; `searchTerms` produces an `Observable` of strings, the filter criteria for the name search.

Each call to `search()` puts a new string into this subject's *observable* stream by calling `next()`.

Initialize the `heroes` property (`ngOnInit`)

A `Subject` is also an `Observable`. You can turn the stream of search terms into a stream of `Hero` arrays and assign the result to the `heroes` property.

```
1. heroes: Observable<Hero[]>;
2.
3. ngOnInit(): void {
4.   this.heroes = this.searchTerms
5.     .debounceTime(300)           // wait 300ms after each keystroke before
6.       considering the term
7.     .distinctUntilChanged()    // ignore if next search term is same as
8.       previous
9.     .switchMap(term => term    // switch to new observable each time the term
10.      changes
11.        // return the http search observable
12.        ? this.heroSearchService.search(term)
13.          // or the observable of empty heroes if there was no search term
14.          : Observable.of<Hero[]>([]))
15.        .catch(error => {
16.          // TODO: add real error handling
17.          console.log(error);
18.          return Observable.of<Hero[]>([]);
19.        });
20.  }
```

Passing every user keystroke directly to the `HeroSearchService` would create an excessive amount of HTTP requests, taxing server resources and burning through the cellular network data plan.

Instead, you can chain `Observable` operators that reduce the request flow to the string `Observable`. You'll make fewer calls to the `HeroSearchService` and still get timely results. Here's how:

- `debounceTime(300)` waits until the flow of new string events pauses for 300 milliseconds before passing along the latest string. You'll never make requests more frequently than 300ms.
- `distinctUntilChanged` ensures that a request is sent only if the filter text changed.
- `switchMap()` calls the search service for each search term that makes it through `debounce` and `distinctUntilChanged`. It cancels and discards previous search observables, returning only the latest search service observable.

With the [switchMap operator](#) (formerly known as `flatMapLatest`), every qualifying key event can trigger an `http()` method call. Even with a 300ms pause between requests, you could have multiple HTTP requests in flight and they may not return in the order sent.

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

If the search text is empty, the `http()` method call is also short circuited and an observable containing an empty array is returned.

Note that until the service supports that feature, *cancelling* the `HeroSearchService` Observable doesn't actually abort a pending HTTP request. For now, unwanted results are discarded.

- `catch` intercepts a failed observable. The simple example prints the error to the console; a real life app would do better. Then to clear the search result, you return an observable containing an empty array.

Import RxJS operators

Most RxJS operators are not included in Angular's base `Observable` implementation. The base implementation includes only what Angular itself requires.

When you need more RxJS features, extend `Observable` by *importing* the libraries in which they are defined. Here are all the RxJS imports that *this* component needs:

src/app/hero-search.component.ts (rxjs imports)

```
import { Observable }      from 'rxjs/Observable';
import { Subject }         from 'rxjs/Subject';

// Observable class extensions
import 'rxjs/add/observable/of';

// Observable operators
import 'rxjs/add/operator/catch';
import 'rxjs/add/operator/debounceTime';
import 'rxjs/add/operator/distinctUntilChanged';
```

The `import 'rxjs/add/...'` syntax may be unfamiliar. It's missing the usual list of symbols between the braces: `{...}`.

You don't need the operator symbols themselves. In each case, the mere act of importing the library loads and executes the library's script file which, in turn, adds the operator to the `Observable` class.

Add the search component to the dashboard

Add the hero search HTML element to the bottom of the `DashboardComponent` template.

src/app/dashboard.component.html

```
<h3>Top Heroes</h3>
<div class="grid grid-pad">
  <a *ngFor="let hero of heroes" [routerLink]=["/detail", hero.id]" class="col-1-4">
    <div class="module hero">
      <h4>{{hero.name}}</h4>
    </div>
  </a>
</div>
<hero-search></hero-search>
```

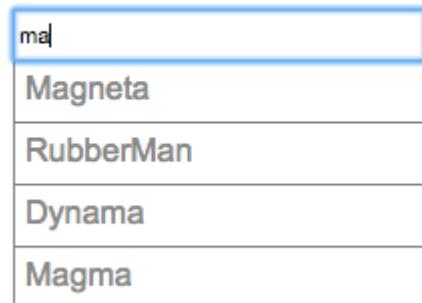
Finally, import `HeroSearchComponent` from `hero-search.component.ts` and add it to the `declarations` array.

src/app/app.module.ts (search)

```
declarations: [
  AppComponent,
  DashboardComponent,
  HeroDetailComponent,
  HeroesComponent,
  HeroSearchComponent
],
```

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

Hero Search



App structure and code

Review the sample source code in the [live example](#) / [download example](#) for this page. Verify that you have the following structure:

```
angular-tour-of-heroes
  src
    app
      app.component.ts
      app.component.css
      app.module.ts
      app-routing.module.ts
      dashboard.component.css
      dashboard.component.html
```

```
  |- dashboard.component.ts
  |- hero.ts
  |- hero-detail.component.css
  |- hero-detail.component.html
  |- hero-detail.component.ts
  |- hero-search.component.html (new)
  |- hero-search.component.css (new)
  |- hero-search.component.ts (new)
  |- hero-search.service.ts (new)
  |- hero.service.ts
  |- heroes.component.css
  |- heroes.component.html
  |- heroes.component.ts
  |- in-memory-data.service.ts (new)

  main.ts
  index.html
  styles.css
  systemjs.config.js
  tsconfig.json

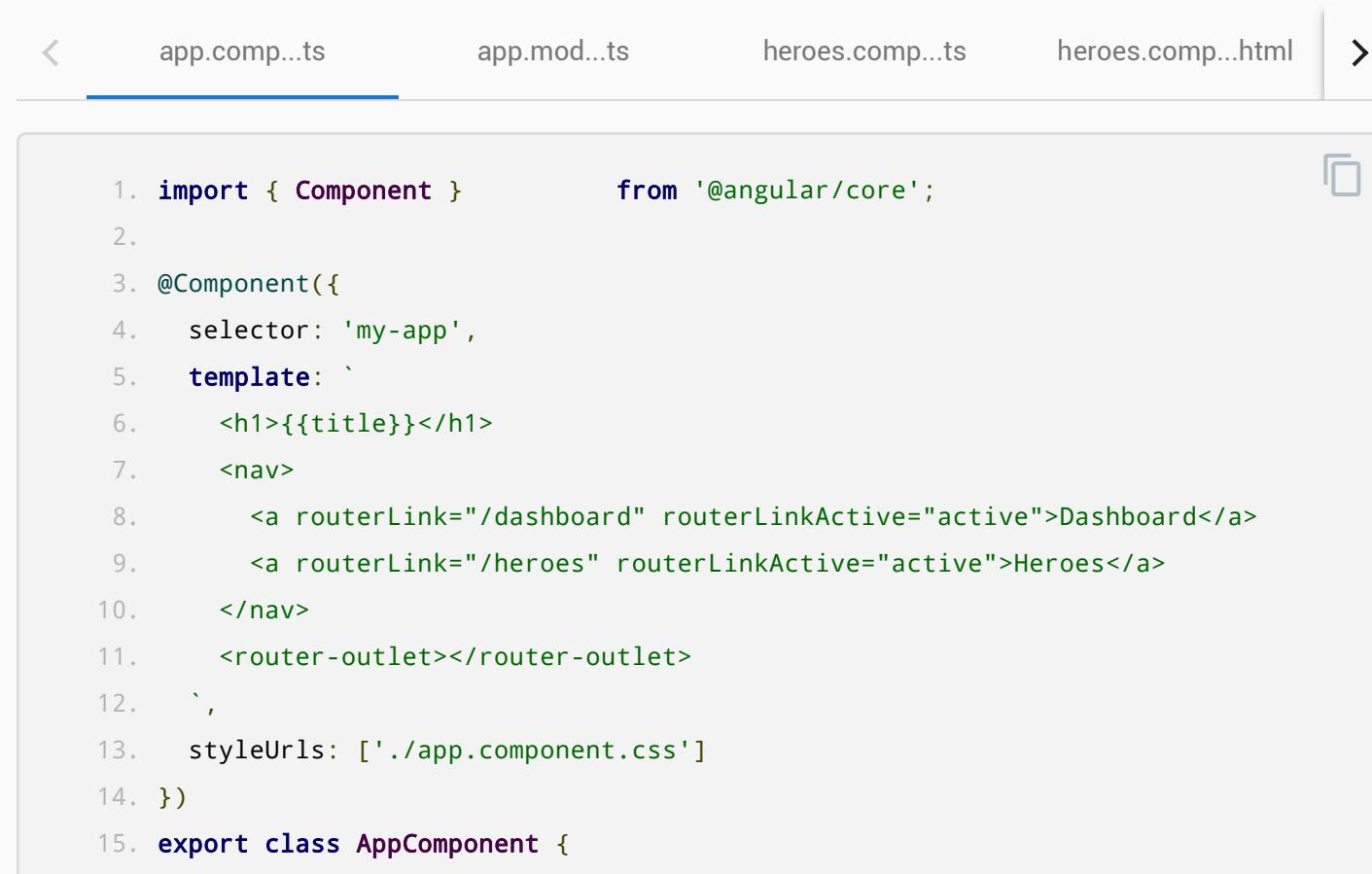
  node_modules ...
  package.json
```

Summary

You're at the end of your journey, and you've accomplished a lot.

- You added the necessary dependencies to use HTTP in the app.
- You refactored `HeroService` to load heroes from a web API.
- You extended `HeroService` to support `post()`, `put()`, and `delete()` methods.
- You updated the components to allow adding, editing, and deleting of heroes.
- You configured an in-memory web API.
- You learned how to use Observables.

Here are the files you added or changed in this page.



The screenshot shows a browser-based file viewer with four tabs at the top: `app.comp...ts`, `app.mod...ts`, `heroes.comp...ts`, and `heroes.comp...html`. The `app.comp...ts` tab is active, indicated by a blue underline. Below the tabs is a code editor window displaying the following TypeScript code:

```
1. import { Component }           from '@angular/core';
2.
3. @Component({
4.   selector: 'my-app',
5.   template: `
6.     <h1>{{title}}</h1>
7.     <nav>
8.       <a routerLink="/dashboard" routerLinkActive="active">Dashboard</a>
9.       <a routerLink="/heroes" routerLinkActive="active">Heroes</a>
10.    </nav>
11.    <router-outlet></router-outlet>
12.  `,
13.   styleUrls: ['./app.component.css']
14. })
15. export class AppComponent {
```

```
16.   title = 'Tour of Heroes';
17. }
```

hero-search.service.ts hero-search.component.ts hero-search.component.html hero-search.compon

```
1. import { Injectable } from '@angular/core';
2. import { Http }      from '@angular/http';
3.
4. import { Observable }    from 'rxjs/Observable';
5. import 'rxjs/add/operator/map';
6.
7. import { Hero }          from './hero';
8.
9. @Injectable()
10. export class HeroSearchService {
11.
12.   constructor(private http: Http) {}
13.
14.   search(term: string): Observable<Hero[]> {
15.     return this.http
16.       .get(`api/heroes/?name=${term}`)
17.       .map(response => response.json().data as Hero[]);
18.   }
19. }
```



Next step

That concludes the "Tour of Heroes" tutorial. You're ready to learn more about Angular development in the fundamentals section, starting with the [Architecture](#) guide.

This is the archived documentation for Angular v4. Please visit angular.io to see documentation for the current version of Angular.

Architecture Overview

[Contents >](#)

Modules

NgModules vs. JavaScript modules

Angular libraries

•••

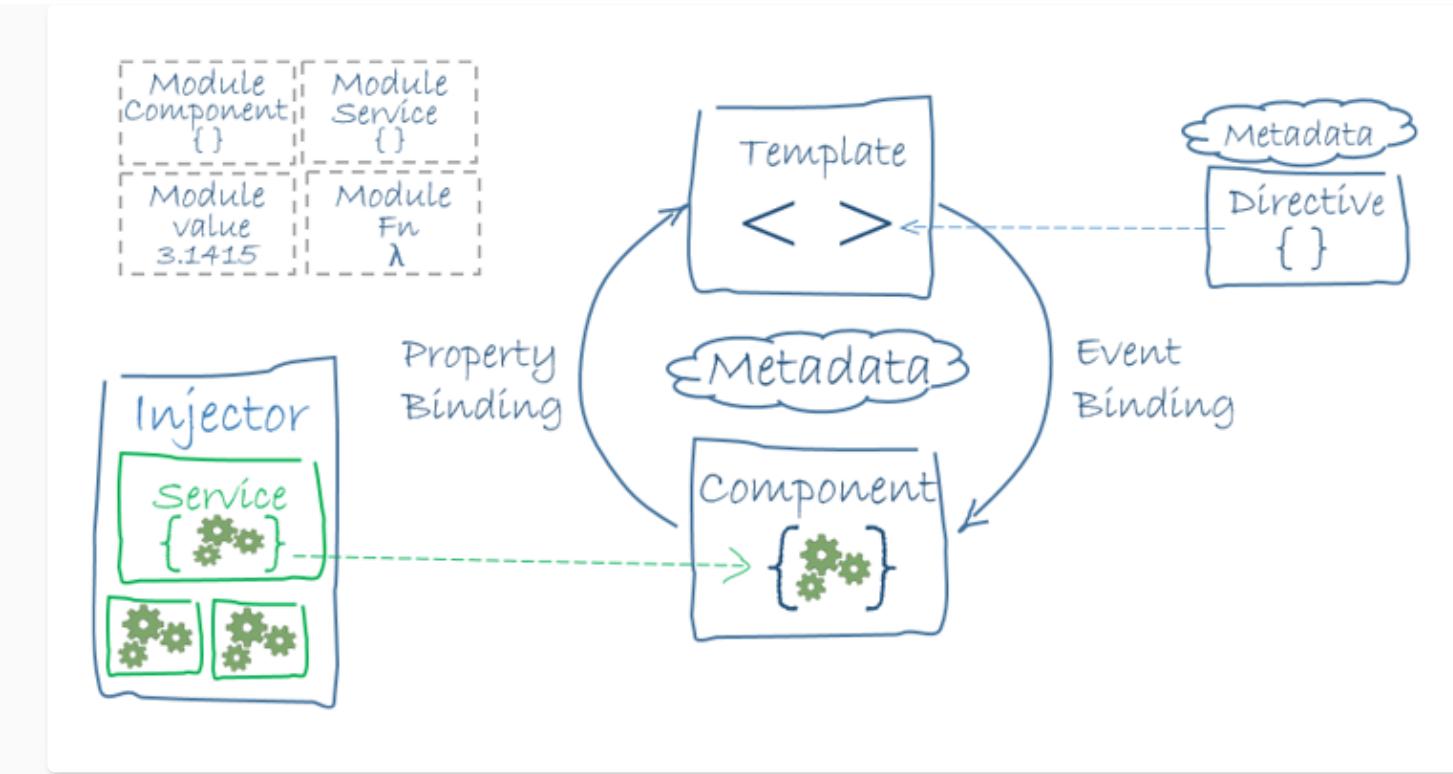
Angular is a framework for building client applications in HTML and either JavaScript or a language like TypeScript that compiles to JavaScript.

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

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.

Module Component { }

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:

src/app/app.module.ts

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

@NgModule({
  imports:      [ BrowserModule ],
  providers:    [ Logger ],
  declarations: [ AppComponent ],
  exports:      [ AppComponent ],
  bootstrap:   [ AppComponent ]
})
export class AppModule { }
```



The `export` of `AppComponent` is just to show how to export; 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.

src/main.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app/app.module';

platformBrowserDynamic().bootstrapModule(AppModule);
```



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.

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



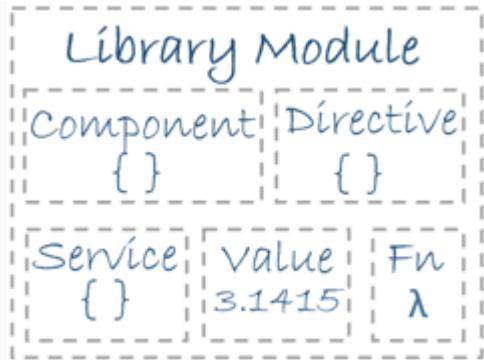
```
export class AppModule { }
```



[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:

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



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

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



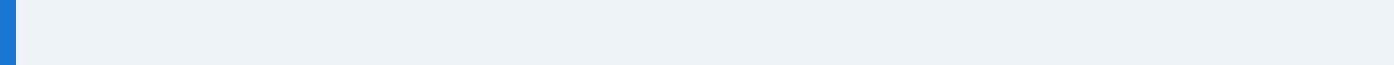
In the example of the simple root module above, the application module needs material from within that `BrowserModule`. To access that material, add it to the `@NgModule` metadata `imports` like this.

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



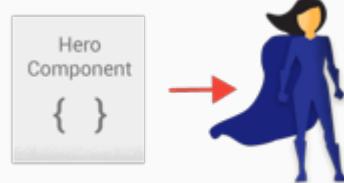
In this way you're using both the Angular and JavaScript module systems *together*.

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](#) 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.

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.

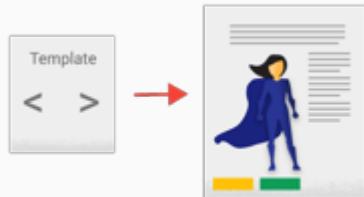
src/app/hero-list.component.ts (class)

```
export class HeroListComponent implements OnInit {  
  heroes: Hero[];  
  selectedHero: Hero;  
  
  constructor(private service: HeroService) {}  
  
  ngOnInit() {  
    this.heroes = this.service.getHeroes();  
  }  
}
```

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

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

src/app/hero-list.component.html

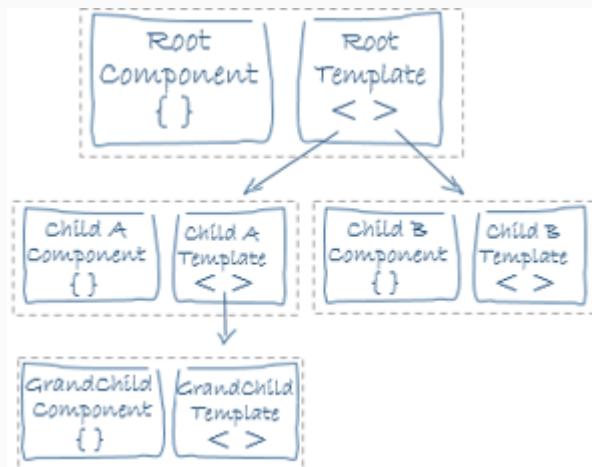
```
<h2>Hero List</h2>  
  
<p><i>Pick a hero from the list</i></p>  
<ul>  
  <li *ngFor="let hero of heroes" (click)="selectHero(hero)">  
    {{hero.name}}  
  </li>  
</ul>  
  
<hero-detail *ngIf="selectedHero" [hero]="selectedHero"></hero-detail>
```

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`:

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

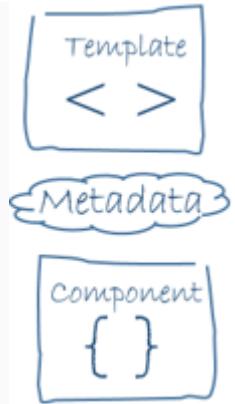
```
@Component({
  selector:    'hero-list',
  templateUrl: './hero-list.component.html',
  providers:  [ HeroService ]
})
export class HeroListComponent implements OnInit {
  /* . . . */
}
```

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.

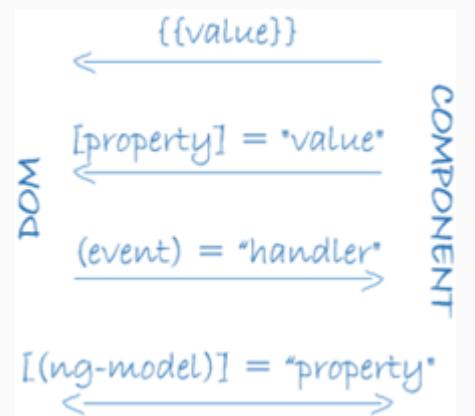
Apply other metadata decorators in a similar fashion to guide Angular behavior.

`@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:

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

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

- The `{{hero.name}}` *interpolation* displays the component's `hero.name` property value within the `` 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.

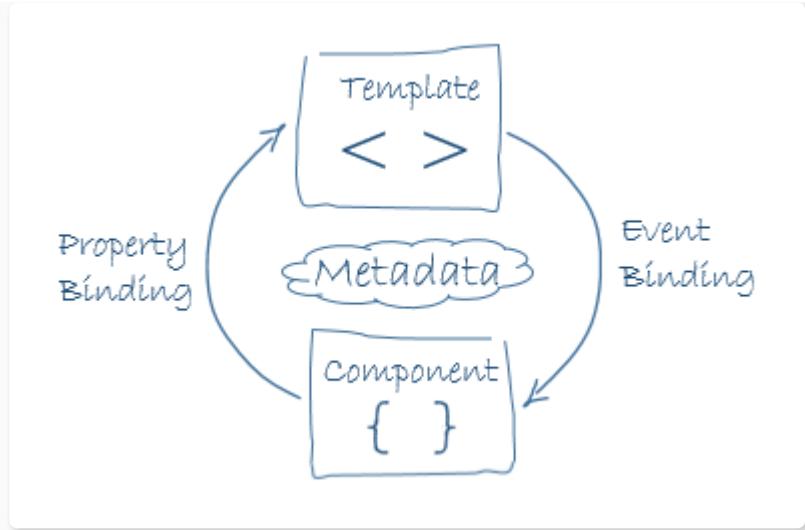
Two-way data binding is an important fourth form that combines property and event binding in a single notation, using the `ngModel` directive. Here's an example from the `HeroDetailComponent` template:

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

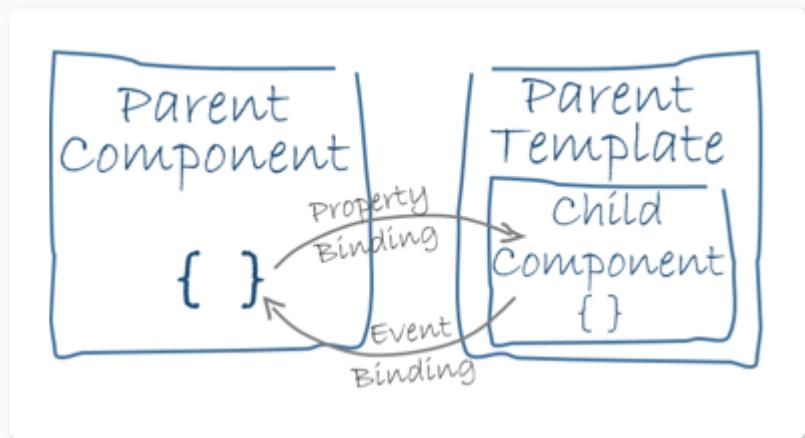
```
<input [(ngModel)]="hero.name">
```

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

Angular processes *a//* data bindings once per JavaScript event cycle, from the root of the application component tree through all child components.

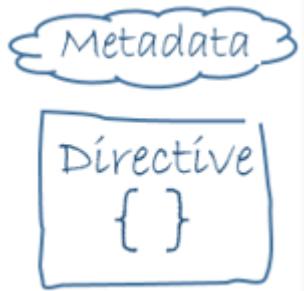


Data binding plays an important role in communication between a template and its component.



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:

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

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



- `*ngFor` tells Angular to stamp out one `` 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.

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

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



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:

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

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

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.

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

```
export class HeroService {  
  private heroes: Hero[] = [];  
  
  constructor(  
    private backend: BackendService,  
    private logger: Logger) {}  
  
  getHeroes() {  
    this.backend.getAll(Hero).then( (heroes: Hero[]) => {  
      this.logger.log(`Fetched ${heroes.length} heroes.`);  
      this.heroes.push(...heroes); // fill cache  
    })  
  }  
}
```

```
});  
return this.heroes;  
}  
}
```

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`:

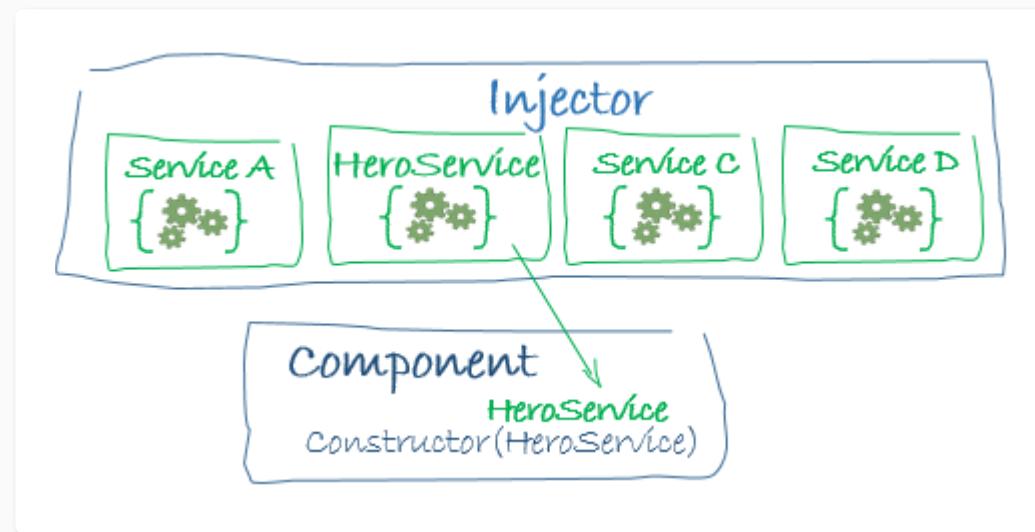
```
constructor(private service: HeroService) { }
```



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.

src/app/app.module.ts (module providers)

```
providers: [
  BackendService,
  HeroService,
  Logger
],
```

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

src/app/hero-list.component.ts (component providers)

```
@Component({
  selector:    'hero-list',
  templateUrl: './hero-list.component.html',
  providers:  [ HeroService ]
})
```

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*.

This is the archived documentation for Angular v4. Please visit [angular.io](#) to see documentation for the current version of Angular.

Displaying Data

[Contents >](#)

[Showing component properties with interpolation](#)

[Template inline or template file?](#)

[Constructor or variable initialization?](#)

•••

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

Showing component properties with interpolation

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

```
 {{myHero}} .
```

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

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

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

src/app/app.component.ts

```
1. import { Component } from '@angular/core';
2.
3. @Component({
4.   selector: 'my-app',
5.   template: `
6.     <h1>{{title}}</h1>
7.     <h2>My favorite hero is: {{myHero}}</h2>
8.   `
9. })
10. export class AppComponent {
11.   title = 'Tour of Heroes';
12.   myHero = 'Windstorm';
13. }
```



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

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

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

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



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

which makes the HTML more readable.

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

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

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

The CSS `selector` in the `@Component` decorator specifies an element named `<my-app>`. That element is a placeholder in the body of your `index.html` file:

src/index.html (body)

```
<body>
  <my-app>loading...</my-app>
</body>
```

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

Constructor or variable initialization?

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

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

```
export class AppCtorComponent {  
  title: string;  
  myHero: string;
```



```
constructor() {
  this.title = 'Tour of Heroes';
  this.myHero = 'Windstorm';
}
}
```

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

Showing an array property with *ngFor

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

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

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

Now use the Angular `ngFor` directive in the template to display each item in the `heroes` list.

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

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

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

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

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

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



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

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

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

In this case, `ngFor` is displaying an array, but `ngFor` can repeat items for any [iterable](#) object.

Now the heroes appear in an unordered list.

Tour of Heroes

**My favorite hero is:
Windstorm**

Heroes:

- Windstorm
- Bombasto
- Magneta
- Tornado

Creating a class for the data

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

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

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

Create a new file in the `app` folder called `hero.ts` with the following code:

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

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

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

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

Consider the first parameter:

src/app/hero.ts (id)

```
public id: number,
```

That brief syntax does a lot:

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

Using the Hero class

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

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

```
heroes = [
```

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

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

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

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

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

Conditional display with NgIf

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

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

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

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

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



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

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

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

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

Summary

Now you know how to use:

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

Here's the final code:

src/app/app.component.ts

src/app/hero.ts

src/app/app.module.ts

main.ts

```
1. import { Component } from '@angular/core';
2.
3. import { Hero } from './hero';
4.
5. @Component({
6.   selector: 'my-app',
7.   template: `
8.     <h1>{{title}}</h1>
9.     <h2>My favorite hero is: {{myHero.name}}</h2>
10.    <p>Heroes:</p>
11.    <ul>
12.      <li *ngFor="let hero of heroes">
13.        {{ hero.name }}
14.      </li>
15.    </ul>
16.    <p *ngIf="heroes.length > 3">There are many heroes!</p>
17.  `
18. })
19. export class AppComponent {
```

```
20.     title = 'Tour of Heroes';
21.     heroes = [
22.       new Hero(1, 'Windstorm'),
23.       new Hero(13, 'Bombasto'),
24.       new Hero(15, 'Magneta'),
25.       new Hero(20, 'Tornado')
26.     ];
27.     myHero = this.heroes[0];
28. }
```


This is the archived documentation for Angular v4. Please visit [angular.io](#) to see documentation for the current version of Angular.

Template Syntax

[Contents >](#)

[HTML in templates](#)

[Interpolation \({{...}} \)](#)

[Template expressions](#)

•••

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

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

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

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

HTML in templates

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

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

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

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

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

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

src/app/app.component.html

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



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

src/app/app.component.html

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



```
  
</h3>
```

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

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

src/app/app.component.html

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

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

src/app/app.component.html

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

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

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

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

Template expressions

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

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

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

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

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

Other notable differences from JavaScript syntax include:

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

Expression context

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

src/app/app.component.html

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



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

src/app/app.component.html

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



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

The previous example presents such a name collision. The component has a `hero` property and the `*ngFor` defines a `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.

Expression guidelines

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

- [No visible side effects](#)

- [Quick execution](#)
- [Simplicity](#)
- [Idempotence](#)

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

No visible side effects

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

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

Quick execution

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

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

Simplicity

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

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

Idempotence

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

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

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

Template statements

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

src/app/app.component.html

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



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

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

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

However, certain JavaScript syntax is not allowed:

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

Statement context

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

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

src/app/app.component.html

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

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

src/app/app.component.html

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

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

Template statements cannot refer to anything in the global namespace. They can't refer to `window` or `document`. They can't call `console.log` or `Math.max`.

Statement guidelines

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

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

Binding syntax: An overview

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

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

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

Data direction	Syntax	Type

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.

src/app/app.component.html

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

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



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

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

src/app/app.component.html

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



That's HTML Plus.

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

src/app/app.component.html

```
<!-- Bind button disabled state to `isUnchanged` property -->
```



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

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

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

HTML attribute vs. DOM property

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

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

- A few HTML attributes have 1:1 mapping to properties. `id` is one example.
- Some HTML attributes don't have corresponding properties. `colspan` is one example.
- Some DOM properties don't have corresponding attributes. `textContent` is one example.
- Many HTML attributes appear to map to properties ... but not in the way you might think!

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

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

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

When the user enters "Sally" into the input box, the DOM element `value` *property* becomes "Sally". But the HTML `value` *attribute* remains unchanged as you discover if you ask the input element

about that attribute: `input.getAttribute('value')` returns "Bob".

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

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

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

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

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

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

A WORLD WITHOUT ATTRIBUTES

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

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

Binding targets

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

Type	Target	Examples
Property	Element property Component property Directive property	<p>src/app/app.component.html</p> <pre> <hero-detail [hero]="currentHero"></hero- detail> <div [ngClass]="{{'special': isSpecial}}></div></pre>
Event	Element event Component event Directive event	<p>src/app/app.component.html</p> <pre><button (click)="onSave()">Save</button> <hero-detail (deleteRequest)="deleteHero()"> </hero-detail> <div (myClick)="clicked=\$event" clickable>click me</div></pre>
Two-way	Event and property	<p>src/app/app.component.html</p>

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

Attribute
(the exception)

src/app/app.component.html

```
<button [attr.aria-label]="help">help</button>
```

Class
class property

src/app/app.component.html

```
<div [class.special]="isSpecial">Special</div>
```

Style
style property

src/app/app.component.html

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

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

Property binding ([property])

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

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

src/app/app.component.html

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

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

src/app/app.component.html

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

Another is setting a property of a directive:

src/app/app.component.html

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

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

src/app/app.component.html

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

One-way *in*

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

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

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

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

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

Binding target

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

src/app/app.component.html

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

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

src/app/app.component.html

```

```

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

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

src/app/app.component.html

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

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

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

Avoid side effects

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

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

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

Return the proper type

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

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

src/app/app.component.html

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

Remember the brackets

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

Don't make the following mistake:

src/app/app.component.html

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

One-time string initialization

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

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

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

src/app/app.component.html

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

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

Property binding or interpolation?

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

src/app/app.component.html

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

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

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

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

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

Content security

Imagine the following *malicious content*.

src/app/app.component.ts

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

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

src/app/app.component.html

```
<!--
```

Angular generates warnings for these two lines as it sanitizes them

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

```
-->
```

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

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

Interpolation handles the script tags differently than property binding but both approaches render the content harmlessly.

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

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

Attribute, class, and style bindings

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

Attribute binding

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

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

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

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

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

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

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

And you get this error:

```
Template parse errors:  
Can't bind to 'colspan' since it isn't a known native property
```

As the message says, the `<td>` element does not have a `colspan` property. It has the "colspan" *attribute*, but interpolation and property binding can set only *properties*, not attributes.

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

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

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

src/app/app.component.html

```
<table border=1>  
  <!-- expression calculates colspan=2 -->  
  <tr><td [attr.colspan]="1 + 1">One-Two</td></tr>  
  
<!-- ERROR: There is no 'colspan' property to set!
```

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

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

Here's how the table renders:

One-Two
Five Six

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

src/app/app.component.html

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



Class binding

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

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

The following examples show how to add and remove the application's "special" class with class bindings.

Here's how to set the attribute without binding:

src/app/app.component.html

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



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

src/app/app.component.html

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



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

src/app/app.component.html

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

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



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

Style binding

You can set inline styles with a style binding.

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

src/app/app.component.html

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

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

src/app/app.component.html

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

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

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

Event binding ((event))

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

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

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

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

src/app/app.component.html

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



Target event

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

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

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



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

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

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



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

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

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



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

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

\$event and event handling statements

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

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

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

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

Consider this example:

src/app/app.component.html

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

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

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

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

Custom events with `EventEmitter`

Directives typically raise custom events with an Angular [EventEmitter](#). The directive creates an `EventEmitter` and exposes it as a property. The directive calls `EventEmitter.emit(payload)` to fire an

event, passing in a message payload, which can be anything. Parent directives listen for the event by binding to this property and accessing the payload through the `$event` object.

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

Here are the pertinent excerpts from that `HeroDetailComponent`:

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

```
template: `

<div>

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

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

```
// This component makes a request but it can't actually delete a hero.

deleteRequest = new EventEmiter<Hero>();

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

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

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

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

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

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

Template statements have side effects

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

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

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

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

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

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

`[(x)] = BANANA IN A BOX`

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

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

src/app/sizer.component.ts

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

```
17. inc() { this.resize(+1); }
18.
19. resize(delta: number) {
20.   this.size = Math.min(40, Math.max(8, +this.size + delta));
21.   this.sizeChange.emit(this.size);
22. }
23. }
```

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

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

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

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

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

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

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

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

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

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

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

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

Built-in directives

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

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

src/app/app.component.html

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

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

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

Built-in *attribute* directives

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

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

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

NgClass

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

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

src/app/app.component.html

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

To add or remove *many* CSS classes at the same time, the `NgClass` directive may be the better choice.

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

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

src/app/app.component.ts

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

Adding an `ngClass` property binding to `currentClasses` sets the element's classes accordingly:

src/app/app.component.html

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

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

NgStyle

You can set inline styles dynamically, based on the state of the component. With `NgStyle` you can set many inline styles simultaneously.

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

src/app/app.component.html

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

To set *many* inline styles at the same time, the `NgStyle` directive may be the better choice.

Try binding `ngStyle` to a key:value control object. Each key of the object is a style name; its value is whatever is appropriate for that style.

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

src/app/app.component.ts

```
currentStyles: {};  
setCurrentStyles() {  
  // CSS styles: set per current state of component properties  
  this.currentStyles = {  
    'font-style': this.canSave ? 'italic' : 'normal',  
    'font-weight': !this.isUnchanged ? 'bold' : 'normal',  
    'font-size': this.isSpecial ? '24px' : '12px'  
  };  
}
```

Adding an `ngStyle` property binding to `currentStyles` sets the element's styles accordingly:

src/app/app.component.html

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

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

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

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

Two-way data binding with the `NgModel` directive makes that easy. Here's an example:

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

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

FormsModule is required to use `ngModel`

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

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

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

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

/* Other imports */

@NgModule({
  imports: [
    BrowserModule,
    FormsModule // <-- import into the NgModule
  ],
  /* Other module metadata */
})
export class AppModule { }
```



Inside [(ngModel)]

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

src/app/app.component.html

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



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

That `ngModel` directive hides these onerous details behind its own `ngModel` input and `ngModelChange` output properties.

src/app/app.component.html

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



The `ngModel` data property sets the element's value property and the `ngModelChange` event property listens for changes to the element's value.

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

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

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

Separate `ngModel` bindings is an improvement over binding to the element's native properties. You can do better.

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

src/app/app.component.html

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



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

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

The following contrived example forces the input value to uppercase:

src/app/app.component.html

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



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

NgModel Binding

Result: Hercules

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

Built-in *structural*/directives

Structural directives are responsible for HTML layout. They shape or reshape the DOM's *structure*, typically by adding, removing, and manipulating the host elements to which they are attached.

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

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

This section is an introduction to the common structural directives:

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

NgIf

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

src/app/app.component.html

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



Don't forget the asterisk (`*`) in front of `ngIf` .

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

Show/hide is not the same thing

You can control the visibility of an element with a `class` or `style` binding:

src/app/app.component.html

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

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

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

Hiding an element is quite different from removing an element with `NgIf`.

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

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

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

Guard against null

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

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

src/app/app.component.html

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

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

NgFor

`NgFor` is a *repeater* directive – a way to present a list of items. You define a block of HTML that defines how a single item should be displayed. You tell Angular to use that block as a template for rendering each item in the list.

Here is an example of `NgFor` applied to a simple `<div>`:

src/app/app.component.html

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

You can also apply an `NgFor` to a component element, as in this example:

src/app/app.component.html

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



Don't forget the asterisk (`*`) in front of `ngFor` .

The text assigned to `*ngFor` is the instruction that guides the repeater process.

`*ngFor` microsyntax

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

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

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

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

Template input variables

The `let` keyword before `hero` creates a *template input variable* called `hero`. The `ngFor` 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 `ngFor` host element (and within its descendants) to access the hero's properties. Here it is referenced first in an interpolation and then passed in a binding to the `hero` property of the `<hero-detail>` component.

src/app/app.component.html

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



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

*ngFor with *index*

The `index` property of the `NgFor` directive context returns the zero-based index of the item in each iteration. You can capture the `index` in a template input variable and use it in the template.

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

src/app/app.component.html

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



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

*ngFor with *trackBy*

The `NgFor` 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 `NgFor should track`. In this case, that value is the hero's `id`.

src/app/app.component.ts

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

In the microsyntax expression, set `trackBy` to this method.

src/app/app.component.html

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

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

Reset heroes 

Change ids

Clear counts

without trackBy

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

with trackBy

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

The *NgSwitch* directives

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

`NgSwitch` is actually a set of three, cooperating directives: `NgSwitch`, `NgSwitchCase`, and `NgSwitchDefault` as seen in this example.

src/app/app.component.html

```
<div [ngSwitch]="currentHero.emotion">
  <happy-hero    *ngSwitchCase="'happy'"    [hero]="currentHero"></happy-hero>
  <sad-hero      *ngSwitchCase="'sad'"      [hero]="currentHero"></sad-hero>
  <confused-hero *ngSwitchCase="'confused'" [hero]="currentHero"></confused-hero>
  <unknown-hero  *ngSwitchDefault        [hero]="currentHero"></unknown-hero>
</div>
```



Pick your favorite hero

Hercules Mr. Nice Narco Windstorm Magneta

Wow. You've selected Hercules. What a happy hero ... just like you.

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

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

Bind to `*ngSwitchCase` and `*ngSwitchDefault`. The `NgSwitchCase` and `NgSwitchDefault` directives are *structural*/directives because they add or remove elements from the DOM.

- `NgSwitchCase` adds its element to the DOM when its bound value equals the switch value.
- `NgSwitchDefault` adds its element to the DOM when there is no selected `NgSwitchCase`.

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

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

src/app/app.component.html

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

Template reference variables (#var)

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

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

src/app/app.component.html

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

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

src/app/app.component.html

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

<!-- lots of other elements -->

<!-- phone refers to the input element; pass its `value` to an event handler -->
<button (click)="callPhone(phone.value)">Call</button>
```

How a reference variable gets its value

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

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

src/app/hero-form.component.html

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

```
  {{submitMessage}}  
  </div>
```

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

If Angular hadn't taken it over when you imported the `FormsModule`, it would be the [HTMLFormElement](#).

The `heroForm` is actually a reference to an Angular [NgForm](#) directive with the ability to track the value and validity of every control in the form.

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

Template reference variable warning notes

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

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

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

src/app/app.component.html

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

Input and output properties (@Input and @Output)

So far, you've focused mainly on binding to component members within template expressions and statements that appear on the *right side of the binding declaration*. A member in that position is a data binding source.

This section concentrates on binding to targets, which are directive properties on the *left side of the binding declaration*. These directive properties must be declared as inputs or outputs.

Remember: All components are directives.

Note the important distinction between a data binding target and a data binding source.

The *target* of a binding is to the *left* of the `=`. The *source* is on the *right* of the `=`.

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

The *source* is either inside quotes (`" "`) or within an interpolation (`{{}}`).

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

You have *limited* access to members of a target directive. You can only bind to properties that are explicitly identified as *inputs* and *outputs*.

In the following snippet, `iconUrl` and `onSave` are data-bound members of the `AppComponent` and are referenced within quoted syntax to the *right* of the equals (`=`).

`src/app/app.component.html`

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

They are *neither inputs nor outputs* of the component. They are sources for their bindings. The targets are the native `` and `<button>` elements.

Now look at another snippet in which the `HeroDetailComponent` is the target of a binding on the *left* of the equals (`=`).

src/app/app.component.html

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

Both `HeroDetailComponent.hero` and `HeroDetailComponent.deleteRequest` are on the left side of binding declarations. `HeroDetailComponent.hero` is inside brackets; it is the target of a property binding.

`HeroDetailComponent.deleteRequest` is inside parentheses; it is the target of an event binding.

Declaring input and output properties

Target properties must be explicitly marked as inputs or outputs.

In the `HeroDetailComponent`, such properties are marked as input or output properties using decorators.

src/app/hero-detail.component.ts

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

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

src/app/hero-detail.component.ts

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



You can specify an input/output property either with a decorator or in a metadata array. Don't do both!

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)">
```

Input

Output

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

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

Aliasing input/output properties

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

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

src/app/app.component.html

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



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

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

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

src/app/click.directive.ts

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



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

src/app/click.directive.ts

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

Template expression operators

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

The pipe operator (|)

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

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

src/app/app.component.html

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

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

You can chain expressions through multiple pipes:

src/app/app.component.html

```
<!-- Pipe chaining: convert title to uppercase, then to lowercase -->
<div>
  Title through a pipe chain:
  {{title | uppercase | lowercase}}
</div>
```



And you can also [apply parameters](#) to a pipe:

src/app/app.component.html

```
<!-- pipe with configuration argument => "February 25, 1970" -->
<div>Birthdate: {{currentHero?.birthdate | date:'longDate'}}</div>
```



The `json` pipe is particularly helpful for debugging bindings:

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

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



The generated output would look something like this

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



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

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

src/app/app.component.html

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



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

src/app/app.component.html

```
The title is {{title}}
```



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

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

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



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



```
TypeError: Cannot read property 'name' of null in [null].
```

Worse, the *entire view disappears*.

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

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

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

Unfortunately, the app crashes when the `currentHero` is null.

You could code around that problem with `*ngIf`.

src/app/app.component.html

```
<!--No hero, div not displayed, no error -->  
<div *ngIf="nullHero">The null hero's name is {{nullHero.name}}</div>
```

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

src/app/app.component.html

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

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

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

src/app/app.component.html

```
<!-- No hero, no problem! -->  
The null hero's name is {{nullHero?.name}}
```

It works perfectly with long property paths such as `a?.b?.c?.d`.

The non-null assertion operator (`!`)

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

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

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

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

For example, after you use `*ngIf` to check that `hero` is defined, you can assert that `hero` properties are also defined.



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

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

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

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

[back to top](#)

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.

This is the archived documentation for Angular v4. Please visit [angular.io](#) to see documentation for the current version of Angular.

Lifecycle Hooks

[Contents >](#)

[Component lifecycle hooks overview](#)

[Lifecycle sequence](#)

[Interfaces are optional \(technically\)](#)

...

`constructor`

A component has a lifecycle managed by Angular.

`ngOnChanges`

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.

`ngOnInit`

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

`ngDoCheck`

A directive has the same set of lifecycle hooks, minus the hooks that are specific to component content and views.

`ngAfterContentInit`

`ngAfterContentChecked`

`ngAfterViewInit`

`ngAfterViewChecked`

`ngOnDestroy`

Component lifecycle hooks overview

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

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

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

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

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

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

No directive or component will implement all of the lifecycle hooks and some of the hooks only make sense for components. Angular only calls a directive/component hook method *if it is defined*.

Lifecycle sequence

After creating a component/directive by calling its constructor, Angular calls the lifecycle hook methods in the following sequence at specific moments:

Hook	Purpose and Timing
<code>ngOnChanges()</code>	<p>Respond when Angular (re)sets data-bound input properties. The method receives a <code>SimpleChanges</code> object of current and previous property values.</p> <p>Called before <code>ngOnInit()</code> and whenever one or more data-bound input properties change.</p>
<code>ngOnInit()</code>	<p>Initialize the directive/component after Angular first displays the data-bound properties and sets the directive/component's input properties.</p> <p>Called <i>once</i>, after the <i>first</i> <code>ngOnChanges()</code>.</p>
<code>ngDoCheck()</code>	<p>Detect and act upon changes that Angular can't or won't detect on its own.</p> <p>Called during every change detection run, immediately after <code>ngOnChanges()</code> and <code>ngOnInit()</code>.</p>
<code>ngAfterContentInit()</code>	<p>Respond after Angular projects external content into the component's view.</p> <p>Called <i>once</i> after the <i>first</i> <code>ngDoCheck()</code>.</p> <p><i>A component-only hook.</i></p>
<code>ngAfterContentChecked()</code>	<p>Respond after Angular checks the content projected into the component.</p> <p>Called after the <code>ngAfterContentInit()</code> and every subsequent <code>ngDoCheck()</code>.</p> <p><i>A component-only hook.</i></p>

<code>ngAfterViewInit()</code>	Respond after Angular initializes the component's views and child views. Called <i>once</i> after the first <code>ngAfterContentChecked()</code> . <i>A component-only hook.</i>
<code>ngAfterViewChecked()</code>	Respond after Angular checks the component's views and child views. Called after the <code>ngAfterViewInit</code> and every subsequent <code>ngAfterContentChecked()</code> . <i>A component-only hook.</i>
<code>ngOnDestroy</code>	Cleanup just before Angular destroys the directive/component. Unsubscribe Observables and detach event handlers to avoid memory leaks. Called <i>just before</i> Angular destroys the directive/component.

Interfaces are optional (technically)

The interfaces are optional for JavaScript and Typescript developers from a purely technical perspective. The JavaScript language doesn't have interfaces. Angular can't see TypeScript interfaces at runtime because they disappear from the transpiled JavaScript.

Fortunately, they aren't necessary. You don't have to add the lifecycle hook interfaces to directives and components to benefit from the hooks themselves.

Angular instead inspects directive and component classes and calls the hook methods *if they are defined*. Angular finds and calls methods like `ngOnInit()`, with or without the interfaces.

Nonetheless, it's good practice to add interfaces to TypeScript directive classes in order to benefit from strong typing and editor tooling.

Other Angular lifecycle hooks

Other Angular sub-systems may have their own lifecycle hooks apart from these component hooks.

3rd party libraries might implement their hooks as well in order to give developers more control over how these libraries are used.

Lifecycle examples

The [live example](#) / [download example](#) demonstrates the lifecycle hooks in action through a series of exercises presented as components under the control of the root `AppComponent`.

They follow a common pattern: a *parent* component serves as a test rig for a *child* component that illustrates one or more of the lifecycle hook methods.

Here's a brief description of each exercise:

Component	Description
Peek-a-boo	Demonstrates every lifecycle hook. Each hook method writes to the on-screen log.
Spy	Directives have lifecycle hooks too. A <code>SpyDirective</code> can log when the element it spies upon is created or destroyed using the <code>ngOnInit</code> and <code>ngOnDestroy</code> hooks. This example applies the <code>SpyDirective</code> to a <code><div></code> in an <code>ngFor</code> <i>hero</i> repeater managed by the parent <code>SpyComponent</code> .
OnChanges	See how Angular calls the <code>ngOnChanges()</code> hook with a <code>changes</code> object every

time one of the component input properties changes. Shows how to interpret the `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.

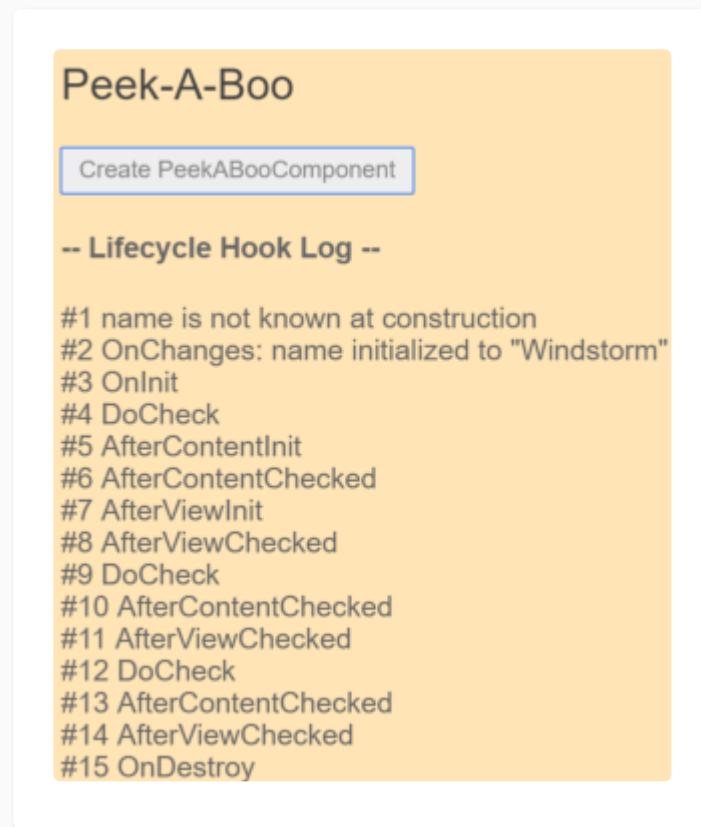
The remainder of this page discusses selected exercises in further detail.

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.



The sequence of log messages follows the prescribed hook calling order: `OnChanges` , `OnInit` ,
`DoCheck` (3x), `AfterContentInit` , `AfterContentChecked` (3x), `AfterViewInit` , `AfterViewChecked` (3x),
and `OnDestroy` .

The constructor isn't an Angular hook *per se*. The log confirms that input properties (the `name` property in this case) have no assigned values at construction.

Had the user clicked the *Update Hero* button, the log would show another `OnChanges` and two more triplets of `DoCheck`, `AfterContentChecked` and `AfterViewChecked`. Clearly these three hooks fire *often*. Keep the logic in these hooks as lean as possible!

The next examples focus on hook details.

Spying `OnInit` and `OnDestroy`

Go undercover with these two spy hooks to discover when an element is initialized or destroyed.

This is the perfect infiltration job for a directive. The heroes will never know they're being watched.

Kidding aside, pay attention to two key points:

1. Angular calls hook methods for *directives* as well as components.
2. A spy directive can provide insight into a DOM object that you cannot change directly.

Obviously you can't touch the implementation of a native `<div>`. You can't modify a third party component either. But you can watch both with a directive.

The sneaky spy directive is simple, consisting almost entirely of `ngOnInit()` and `ngOnDestroy()` hooks that log messages to the parent via an injected `LoggerService`.

src/app/spy.directive.ts

```
// Spy on any element to which it is applied.  
// Usage: <div mySpy>...</div>  
@Directive({selector: '[mySpy]'}  
export class SpyDirective implements OnInit, OnDestroy {
```



```
constructor(private logger: LoggerService) { }

ngOnInit()    { this.logIt(`onInit`); }

ngOnDestroy() { this.logIt(`onDestroy`); }

private logIt(msg: string) {
  this.logger.log(`Spy #${nextId++} ${msg}`);
}

}
```

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

src/app/spy.component.html

```
<div *ngFor="let hero of heroes" mySpy class="heroes">
  {{hero}}
</div>
```



Each spy's birth and death marks the birth and death of the attached hero `<div>` with an entry in the *Hook Log* as seen here:

Spy Directive

Herbie Add Hero Reset Heroes

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.

OnInit()

Use `ngOnInit()` for two main reasons:

1. To perform complex initializations shortly after construction.

2. To set up the component after Angular sets the input properties.

Experienced developers agree that components should be cheap and safe to construct.

Misko Hevery, Angular team lead, [explains why](#) you should avoid complex constructor logic.

Don't fetch data in a component constructor. You shouldn't worry that a new component will try to contact a remote server when created under test or before you decide to display it. Constructors should do no more than set the initial local variables to simple values.

An `ngOnInit()` is a good place for a component to fetch its initial data. The [Tour of Heroes Tutorial](#) guide shows how.

Remember also that a directive's data-bound input properties are not set until *after construction*. That's a problem if you need to initialize the directive based on those properties. They'll have been set when `ngOnInit()` runs.

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

You can count on Angular to call the `ngOnInit()` method *soon* after creating the component. That's where the heavy initialization logic belongs.

OnDestroy()

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

This is the time to notify another part of the application that the component is going away.

This is the place to free resources that won't be garbage collected automatically. Unsubscribe from Observables and DOM events. Stop interval timers. Unregister all callbacks that this directive registered with global or application services. You risk memory leaks if you neglect to do so.

OnChanges()

Angular calls its `ngOnChanges()` method whenever it detects changes to *input properties* of the component (or directive). This example monitors the `OnChanges` hook.

on-changes.component.ts (excerpt)

```
ngOnChanges(changes: SimpleChanges) {  
  for (let propName in changes) {  
    let chng = changes[propName];  
    let cur = JSON.stringify(chng.currentValue);  
    let prev = JSON.stringify(chng.previousValue);  
    this.changeLog.push(` ${propName}: currentValue = ${cur}, previousValue =  
    ${prev}`);  
  }  
}
```

The `ngOnChanges()` method takes an object that maps each changed property name to a `SimpleChange` object holding the current and previous property values. This hook iterates over the changed properties and logs them.

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

src/app/on-changes.component.ts

```
@Input() hero: Hero;
```

```
@Input() power: string;
```

The host `OnChangesParentComponent` binds to them like this:

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

```
<on-changes [hero]="hero" [power]="power"></on-changes>
```



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

OnChanges

Power:

sing

Hero.name: Windstorm

Reset Log

Windstorm can sing

-- Change Log --

```
hero: currentValue = {"name": "Windstorm"}, previousValue = {}  
power: currentValue = "sing", previousValue = {}
```

[back to top](#)

The log entries appear as the string value of the `power` property changes. But the `ngOnChanges` does not catch changes to `hero.name`. That's surprising at first.

Angular only calls the hook when the value of the input property changes. The value of the `hero` property is the *reference to the hero object*. Angular doesn't care that the hero's own `name` property changed. The hero object *reference* didn't change so, from Angular's perspective, there is no change to report!

DoCheck()

Use the `DoCheck` hook to detect and act upon changes that Angular doesn't catch on its own.

Use this method to detect a change that Angular overlooked.

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

DoCheckComponent (ngDoCheck)

```
ngDoCheck() {  
  
  if (this.hero.name !== this.oldHeroName) {  
    this.changeDetected = true;  
    this.changeLog.push(`DoCheck: Hero name changed to "${this.hero.name}" from  
"${this.oldHeroName}"`);  
    this.oldHeroName = this.hero.name;  
  }  
  
  if (this.power !== this.oldPower) {  
    this.changeDetected = true;  
    this.changeLog.push(`DoCheck: Power changed to "${this.power}" from  
"${this.oldPower}"`);  
    this.oldPower = this.power;  
  }  
  
  if (this.changeDetected) {  
    this.noChangeCount = 0;  
  }  
}
```

```
    } else {
        // log that hook was called when there was no relevant change.
        let count = this.noChangeCount += 1;
        let noChangeMsg = `DoCheck called ${count}x when no change to hero or power`;
        if (count === 1) {
            // add new "no change" message
            this.changeLog.push(noChangeMsg);
        } else {
            // update last "no change" message
            this.changeLog[this.changeLog.length - 1] = noChangeMsg;
        }
    }

    this.changeDetected = false;
}
```

This code inspects certain *values of interest*, capturing and comparing their current state against previous values. It writes a special message to the log when there are no substantive changes to the `hero` or the `power` so you can see how often `DoCheck` is called. The results are illuminating:

DoCheck

Power:

Hero.name:

[Reset Log](#)

Windstorm can sing

-- Change Log --

OnChanges: hero: currentValue = {"name": "Windstorm"}, previousValue = {}

OnChanges: power: currentValue = "sing", previousValue = {}

DoCheck: Hero name changed to "Windstorm" from ""

DoCheck: Power changed to "sing" from ""

DoCheck called 26x when no change to hero or power

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

Most of these initial checks are triggered by Angular's first rendering of *unrelated data elsewhere on the page*. Mere mousing into another `<input>` triggers a call. Relatively few calls reveal actual changes to pertinent data. Clearly our implementation must be very lightweight or the user experience suffers.

AfterView

The `AfterView` sample explores the `AfterViewInit()` and `AfterViewChecked()` hooks that Angular calls *after* it creates a component's child views.

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

ChildComponent

```
@Component({
  selector: 'my-child-view',
  template: '<input [(ngModel)]="hero">'
})
export class ChildViewComponent {
  hero = 'Magneta';
}
```

The `AfterViewComponent` displays this child view *within its template*:

AfterViewComponent (template)

```
template: `
<div>-- child view begins --</div>
<my-child-view></my-child-view>
<div>-- child view ends --</div>`
```

The following hooks take action based on changing values *within the child view*, which can only be reached by querying for the child view via the property decorated with `@ViewChild`.

AfterViewComponent (class excerpts)

```
export class AfterViewComponent implements AfterViewChecked, AfterViewInit {  
  private prevHero = '';  
  
  // Query for a VIEW child of type 'ChildViewComponent'  
  @ViewChild(ChildViewComponent) viewChild: ChildViewComponent;  
  
  ngAfterViewInit() {  
    // viewChild is set after the view has been initialized  
    this.logIt('AfterViewInit');  
    this.doSomething();  
  }  
  
  ngAfterViewChecked() {  
    // viewChild is updated after the view has been checked  
    if (this.prevHero === this.viewChild.hero) {  
      this.logIt('AfterViewChecked (no change)');  
    } else {  
      this.prevHero = this.viewChild.hero;  
      this.logIt('AfterViewChecked');  
      this.doSomething();  
    }  
  }  
  // ...  
}
```

Abide by the unidirectional data flow rule

The `doSomething()` method updates the screen when the hero name exceeds 10 characters.

AfterViewComponent (doSomething)

```
// This surrogate for real business logic sets the `comment`  
private doSomething() {  
  let c = this.viewChild.hero.length > 10 ? `That's a long name` : '';  
  if (c !== this.comment) {  
    // Wait a tick because the component's view has already been checked  
    this.logger.tick_then(() => this.comment = c);  
  }  
}
```



Why does the `doSomething()` method wait a tick before updating `comment`?

Angular's unidirectional data flow rule forbids updates to the view *after* it has been composed. Both of these hooks fire *after* the component's view has been composed.

Angular throws an error if the hook updates the component's data-bound `comment` property immediately (try it!). The `LoggerService.tick_then()` postpones the log update for one turn of the browser's JavaScript cycle and that's just long enough.

Here's *AfterView* in action:

AfterView

-- child view begins --

Magneta

-- child view ends --

-- AfterView Logs --

Reset

AfterView constructor: no child view

AfterViewInit: Magneta child view

AfterViewChecked: Magneta child view

[back to top](#)

Notice that Angular frequently calls `AfterViewChecked()`, often when there are no changes of interest.

Write lean hook methods to avoid performance problems.

AfterContent

The `AfterContent` sample explores the `AfterContentInit()` and `AfterContentChecked()` hooks that Angular calls *after* Angular projects external content into the component.

Content projection

Content projection is a way to import HTML content from outside the component and insert that content into the component's template in a designated spot.

AngularJS developers know this technique as *transclusion*.

Consider this variation on the previous [AfterView](#) example. This time, instead of including the child view within the template, it imports the content from the `AfterContentComponent`'s parent. Here's the parent's template:

AfterContentParentComponent (template excerpt)

```
`<after-content>
  <my-child></my-child>
</after-content>`
```

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:

AfterContentComponent (template)

```
template: `
<div>-- projected content begins --</div>
```

```
<ng-content></ng-content>  
<div>-- projected content ends --</div>
```

The `<ng-content>` tag is a *placeholder* for the external content. It tells Angular where to insert that content. In this case, the projected content is the `<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 `<ng-content>` tags in the component's template.

AfterContent hooks

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

- The *AfterView* hooks concern `ViewChildren`, the child components whose element tags appear *within* the component's template.
- The *AfterContent* hooks concern `ContentChildren`, the child components that Angular projected into the component.

The following *AfterContent* hooks take action based on changing values in a *content child*, which can only be reached by querying for them via the property decorated with `@ContentChild`.

AfterContentComponent (class excerpts)

```
export class AfterContentComponent implements AfterContentChecked, AfterContentInit {  
  private prevHero = '';  
  comment = '';  
  
  // Query for a CONTENT child of type `ChildComponent`  
  @ContentChild(ChildComponent) contentChild: ChildComponent;  
  
  ngAfterContentInit() {  
    // contentChild is set after the content has been initialized  
    this.logIt('AfterContentInit');  
    this.doSomething();  
  }  
  
  ngAfterContentChecked() {  
    // contentChild is updated after the content has been checked  
    if (this.prevHero === this.contentChild.hero) {  
      this.logIt('AfterContentChecked (no change)');  
    } else {  
      this.prevHero = this.contentChild.hero;  
      this.logIt('AfterContentChecked');  
      this.doSomething();  
    }  
  }  
  // ...  
}
```

No unidirectional flow worries with *AfterContent*

This component's `doSomething()` method update's the component's data-bound `comment` property immediately. There's no [need to wait](#).

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

This is the archived documentation for Angular v4. Please visit [angular.io](#) to see documentation for the current version of Angular.

Component Interaction

[Contents >](#)

[Pass data from parent to child with input binding](#)

[Intercept input property changes with a setter](#)

[Intercept input property changes with `ngOnChanges\(\)`](#)

•••

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

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

Pass data from parent to child with input binding

`HeroChildComponent` has two *input properties*, typically adorned with [@Input](#) decorations.

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

```
1. import { Component, Input } from '@angular/core';
2.
3. import { Hero } from './hero';
4.
```



```
5. @Component({
6.   selector: 'hero-child',
7.   template: `
8.     <h3>{{hero.name}} says:</h3>
9.     <p>I, {{hero.name}}, am at your service, {{masterName}}.</p>
10.    `
11.  })
12. export class HeroChildComponent {
13.   @Input() hero: Hero;
14.   @Input('master') masterName: string;
15. }
```

The second `@Input` aliases the child component property name `masterName` as `'master'`.

The `HeroParentComponent` nests the child `HeroChildComponent` inside an `*ngFor` repeater, binding its `master` string property to the child's `master` alias, and each iteration's `hero` instance to the child's `hero` property.

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

```
1. import { Component } from '@angular/core';
2.
3. import { HEROES } from './hero';
4.
5. @Component({
6.   selector: 'hero-parent',
7.   template: `
8.     <h2>{{master}} controls {{heroes.length}} heroes</h2>
9.     <hero-child *ngFor="let hero of heroes"
10.       [hero]="hero"
11.       [master]="master">
```

```
12.      </hero-child>
13.      .
14.  })
15. export class HeroParentComponent {
16.   heroes = HEROES;
17.   master = 'Master';
18. }
```

The running application displays three heroes:

Master controls 3 heroes

Mr. IQ says:

I, Mr. IQ, am at your service, Master.

Magneta says:

I, Magneta, am at your service, Master.

Bombasto says:

I, Bombasto, am at your service, Master.

Test it

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

component-interaction/e2e-spec.ts

```
1. // ...
2. let _heroNames = ['Mr. IQ', 'Magneta', 'Bombasto'];
3. let _masterName = 'Master';
4.
5. it('should pass properties to children properly', function () {
6.   let parent = element.all(by.tagName('hero-parent')).get(0);
7.   let heroes = parent.all(by.tagName('hero-child'));
8.
9.   for (let i = 0; i < _heroNames.length; i++) {
10.     let childTitle = heroes.get(i).element(by.tagName('h3')).getText();
11.     let childDetail = heroes.get(i).element(by.tagName('p')).getText();
12.     expect(childTitle).toEqual(_heroNames[i] + ' says:');
13.     expect(childDetail).toContain(_masterName);
14.   }
15. });
16. // ...
```

[Back to top](#)

Intercept input property changes with a setter

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

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

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

```
1. import { Component, Input } from '@angular/core';
2.
```

```
3. @Component({
4.   selector: 'name-child',
5.   template: '<h3>{{name}}</h3>'
6. })
7. export class NameChildComponent {
8.   private _name = '';
9.
10.  @Input()
11.  set name(name: string) {
12.    this._name = (name && name.trim()) || '<no name set>';
13.  }
14.
15.  get name(): string { return this._name; }
16. }
```

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

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

```
1. import { Component } from '@angular/core';
2.
3. @Component({
4.   selector: 'name-parent',
5.   template: `
6.     <h2>Master controls {{names.length}} names</h2>
7.     <name-child *ngFor="let name of names" [name]="name"></name-child>
8.   `
9. })
10. export class NameParentComponent {
11.   // Displays 'Mr. IQ', '<no name set>', 'Bombasto'
```

```
12. names = ['Mr. IQ', ' ', ' Bombasto '];  
13. }
```

Master controls 3 names

"Mr. IQ"

"<no name set>"

"Bombasto"

Test it

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

component-interaction/e2e-spec.ts

```
1. // ...  
2. it('should display trimmed, non-empty names', function () {  
3.   let _nonEmptyNameIndex = 0;  
4.   let _nonEmptyName = '"Mr. IQ"';  
5.   let parent = element.all(by.tagName('name-parent')).get(0);  
6.   let hero = parent.all(by.tagName('name-child')).get(_nonEmptyNameIndex);  
7.  
8.   let displayName = hero.element(by.tagName('h3')).getText();  
9.   expect(displayName).toEqual(_nonEmptyName);
```

```
10. });
11.
12. it('should replace empty name with default name', function () {
13.   let _emptyNameIndex = 1;
14.   let _defaultName = '<no name set>';
15.   let parent = element.all(by.tagName('name-parent')).get(0);
16.   let hero = parent.all(by.tagName('name-child')).get(_emptyNameIndex);
17.
18.   let displayName = hero.element(by.tagName('h3')).getText();
19.   expect(displayName).toEqual(_defaultName);
20. });
21. // ...
```

[Back to top](#)

Intercept input property changes with `ngOnChanges()`

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

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

Learn about `ngOnChanges()` in the [LifeCycle Hooks](#) chapter.

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

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

```
1. import { Component, Input, OnChanges, SimpleChange } from '@angular/core';
2.
3. @Component({
4.   selector: 'version-child',
5.   template: `
6.     <h3>Version {{major}}.{{minor}}</h3>
7.     <h4>Change log:</h4>
8.     <ul>
9.       <li *ngFor="let change of changeLog">{{change}}</li>
10.    </ul>
11.  `
12. })
13. export class VersionChildComponent implements OnChanges {
14.   @Input() major: number;
15.   @Input() minor: number;
16.   changeLog: string[] = [];
17.
18.   ngOnChanges(changes: {[propKey: string]: SimpleChange}) {
19.     let log: string[] = [];
20.     for (let propName in changes) {
21.       let changedProp = changes[propName];
22.       let to = JSON.stringify(changedProp.currentValue);
23.       if (changedProp.isFirstChange()) {
24.         log.push(`Initial value of ${propName} set to ${to}`);
25.       } else {
26.         let from = JSON.stringify(changedProp.previousValue);
27.         log.push(`${propName} changed from ${from} to ${to}`);
28.       }
29.     }
30.     console.log(log.join('\n'));
31.   }
32. }
```

```
29.      }
30.      this.changeLog.push(log.join(', '));
31.  }
32. }
```

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

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

```
1. import { Component } from '@angular/core';
2.
3. @Component({
4.   selector: 'version-parent',
5.   template: `
6.     <h2>Source code version</h2>
7.     <button (click)="newMinor()">New minor version</button>
8.     <button (click)="newMajor()">New major version</button>
9.     <version-child [major]="major" [minor]="minor"></version-child>
10.    `
11.  })
12. export class VersionParentComponent {
13.   major = 1;
14.   minor = 23;
15.
16.   newMinor() {
17.     this.minor++;
18.   }
19.
20.   newMajor() {
```

```
21.     this.major++;
22.     this.minor = 0;
23. }
24. }
```

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

Source code version

New minor version

New major version

Version 1.23

Change log:

- Initial value of major set to 1, Initial value of minor set to 23

Test it

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

component-interaction/e2e-spec.ts

```
1. // ...
2. // Test must all execute in this exact order
```



```
3. it('should set expected initial values', function () {
4.   let actual = getActual();
5.
6.   let initialLabel = 'Version 1.23';
7.   let initialLog = 'Initial value of major set to 1, Initial value of minor
     set to 23';
8.
9.   expect(actual.label).toBe(initialLabel);
10.  expect(actual.count).toBe(1);
11.  expect(actual.logs.get(0).getText()).toBe(initialLog);
12. });
13.
14. it('should set expected values after clicking \'Minor\' twice', function ()
  {
15.   let repoTag = element(by.tagName('version-parent'));
16.   let newMinorButton = repoTag.all(by.tagName('button')).get(0);
17.
18.   newMinorButton.click().then(function() {
19.     newMinorButton.click().then(function() {
20.       let actual = getActual();
21.
22.       let labelAfter2Minor = 'Version 1.25';
23.       let logAfter2Minor = 'minor changed from 24 to 25';
24.
25.       expect(actual.label).toBe(labelAfter2Minor);
26.       expect(actual.count).toBe(3);
27.       expect(actual.logs.get(2).getText()).toBe(logAfter2Minor);
28.     });
29.   });
30. });
```

```
31.  
32. it('should set expected values after clicking \'Major\' once', function () {  
33.   let repoTag = element(by.tagName('version-parent'));  
34.   let newMajorButton = repoTag.all(by.tagName('button')).get(1);  
35.  
36.   newMajorButton.click().then(function() {  
37.     let actual = getActual();  
38.  
39.     let labelAfterMajor = 'Version 2.0';  
40.     let logAfterMajor = 'major changed from 1 to 2, minor changed from 25 to  
0';  
41.  
42.     expect(actual.label).toBe(labelAfterMajor);  
43.     expect(actual.count).toBe(4);  
44.     expect(actual.logs.get(3).getText()).toBe(logAfterMajor);  
45.   });  
46. });  
47.  
48. function getActual() {  
49.   let versionTag = element(by.tagName('version-child'));  
50.   let label = versionTag.element(by.tagName('h3')).getText();  
51.   let ul = versionTag.element((by.tagName('ul')));  
52.   let logs = ul.all(by.tagName('li'));  
53.  
54.   return {  
55.     label: label,  
56.     logs: logs,  
57.     count: logs.count()  
58.   };  
59. }
```

```
60. // ...
```

[Back to top](#)

Parent listens for child event

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

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

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

```
1. import { Component, EventEmitter, Input, Output } from '@angular/core';
2.
3. @Component({
4.   selector: 'my-voter',
5.   template: `
6.     <h4>{{name}}</h4>
7.     <button (click)="vote(true)" [disabled]="voted">Agree</button>
8.     <button (click)="vote(false)" [disabled]="voted">Disagree</button>
9.   `,
10. })
11. export class VoterComponent {
12.   @Input() name: string;
13.   @Output() onVoted = new EventEmitter<boolean>();
14.   voted = false;
15. }
```

```
16.   vote(agreed: boolean) {  
17.     this.onVoted.emit(agreed);  
18.     this.voted = true;  
19.   }  
20. }
```

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

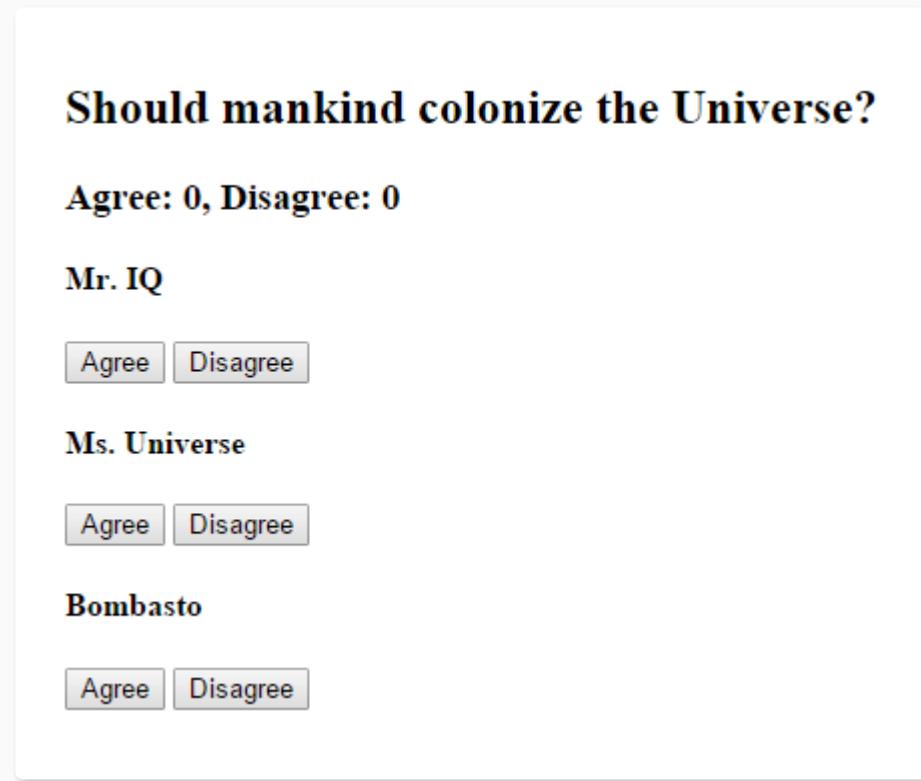
The parent `VoteTakerComponent` binds an event handler called `onVoted()` that responds to the child event payload `$event` and updates a counter.

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

```
1. import { Component }      from '@angular/core';  
2.  
3. @Component({  
4.   selector: 'vote-taker',  
5.   template: `  
6.     <h2>Should mankind colonize the Universe?</h2>  
7.     <h3>Agree: {{agreed}}, Disagree: {{disagreed}}</h3>  
8.     <my-voter *ngFor="let voter of voters"  
9.       [name]="voter"  
10.      (onVoted)="onVoted($event)">  
11.    </my-voter>  
12.  `  
13. })  
14. export class VoteTakerComponent {  
15.   agreed = 0;  
16.   disagreed = 0;  
17.   voters = ['Mr. IQ', 'Ms. Universe', 'Bombasto'];  
18. }
```

```
19.   onVoted(agreed: boolean) {  
20.     agreed ? this.agreed++ : this.disagreed++;  
21.   }  
22. }
```

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



Test it

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

[component-interaction/e2e-spec.ts](#)



```
1. // ...
2. it('should not emit the event initially', function () {
3.   let voteLabel = element(by.tagName('vote-taker'))
4.     .element(by.tagName('h3')).getText();
5.   expect(voteLabel).toBe('Agree: 0, Disagree: 0');
6. });
7.
8. it('should process Agree vote', function () {
9.   let agreeButton1 = element.all(by.tagName('my-voter')).get(0)
10.    .all(by.tagName('button')).get(0);
11.   agreeButton1.click().then(function() {
12.     let voteLabel = element(by.tagName('vote-taker'))
13.       .element(by.tagName('h3')).getText();
14.     expect(voteLabel).toBe('Agree: 1, Disagree: 0');
15.   });
16. });
17.
18. it('should process Disagree vote', function () {
19.   let agreeButton1 = element.all(by.tagName('my-voter')).get(1)
20.    .all(by.tagName('button')).get(1);
21.   agreeButton1.click().then(function() {
22.     let voteLabel = element(by.tagName('vote-taker'))
23.       .element(by.tagName('h3')).getText();
24.     expect(voteLabel).toBe('Agree: 1, Disagree: 1');
25.   });
26. });
27. // ...
```

Parent interacts with child via *local variable*

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

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

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

```
1. import { Component, OnDestroy, OnInit } from '@angular/core';
2.
3. @Component({
4.   selector: 'countdown-timer',
5.   template: '<p>{{message}}</p>'
6. })
7. export class CountdownTimerComponent implements OnInit, OnDestroy {
8.
9.   intervalId = 0;
10.  message = '';
11.  seconds = 11;
12.
13.  clearTimer() { clearInterval(this.intervalId); }
14.
15.  ngOnInit()    { this.start(); }
16.  ngOnDestroy() { this.clearTimer(); }
17.
18.  start() { this.countDown(); }
19.  stop() { }
```

```

20.     this.clearTimer();
21.     this.message = `Holding at T-${this.seconds} seconds`;
22.   }
23.
24.   private countDown() {
25.     this.clearTimer();
26.     this.intervalId = window.setInterval(() => {
27.       this.seconds -= 1;
28.       if (this.seconds === 0) {
29.         this.message = 'Blast off!';
30.       } else {
31.         if (this.seconds < 0) { this.seconds = 10; } // reset
32.         this.message = `T-${this.seconds} seconds and counting`;
33.       }
34.     }, 1000);
35.   }
36. }

```

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

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

```

1. import { Component }           from '@angular/core';
2. import { CountdownTimerComponent } from './countdown-timer.component';
3.
4. @Component({
5.   selector: 'countdown-parent-lv',
6.   template: `
7.     <h3>Countdown to Liftoff (via local variable)</h3>
8.     <button (click)="timer.start()">Start</button>

```

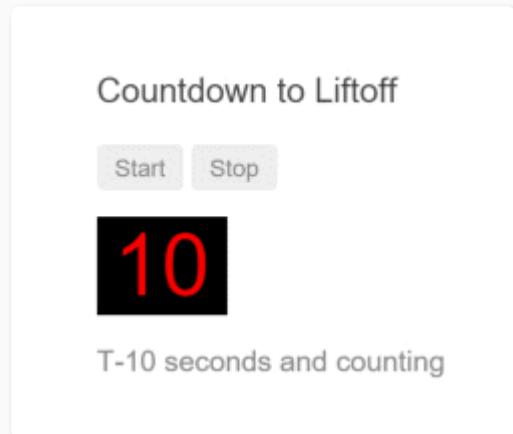
```
9.   <button (click)="timer.stop()">Stop</button>
10.  <div class="seconds">{{timer.seconds}}</div>
11.  <countdown-timer #timer></countdown-timer>
12.  `,
13.  styleUrls: ['demo.css']
14. })
15. export class CountdownLocalVarParentComponent { }
```

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

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

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

Here we see the parent and child working together.



Test it

Test that the seconds displayed in the parent template match the seconds displayed in the child's status message. Test also that clicking the *Stop* button pauses the countdown timer:

component-interaction/e2e-spec.ts

```
1. // ...
2. it('timer and parent seconds should match', function () {
3.   let parent = element(by.tagName(parentTag));
4.   let message = parent.element(by.tagName('countdown-timer')).getText();
5.   browser.sleep(10); // give `seconds` a chance to catchup with `message`
6.   let seconds = parent.element(by.className('seconds')).getText();
7.   expect(message).toContain(seconds);
8. });
9.
10. it('should stop the countdown', function () {
11.   let parent = element(by.tagName(parentTag));
12.   let stopButton = parent.all(by.tagName('button')).get(1);
13.
14.   stopButton.click().then(function() {
15.     let message = parent.element(by.tagName('countdown-timer')).getText();
16.     expect(message).toContain('Holding');
17.   });
18. });
19. // ...
```



[Back to top](#)

Parent calls an `@ViewChild()`

The *local variable* approach is simple and easy. But it is limited because the parent-child wiring must be done entirely within the parent template. The parent component *itself* has no access to the child.

You can't use the *local variable* technique if an instance of the parent component *class* must read or write child component values or must call child component methods.

When the parent component *class* requires that kind of access, *inject* the child component into the parent as a *ViewChild*.

The following example illustrates this technique with the same [Countdown Timer](#) example. Neither its appearance nor its behavior will change. The child [CountdownTimerComponent](#) is the same as well.

The switch from the *local variable* to the *ViewChild* technique is solely for the purpose of demonstration.

Here is the parent, `CountdownViewChildParentComponent` :

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

```
1. import { AfterViewInit, ViewChild } from '@angular/core';
2. import { Component }                  from '@angular/core';
3. import { CountdownTimerComponent }   from './countdown-timer.component';
4.
5. @Component({
6.   selector: 'countdown-parent-vc',
7.   template: `
8.     <h3>Countdown to Liftoff (via ViewChild)</h3>
9.     <button (click)="start()">Start</button>
10.    <button (click)="stop()">Stop</button>
11.    <div class="seconds">{{ seconds() }}</div>
```

```
12.  <countdown-timer></countdown-timer>
13.  `,
14.  styleUrls: ['demo.css']
15. })
16. export class CountdownViewChildParentComponent implements AfterViewInit {
17.
18. @ViewChild(CountdownTimerComponent)
19. private timerComponent: CountdownTimerComponent;
20.
21. seconds() { return 0; }
22.
23. ngAfterViewInit() {
24.   // Redefine `seconds()` to get from the
25.   // `CountdownTimerComponent.seconds` ...
26.   // but wait a tick first to avoid one-time devMode
27.   // unidirectional-data-flow-violation error
28.   setTimeout(() => this.seconds = () => this.timerComponent.seconds, 0);
29. }
30. start() { this.timerComponent.start(); }
31. stop() { this.timerComponent.stop(); }
32. }
```

It takes a bit more work to get the child view into the parent component *class*.

First, you have to import references to the `ViewChild` decorator and the `AfterViewInit` lifecycle hook.

Next, inject the child `CountdownTimerComponent` into the private `timerComponent` property via the `@ViewChild` property decoration.

The `#timer` local variable is gone from the component metadata. Instead, bind the buttons to the parent component's own `start` and `stop` methods and present the ticking seconds in an interpolation around the parent component's `seconds` method.

These methods access the injected timer component directly.

The `ngAfterViewInit()` lifecycle hook is an important wrinkle. The timer component isn't available until *after* Angular displays the parent view. So it displays `0` seconds initially.

Then Angular calls the `ngAfterViewInit` lifecycle hook at which time it is *too late* to update the parent view's display of the countdown seconds. Angular's unidirectional data flow rule prevents updating the parent view's in the same cycle. The app has to *wait one turn* before it can display the seconds.

Use `setTimeout()` to wait one tick and then revise the `seconds()` method so that it takes future values from the timer component.

Test it

Use [the same countdown timer tests](guide/component-interaction#countdown-tests) as before.

[Back to top](#)

Parent and children communicate via a service

A parent component and its children share a service whose interface enables bi-directional communication *within the family*.

The scope of the service instance is the parent component and its children. Components outside this component subtree have no access to the service or their communications.

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

`component-interaction/src/app/mission.service.ts`

```
1. import { Injectable } from '@angular/core';
2. import { Subject }    from 'rxjs/Subject';
3.
4. @Injectable()
5. export class MissionService {
6.
7.   // Observable string sources
8.   private missionAnnouncedSource = new Subject<string>();
9.   private missionConfirmedSource = new Subject<string>();
10.
11.  // Observable string streams
12.  missionAnnounced$ = this.missionAnnouncedSource.asObservable();
13.  missionConfirmed$ = this.missionConfirmedSource.asObservable();
14.
15.  // Service message commands
16.  announceMission(mission: string) {
17.    this.missionAnnouncedSource.next(mission);
18.  }
19.
20.  confirmMission(astronaut: string) {
21.    this.missionConfirmedSource.next(astronaut);
22.  }
23. }
```

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

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

```
1. import { Component }           from '@angular/core';
2.
3. import { MissionService }     from './mission.service';
4.
5. @Component({
6.   selector: 'mission-control',
7.   template: `
8.     <h2>Mission Control</h2>
9.     <button (click)="announce()">Announce mission</button>
10.    <my-astronaut *ngFor="let astronaut of astronauts"
11.      [astronaut]="astronaut">
12.    </my-astronaut>
13.    <h3>History</h3>
14.    <ul>
15.      <li *ngFor="let event of history">{{event}}</li>
16.    </ul>
17.  `,
18.   providers: [MissionService]
19. })
20. export class MissionControlComponent {
21.   astronauts = ['Lovell', 'Swigert', 'Haise'];
22.   history: string[] = [];
23.   missions = ['Fly to the moon!',
24.               'Fly to mars!',
25.               'Fly to Vegas!'];
26.   nextMission = 0;
27.
28.   constructor(private missionService: MissionService) {
29.     missionService.missionConfirmed$.subscribe(
30.       astronaut => {
```

```
31.         this.history.push(`${astronaut} confirmed the mission`);
32.     });
33. }
34.
35. announce() {
36.     let mission = this.missions[this.nextMission++];
37.     this.missionService.announceMission(mission);
38.     this.history.push(`Mission "${mission}" announced`);
39.     if (this.nextMission >= this.missions.length) { this.nextMission = 0; }
40. }
41. }
```

The `AstronautComponent` also injects the service in its constructor. Each `AstronautComponent` is a child of the `MissionControlComponent` and therefore receives its parent's service instance:

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

```
1. import { Component, Input, OnDestroy } from '@angular/core';
2.
3. import { MissionService } from './mission.service';
4. import { Subscription } from 'rxjs/Subscription';
5.
6. @Component({
7.   selector: 'my-astronaut',
8.   template: `
9.     <p>
10.       {{astronaut}}: <strong>{{mission}}</strong>
11.       <button
12.         (click)="confirm()"
13.         [disabled]="${!announced || confirmed}">
```

```
14.      Confirm
15.    </button>
16.  </p>
17.  `

18. })
19. export class AstronautComponent implements OnDestroy {
20.   @Input() astronaut: string;
21.   mission = '<no mission announced>';
22.   confirmed = false;
23.   announced = false;
24.   subscription: Subscription;
25.
26.   constructor(private missionService: MissionService) {
27.     this.subscription = missionService.missionAnnounced$.subscribe(
28.       mission => {
29.         this.mission = mission;
30.         this.announced = true;
31.         this.confirmed = false;
32.       });
33.   }
34.
35.   confirm() {
36.     this.confirmed = true;
37.     this.missionService.confirmMission(this.astronaut);
38.   }
39.
40.   ngOnDestroy() {
41.     // prevent memory leak when component destroyed
42.     this.subscription.unsubscribe();
43.   }

```

Notice that this example captures the `subscription` and `unsubscribe()` when the `AstronautComponent` is destroyed. This is a memory-leak guard step. There is no actual risk in this app because the lifetime of a `AstronautComponent` is the same as the lifetime of the app itself. That *would not* always be true in a more complex application.

You don't add this guard to the `MissionControlComponent` because, as the parent, it controls the lifetime of the `MissionService`.

The `History` log demonstrates that messages travel in both directions between the parent `MissionControlComponent` and the `AstronautComponent` children, facilitated by the service:

Mission Control

Announce mission

Lovell: <no mission announced> Confirm

Swigert: <no mission announced> Confirm

Haise: <no mission announced> Confirm

History

Test it

Tests click buttons of both the parent `MissionControlComponent` and the `AstronautComponent` children and verify that the history meets expectations:

component-interaction/e2e-spec.ts

```
1. // ...
2. it('should announce a mission', function () {
3.   let missionControl = element(by.tagName('mission-control'));
4.   let announceButton = missionControl.all(by.tagName('button')).get(0);
5.   announceButton.click().then(function () {
```

```
6.   let history = missionControl.all(by.tagName('li'));
7.   expect(history.count()).toBe(1);
8.   expect(history.get(0).getText()).toMatch(/Mission.* announced/);
9. });
10. });
11.
12. it('should confirm the mission by Lovell', function () {
13.   testConfirmMission(1, 2, 'Lovell');
14. });
15.
16. it('should confirm the mission by Haise', function () {
17.   testConfirmMission(3, 3, 'Haise');
18. });
19.
20. it('should confirm the mission by Swigert', function () {
21.   testConfirmMission(2, 4, 'Swigert');
22. });
23.
24. function testConfirmMission(buttonIndex: number, expectedLogCount: number,
astronaut: string) {
25.   let _confirmedLog = ' confirmed the mission';
26.   let missionControl = element(by.tagName('mission-control'));
27.   let confirmButton =
missionControl.all(by.tagName('button')).get(buttonIndex);
28.   confirmButton.click().then(function () {
29.     let history = missionControl.all(by.tagName('li'));
30.     expect(history.count()).toBe(expectedLogCount);
31.     expect(history.get(expectedLogCount - 1).getText()).toBe(astronaut +
_confirmedLog);
32.   });

```

33. }

34. // ...

[Back to top](#)

This is the archived documentation for Angular v4. Please visit [angular.io](#) to see documentation for the current version of Angular.

Component Styles

[Contents >](#)

[Using component styles](#)

[Special selectors](#)

[:host](#)

•••

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

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

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

You can run the [live example](#) / [download example](#) in 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:

src/app/hero-app.component.ts

```
@Component({
  selector: 'hero-app',
  template: `
    <h1>Tour of Heroes</h1>
    <hero-app-main [hero]=hero></hero-app-main>`,
  styles: ['h1 { font-weight: normal; }']
})
export class HeroAppComponent {
  /* . . . */
}
```



The selectors you put into a component's styles apply only within the template of that component. The `h1` selector in the preceding example applies only to the `<h1>` tag in the template of `HeroAppComponent`. Any `<h1>` elements elsewhere in the application are unaffected.

This is a big improvement in modularity compared to how CSS traditionally works.

- You can use the CSS class names and selectors that make the most sense in the context of each component.
- Class names and selectors are local to the component and don't collide with classes and selectors used elsewhere in the application.
- Changes to styles elsewhere in the application don't affect the component's styles.
- You can co-locate the CSS code of each component with the TypeScript and HTML code of the component, which leads to a neat and tidy project structure.
- You can change or remove component CSS code without searching through the whole application to find where else the code is used.

Special selectors

Component styles have a few special *selectors* from the world of shadow DOM style scoping (described in the [CSS Scoping Module Level 1](#) page on the [W3C](#) site). The following sections describe these selectors.

:host

Use the `:host` pseudo-class selector to target styles in the element that *hosts* the component (as opposed to targeting elements *inside* the component's template).

src/app/hero-details.component.css

```
:host {  
  display: block;  
  border: 1px solid black;  
}
```

The `:host` selector is the only way to target the host element. You can't reach the host element from inside the component with other selectors because it's not part of the component's own template. The host element is in a parent component's template.

Use the *function form* to apply host styles conditionally by including another selector inside parentheses after `:host`.

The next example targets the host element again, but only when it also has the `active` CSS class.

src/app/hero-details.component.css

```
:host(.active) {  
  border-width: 3px;  
}
```

:host-context

Sometimes it's useful to apply styles based on some condition *outside* of a component's view. For example, a CSS theme class could be applied to the document `<body>` element, and you want to change how your component looks based on that.

Use the `:host-context()` pseudo-class selector, which works just like the function form of `:host()`. The `:host-context()` selector looks for a CSS class in any ancestor of the component host element, up to the document root. The `:host-context()` selector is useful when combined with another selector.

The following example applies a `background-color` style to all `<h2>` elements *inside* the component, only if some ancestor element has the CSS class `theme-light`.

src/app/hero-details.component.css

```
:host-context(.theme-light) h2 {  
  background-color: #eef;  
}
```

(deprecated) /deep/, >>>, and ::ng-deep

Component styles normally apply only to the HTML in the component's own template.

Use the `/deep/` shadow-piercing descendant combinator to force a style down through the child component tree into all the child component views. The `/deep/` combinator works to any depth of nested components, and it applies to both the view children and content children of the component.

The following example targets all `<h3>` elements, from the host element down through this component to all of its child elements in the DOM.

```
:host /deep/ h3 {  
  font-style: italic;  
}
```



The `/deep/` combinator also has the aliases `>>>`, and `::ng-deep`.

Use `/deep/`, `>>>` and `::ng-deep` only with *emulated* view encapsulation. Emulated is the default and most commonly used view encapsulation. For more information, see the [Controlling view encapsulation](#) section.

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

Loading component styles

There are several ways to add styles to a component:

- By setting `styles` or `styleUrls` metadata.
- Inline in the template HTML.
- With CSS imports.

The scoping rules outlined earlier apply to each of these loading patterns.

Styles in metadata

You can add a `styles` array property to the `@Component` decorator. Each string in the array (usually just one string) defines the CSS.

src/app/hero-app.component.ts

```
@Component({
  selector: 'hero-app',
  template: `
    <h1>Tour of Heroes</h1>
    <hero-app-main [hero]=hero></hero-app-main>`,
  styles: ['h1 { font-weight: normal; }']
})
export class HeroAppComponent {
  /* . . . */
}
```

Style URLs in metadata

You can load styles from external CSS files by adding a `styleUrls` attribute into a component's `@Component` decorator:

src/app/hero-details.component.ts

```
1. @Component({
2.   selector: 'hero-details',
3.   template: `
4.     <h2>{{hero.name}}</h2>
```

```
5.      <hero-team [hero]=hero></hero-team>
6.      <ng-content></ng-content>
7.      `,
8.      styleUrls: ['app/hero-details.component.css']
9.  })
10. export class HeroDetailsComponent {
11. /* . . . */
12. }
```

The URL is relative to the *application root*, which is usually the location of the `index.html` web page that hosts the application. The style file URL is *not* relative to the component file. That's why the example URL begins `src/app/`. To specify a URL relative to the component file, see [Appendix 2](#).

If you use module bundlers like Webpack, you can also use the `styles` attribute to load styles from external files at build time. You could write:

```
styles: [require('my.component.css')]
```

Set the `styles` property, not the `styleUrls` property. The module bundler loads the CSS strings, not Angular. Angular sees the CSS strings only after the bundler loads them. To Angular, it's as if you wrote the `styles` array by hand. For information on loading CSS in this manner, refer to the module bundler's documentation.

Template inline styles

You can embed styles directly into the HTML template by putting them inside `<style>` tags.

src/app/hero-controls.component.ts

```
1. @Component({
2.   selector: 'hero-controls',
3.   template: `
4.     <style>
5.       button {
6.         background-color: white;
7.         border: 1px solid #777;
8.       }
9.     </style>
10.    <h3>Controls</h3>
11.    <button (click)="activate()">Activate</button>
12.  `
13. })
```



Template link tags

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

As with `styleUrls`, the link tag's `href` URL is relative to the application root, not the component file.

src/app/hero-team.component.ts

```
@Component({
  selector: 'hero-team',
  template: `
    <link rel="stylesheet" href="app/hero-team.component.css">
```



```
<h3>Team</h3>
<ul>
  <li *ngFor="let member of hero.team">
    {{member}}
  </li>
</ul>
})
```

CSS @imports

You can also import CSS files into the CSS files using the standard CSS `@import` rule. For details, see [@import](#) on the [MDN](#) site.

In this case, the URL is relative to the CSS file into which you're importing.

src/app/hero-details.component.css (excerpt)

```
@import 'hero-details-box.css';
```



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:

src/app/quest-summary.component.ts

```
// warning: few browsers support shadow DOM encapsulation at this time
encapsulation: ViewEncapsulation.Native
```

`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.

Appendix: 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.

Appendix: Loading with relative URLs

It's common practice to split a component's code, HTML, and CSS into three separate files in the same directory:

```
quest-summary.component.ts
quest-summary.component.html
quest-summary.component.css
```



You include the template and CSS files by setting the `templateUrl` and `styleUrls` metadata properties respectively. Because these files are co-located with the component, it would be nice to refer to them by name without also having to specify a path back to the root of the application.

You can use a relative URL by prefixing your filenames with `./`:

```
src/app/quest-summary.component.ts
```

```
@Component({
  selector: 'quest-summary',
  templateUrl: './quest-summary.component.html',
  styleUrls: ['./quest-summary.component.css']
})
export class QuestSummaryComponent { }
```



This is the archived documentation for Angular v4. Please visit [angular.io](#) to see documentation for the current version of Angular.

Dynamic Component Loader

[Contents >](#)

[Dynamic component loading](#)

[The anchor directive](#)

[Loading components](#)

...

Component templates are not always fixed. An application may need to load new components at runtime.

This cookbook shows you how to use `ComponentFactoryResolver` to add components dynamically.

See the [live example](#) / [download example](#) of the code in this cookbook.

Dynamic component loading

The following example shows how to build a dynamic ad banner.

The hero agency is planning an ad campaign with several different ads cycling through the banner. New ad components are added frequently by several different teams. This makes it impractical to use a template with a static component structure.

Instead, you need a way to load a new component without a fixed reference to the component in the ad banner's template.

Angular comes with its own API for loading components dynamically.

The anchor directive

Before you can add components you have to define an anchor point to tell Angular where to insert components.

The ad banner uses a helper directive called `AdDirective` to mark valid insertion points in the template.

src/app/ad.directive.ts

```
import { Directive, ViewContainerRef } from '@angular/core';

@Directive({
  selector: '[ad-host]',
})
export class AdDirective {
  constructor(public viewContainerRef: ViewContainerRef) { }
}
```

`AdDirective` injects `ViewContainerRef` to gain access to the view container of the element that will host the dynamically added component.

In the `@Directive` decorator, notice the selector name, `ad-host`; that's what you use to apply the directive to the element. The next section shows you how.

Loading components

Most of the ad banner implementation is in `ad-banner.component.ts`. To keep things simple in this example, the HTML is in the `@Component` decorator's `template` property as a template string.

The `<ng-template>` element is where you apply the directive you just made. To apply the `AdDirective`, recall the selector from `ad.directive.ts`, `ad-host`. Apply that to `<ng-template>` without the square brackets. Now Angular knows where to dynamically load components.

src/app/ad-banner.component.ts (template)

```
template: `

  <div class="ad-banner">
    <h3>Advertisements</h3>
    <ng-template ad-host></ng-template>
  </div>
`
```



The `<ng-template>` element is a good choice for dynamic components because it doesn't render any additional output.

Resolving components

Take a closer look at the methods in `ad-banner.component.ts`.

`AdBannerComponent` takes an array of `AdItem` objects as input, which ultimately comes from `AdService`. `AdItem` objects specify the type of component to load and any data to bind to the component. `AdService` returns the actual ads making up the ad campaign.

Passing an array of components to `AdBannerComponent` allows for a dynamic list of ads without static elements in the template.

With its `getAds()` method, `AdBannerComponent` cycles through the array of `AdItems` and loads a new component every 3 seconds by calling `loadComponent()`.

src/app/ad-banner.component.ts (excerpt)

```
export class AdBannerComponent implements AfterViewInit, OnDestroy {  
  @Input() ads: AdItem[];  
  currentAddIndex: number = -1;  
  @ViewChild(AdDirective) adHost: AdDirective;  
  subscription: any;  
  interval: any;  
  
  constructor(private componentFactoryResolver: ComponentFactoryResolver) { }  
  
  ngAfterViewInit() {  
    this.loadComponent();  
    this.getAds();  
  }  
  
  ngOnDestroy() {  
    clearInterval(this.interval);  
  }  
  
  loadComponent() {  
    this.currentAddIndex = (this.currentAddIndex + 1) % this.ads.length;  
    let adItem = this.ads[this.currentAddIndex];  
  
    let componentFactory =  
      this.componentFactoryResolver.resolveComponentFactory(adItem.component);  
  
    let viewContainerRef = this.adHost.viewContainerRef;  
    viewContainerRef.clear();  
  
    let componentRef = viewContainerRef.createComponent(componentFactory);  
    (<AdComponent>componentRef.instance).data = adItem.data;  
  }  
}
```

```
}

getAds() {
  this.interval = setInterval(() => {
    this.loadComponent();
  }, 3000);
}

}
```

The `loadComponent()` method is doing a lot of the heavy lifting here. Take it step by step. First, it picks an ad.

How `loadComponent()` chooses an ad

The `loadComponent()` method chooses an ad using some math.

First, it sets the `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.

Selector references

Generally, the Angular compiler generates a `ComponentFactory` for any component referenced in a template. However, there are no selector references in the templates for dynamically loaded components since they load at runtime.

To ensure that the compiler still generates a factory, add dynamically loaded components to the

`NgModule`'s `entryComponents` array:

src/app/app.module.ts (entry components)

```
entryComponents: [ HeroJobAdComponent, HeroProfileComponent ],
```

The *AdComponent* interface

In the ad banner, all components implement a common `AdComponent` interface to standardize the API for passing data to the components.

Here are two sample components and the `AdComponent` interface for reference:

hero-job-ad.component.ts hero-profile.component.ts ad.component.ts

```
1. import { Component, Input } from '@angular/core';
2.
```

```
3. import { AdComponent }      from './ad.component';
4.
5. @Component({
6.   template: `
7.     <div class="job-ad">
8.       <h4>{{data.headline}}</h4>
9.
10.      {{data.body}}
11.    </div>
12.  `
13. })
14. export class HeroJobAdComponent implements AdComponent {
15.   @Input() data: any;
16.
17. }
```

Final ad banner

The final ad banner looks like this:

Featured Hero Profile

Bombasto

Brave as they come

Hire this hero today!

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

This is the archived documentation for Angular v4. Please visit [angular.io](#) to see documentation for the current version of Angular.

Attribute Directives

[Contents >](#)

[Directives overview](#)

[Build a simple attribute directive](#)

[Write the directive code](#)

•••

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

Try the [Attribute Directive example](#) / [download example](#).

Directives overview

There are three kinds of directives in Angular:

1. Components—directives with a template.
2. Structural directives—change the DOM layout by adding and removing DOM elements.
3. Attribute directives—change the appearance or behavior of an element, component, or another directive.

Components are the most common of the three directives. You saw a component for the first time in the [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 *myHighlight* attribute directive to set an element's background color when the user hovers over that element. You can apply it like this:

src/app/app.component.html (applied)

```
<p myHighlight>Highlight me!</p>
```

Write the directive code

Follow the [setup](#) instructions for creating a new local project named `attribute-directives`.

Create the following source file in the indicated folder:

src/app/highlight.directive.ts

```
import { Directive, ElementRef, Input } from '@angular/core';

@Directive({ selector: '[myHighlight]' })
```

```
export class HighlightDirective {
  constructor(el: ElementRef) {
    el.nativeElement.style.backgroundColor = 'yellow';
  }
}
```

The `import` statement specifies symbols from the Angular `core`:

1. `Directive` provides the functionality of the `@Directive` decorator.
2. `ElementRef` injects into the directive's constructor so the code can access the DOM element.
3. `Input` allows data to flow from the binding expression into the directive.

Next, the `@Directive` decorator function contains the directive metadata in a configuration object as an argument.

`@Directive` requires a CSS selector to identify the HTML in the template that is associated with the directive. The [CSS selector for an attribute](#) is the attribute name in square brackets. Here, the directive's selector is `[myHighlight]`. Angular locates all elements in the template that have an attribute named `myHighlight`.

Why not call it "highlight"?

Though `highlight` is a more concise name than `myHighlight` and would work, a 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.

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. For a simple demo, the short prefix, `my`, helps distinguish your custom directive.

After the `@Directive` metadata comes the directive's controller class, called `HighlightDirective`, which contains the logic for the directive. Exporting `HighlightDirective` makes it accessible to other components.

Angular creates a new instance of the directive's controller class for each matching element, injecting an Angular `ElementRef` into the constructor. `ElementRef` is a service that grants direct access to the DOM element through its `nativeElement` property.

Apply the attribute directive

To use the new `HighlightDirective`, create a template that applies the directive as an attribute to a paragraph (`<p>`) element. In Angular terms, the `<p>` element is the attribute host.

Put the template in its own `app.component.html` file that looks like this:

src/app/app.component.html

```
<h1>My First Attribute Directive</h1>
<p myHighlight>Highlight me!</p>
```

Now reference this template in the `AppComponent`:

src/app/app.component.ts

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

@Component({
  selector: 'my-app',
  templateUrl: './app.component.html'
})
```

```
export class AppComponent {  
  color: string;  
}  
}
```

Next, add an `import` statement to fetch the `Highlight` directive and add that class to the `declarations` NgModule metadata. This way Angular recognizes the directive when it encounters `myHighlight` in the template.

src/app/app.module.ts

```
1. import { NgModule } from '@angular/core';  
2. import { BrowserModule } from '@angular/platform-browser';  
3.  
4. import { AppComponent } from './app.component';  
5. import { HighlightDirective } from './highlight.directive';  
6.  
7. @NgModule({  
8.   imports: [ BrowserModule ],  
9.   declarations: [  
10.    AppComponent,  
11.    HighlightDirective  
12.   ],  
13.   bootstrap: [ AppComponent ]  
14. })  
15. export class AppModule {}
```



Now when the app runs, the `myHighlight` directive highlights the paragraph text.

My First Angular 2 App

Highlight me!

Your directive isn't working?

Did you remember to add the directive to the `declarations` attribute of `@NgModule`? It is easy to forget! Open the console in the browser tools and look for an error like this:

EXCEPTION: `Template` parse errors:

`Can't bind to 'myHighlight' since it isn't a known property of 'p'.`

Angular detects that you're trying to bind to *something* but it can't find this directive in the module's `declarations` array. After specifying `HighlightDirective` in the `declarations` array, Angular knows it can apply the directive to components declared in this module.

To summarize, Angular found the `myHighlight` attribute on the `<p>` element. It created an instance of the `HighlightDirective` class and injected a reference to the `<p>` element into the directive's constructor which sets the `<p>` element's background style to yellow.

Respond to user-initiated events

Currently, `myHighlight` 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; add the `Input` symbol as well because you'll need it soon.

src/app/highlight.directive.ts (imports)

```
import { Directive, ElementRef, HostListener, Input } from '@angular/core';
```



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

src/app/highlight.directive.ts (mouse-methods)

```
@HostListener('mouseenter') onMouseEnter() {
  this.highlight('yellow');
}

@HostListener('mouseleave') onMouseLeave() {
  this.highlight(null);
}

private highlight(color: string) {
  this.el.nativeElement.style.backgroundColor = color;
}
```



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

Of course you could reach into the DOM with standard JavaScript and attach event listeners manually. There are at least three problems with *that* approach:

1. You have to write the listeners correctly.
2. The code must *detach* the listener when the directive is destroyed to avoid memory leaks.
3. Talking to DOM API directly isn't a best practice.

The handlers delegate to a helper method that sets the color on the DOM element, `el`, which you declare and initialize in the constructor.

src/app/highlight.directive.ts (constructor)

```
constructor(private el: ElementRef) { }
```

Here's the updated directive in full:

src/app/highlight.directive.ts

```
1. import { Directive, ElementRef, HostListener, Input } from '@angular/core';  
2.  
3. @Directive({  
4.   selector: '[myHighlight]'  
5. })  
6. export class HighlightDirective {  
7.   constructor(private el: ElementRef) { }  
8.  
9.   @HostListener('mouseenter') onMouseEnter() {  
10.     this.highlight('yellow');  
11.   }  
12.  
13.   @HostListener('mouseleave') onMouseLeave() {  
14.     this.highlight(null);  
15.   }  
16.  
17.   private highlight(color: string) {  
18.     this.el.nativeElement.style.backgroundColor = color;  
19.   }  
20. }  
21.  
22. <div myHighlight>Hello World</div>
```

```
15. }
16.
17. private highlight(color: string) {
18.   this.el.nativeElement.style.backgroundColor = color;
19. }
```

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



Highlight me!

Pass values into the directive with an `@Input` data binding

Currently the highlight color is hard-coded *within* the directive. That's inflexible. In this section, you give the developer the power to set the highlight color while applying the directive.

Start by adding a `highlightColor` property to the directive class like this:

src/app/highlight.directive.ts (highlightColor)

```
@Input() highlightColor: string;
```



Binding to an `@Input` property

Notice the `@Input` decorator. It adds metadata to the class that makes the directive's `highlightColor` property available for binding.

It's called an *input* property because data flows from the binding expression *into* the directive. Without that input metadata, Angular rejects the binding; see [below](#) for more about that.

Try it by adding the following directive binding variations to the `AppComponent` template:

src/app/app.component.html (excerpt)

```
<p myHighlight highlightColor="yellow">Highlighted in yellow</p>
<p myHighlight [highlightColor]="'orange'">Highlighted in orange</p>
```

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

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

```
export class AppComponent {
  color = 'yellow';
}
```

Let it control the highlight color with a property binding.

src/app/app.component.html (excerpt)

```
<p myHighlight [highlightColor]="color">Highlighted with parent component's
color</p>
```

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

src/app/app.component.html (color)

```
<p [myHighlight]="color">Highlight me!</p>
```



The `[myHighlight]` 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 (`[myHighlight]`) to do both jobs. That's a crisp, compact syntax.

You'll have to rename the directive's `highlightColor` property to `myHighlight` because that's now the color property binding name.

src/app/highlight.directive.ts (renamed to match directive selector)

```
@Input() myHighlight: string;
```



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

Bind to an `@Input` alias

Fortunately you can name the directive property whatever you want *and alias it* for binding purposes.

Restore the original property name and specify the selector as the alias in the argument to `@Input`.

src/app/highlight.directive.ts (color property with alias)

```
@Input('myHighlight') highlightColor: string;
```



Inside the directive the property is known as `highlightColor` . *Outside* the directive, where you bind to it, it's known as `myHighlight` .

You get the best of both worlds: the property name you want and the binding syntax you want:

src/app/app.component.html (color)

```
<p [myHighlight]="color">Highlight me!</p>
```

Now that you're binding to `highlightColor` , modify the `onMouseEnter()` method to use it. If someone neglects to bind to `highlightColor` , highlight in red:

src/app/highlight.directive.ts (mouse enter)

```
@HostListener('mouseenter') onMouseEnter() {
  this.highlight(this.highlightColor || 'red');
}
```

Here's the latest version of the directive class.

src/app/highlight.directive.ts (excerpt)

```
import { Directive, ElementRef, HostListener, Input } from '@angular/core';

@Directive({
  selector: '[myHighlight]'
})
export class HighlightDirective {

  constructor(private el: ElementRef) { }
```

```
@Input('myHighlight') highlightColor: string;

@HostListener('mouseenter') onMouseEnter() {
  this.highlight(this.highlightColor || 'red');
}

@HostListener('mouseleave') onMouseLeave() {
  this.highlight(null);
}

private highlight(color: string) {
  this.el.nativeElement.style.backgroundColor = color;
}
```

Write a harness to try it

It may be difficult to imagine how this directive actually works. In this section, you'll turn `AppComponent` into a harness that lets you pick the highlight color with a radio button and bind your color choice to the directive.

Update `app.component.html` as follows:

src/app/app.component.html (v2)

```
<h1>My First Attribute Directive</h1>
```



```
<h4>Pick a highlight color</h4>
<div>
```

```
<input type="radio" name="colors" (click)="color='lightgreen'">Green  
<input type="radio" name="colors" (click)="color='yellow'">Yellow  
<input type="radio" name="colors" (click)="color='cyan'">Cyan  
</div>  
<p [myHighlight]="color">Highlight me!</p>
```

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

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

```
export class AppComponent {  
  color: string;  
}
```



Here are the harness and directive in action.

My First Attribute Directive

Pick a highlight color

Green Yellow Cyan

Highlight me!

Bind to a second property

This highlight directive has a single customizable property. In a real app, it may need more.

At the moment, the default color—the color that prevails until the user picks a highlight color—is hard-coded as "red". Let the template developer set the default color.

Add a second input property to `HighlightDirective` called `defaultColor`:

src/app/highlight.directive.ts (defaultColor)

```
@Input() defaultColor: string;
```

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

src/app/highlight.directive.ts (mouse-enter)

```
@HostListener('mouseenter') onMouseEnter() {
  this.highlight(this.highlightColor || this.defaultColor || 'red');
}
```

How do you bind to a second property when you're already binding to the `myHighlight` attribute name?

As with components, you can add as many directive property bindings as you need by stringing them along in the template. The developer should be able to write the following template HTML to both bind to the `AppComponent.color` and fall back to "violet" as the default color.

src/app/app.component.html (defaultColor)

```
<p [myHighlight]="color" defaultColor="violet">
  Highlight me too!
</p>
```

Angular knows that the `defaultColor` binding belongs to the `HighlightDirective` because you made it `public` with the `@Input` decorator.

Here's how the harness should work when you're done coding.



Summary

This page covered how to:

- [Build an attribute directive](#) that modifies the behavior of an element.
- [Apply the directive](#) to an element in a template.
- [Respond to events](#) that change the directive's behavior.
- [Bind values to the directive](#).

The final source code follows:

< app/app.component.ts app/app.component.html app/highlight.directive.ts app/app.mod >

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

@Component({
  selector: 'my-app',
  templateUrl: './app.component.html'
})
export class AppComponent {
  color: string;
}
```

You can also experience and download the [Attribute Directive example / download example](#).

Appendix: Why add `@Input`?

In this demo, the `highlightColor` property is an *input* property of the `HighlightDirective`. You've seen it applied without an alias:

src/app/highlight.directive.ts (color)

```
@Input() highlightColor: string;
```

You've seen it with an alias:

src/app/highlight.directive.ts (color)

```
@Input('myHighlight') highlightColor: string;
```

Either way, the `@Input` decorator tells Angular that this property is *public* and available for binding by a parent component. Without `@Input`, Angular refuses to bind to the property.

You've bound template HTML to component properties before and never used `@Input`. What's different?

The difference is a matter of trust. Angular treats a component's template as *belonging* to the component. The component and its template trust each other implicitly. Therefore, the component's own template may bind to *any* property of that component, with or without the `@Input` decorator.

But a component or directive shouldn't blindly trust *other* components and directives. The properties of a component or directive are hidden from binding by default. They are *private* from an Angular binding perspective. When adorned with the `@Input` decorator, the property becomes *public* from an Angular binding perspective. Only then can it be bound by some other component or directive.

You can tell if `@Input` is needed by the position of the property name in a binding.

- When it appears in the template expression to the *right* of the equals (=), it belongs to the template's component and does not require the `@Input` decorator.
- When it appears in square brackets ([]) to the left of the equals (=), the property belongs to some *other* component or directive; that property must be adorned with the `@Input` decorator.

Now apply that reasoning to the following example:

src/app/app.component.html (color)

```
<p [myHighlight]="color">Highlight me!</p>
```



- The `color` property in the expression on the right belongs to the template's component. The template and its component trust each other. The `color` property doesn't require the `@Input` decorator.
- The `myHighlight` property on the left refers to an *aliased* property of the `HighlightDirective`, not a property of the template's component. There are trust issues. Therefore, the directive property must

carry the `@Input` decorator.

This is the archived documentation for Angular v4. Please visit [angular.io](#) to see documentation for the current version of Angular.

Structural Directives

[Contents >](#)

[What are structural directives?](#)

[NgIf case study](#)

[Why *remove* rather than *hide*?](#)

•••

This guide looks at how Angular manipulates the DOM with structural directives and how you can write your own structural directives to do the same thing.

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

What are structural directives?

Structural directives are responsible for HTML layout. They shape or reshape the DOM's *structure*, typically by adding, removing, or manipulating elements.

As with other directives, you apply a structural directive to a *host element*. The directive then does whatever it's supposed to do with that host element and its descendants.

Structural directives are easy to recognize. An asterisk (*) precedes the directive attribute name as in this example.

src/app/app.component.html (ngif)

```
<div *ngIf="hero" >{{hero.name}}</div>
```



No brackets. No parentheses. Just `*ngIf` set to a string.

You'll learn in this guide that the [asterisk \(*\) is a convenience notation](#) and the string is a [*microsyntax*](#) rather than the usual [template expression](#). Angular desugars this notation into a marked-up `<ng-template>` that surrounds the host element and its descendants. Each structural directive does something different with that template.

Three of the common, built-in structural directives—[NgIf](#), [NgFor](#), and [NgSwitch...](#)—are described in the [Template Syntax](#) guide and seen in samples throughout the Angular documentation. Here's an example of them in a template:

src/app/app.component.html (built-in)

```
<div *ngIf="hero" >{{hero.name}}</div>

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

<div [ngSwitch]="hero?.emotion">
  <happy-hero    *ngSwitchCase="'happy'"    [hero]="hero"></happy-hero>
  <sad-hero     *ngSwitchCase="'sad'"      [hero]="hero"></sad-hero>
  <confused-hero *ngSwitchCase="'confused'" [hero]="hero"></confused-hero>
  <unknown-hero  *ngSwitchDefault        [hero]="hero"></unknown-hero>
</div>
```



This guide won't repeat how to *use* them. But it does explain *how they work* and how to [write your own structural directive](#).

DIRECTIVE SPELLING

Throughout this guide, you'll see a directive spelled in both *UpperCamelCase* and *lowerCamelCase*.

Already you've seen `NgIf` and `ngIf`. There's a reason. `NgIf` refers to the directive *class*; `ngIf` refers to the directive's *attribute name*.

A directive *class* is spelled in *UpperCamelCase* (`NgIf`). A directive's *attribute name* is spelled in *lowerCamelCase* (`ngIf`). The guide refers to the directive *class* when talking about its properties and what the directive does. The guide refers to the *attribute name* when describing how you apply the directive to an element in the HTML template.

There are two other kinds of Angular directives, described extensively elsewhere: (1) components and (2) attribute directives.

A *component* manages a region of HTML in the manner of a native HTML element. Technically it's a directive with a template.

An [attribute directive](#) changes the appearance or behavior of an element, component, or another directive. For example, the built-in `NgStyle` directive changes several element styles at the same time.

You can apply many *attribute* directives to one host element. You can [only apply one structural](#) directive to a host element.

NgIf case study

`NgIf` is the simplest structural directive and the easiest to understand. It takes a boolean expression and makes an entire chunk of the DOM appear or disappear.

src/app/app.component.html (ngif-true)

```
<p *ngIf="true">  
  Expression is true and ngIf is true.  
  This paragraph is in the DOM.  
</p>  
<p *ngIf="false">  
  Expression is false and ngIf is false.  
  This paragraph is not in the DOM.  
</p>
```

The `ngIf` directive doesn't hide elements with CSS. It adds and removes them physically from the DOM. Confirm that fact using browser developer tools to inspect the DOM.

```
<p _ngcontent-yhe-0>  
  Expression is true and ngIf is true.  
  This paragraph is in the DOM.  
</p>  
<!--template bindings={  
  "ng-reflect-ng-if": null  
}-->
```

The top paragraph is in the DOM. The bottom, disused paragraph is not; in its place is a comment about "bindings" (more about that [later](#)).

When the condition is false, `NgIf` removes its host element from the DOM, detaches it from DOM events (the attachments that it made), detaches the component from Angular change detection, and destroys it. The component and DOM nodes can be garbage-collected and free up memory.

Why *remove* rather than *hide*?

A directive could hide the unwanted paragraph instead by setting its `display` style to `none`.

src/app/app.component.html (display-none)

```
<p [style.display]="'block'">
  Expression sets display to "block".
  This paragraph is visible.
</p>
<p [style.display]="'none'">
  Expression sets display to "none".
  This paragraph is hidden but still in the DOM.
</p>
```

While invisible, the element remains in the DOM.

```
<p _ngcontent-fwv-0 style="display: block;">
  Expression sets display to "block" .
  This paragraph is visible.
</p>
<p _ngcontent-fwv-0 style="display: none;">
  Expression sets display to "none" .
  This paragraph is hidden but still in the DOM.
</p>
```

The difference between hiding and removing doesn't matter for a simple paragraph. It does matter when the host element is attached to a resource intensive component. Such a component's behavior continues even when hidden. The component stays attached to its DOM element. It keeps listening to events. Angular keeps checking for changes that could affect data bindings. Whatever the component was doing, it keeps doing.

Although invisible, the component—and all of its descendant components—tie up resources. The performance and memory burden can be substantial, responsiveness can degrade, and the user sees nothing.

On the positive side, showing the element again is quick. The component's previous state is preserved and ready to display. The component doesn't re-initialize—an operation that could be expensive. So hiding and showing is sometimes the right thing to do.

But in the absence of a compelling reason to keep them around, your preference should be to remove DOM elements that the user can't see and recover the unused resources with a structural directive like `NgIf`.

These same considerations apply to every structural directive, whether built-in or custom. Before applying a structural directive, you might want to pause for a moment to consider the consequences of adding and removing elements and of creating and destroying components.

The asterisk (*) prefix

Surely you noticed the asterisk (*) prefix to the directive name and wondered why it is necessary and what it does.

Here is `*ngIf` displaying the hero's name if `hero` exists.

src/app/app.component.html (asterisk)

```
<div *ngIf="hero" >{{hero.name}}</div>
```

The asterisk is "syntactic sugar" for something a bit more complicated. Internally, Angular desugars it in two stages. First, it translates the `*ngIf="..."` into a template *attribute*, `template="ngIf ..."`, like this.

src/app/app.component.html (ngif-template-attr)

```
<div template="ngIf hero">{{hero.name}}</div>
```

Then it translates the template *attribute* into a `<ng-template>` element, wrapped around the host element, like this.

src/app/app.component.html (ngif-template)

```
<ng-template [ngIf]="hero">
  <div>{{hero.name}}</div>
</ng-template>
```

- The `*ngIf` directive moved to the `<ng-template>` element where it became a property binding, `[ngIf]`.

- The rest of the `<div>`, including its class attribute, moved inside the `<ng-template>` element.

None of these forms are actually rendered. Only the finished product ends up in the DOM.

```
<!--template bindings={  
  "ng-reflect-ng-if": "[object Object]"  
}-->  
<div ngcontent-exn-0>Mr. Nice</div>
```

Angular consumed the `<ng-template>` content during its actual rendering and replaced the `<ng-template>` with a diagnostic comment.

The `NgFor` and `NgSwitch...` directives follow the same pattern.

Inside `*ngFor`

Angular transforms the `*ngFor` in similar fashion from asterisk (*) syntax through template *attribute* to `<ng-template>` *element*.

Here's a full-featured application of `NgFor`, written all three ways:

src/app/app.component.html (inside-ngfor)

```
<div *ngFor="let hero of heroes; let i=index; let odd=odd; trackBy: trackById"  
[class.odd]="odd">  
  ({{i}}) {{hero.name}}  
</div>  
  
<div template="ngFor let hero of heroes; let i=index; let odd=odd; trackBy:  
trackById" [class.odd]="odd">  
  ({{i}}) {{hero.name}}
```

```
</div>

<ng-template ngFor let-hero [ngForOf]="heroes" let-i="index" let-odd="odd"
[ngForTrackBy]="trackById">
  <div [class.odd]="odd">{{i}} {{hero.name}}</div>
</ng-template>
```

This is manifestly more complicated than `ngIf` and rightly so. The `NgFor` directive has more features, both required and optional, than the `NgIf` shown in this guide. At minimum `NgFor` needs a looping variable (`let hero`) and a list (`heroes`).

You enable these features in the string assigned to `ngFor`, which you write in Angular's [microsyntax](#).

Everything *outside* the `ngFor` string stays with the host element (the `<div>`) as it moves inside the `<ng-template>`. In this example, the `[ngClass]="odd"` stays on the `<div>`.

Microsyntax

The Angular microsyntax lets you configure a directive in a compact, friendly string. The microsyntax parser translates that string into attributes on the `<ng-template>`:

- The `let` keyword declares a [*template input variable*](#) that you reference within the template. The input variables in this example are `hero`, `i`, and `odd`. The parser translates `let hero`, `let i`, and `let odd` into variables named, `let-hero`, `let-i`, and `let-odd`.
- The microsyntax parser takes `of` and `trackby`, title-cases them (`of -> Of`, `trackBy -> TrackBy`), and prefixes them with the directive's attribute name (`ngFor`), yielding the names `ngForOf` and `ngForTrackBy`. Those are the names of two `NgFor` [*input properties*](#). That's how the directive learns that the list is `heroes` and the track-by function is `trackById`.

- As the `NgFor` directive loops through the list, it sets and resets properties of its own `context` object. These properties include `index` and `odd` and a special property named `$implicit`.
- The `let-i` and `let-odd` variables were defined as `let i=index` and `let odd=odd`. Angular sets them to the current value of the context's `index` and `odd` properties.
- The context property for `let-hero` wasn't specified. Its intended source is `implicit`. Angular sets `let-hero` to the value of the context's `$implicit` property which `NgFor` has initialized with the hero for the current iteration.
- The [API guide](#) describes additional `NgFor` directive properties and context properties.
- `NgFor` is implemented by the `NgForOf` directive. Read more about additional `NgForOf` directive properties and context properties [NgForOf API reference](#).

These microsyntax mechanisms are available to you when you write your own structural directives.

Studying the [source code for NgIf](#) and [NgForOf](#) is a great way to learn more.

Template input variable

A *template input variable* is a variable whose value you can reference *within* a single instance of the template. There are several such variables in this example: `hero`, `i`, and `odd`. All are preceded by the keyword `let`.

A *template input variable* is *not* the same as a [template reference variable](#), neither *semantically* nor *syntactically*.

You declare a template *input* variable using the `let` keyword (`let hero`). The variable's scope is limited to a *single instance* of the repeated template. You can use the same variable name again in the definition of other structural directives.

You declare a template *reference* variable by prefixing the variable name with `#` (`#var`). A *reference* variable refers to its attached element, component or directive. It can be accessed *anywhere* in the *entire template*.

Template *input* and *reference* variable names have their own namespaces. The `hero` in `let hero` is never the same variable as the `hero` declared as `#hero`.

One structural directive per host element

Someday you'll want to repeat a block of HTML but only when a particular condition is true. You'll *try* to put both an `*ngFor` and an `*ngIf` on the same host element. Angular won't let you. You may apply only one *structural* directive to an element.

The reason is simplicity. Structural directives can do complex things with the host element and its descendants. When two directives lay claim to the same host element, which one takes precedence? Which should go first, the `NgIf` or the `NgFor`? Can the `NgIf` cancel the effect of the `NgFor`? If so (and it seems like it should be so), how should Angular generalize the ability to cancel for other structural directives?

There are no easy answers to these questions. Prohibiting multiple structural directives makes them moot. There's an easy solution for this use case: put the `*ngIf` on a container element that wraps the `*ngFor` element. One or both elements can be an `ng-container` so you don't have to introduce extra levels of HTML.

Inside *NgSwitch* directives

The Angular *NgSwitch* is actually a set of cooperating directives: `NgSwitch`, `NgSwitchCase`, and `NgSwitchDefault`.

Here's an example.

src/app/app.component.html (ngswitch)

```
<div [ngSwitch]="hero?.emotion">  
  <happy-hero *ngSwitchCase="'happy'" [hero]="hero"></happy-hero>
```

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

The switch value assigned to `NgSwitch` (`hero.emotion`) determines which (if any) of the switch cases are displayed.

`NgSwitch` itself is not a structural directive. It's an *attribute* directive that controls the behavior of the other two switch directives. That's why you write `[ngSwitch]`, never `*ngSwitch`.

`NgSwitchCase` and `NgSwitchDefault` *are* structural directives. You attach them to elements using the asterisk (*) prefix notation. An `NgSwitchCase` displays its host element when its value matches the switch value. The `NgSwitchDefault` displays its host element when no sibling `NgSwitchCase` matches the switch value.

The element to which you apply a directive is its *host element*. The `<happy-hero>` is the host element for the happy `*ngSwitchCase`. The `<unknown-hero>` is the host element for the `*ngSwitchDefault`.

As with other structural directives, the `NgSwitchCase` and `NgSwitchDefault` can be desugared into the template *attribute* form.

src/app/app.component.html (ngswitch-template-attr)

```
<div [ngSwitch]="hero?.emotion">
  <happy-hero    template="ngSwitchCase 'happy'"    [hero]="hero"></happy-hero>
  <sad-hero     template="ngSwitchCase 'sad'"      [hero]="hero"></sad-hero>
  <confused-hero template="ngSwitchCase 'confused'" [hero]="hero"></confused-hero>
```

```
<unknown-hero template="ngSwitchDefault" [hero]="hero"></unknown-hero>
</div>
```

That, in turn, can be desugared into the `<ng-template>` element form.

src/app/app.component.html (ngswitch-template)

```
<div [ngSwitch]="hero?.emotion">
  <ng-template [ngSwitchCase]="'happy'">
    <happy-hero [hero]="hero"></happy-hero>
  </ng-template>
  <ng-template [ngSwitchCase]="'sad'">
    <sad-hero [hero]="hero"></sad-hero>
  </ng-template>
  <ng-template [ngSwitchCase]="'confused'">
    <confused-hero [hero]="hero"></confused-hero>
  </ng-template>
  <ng-template ngSwitchDefault>
    <unknown-hero [hero]="hero"></unknown-hero>
  </ng-template>
</div>
```

Prefer the asterisk (*) syntax.

The asterisk (*) syntax is more clear than the other desugared forms. Use `<ng-container>` when there's no single element to host the directive.

While there's rarely a good reason to apply a structural directive in template *attribute* or *element* form, it's still important to know that Angular creates a `<ng-template>` and to understand how it works. You'll refer

to the `<ng-template>` when you [write your own structural directive](#).

The `<ng-template>`

The `<ng-template>` is an Angular element for rendering HTML. It is never displayed directly. In fact, before rendering the view, Angular *replaces* the `<ng-template>` and its contents with a comment.

If there is no structural directive and you merely wrap some elements in a `<ng-template>`, those elements disappear. That's the fate of the middle "Hip!" in the phrase "Hip! Hip! Hooray!".

src/app/app.component.html (template-tag)

```
<p>Hip!</p>
<ng-template>
  <p>Hip!</p>
</ng-template>
<p>Hooray!</p>
```

Angular erases the middle "Hip!", leaving the cheer a bit less enthusiastic.

```
<p _ngcontent-kun-0>Hip!</p>
<!--template bindings={}-->
<p _ngcontent-kun-0>Hooray!</p>
```

Hip!
Hooray!

A structural directive puts a `<ng-template>` to work as you'll see when you [write your own structural directive](#).

Group sibling elements with <ng-container>

There's often a *root* element that can and should host the structural directive. The list element (``) is a typical host element of an `NgFor` repeater.

src/app/app.component.html (ngfor-li)

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

When there isn't a host element, you can usually wrap the content in a native HTML container element, such as a `<div>`, and attach the directive to that wrapper.

src/app/app.component.html (ngif)

```
<div *ngIf="hero" >{{hero.name}}</div>
```

Introducing another container element—typically a `` or `<div>`—to group the elements under a single *root* is usually harmless. *Usually...* but not *always*.

The grouping element may break the template appearance because CSS styles neither expect nor accommodate the new layout. For example, suppose you have the following paragraph layout.

src/app/app.component.html (ngif-span)

```
<p>  
  I turned the corner  
  <span *ngIf="hero">  
    and saw {{hero.name}}. I waved  
  </span>
```

and continued on my way.

```
</p>
```

You also have a CSS style rule that happens to apply to a `` within a `<p>` aragraph.

src/app/app.component.css (p-span)

```
p span { color: red; font-size: 70%; }
```



The constructed paragraph renders strangely.

I turned the corner and saw Mr. Nice. I waved and continued on my way.

The `p span` style, intended for use elsewhere, was inadvertently applied here.

Another problem: some HTML elements require all immediate children to be of a specific type. For example, the `<select>` element requires `<option>` children. You can't wrap the *options* in a conditional `<div>` or a ``.

When you try this,

src/app/app.component.html (select-span)

```
<div>
  Pick your favorite hero
  (<label><input type="checkbox" checked (change)="showSad = !showSad">show
  sad</label>)
</div>
<select [(ngModel)]="hero">
```



```
<span *ngFor="let h of heroes">
  <span *ngIf="showSad || h.emotion !== 'sad'">
    <option [ngValue]="h">{{h.name}} ({{h.emotion}})</option>
  </span>
</span>
</select>
```

the drop down is empty.

Pick your favorite hero, who is not sad



The browser won't display an `<option>` within a ``.

<ng-container> to the rescue

The Angular `<ng-container>` is a grouping element that doesn't interfere with styles or layout because Angular *doesn't put it in the DOM*.

Here's the conditional paragraph again, this time using `<ng-container>`.

src/app/app.component.html (ngif-ngcontainer)

```
<p>
  I turned the corner
  <ng-container *ngIf="hero">
    and saw {{hero.name}}. I waved
  </ng-container>
```

and continued on my way.

```
</p>
```

It renders properly.

```
I turned the corner and saw Mr. Nice. I waved and continued on my way.
```

Now conditionally exclude a *select* `<option>` with `<ng-container>`.

src/app/app.component.html (select-ngcontainer)

```
<div>
  Pick your favorite hero
  (<label><input type="checkbox" checked (change)="showSad = !showSad">show
  sad</label>)
</div>
<select [(ngModel)]="hero">
  <ng-container *ngFor="let h of heroes">
    <ng-container *ngIf="showSad || h.emotion !== 'sad'">
      <option [ngValue]="h">{{h.name}} ({{h.emotion}})</option>
    </ng-container>
  </ng-container>
</select>
```

The drop down works properly.

Pick your favorite hero, who is not sad

Mr. Nice (happy)



The `<ng-container>` is a syntax element recognized by the Angular parser. It's not a directive, component, class, or interface. It's more like the curly braces in a JavaScript `if`-block:

```
if (someCondition) {  
    statement1;  
    statement2;  
    statement3;  
}
```



Without those braces, JavaScript would only execute the first statement when you intend to conditionally execute all of them as a single block. The `<ng-container>` satisfies a similar need in Angular templates.

Write a structural directive

In this section, you write an `UnlessDirective` structural directive that does the opposite of `NgIf`. `NgIf` displays the template content when the condition is `true`. `UnlessDirective` displays the content when the condition is `false`.

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

```
<p *myUnless="condition">Show this sentence unless the condition is true.</p>
```



Creating a directive is similar to creating a component.

- Import the `Directive` decorator (instead of the `Component` decorator).
- Import the `Input`, `TemplateRef`, and `ViewContainerRef` symbols; you'll need them for *any* structural directive.
- Apply the decorator to the directive class.
- Set the CSS *attribute selector* that identifies the directive when applied to an element in a template.

Here's how you might begin:

src/app/unless.directive.ts (skeleton)

```
import { Directive, Input, TemplateRef, ViewContainerRef } from '@angular/core';

@Directive({ selector: '[myUnless]' })
export class UnlessDirective {
}
```

The directive's *selector* is typically the directive's attribute name in square brackets, `[myUnless]`. 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 `my`.

The directive *class name* ends in `Directive` per the [style guide](#). Angular's own directives do not.

TemplateRef and *ViewContainerRef*

A simple structural directive like this one creates an [embedded view](#) from the Angular-generated `<ng-template>` and inserts that view in a [view container](#) adjacent to the directive's original `<p>` host element.

You'll acquire the `<ng-template>` contents with a `TemplateRef` and access the *view container* through a `ViewContainerRef`.

You inject both in the directive constructor as private variables of the class.

src/app/unless.directive.ts (ctor)

```
constructor(  
  private templateRef: TemplateRef<any>,  
  private viewContainer: ViewContainerRef) { }
```

The *myUnless* property

The directive consumer expects to bind a true/false condition to `[myUnless]`. That means the directive needs a `myUnless` property, decorated with `@Input`

Read about `@Input` in the [Template Syntax](#) guide.

src/app/unless.directive.ts (set)

```
@Input() set myUnless(condition: boolean) {  
  if (!condition && !this.hasView) {  
    this.viewContainer.createEmbeddedView(this.templateRef);  
    this.hasView = true;  
  } else if (condition && this.hasView) {  
    this.viewContainer.clear();  
    this.hasView = false;  
  }  
}
```

```
    }
}
```

Angular sets the `myUnless` property whenever the value of the condition changes. Because the `myUnless` 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 `myUnless` property so it doesn't need a getter.

The completed directive code looks like this:

src/app/unless.directive.ts (excerpt)

```
import { Directive, Input, TemplateRef, ViewContainerRef } from '@angular/core';

/**
 * Add the template content to the DOM unless the condition is true.
 */
@Directive({ selector: '[myUnless]' })
export class UnlessDirective {
  private hasView = false;

  constructor(
    private templateRef: TemplateRef<any>,
    private viewContainer: ViewContainerRef) { }

  @Input() set myUnless(condition: boolean) {
```

```
if (!condition && !this.hasView) {  
    this.viewContainer.createEmbeddedView(this.templateRef);  
    this.hasView = true;  
} else if (condition && this.hasView) {  
    this.viewContainer.clear();  
    this.hasView = false;  
}  
}  
}
```

Add this directive to the `declarations` array of the AppModule.

Then create some HTML to try it.

src/app/app.component.html (myUnless)

```
<p *myUnless="condition" class="unless a">  
    (A) This paragraph is displayed because the condition is false.  
</p>  
  
<p *myUnless="!condition" class="unless b">  
    (B) Although the condition is true,  
        this paragraph is displayed because myUnless is set to false.  
</p>
```



When the `condition` is falsy, the top (A) paragraph appears and the bottom (B) paragraph disappears.

When the `condition` is truthy, the top (A) paragraph is removed and the bottom (B) paragraph appears.

The condition is currently **false**. [Toggle condition to true](#)

(A) This paragraph is displayed because the condition is false.

Summary

You can both try and download the source code for this guide in the [live example](#) / [download example](#).

Here is the source from the `src/app/` folder.



`app.component.ts`

`app.component.html`

`app.component.css`

`app.module.ts`



```
1. import { Component } from '@angular/core';
2.
3. import { Hero, heroes } from './hero';
4.
5. @Component({
6.   selector: 'my-app',
7.   templateUrl: './app.component.html',
8.   styleUrls: [ './app.component.css' ]
9. })
10. export class AppComponent {
11.   heroes = heroes;
12.   hero = this.heroes[0];
13.
14.   condition = false;
```



```
15.  logs: string[] = [];
16.  showSad = true;
17.  status = 'ready';
18.
19.  trackById(index: number, hero: Hero): number { return hero.id; }
20. }
```

You learned

- that structural directives manipulate HTML layout.
- to use `<ng-container>` as a grouping element when there is no suitable host element.
- that the Angular desugars `asterisk (*) syntax` into a `<ng-template>`.
- how that works for the `NgIf`, `NgFor` and `NgSwitch` built-in directives.
- about the `microsyntax` that expands into a `<ng-template>`.
- to write a `custom structural directive`, `UnlessDirective`.

This is the archived documentation for Angular v4. Please visit [angular.io](#) to see documentation for the current version of Angular.

Pipes

[Contents >](#)

[Using pipes](#)

[Built-in pipes](#)

[Parameterizing a pipe](#)

•••

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

src/app/hero-birthday1.component.ts

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

@Component({
  selector: 'hero-birthday',
  template: `<p>The hero's birthday is {{ birthday | date }}</p>`
})
export class HeroBirthdayComponent {
  birthday = new Date(1988, 3, 15); // April 15, 1988
}
```



Focus on the component's template.

src/app/app.component.html

```
<p>The hero's birthday is {{ birthday | date }}</p>
```



Inside the interpolation expression, you flow the component's `birthday` value through the [pipe operator \(|\)](#) to the [Date pipe](#) function on the right. All pipes work this way.

The `Date` and `Currency` pipes need the [ECMAScript Internationalization API](#). Safari and other older browsers don't support it. You can add support with a polyfill.

```
<script src="https://cdn.polyfill.io/v2/polyfill.min.js?  
features=Intl~locale.en"></script>
```

Built-in pipes

Angular comes with a stock of pipes such as `DatePipe`, `UpperCasePipe`, `LowerCasePipe`, `CurrencyPipe`, and `PercentPipe`. They are all available for use in any template.

Read more about these and many other built-in pipes in the [pipes topics](#) of the [API Reference](#); filter for entries that include the word "pipe".

Angular doesn't have a `FilterPipe` or an `OrderByPipe` for reasons explained in the [Appendix](#) of this page.

Parameterizing a pipe

A pipe can accept any number of optional parameters to fine-tune its output. To add parameters to a pipe, follow the pipe name with a colon (:) and then the parameter value (such as `currency: 'EUR'`). If the pipe accepts multiple parameters, separate the values with colons (such as `slice:1:5`)

Modify the birthday template to give the date pipe a format parameter. After formatting the hero's April 15th birthday, it renders as 04/15/88:

src/app/app.component.html

```
<p>The hero's birthday is {{ birthday | date:"MM/dd/yy" }} </p>
```

The parameter value can be any valid template expression, (see the [Template expressions](#) section of the [Template Syntax](#) page) such as a string literal or a component property. In other words, you can control the format through a binding the same way you control the birthday value through a binding.

Write a second component that *binds* the pipe's format parameter to the component's `format` property. Here's the template for that component:

src/app/hero-birthday2.component.ts (template)

```
template: `
  <p>The hero's birthday is {{ birthday | date:format }}</p>
  <button (click)="toggleFormat()">Toggle Format</button>
`
```

You also added a button to the template and bound its click event to the component's `toggleFormat()` method. That method toggles the component's `format` property between a short form (`'shortDate'`) and a longer form (`'fullDate'`).

src/app/hero-birthday2.component.ts (class)

```
export class HeroBirthday2Component {
  birthday = new Date(1988, 3, 15); // April 15, 1988
  toggle = true; // start with true == shortDate

  get format() { return this.toggle ? 'shortDate' : 'fullDate'; }
  toggleFormat() { this.toggle = !this.toggle; }
}
```

As you click the button, the displayed date alternates between "04/15/1988" and "Friday, April 15, 1988".

The hero's birthday is 4/15/1988

[Toggle Format](#)

Read more about the `DatePipe` format options in the [Date Pipe API Reference](#) page.

Chaining pipes

You can chain pipes together in potentially useful combinations. In the following example, to display the birthday in uppercase, the birthday is chained to the `DatePipe` and on to the `UpperCasePipe`. The birthday displays as APR 15, 1988.

src/app/app.component.html

```
The chained hero's birthday is  
{{ birthday | date | uppercase}}
```

This example—which displays FRIDAY, APRIL 15, 1988—chains the same pipes as above, but passes in a parameter to `date` as well.

src/app/app.component.html

```
The chained hero's birthday is
```

```
 {{ birthday | date:'fullDate' | uppercase}}
```

Custom pipes

You can write your own custom pipes. Here's a custom pipe named `ExponentialStrengthPipe` that can boost a hero's powers:

src/app/exponential-strength.pipe.ts

```
import { Pipe, PipeTransform } from '@angular/core';
/*
 * Raise the value exponentially
 * Takes an exponent argument that defaults to 1.
 * Usage:
 *   value | exponentialStrength:exponent
 * Example:
 *   {{ 2 | exponentialStrength:10 }}
 *   formats to: 1024
 */
@Pipe({name: 'exponentialStrength'})
export class ExponentialStrengthPipe implements PipeTransform {
  transform(value: number, exponent: string): number {
    let exp = parseFloat(exponent);
    return Math.pow(value, isNaN(exp) ? 1 : exp);
  }
}
```



This pipe definition reveals the following key points:

- A pipe is a class decorated with pipe metadata.
- The pipe class implements the `PipeTransform` interface's `transform` method that accepts an input value followed by optional parameters and returns the transformed value.
- There will be one additional argument to the `transform` method for each parameter passed to the pipe. Your pipe has one such parameter: the `exponent`.
- To tell Angular that this is a pipe, you apply the `@Pipe` decorator, which you import from the core Angular library.
- The `@Pipe` decorator allows you to define the pipe name that you'll use within template expressions. It must be a valid JavaScript identifier. Your pipe's name is `exponentialStrength`.

The `PipeTransform` interface

The `transform` method is essential to a pipe. The `PipeTransform` *interface* defines that method and guides both tooling and the compiler. Technically, it's optional; Angular looks for and executes the `transform` method regardless.

Now you need a component to demonstrate the pipe.

src/app/power-booster.component.ts

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

@Component({
  selector: 'power-booster',
  template: `
    <h2>Power Booster</h2>
    <p>Super power boost: {{2 | exponentialStrength: 10}}</p>
  `
```

```
)  
export class PowerBoosterComponent {}
```

Power Booster

Super power boost: 1024

Note the following:

- You use your custom pipe the same way you use built-in pipes.
- You must include your pipe in the `declarations` array of the `AppModule`.

REMEMBER THE DECLARATIONS ARRAY

You must manually register custom pipes. If you don't, Angular reports an error. In the previous example, you didn't list the `DatePipe` because all Angular built-in pipes are pre-registered.

To probe the behavior in the [live example](#) / [download example](#), change the value and optional exponent in the template.

Power Boost Calculator

It's not much fun updating the template to test the custom pipe. Upgrade the example to a "Power Boost Calculator" that combines your pipe and two-way data binding with `ngModel`.

```
1. import { Component } from '@angular/core';
2.
3. @Component({
4.   selector: 'power-boost-calculator',
5.   template: `
6.     <h2>Power Boost Calculator</h2>
7.     <div>Normal power: <input [(ngModel)]="power"></div>
8.     <div>Boost factor: <input [(ngModel)]="factor"></div>
9.     <p>
10.       Super Hero Power: {{power | exponentialStrength: factor}}
11.     </p>
12.   `,
13. })
14. export class PowerBoostCalculatorComponent {
15.   power = 5;
16.   factor = 1;
17. }
```



Power Boost Calculator

Normal power: 5

Boost factor: 1

Super Hero Power: 5

Pipes and change detection

Angular looks for changes to data-bound values through a *change detection* process that runs after every DOM event: every keystroke, mouse move, timer tick, and server response. This could be expensive. Angular strives to lower the cost whenever possible and appropriate.

Angular picks a simpler, faster change detection algorithm when you use a pipe.

No pipe

In the next example, the component uses the default, aggressive change detection strategy to monitor and update its display of every hero in the `heroes` array. Here's the template:

src/app/flying-heroes.component.html (v1)

```
New hero:  
<input type="text" #box  
       (keyup.enter)="addHero(box.value); box.value=''"  
       placeholder="hero name">  
<button (click)="reset()">Reset</button>  
<div *ngFor="let hero of heroes">  
  {{hero.name}}  
</div>
```

The companion component class provides heroes, adds heroes into the array, and can reset the array.

src/app/flying-heroes.component.ts (v1)

```
export class FlyingHeroesComponent {  
  heroes: any[] = [];
```

```
canFly = true;  
constructor() { this.reset(); }  
  
addHero(name: string) {  
  name = name.trim();  
  if (!name) { return; }  
  let hero = {name, canFly: this.canFly};  
  this.heroes.push(hero);  
}  
  
reset() { this.heroes = HEROES.slice(); }  
}
```

You can add heroes and Angular updates the display when you do. If you click the `reset` button, Angular replaces `heroes` with a new array of the original heroes and updates the display. If you added the ability to remove or change a hero, Angular would detect those changes and update the display as well.

FlyingHeroesPipe

Add a `FlyingHeroesPipe` to the `*ngFor` repeater that filters the list of heroes to just those heroes who can fly.

src/app/flying-heroes.component.html (flyers)

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



Here's the `FlyingHeroesPipe` implementation, which follows the pattern for custom pipes described earlier.

src/app/flying-heroes.pipe.ts

```
import { Pipe, PipeTransform } from '@angular/core';

import { Flyer } from './heroes';

@Pipe({ name: 'flyingHeroes' })
export class FlyingHeroesPipe implements PipeTransform {
  transform(allHeroes: Flyer[]) {
    return allHeroes.filter(hero => hero.canFly);
  }
}
```



Notice the odd behavior in the [live example](#) / [download example](#) : when you add flying heroes, none of them are displayed under "Heroes who fly."

Although you're not getting the behavior you want, Angular isn't broken. It's just using a different change-detection algorithm that ignores changes to the list or any of its items.

Notice how a hero is added:

src/app/flying-heroes.component.ts

```
this.heroes.push(hero);
```



You add the hero into the `heroes` array. The reference to the array hasn't changed. It's the same array. That's all Angular cares about. From its perspective, *same array, no change, no display update*.

To fix that, create an array with the new hero appended and assign that to `heroes`. This time Angular detects that the array reference has changed. It executes the pipe and updates the display with the new array, which includes the new flying hero.

If you *mutate* the array, no pipe is invoked and the display isn't updated; if you *replace* the array, the pipe executes and the display is updated. The Flying Heroes application extends the code with checkbox switches and additional displays to help you experience these effects.

Flying Heroes

New flying hero: can fly

Mutate array

Heroes who fly (piped)

Windstorm
Tornado

All Heroes (no pipe)

Windstorm
Bombasto
Magneto
Tornado

Replacing the array is an efficient way to signal Angular to update the display. When do you replace the array? When the data change. That's an easy rule to follow in *this* example where the only way to change the data is by adding a hero.

More often, you don't know when the data have changed, especially in applications that mutate data in many ways, perhaps in application locations far away. A component in such an application usually can't

know about those changes. Moreover, it's unwise to distort the component design to accommodate a pipe. Strive to keep the component class independent of the HTML. The component should be unaware of pipes.

For filtering flying heroes, consider an *impure pipe*.

Pure and impure pipes

There are two categories of pipes: *pure* and *impure*. Pipes are pure by default. Every pipe you've seen so far has been pure. You make a pipe impure by setting its pure flag to false. You could make the `FlyingHeroesPipe` impure like this:

src/app/flying-heroes.pipe.ts

```
@Pipe({
  name: 'flyingHeroesImpure',
  pure: false
})
```

Before doing that, understand the difference between pure and impure, starting with a pure pipe.

Pure pipes

Angular executes a *pure pipe* only when it detects a *pure change* to the input value. A pure change is either a change to a primitive input value (`String`, `Number`, `Boolean`, `Symbol`) or a changed object reference (`Date`, `Array`, `Function`, `Object`).

Angular ignores changes within (composite) objects. It won't call a pure pipe if you change an input month, add to an input array, or update an input object property.

This may seem restrictive but it's also fast. An object reference check is fast—much faster than a deep check for differences—so Angular can quickly determine if it can skip both the pipe execution and a view update.

For this reason, a pure pipe is preferable when you can live with the change detection strategy. When you can't, you *can* use the impure pipe.

Or you might not use a pipe at all. It may be better to pursue the pipe's purpose with a property of the component, a point that's discussed later in this page.

Impure pipes

Angular executes an *impure pipe* during every component change detection cycle. An impure pipe is called often, as often as every keystroke or mouse-move.

With that concern in mind, implement an impure pipe with great care. An expensive, long-running pipe could destroy the user experience.

An impure *FlyingHeroesPipe*

A flip of the switch turns the `FlyingHeroesPipe` into a `FlyingHeroesImpurePipe`. The complete implementation is as follows:

`FlyingHeroesImpurePipe` `FlyingHeroesPipe`

```
@Pipe({  
  name: 'flyingHeroesImpure',  
})
```

```
    pure: false
  })
export class FlyingHeroesImpurePipe extends FlyingHeroesPipe {}
```

You inherit from `FlyingHeroesPipe` to prove the point that nothing changed internally. The only difference is the `pure` flag in the pipe metadata.

This is a good candidate for an impure pipe because the `transform` function is trivial and fast.

src/app/flying-heroes.pipe.ts (filter)

```
return allHeroes.filter(hero => hero.canFly);
```

You can derive a `FlyingHeroesImpureComponent` from `FlyingHeroesComponent`.

src/app/flying-heroes-impure.component.html (excerpt)

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

The only substantive change is the pipe in the template. You can confirm in the [live example / download example](#) that the *flying heroes* display updates as you add heroes, even when you mutate the `heroes` array.

The impure *AsyncPipe*

The Angular `AsyncPipe` is an interesting example of an impure pipe. The `AsyncPipe` accepts a `Promise` or `Observable` as input and subscribes to the input automatically, eventually returning the emitted values.

The `AsyncPipe` is also stateful. The pipe maintains a subscription to the input `Observable` and keeps delivering values from that `Observable` as they arrive.

This next example binds an `Observable` of message strings (`message$`) to a view with the `async` pipe.

src/app/hero-async-message.component.ts

```
1. import { Component } from '@angular/core';
2.
3. import { Observable } from 'rxjs/Observable';
4. import 'rxjs/add/observable/interval';
5. import 'rxjs/add/operator/map';
6. import 'rxjs/add/operator/take';
7.
8. @Component({
9.   selector: 'hero-message',
10.  template: `
11.    <h2>Async Hero Message and AsyncPipe</h2>
12.    <p>Message: {{ message$ | async }}</p>
13.    <button (click)="resend()">Resend</button>`,
14. })
15. export class HeroAsyncMessageComponent {
16.   message$: Observable<string>;
17.
18.   private messages = [
19.     'You are my hero!',
20.     'You are the best hero!',
21.     'Will you be my hero?'
22.   ];
23.
24.   constructor() { this.resend(); }
```

```
25.  
26. resend() {  
27.   this.message$ = Observable.interval(500)  
28.     .map(i => this.messages[i])  
29.     .take(this.messages.length);  
30.   }  
31. }
```

The Async pipe saves boilerplate in the component code. The component doesn't have to subscribe to the async data source, extract the resolved values and expose them for binding, and have to unsubscribe when it's destroyed (a potent source of memory leaks).

An impure caching pipe

Write one more impure pipe, a pipe that makes an HTTP request.

Remember that impure pipes are called every few milliseconds. If you're not careful, this pipe will punish the server with requests.

In the following code, the pipe only calls the server when the request URL changes and it caches the server response. The code uses the [Angular http](#) client to retrieve data:

src/app/fetch-json.pipe.ts

```
1. import { Pipe, PipeTransform } from '@angular/core';  
2. import { Http } from '@angular/http';  
3.  
4. import 'rxjs/add/operator/map';  
5.  
6. @Pipe({  
7.   name: 'fetch',
```

```
8.    pure: false
9. })
10. export class FetchJsonPipe implements PipeTransform {
11.   private cachedData: any = null;
12.   private cachedUrl = '';
13.
14.   constructor(private http: Http) { }
15.
16.   transform(url: string): any {
17.     if (url !== this.cachedUrl) {
18.       this.cachedData = null;
19.       this.cachedUrl = url;
20.       this.http.get(url)
21.         .map( result => result.json() )
22.         .subscribe( result => this.cachedData = result );
23.     }
24.
25.     return this.cachedData;
26.   }
27. }
```

Now demonstrate it in a harness component whose template defines two bindings to this pipe, both requesting the heroes from the `heroes.json` file.

src/app/hero-list.component.ts

```
1. import { Component } from '@angular/core';
2.
3. @Component({
4.   selector: 'hero-list',
```

```
5.  template: ` 
6.    <h2>Heroes from JSON File</h2>
7.
8.    <div *ngFor="let hero of ('heroes.json' | fetch) ">
9.      {{hero.name}}
10.     </div>
11.
12.    <p>Heroes as JSON:
13.      {{'heroes.json' | fetch | json}}
14.    </p>
15.  )
16. export class HeroListComponent { }
```

The component renders as the following:

Heroes from JSON File

Windstorm
Bombasto
Magneto
Tornado

Heroes as JSON: [{ "name": "Windstorm" }, {
 "name": "Bombasto" }, { "name": "Magneto" }, {
 "name": "Tornado" }]

A breakpoint on the pipe's request for data shows the following:

- Each binding gets its own pipe instance.

- Each pipe instance caches its own URL and data.
- Each pipe instance only calls the server once.

JsonPipe

In the previous code sample, the second `fetch` pipe binding demonstrates more pipe chaining. It displays the same hero data in JSON format by chaining through to the built-in `JsonPipe`.

DEBUGGING WITH THE JSON PIPE

The `JsonPipe` provides an easy way to diagnosis a mysteriously failing data binding or inspect an object for future binding.

Pure pipes and pure functions

A pure pipe uses pure functions. Pure functions process inputs and return values without detectable side effects. Given the same input, they should always return the same output.

The pipes discussed earlier in this page are implemented with pure functions. The built-in `DatePipe` is a pure pipe with a pure function implementation. So are the `ExponentialStrengthPipe` and `FlyingHeroesPipe`. A few steps back, you reviewed the `FlyingHeroesImpurePipe` —an impure pipe with a pure function.

But always implement a *pure pipe* with a *pure function*. Otherwise, you'll see many console errors regarding expressions that changed after they were checked.

Next steps

Pipes are a great way to encapsulate and share common display-value transformations. Use them like styles, dropping them into your template's expressions to enrich the appeal and usability of your views.

Explore Angular's inventory of built-in pipes in the [API Reference](#). Try writing a custom pipe and perhaps contributing it to the community.

Appendix: No *FilterPipe* or *OrderByPipe*

Angular doesn't provide pipes for filtering or sorting lists. Developers familiar with AngularJS know these as `filter` and `orderBy`. There are no equivalents in Angular.

This isn't an oversight. Angular doesn't offer such pipes because they perform poorly and prevent aggressive minification. Both `filter` and `orderBy` require parameters that reference object properties. Earlier in this page, you learned that such pipes must be [impure](#) and that Angular calls impure pipes in almost every change-detection cycle.

Filtering and especially sorting are expensive operations. The user experience can degrade severely for even moderate-sized lists when Angular calls these pipe methods many times per second. `filter` and `orderBy` have often been abused in AngularJS apps, leading to complaints that Angular itself is slow. That charge is fair in the indirect sense that AngularJS prepared this performance trap by offering `filter` and `orderBy` in the first place.

The minification hazard is also compelling, if less obvious. Imagine a sorting pipe applied to a list of heroes. The list might be sorted by hero `name` and `planet` of origin properties in the following way:

```
<!-- NOT REAL CODE! -->  
<div *ngFor="let hero of heroes | orderBy:'name,planet'"></div>
```

You identify the sort fields by text strings, expecting the pipe to reference a property value by indexing (such as `hero['name']`). Unfortunately, aggressive minification manipulates the `Hero` property names so that

`Hero.name` and `Hero.planet` become something like `Hero.a` and `Hero.b`. Clearly `hero['name']` doesn't work.

While some may not care to minify this aggressively, the Angular product shouldn't prevent anyone from minifying aggressively. Therefore, the Angular team decided that everything Angular provides will minify safely.

The Angular team and many experienced Angular developers strongly recommend moving filtering and sorting logic into the component itself. The component can expose a `filteredHeroes` or `sortedHeroes` property and take control over when and how often to execute the supporting logic. Any capabilities that you would have put in a pipe and shared across the app can be written in a filtering/sorting service and injected into the component.

If these performance and minification considerations don't apply to you, you can always create your own such pipes (similar to the [FlyingHeroesPipe](#)) or find them in the community.

This is the archived documentation for Angular v4. Please visit [angular.io](#) to see documentation for the current version of Angular.

Animations

[Contents >](#)

[Overview](#)

[Setup](#)

[Transitioning between two states](#)

•••

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](#) and run natively on [browsers that support it](#).

For other browsers, a polyfill is required. Grab [web-animations.min.js](#) from GitHub and add it to your page.

The examples in this page are available as a [live example](#) / [download example](#).

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.

app.module.ts (animation module import excerpt)

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

@NgModule({
  imports: [ BrowserModule, BrowserAnimationsModule ],
  // ... more stuff ...
})
export class AppModule { }
```



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.

hero.service.ts (Hero class)

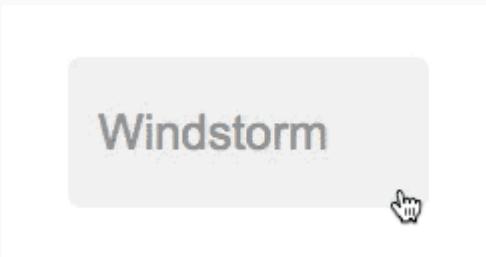
```
export class Hero {  
  constructor(public name: string, public state = 'inactive') {}  
  
  toggleState() {  
    this.state = this.state === 'active' ? 'inactive' : 'active';  
  }  
}
```

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.

Transitioning between two states

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.



hero-list-basic.component.ts

```
import {  
  Component,  
}
```

```
    Input
} from '@angular/core';
import {
  trigger,
  state,
  style,
  animate,
  transition
} from '@angular/animations';
```

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.

hero-list-basic.component.ts (@Component excerpt)

```
  animations: [
    trigger('heroState', [
      state('inactive', style({
        backgroundColor: '#eee',
        transform: 'scale(1)'
      })),
      state('active',   style({
        backgroundColor: '#cf8dc',
        transform: 'scale(1.1)'
      })),
      transition('inactive => active', animate('100ms ease-in')),
      transition('active => inactive', animate('100ms ease-out'))
    ])
]
```

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.

hero-list-basic.component.ts (excerpt)

```
template: `

<ul>
  <li *ngFor="let hero of heroes"
      [@heroState]="hero.state"
      (click)="hero.toggleState()"
      {{hero.name}}
  </li>
</ul>
`
```

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:

hero-list-basic.component.ts

```
1. import {
```

```
2.  Component,
3.  Input
4. } from '@angular/core';
5. import {
6.   trigger,
7.   state,
8.   style,
9.   animate,
10.  transition
11. } from '@angular/animations';
12.
13. import { Hero } from './hero.service';
14.
15. @Component({
16.   selector: 'hero-list-basic',
17.   template: `
18.     <ul>
19.       <li *ngFor="let hero of heroes"
20.           [@heroState]="hero.state"
21.           (click)="hero.toggleState()">
22.             {{hero.name}}
23.           </li>
24.         </ul>
25.       `,
26.   styleUrls: ['./hero-list.component.css'],
27.   animations: [
28.     trigger('heroState', [
29.       state('inactive', style({
30.         backgroundColor: '#eee',
31.         transform: 'scale(1)'
```

```
32.     })),  
33.     state('active', style({  
34.       backgroundColor: '#cf8dc',  
35.       transform: 'scale(1.1)'  
36.     })),  
37.     transition('inactive => active', animate('100ms ease-in')),  
38.     transition('active => inactive', animate('100ms ease-out'))  
39.   ]  
40. )  
41. )  
42. export class HeroListBasicComponent {  
43.   @Input() heroes: Hero[];  
44. }
```

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:

src/app/hero-list-basic.component.ts

```
state('inactive', style({  
  backgroundColor: '#eee',  
  transform: 'scale(1)'
```

```
    }),  
    state('active', style({  
      backgroundColor: '#cf8dc',  
      transform: 'scale(1.1)'  
    }),
```

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.

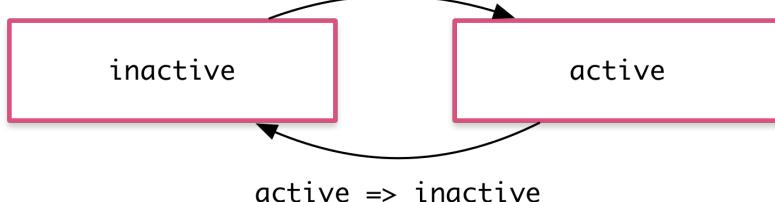
After you define states, you can define *transitions* between the states. Each transition controls the timing of switching between one set of styles and the next:

src/app/hero-list-basic.component.ts

```
transition('inactive => active', animate('100ms ease-in')),  
transition('active => inactive', animate('100ms ease-out'))
```



inactive => active



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

src/app/hero-list-combined-transitions.component.ts

```
transition('inactive => active, active => inactive',
  animate('100ms ease-out'))
```



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

src/app/hero-list-twoway.component.ts

```
transition('inactive <=> active', animate('100ms ease-out'))
```



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`.

src/app/hero-list-inline-styles.component.ts

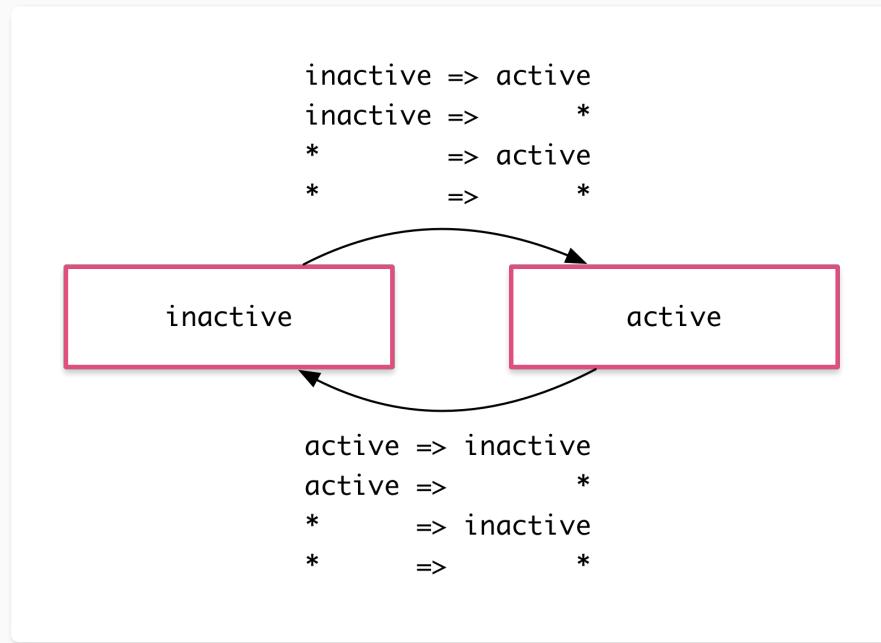
```
transition('inactive => active', [
  style({
    backgroundColor: '#cf8dc',
    transform: 'scale(1.3)'
  }),
  animate('80ms ease-in', style({
    backgroundColor: '#eee',
    transform: 'scale(1)'
  }))
],
```



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:

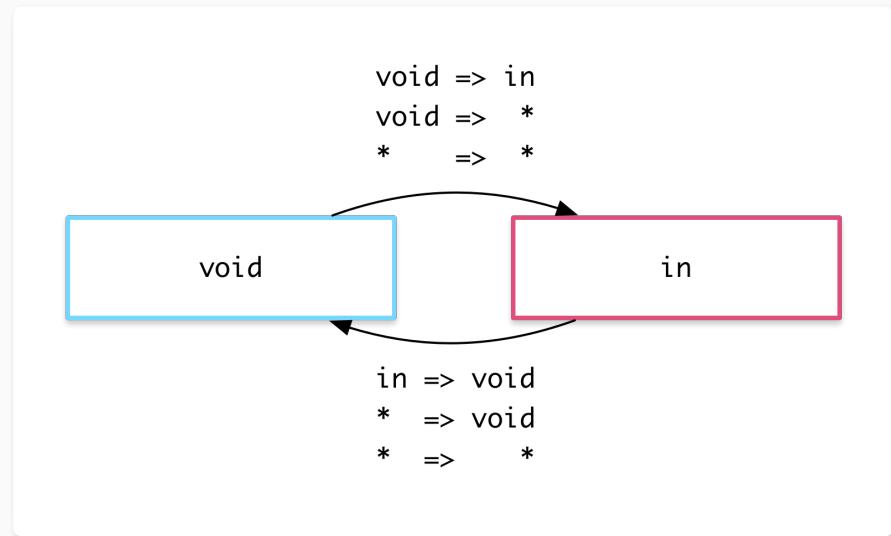
- The `active => *` transition applies when the element's state changes from `active` to anything else.
- The `* => *` transition applies when *any* change between two states takes place.



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.

hero-list-enter-leave.component.ts (excerpt)

```
animations: [
  trigger('flyInOut', [
    state('in', style({transform: 'translateX(0)'})),
    transition('void => *', [
      style({transform: 'translateX(-100%)'}),
      animate(100)
    ]),
    transition('* => void', [
      animate(100, style({transform: 'translateX(100%)'}))
    ])
  ])
]
```

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
```

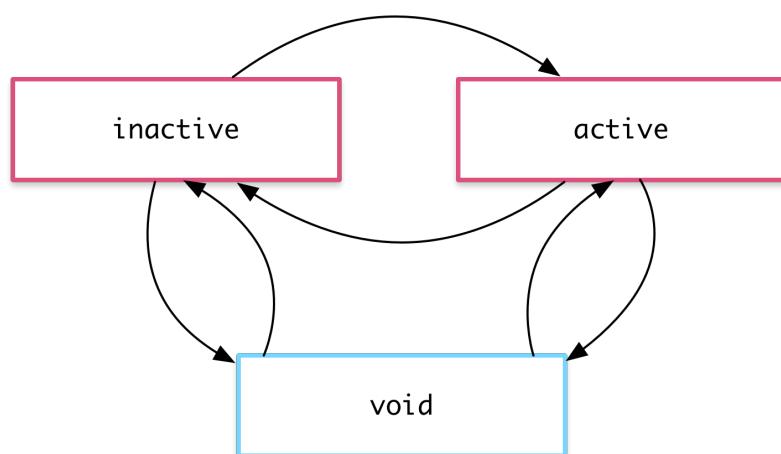
Example: Entering and leaving from different states

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:



hero-list-enter-leave.component.ts (excerpt)

```
animations: [
  trigger('heroState', [
    state('inactive', style({transform: 'translateX(0) scale(1)' })),
    state('active', style({transform: 'translateX(0) scale(1.1)' })),
    transition('inactive => active', animate('100ms ease-in')),
    transition('active => inactive', animate('100ms ease-out')),
    transition('void => inactive', [
      style({transform: 'translateX(-100%) scale(1)'})
    ])
  ])
]
```

```
    animate(100)
  ]),
  transition('inactive => void', [
    animate(100, style({transform: 'translateX(100%) scale(1)'})
  ]),
  transition('void => active', [
    style({transform: 'translateX(0) scale(0)'}),
    animate(200)
  ]),
  transition('active => void', [
    animate(200, style({transform: 'translateX(0) scale(0)'}))
  ])
])
]
```

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:

src/app/hero-list-auto.component.ts

```
animations: [
  trigger('shrinkOut', [
    state('in', style({height: '*' })),
    transition('* => void', [
      style({height: '*' }),
      animate(250, style({height: 0}))
    ])
  ])
]
```



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'`:

hero-list-timings.component.ts (excerpt)

```
animations: [
  trigger('flyInOut', [
    state('in', style({opacity: 1, transform: 'translateX(0)'})),
    transition('void => *', [
      style({
        opacity: 0,
        transform: 'translateX(-100%)'
      }),
      animate('0.2s ease-in')
    ]),
    transition('* => void', [
      animate('0.2s 0.1s ease-out', style({
        opacity: 0,
        transform: 'translateX(100%)'
      }))
    ])
]
```

```
    ]  
  ]  
]
```

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:

hero-list-multistep.component.ts (excerpt)

```
animations: [  
  trigger('flyInOut', [  
    state('in', style({transform: 'translateX(0')})),  
    transition('void => *', [  
      animate(300, keyframes [  
        style({opacity: 0, transform: 'translateX(-100%)', offset: 0}),  
        style({opacity: 1, transform: 'translateX(15px)', offset: 0.3}),  
        style({opacity: 1, transform: 'translateX(0)', offset: 1.0})  
      ])  
    ]),  
    transition('* => void', [  
    ])
```

```
    animate(300, keyframes([
      style({opacity: 1, transform: 'translateX(0)', offset: 0}),
      style({opacity: 1, transform: 'translateX(-15px)', offset: 0.7}),
      style({opacity: 0, transform: 'translateX(100%)', offset: 1.0})
    ]))
  ]
]
```

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:

hero-list-groups.component.ts (excerpt)



```
animations: [
  trigger('flyInOut', [
    state('in', style({width: 120, transform: 'translateX(0)', opacity: 1})),
    transition('void => *', [
      style({width: 10, transform: 'translateX(50px)', opacity: 0}),
      group([
        animate('0.3s 0.1s ease', style({
          transform: 'translateX(0)',
          width: 120
        })),
        animate('0.3s ease', style({
          opacity: 1
        }))
      ])
    ]),
    transition('* => void', [
      group([
        animate('0.3s ease', style({
          transform: 'translateX(50px)',
          width: 10
        })),
        animate('0.3s 0.2s ease', style({
          opacity: 0
        }))
      ])
    ])
  ])
]
```

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:

hero-list-multistep.component.ts (excerpt)

```
template: `

<ul>
  <li *ngFor="let hero of heroes"
      (@flyInOut.start)="animationStarted($event)"
      (@flyInOut.done)="animationDone($event)"
      [@flyInOut]="'in'"
      {{hero.name}}
    </li>
  </ul>
`,
```



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.

This is the archived documentation for Angular v4. Please visit [angular.io](#) to see documentation for the current version of Angular.

User Input

[Contents >](#)

[Binding to user input events](#)

[Get user input from the \\$event object](#)

[Type the \\$event](#)

•••

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

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

Binding to user input events

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

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

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

src/app/click-me.component.ts

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



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

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

src/app/click-me.component.ts

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

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



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

Get user input from the \$event object

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

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

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

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

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

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

```
export class KeyUpComponent_v1 {
  values = '';

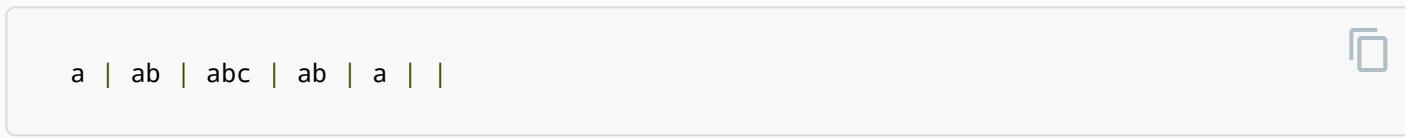
  onKey(event: any) { // without type info
    this.values += event.target.value + ' | ';
  }
}
```

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

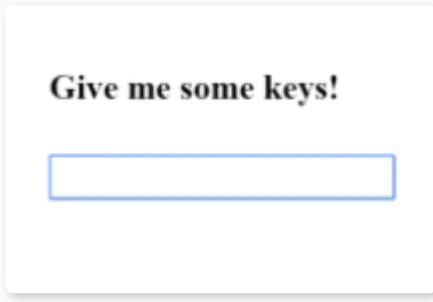
All [standard DOM event objects](#) have a `target` property, a reference to the element that raised the event. In this case, `target` refers to the `<input>` element and `event.target.value` returns the current contents of that element.

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:

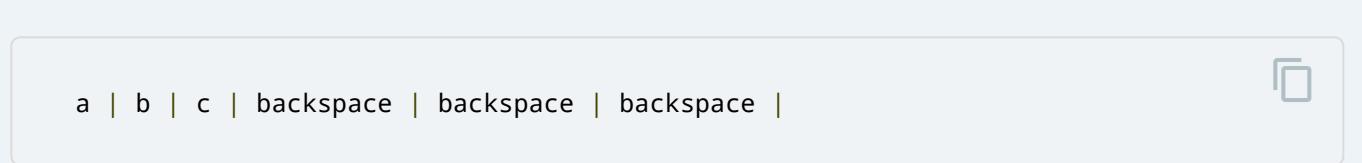


a | ab | abc | ab | a | |



Give me some keys!

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



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

Type the `$event`

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

The following example rewrites the method with types:

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

```
export class KeyUpComponent_v1 {
  values = '';

  onKey(event: KeyboardEvent) { // with type info
    this.values += (<HTMLInputElement>event.target).value + ' | ';
  }
}
```

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

Passing `$event` is a dubious practice

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

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

Get user input from a template reference variable

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

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

src/app/loop-back.component.ts

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

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

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

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

keyup loop-back component

↖

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

Angular updates the bindings (and therefore the screen) only if the app does something in response to asynchronous events, such as keystrokes. This example code binds the `keyup` event to the number 0, the shortest template statement possible. While the statement does nothing useful, it satisfies Angular's requirement so that Angular will update the screen.

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

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

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

```
onKey(value: string) {
  this.values += value + ' | ';
}
}
```

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

Key event filtering (with key.enter)

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

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

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

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

Here's how it works.

Type away! Press [enter] when done

On blur

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

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

src/app/keyup.components.ts (v4)

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

    <p>{{value}}</p>
  `})

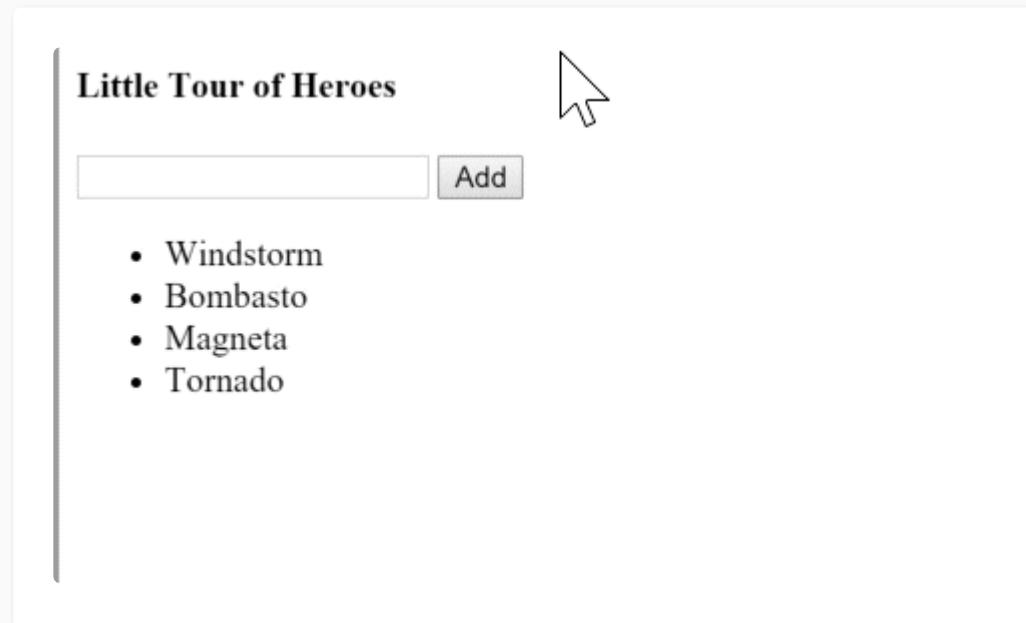
```

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

Put it all together

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

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



Below is the "Little Tour of Heroes" component.

src/app/little-tour.component.ts

```

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

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

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

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

```

Observations

- Use template variables to refer to elements – The `newHero` template variable refers to the `<input>` element. You can reference `newHero` from any sibling or child of the `<input>` element.
- Pass values, not elements – Instead of passing the `newHero` into the component's `addHero` method, get the input box value and pass *that* to `addHero`.
- Keep template statements simple – The `(blur)` event is bound to two JavaScript statements. The first statement calls `addHero`. The second statement, `newHero.value=' '`, clears the input box after a

new hero is added to the list.

Source code

Following is all the code discussed in this page.

[click-me.component.ts](#) [keyup.components.ts](#) [loop-back.component.ts](#) [little-tour.component.ts](#)

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



Summary

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

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

This is the archived documentation for Angular v4. Please visit [angular.io](#) to see documentation for the current version of Angular.

Forms

[Contents >](#)

[Template-driven forms](#)

[Setup](#)

[Create the Hero model class](#)

•••

Forms are the mainstay of business applications. You use forms to log in, submit a help request, place an order, book a flight, schedule a meeting, and perform countless other data-entry tasks.

In developing a form, it's important to create a data-entry experience that guides the user efficiently and effectively through the workflow.

Developing forms requires design skills (which are out of scope for this page), as well as framework support for *two-way data binding, change tracking, validation, and error handling*, which you'll learn about on this page.

This page shows you how to build a simple form from scratch. Along the way you'll learn how to:

- Build an Angular form with a component and template.
- Use `ngModel` to create two-way data bindings for reading and writing input-control values.
- Track state changes and the validity of form controls.
- Provide visual feedback using special CSS classes that track the state of the controls.

- Display validation errors to users and enable/disable form controls.
- Share information across HTML elements using template reference variables.

You can run the [live example](#) / [download example](#) in Plunker and download the code from there.

Template-driven forms

You can build forms by writing templates in the Angular [template syntax](#) with the form-specific directives and techniques described in this page.

You can also use a reactive (or model-driven) approach to build forms. However, this page focuses on template-driven forms.

You can build almost any form with an Angular template—login forms, contact forms, and pretty much any business form. You can lay out the controls creatively, bind them to data, specify validation rules and display validation errors, conditionally enable or disable specific controls, trigger built-in visual feedback, and much more.

Angular makes the process easy by handling many of the repetitive, boilerplate tasks you'd otherwise wrestle with yourself.

You'll learn to build a template-driven form that looks like this:

Hero Form

Name

Dr IQ

Alter Ego

Chuck Overstreet

Hero Power

Really Smart

Submit

The *Hero Employment Agency* uses this form to maintain personal information about heroes. Every hero needs a job. It's the company mission to match the right hero with the right crisis.

Two of the three fields on this form are required. Required fields have a green bar on the left to make them easy to spot.

If you delete the hero name, the form displays a validation error in an attention-grabbing style:

Hero Form

Name

Name is required

Alter Ego

Hero Power

Submit

Note that the *Submit* button is disabled, and the "required" bar to the left of the input control changes from green to red.

You can customize the colors and location of the "required" bar with standard CSS.

You'll build this form in small steps:

1. Create the `Hero` model class.
2. Create the component that controls the form.

3. Create a template with the initial form layout.
4. Bind data properties to each form control using the `ngModel` two-way data-binding syntax.
5. Add a `name` attribute to each form-input control.
6. Add custom CSS to provide visual feedback.
7. Show and hide validation-error messages.
8. Handle form submission with `ngSubmit`.
9. Disable the form's `Submit` button until the form is valid.

Setup

Follow the [setup](#) instructions for creating a new project named 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`).

In the `app` directory, create the following file with the given content:

src/app/hero.ts

```
export class Hero {  
  
  constructor(  
    public id: number,  
    public name: string,  
    public power: string,  
    public alterEgo: string  
  ) {}  
  
  static readonly idCounter = 0;  
}
```

```
    public name: string,  
    public power: string,  
    public alterEgo?: string  
  ) {  
  
}  
}
```

It's an anemic model with few requirements and no behavior. Perfect for the demo.

The TypeScript compiler generates a public field for each `public` constructor parameter and automatically assigns the parameter's value to that field when you create heroes.

The `alterEgo` is optional, so the constructor lets you omit it; note the question mark (?) in `alterEgo?`.

You can create a new hero like this:

src/app/hero-form.component.ts (SkyDog)

```
let myHero = new Hero(42, 'SkyDog',  
                      'Fetch any object at any distance',  
                      'Leslie Rollover');  
console.log('My hero is called ' + myHero.name); // "My hero is called SkyDog"
```



Create a form component

An Angular form has two parts: an HTML-based *template* and a component *class* to handle data and user interactions programmatically. Begin with the class because it states, in brief, what the hero editor can do.

Create the following file with the given content:

src/app/hero-form.component.ts (v1)

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

@Component({
  selector: 'hero-form',
  templateUrl: './hero-form.component.html'
})
export class HeroFormComponent {

  powers = ['Really Smart', 'Super Flexible',
            'Super Hot', 'Weather Changer'];

  model = new Hero(18, 'Dr IQ', this.powers[0], 'Chuck Overstreet');

  submitted = false;

  onSubmit() { this.submitted = true; }

  // TODO: Remove this when we're done
  get diagnostic() { return JSON.stringify(this.model); }
}
```



There's nothing special about this component, nothing form-specific, nothing to distinguish it from any component you've written before.

Understanding this component requires only the Angular concepts covered in previous pages.

- The code imports the Angular core library and the `Hero` model you just created.

- The `@Component` selector value of "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.

Why the separate template file?

Why don't you write the template inline in the component file as you often do elsewhere?

There is no "right" answer for all occasions. Inline templates are useful when they are short. Most form templates aren't short. TypeScript and JavaScript files generally aren't the best place to write (or read) large stretches of HTML, and few editors help with files that have a mix of HTML and code.

Form templates tend to be large, even when displaying a small number of fields, so it's usually best to put the HTML template in a separate file. You'll write that template file in a moment. First, revise the `app.module.ts` and `app.component.ts` to make use of the new `HeroFormComponent`.

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.

Replace the contents of the "QuickStart" version with the following:

src/app/app.module.ts

```
1. import { NgModule }      from '@angular/core';
2. import { BrowserModule } from '@angular/platform-browser';
3. import { FormsModule }   from '@angular/forms';
4.
5. import { AppComponent }  from './app.component';
6. import { HeroFormComponent } from './hero-form.component';
7.
8. @NgModule({
9.   imports: [
10.     BrowserModule,
11.     FormsModule
12.   ],
13.   declarations: [
14.     AppComponent,
15.     HeroFormComponent
16.   ],
17.   bootstrap: [ AppComponent ]
18. })
19. export class AppModule { }
```



There are three changes:

1. You import `FormsModule` and the new `HeroFormComponent`.
2. You add the `FormsModule` to the list of `imports` defined in the `@NgModule` decorator. This gives the application access to all of the template-driven forms features, including `ngModel`.

3. You add the `HeroFormComponent` to the list of declarations defined in the `@NgModule` decorator. This makes the `HeroFormComponent` component visible throughout this module.

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.ts`

`AppComponent` is the application's root component. It will host the new `HeroFormComponent`.

Replace the contents of the "QuickStart" version with the following:

src/app/app.component.ts

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

@Component({
  selector: 'my-app',
  template: '<hero-form></hero-form>'
})
export class AppComponent { }
```

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. You've also dropped the `name` field from the class body.

Create an initial HTML form template

Create the template file with the following contents:

src/app/hero-form.component.html

```
1. <div class="container">
2.   <h1>Hero Form</h1>
3.   <form>
4.     <div class="form-group">
5.       <label for="name">Name</label>
6.       <input type="text" class="form-control" id="name" required>
7.     </div>
8.
9.     <div class="form-group">
10.      <label for="alterEgo">Alter Ego</label>
11.      <input type="text" class="form-control" id="alterEgo">
12.    </div>
13.
14.    <button type="submit" class="btn btn-success">Submit</button>
15.
16.  </form>
17. </div>
```



The language is simply HTML5. You're presenting two of the `Hero` fields, `name` and `alterEgo`, and opening them up for user input in input boxes.

The `Name` `<input>` control has the HTML5 `required` attribute; the `Alter Ego` `<input>` control does not because `alterEgo` is optional.

You added a `Submit` button at the bottom with some classes on it for styling.

You're not using Angular yet. There are no bindings or extra directives, just layout.

In template driven forms, if you've imported `FormsModule`, you don't have to do anything to the `<form>` tag in order to make use of `FormsModule`. Continue on to see how this works.

The `container`, `form-group`, `form-control`, and `btn` classes come from [Twitter Bootstrap](#). These classes are purely cosmetic. Bootstrap gives the form a little style.

ANGULAR FORMS DON'T REQUIRE A STYLE LIBRARY

Angular makes no use of the `container`, `form-group`, `form-control`, and `btn` classes or the styles of any external library. Angular apps can use any CSS library or none at all.

To add the stylesheet, open `index.html` and add the following link to the `<head>`:

src/index.html (bootstrap)

```
<link rel="stylesheet"
      href="https://unpkg.com/bootstrap@3.3.7/dist/css/bootstrap.min.css">
```

Add powers with `*ngFor`

The hero must choose one superpower from a fixed list of agency-approved powers. You maintain that list internally (in `HeroFormComponent`).

You'll add a `select` to the form and bind the options to the `powers` list using `ngFor`, a technique seen previously in the [Displaying Data](#) page.

Add the following HTML *immediately below* the *Alter Ego* group:

src/app/hero-form.component.html (powers)

```
<div class="form-group">
  <label for="power">Hero Power</label>
  <select class="form-control" id="power" required>
    <option *ngFor="let pow of powers" [value]="pow">{{pow}}</option>
  </select>
</div>
```

This code repeats the `<option>` tag for each power in the list of powers. The `pow` template input variable is a different power in each iteration; you display its name using the interpolation syntax.

Two-way data binding with *ngModel*

Running the app right now would be disappointing.

Hero Form

Name

Alter Ego

Hero Power

Really Smart ▾

You don't see hero data because you're not binding to the `Hero` yet. You know how to do that from earlier pages. [Displaying Data](#) teaches property binding. [User Input](#) shows how to listen for DOM events with an event binding and how to update a component property with the displayed value.

Now you need to display, listen, and extract at the same time.

You could use the techniques you already know, but instead you'll use the new `[(ngModel)]` syntax, which makes binding the form to the model easy.

Find the `<input>` tag for *Name* and update it like this:

src/app/hero-form.component.html (excerpt)

```
<input type="text" class="form-control" id="name"
       required
       [(ngModel)]="model.name" name="name">
```



TODO: remove this: {{model.name}}

You added a diagnostic interpolation after the input tag so you can see what you're doing. You left yourself a note to throw it away when you're done.

Focus on the binding syntax: [(ngModel)]="..." .

You need one more addition to display the data. Declare a template variable for the form. Update the `<form>` tag with `#heroForm="ngForm"` as follows:

src/app/hero-form.component.html (excerpt)

```
<form #heroForm="ngForm">
```

The variable `heroForm` is now a reference to the `NgForm` directive that governs the form as a whole.

The *NgForm* directive

What `NgForm` directive? You didn't add an `NgForm` directive.

Angular did. Angular automatically creates and attaches an `NgForm` directive to the `<form>` tag.

The `NgForm` directive supplements the `form` element with additional features. It holds the controls you created for the elements with an `ngModel` directive and `name` attribute, and monitors their properties, including their validity. It also has its own `valid` property which is true only *if every contained control* is valid.

If you ran the app now and started typing in the `Name` input box, adding and deleting characters, you'd see them appear and disappear from the interpolated text. At some point it might look like this:

Dr IQ 3000

TODO: remove this: Dr IQ 3000

The diagnostic is evidence that values really are flowing from the input box to the model and back again.

That's *two-way data binding*. For more information, see [Two-way binding with NgModel](#) on the [Template Syntax](#) page.

Notice that you also added a `name` attribute to the `<input>` tag and set it to "name", which makes sense for the hero's name. Any unique value will do, but using a descriptive name is helpful. Defining a `name` attribute is a requirement when using `[(ngModel)]` in combination with a form.

Internally, Angular creates `FormControl` instances and registers them with an `NgForm` directive that Angular attaches to the `<form>` tag. Each `FormControl` is registered under the name you assigned to the `name` attribute. Read more in the previous section, [The NgForm directive](#).

Add similar `[(ngModel)]` bindings and `name` attributes to *Alter Ego* and *Hero Power*. You'll ditch the input box binding message and add a new binding (at the top) to the component's `diagnostic` property. Then you can confirm that two-way data binding works *for the entire hero model*.

After revision, the core of the form should look like this:

src/app/hero-form.component.html (excerpt)

```
{{{diagnostic}}}

<div class="form-group">
  <label for="name">Name</label>
  <input type="text" class="form-control" id="name"
    required
    [(ngModel)]="model.name" name="name">
</div>

<div class="form-group">
  <label for="alterEgo">Alter Ego</label>
  <input type="text" class="form-control" id="alterEgo"
    [(ngModel)]="model.alterEgo" name="alterEgo">
</div>

<div class="form-group">
  <label for="power">Hero Power</label>
  <select class="form-control" id="power"
    required
    [(ngModel)]="model.power" name="power">
    <option *ngFor="let pow of powers" [value]="pow">{{pow}}</option>
  </select>
</div>
```

- Each input element has an `id` property that is used by the `label` element's `for` attribute to match the label to its input control.
- Each input element has a `name` property that is required by Angular forms to register the control with the form.

If you run the app now and change every hero model property, the form might display like this:

Hero Form

```
{"id":18,"name":"Dr IQ 3000","power":"Super Flexible","alterEgo":"Chuck OverUnderStreet"}
```

Name

Alter Ego

Hero Power

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.	<code>ng-touched</code>	<code>ng-untouched</code>
The control's value has changed.	<code>ng-dirty</code>	<code>ng-pristine</code>
The control's value is valid.	<code>ng-valid</code>	<code>ng-invalid</code>

Temporarily add a [template reference variable](#) named `spy` to the `Name` `<input>` tag and use it to display the input's CSS classes.

src/app/hero-form.component.html (excerpt)

```
<input type="text" class="form-control" id="name"
  required
  [(ngModel)]="model.name" name="name"
  #spy>
<br>TODO: remove this: {{spy.className}}
```

Now run the app and look at the `Name` input box. Follow these steps *precisely*:

1. Look but don't touch.
2. Click inside the name box, then click outside it.
3. Add slashes to the end of the name.
4. Erase the name.

The actions and effects are as follows:

Dr IQ

TODO: remove this: form-control ng-untouched ng-pristine ng-valid

You should see the following transitions and class names:

Dr IQ

TODO: remove this: form-control **ng-untouched** ng-pristine ng-valid

Untouched

Dr IQ

TODO: remove this: form-control **ng-pristine** ng-valid **ng-touched**

Touched

Dr IQ///

TODO: remove this: form-control **ng-valid** **ng-touched** **ng-dirty**

Changed

TODO: remove this: form-control **ng-touched** **ng-dirty** **ng-invalid**

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:



You achieve this effect by adding these class definitions to a new `forms.css` file that you add to the project as a sibling to `index.html`:

```
src/forms.css
```

```
.ng-valid[required], .ng-valid.required {  
    border-left: 5px solid #42A948; /* green */  
}  
  
.ng-invalid:not(form) {  
    border-left: 5px solid #a94442; /* red */  
}
```

Update the `<head>` of `index.html` to include this style sheet:

src/index.html (styles)

```
<link rel="stylesheet" href="styles.css">
<link rel="stylesheet" href="forms.css">
```



Show and hide validation error messages

You can improve the form. The *Name* input box is required and clearing it turns the bar red. That says something is wrong but the user doesn't know *what* is wrong or what to do about it. Leverage the control's state to reveal a helpful message.

When the user deletes the name, the form should look like this:

A screenshot of a web form. At the top, there is a label 'Name' followed by an input field which is currently empty. Below the input field, a red rectangular box contains the text 'Name is required'. The entire form is enclosed in a light gray rounded rectangle.

To achieve this effect, extend the `<input>` tag with the following:

- A [template reference variable](#).
- The "*is required*" message in a nearby `<div>`, which you'll display only if the control is invalid.

Here's an example of an error message added to the *name* input box:

src/app/hero-form.component.html (excerpt)

```
<label for="name">Name</label>
```



```
<input type="text" class="form-control" id="name"
       required
       [(ngModel)]="model.name" name="name"
       #name="ngModel">
<div [hidden]="name.valid || name.pristine"
      class="alert alert-danger">
  Name is required
</div>
```

You need a template reference variable to access the input box's Angular control from within the template. Here you created a variable called `name` and gave it the value "ngModel".

Why "ngModel"? A directive's `exportAs` property tells Angular how to link the reference variable to the directive. You set `name` to `ngModel` because the `ngModel` directive's `exportAs` property happens to be "ngModel".

You control visibility of the name error message by binding properties of the `name` control to the message `<div>` element's `hidden` property.

src/app/hero-form.component.html (hidden-error-msg)

```
<div [hidden]="name.valid || name.pristine"
      class="alert alert-danger">
```

In this example, you hide the message when the control is valid or pristine; "pristine" means the user hasn't changed the value since it was displayed in this form.

This user experience is the developer's choice. Some developers want the message to display at all times. If you ignore the `pristine` state, you would hide the message only when the value is valid. If you arrive in

this component with a new (blank) hero or an invalid hero, you'll see the error message immediately, before you've done anything.

Some developers want the message to display only when the user makes an invalid change. Hiding the message while the control is "pristine" achieves that goal. You'll see the significance of this choice when you add a new hero to the form.

The hero *Alter Ego* is optional so you can leave that be.

Hero *Power* selection is required. You can add the same kind of error handling to the `<select>` if you want, but it's not imperative because the selection box already constrains the power to valid values.

Now you'll add a new hero in this form. Place a *New Hero* button at the bottom of the form and bind its click event to a `newHero` component method.

src/app/hero-form.component.html (New Hero button)

```
<button type="button" class="btn btn-default" (click)="newHero()">New Hero</button>
```

src/app/hero-form.component.ts (New Hero method)

```
newHero() {
  this.model = new Hero(42, '', '');
}
```

Run the application again, click the *New Hero* button, and the form clears. The *required* bars to the left of the input box are red, indicating invalid `name` and `power` properties. That's understandable as these are required fields. The error messages are hidden because the form is pristine; you haven't changed anything yet.

Enter a name and click *New Hero* again. The app displays a *Name is required* error message. You don't want error messages when you create a new (empty) hero. Why are you getting one now?

Inspecting the element in the browser tools reveals that the *name* input box is *no longer pristine*. The form remembers that you entered a name before clicking *New Hero*. Replacing the hero object *did not restore the pristine state* of the form controls.

You have to clear all of the flags imperatively, which you can do by calling the form's `reset()` method after calling the `newHero()` method.

src/app/hero-form.component.html (Reset the form)

```
<button type="button" class="btn btn-default" (click)="newHero();  
heroForm.reset()">New Hero</button>
```

Now clicking "New Hero" resets both the form and its control flags.

Submit the form with *ngSubmit*

The user should be able to submit this form after filling it in. The *Submit* button at the bottom of the form does nothing on its own, but it will trigger a form submit because of its type (`type="submit"`).

A "form submit" is useless at the moment. To make it useful, bind the form's `ngSubmit` event property to the hero form component's `onSubmit()` method:

src/app/hero-form.component.html (ngSubmit)

```
<form (ngSubmit)="onSubmit()" #heroForm="ngForm">
```

You'd already defined a template reference variable, `#heroForm`, and initialized it with the value "ngForm". Now, use that variable to access the form with the Submit button.

You'll bind the form's overall validity via the `heroForm` variable to the button's `disabled` property using an event binding. Here's the code:

src/app/hero-form.component.html (submit-button)

```
<button type="submit" class="btn btn-success"  
[disabled]="!heroForm.form.valid">Submit</button>
```



If you run the application now, you find that the button is enabled—although it doesn't do anything useful yet.

Now if you delete the Name, you violate the "required" rule, which is duly noted in the error message. The *Submit* button is also disabled.

Not impressed? Think about it for a moment. What would you have to do to wire the button's enable/disabled state to the form's validity without Angular's help?

For you, it was as simple as this:

1. Define a template reference variable on the (enhanced) form element.
2. Refer to that variable in a button many lines away.

Toggle two form regions (extra credit)

Submitting the form isn't terribly dramatic at the moment.

An unsurprising observation for a demo. To be honest, jazzing it up won't teach you anything new about forms. But this is an opportunity to exercise some of your newly won binding skills. If you

aren't interested, skip to this page's conclusion.

For a more strikingly visual effect, hide the data entry area and display something else.

Wrap the form in a `<div>` and bind its `hidden` property to the `HeroFormComponent.submitted` property.

src/app/hero-form.component.html (excerpt)

```
<div [hidden]="submitted">
  <h1>Hero Form</h1>
  <form (ngSubmit)="onSubmit()" #heroForm="ngForm">

    <!-- ... all of the form ... -->

  </form>
</div>
```

The main form is visible from the start because the `submitted` property is false until you submit the form, as this fragment from the `HeroFormComponent` shows:

src/app/hero-form.component.ts (submitted)

```
submitted = false;

onSubmit() { this.submitted = true; }
```

When you click the *Submit* button, the `submitted` flag becomes true and the form disappears as planned.

Now the app needs to show something else while the form is in the submitted state. Add the following HTML below the `<div>` wrapper you just wrote:

src/app/hero-form.component.html (excerpt)

```
<div [hidden]="!submitted">
  <h2>You submitted the following:</h2>
  <div class="row">
    <div class="col-xs-3">Name</div>
    <div class="col-xs-9 pull-left">{{ model.name }}</div>
  </div>
  <div class="row">
    <div class="col-xs-3">Alter Ego</div>
    <div class="col-xs-9 pull-left">{{ model.alterEgo }}</div>
  </div>
  <div class="row">
    <div class="col-xs-3">Power</div>
    <div class="col-xs-9 pull-left">{{ model.power }}</div>
  </div>
  <br>
  <button class="btn btn-primary" (click)="submitted=false">Edit</button>
</div>
```



There's the hero again, displayed read-only with interpolation bindings. This `<div>` appears only while the component is in the submitted state.

The HTML includes an *Edit* button whose click event is bound to an expression that clears the `submitted` flag.

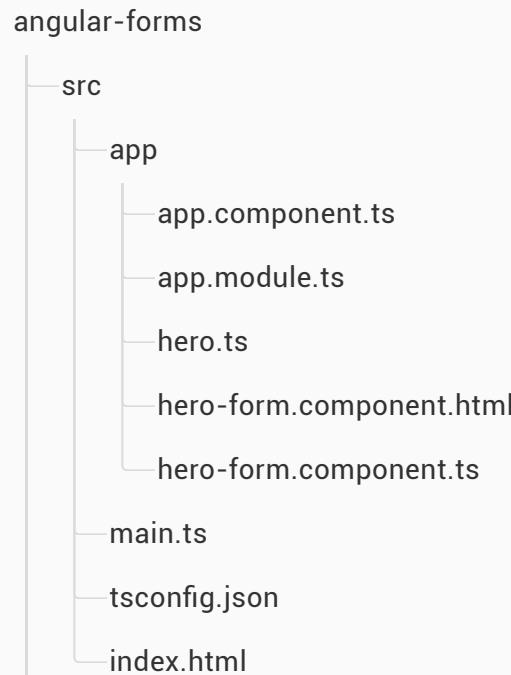
When you click the *Edit* button, this block disappears and the editable form reappears.

Conclusion

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.

The final project folder structure should look like this:



node_modules ...

package.json

Here's the code for the final version of the application:



hero-form.component.ts

hero-form.component.html

hero.ts

app.modul



```
1. import { Component } from '@angular/core';
2.
3. import { Hero }      from './hero';
4.
5. @Component({
6.   selector: 'hero-form',
7.   templateUrl: './hero-form.component.html'
8. })
9. export class HeroFormComponent {
10.
11.   powers = ['Really Smart', 'Super Flexible',
12.             'Super Hot', 'Weather Changer'];
13.
14.   model = new Hero(18, 'Dr IQ', this.powers[0], 'Chuck Overstreet');
15.
16.   submitted = false;
17.
18. onSubmit() { this.submitted = true; }
19.
20. newHero() {
21.   this.model = new Hero(42, '', '');
```



22. }

23. }

This is the archived documentation for Angular v4. Please visit [angular.io](#) to see documentation for the current version of Angular.

Form Validation

[Contents >](#)

[Template-driven validation](#)

[Reactive form validation](#)

[Validator functions](#)

•••

Improve overall data quality by validating user input for accuracy and completeness.

This page shows how to validate user input in the UI and display useful validation messages using both reactive and template-driven forms. It assumes some basic knowledge of the two forms modules.

If you're new to forms, start by reviewing the [Forms](#) and [Reactive Forms](#) guides.

⇒ [Template-driven validation](#)

To add validation to a template-driven form, you add the same validation attributes as you would with [native HTML form validation](#). Angular uses directives to match these attributes with validator functions in the framework.

Every time the value of a form control changes, Angular runs validation and generates either a list of validation errors, which results in an INVALID status, or null, which results in a VALID status.

You can then inspect the control's state by exporting `ngModel` to a local template variable. The following example exports `NgModel` into a variable called `name`:

template/hero-form-template.component.html (name)

```
<input id="name" name="name" class="form-control"
       required minlength="4" forbiddenName="bob"
       [(ngModel)]="hero.name" #name="ngModel" >

<div *ngIf="name.invalid && (name.dirty || name.touched)"
      class="alert alert-danger">

  <div *ngIf="name.errors.required">
    Name is required.
  </div>
  <div *ngIf="name.errors.minlength">
    Name must be at least 4 characters long.
  </div>
  <div *ngIf="name.errors.forbiddenName">
    Name cannot be Bob.
  </div>

</div>
```

Note the following:

- The `<input>` element carries the HTML validation attributes: `required` and `minlength`. It also carries a custom validator directive, `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:

reactive/hero-form-reactive.component.ts (validator functions)

```
ngOnInit(): void {
  this.heroForm = new FormGroup({
    'name': new FormControl(this.hero.name, [
      Validators.required,
      Validators.minLength(4),
      forbiddenNameValidator(/bob/i) // <-- Here's how you pass in the custom
      validator.
    ]),
  });
}
```

```

    'alterEgo': new FormControl(this.hero.alterEgo),
    'power': new FormControl(this.hero.power, Validators.required)
  });
}

get name() { return this.heroForm.get('name'); }

get power() { return this.heroForm.get('power'); }

```

Note that:

- The name control sets up two built-in validators—`Validators.required` and `Validators.minLength(4)`—and one custom validator, `forbiddenNameValidator`. For more details see the [Custom validators](#) section in this guide.
- As these validators are all sync validators, you pass them in as the second argument.
- Support multiple validators by passing the functions in as an array.
- This example adds a few getter methods. In a reactive form, you can always access any form control through the `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.

reactive/hero-form-reactive.component.html (name with error msg)

```

<input id="name" class="form-control"
  formControlName="name" required >

<div *ngIf="name.invalid && (name.dirty || name.touched)"
  class="alert alert-danger">

  <div *ngIf="name.errors.required">

```

```
Name is required.  
</div>  
  Name must be at least 4 characters long.  
</div>  
  Name cannot be Bob.  
</div>  
</div>
```

Key takeaways:

- The form no longer exports any directives, and instead uses the `name` getter defined in the component class.
- The `required` attribute is still present. While it's not necessary for validation purposes, you may want to keep it in your template for CSS styling or accessibility reasons.

Custom validators

Since the built-in validators won't always match the exact use case of your application, sometimes you'll want to create a custom validator.

Consider the `forbiddenNameValidator` function from previous [examples](#) in this guide. Here's what the definition of that function looks like:

shared/forbidden-name.directive.ts (forbiddenNameValidator)

```
/** A hero's name can't match the given regular expression */  
export function forbiddenNameValidator(nameRe: RegExp): ValidatorFn {  
  return (control: AbstractControl): {[key: string]: any} => {
```

```
    const forbidden = nameRe.test(control.value);
    return forbidden ? {'forbiddenName': {value: control.value}} : null;
};

}
```

The function is actually a factory that takes a regular expression to detect a *specific* forbidden name and returns a validator function.

In this sample, the forbidden name is "bob", so the validator will reject any hero name containing "bob". Elsewhere it could reject "alice" or any name that the configuring regular expression matches.

The `forbiddenNameValidator` factory returns the configured validator function. That function takes an Angular control object and returns *either* null if the control value is valid *or* a validation error object. The validation error object typically has a property whose name is the validation key, `'forbiddenName'`, and whose value is an arbitrary dictionary of values that you could insert into an error message, `{name}`.

Custom async validators are similar to sync validators, but they must instead return a Promise or Observable that later emits null or a validation error object. In the case of an Observable, the Observable must complete, at which point the form uses the last value emitted for validation.

Adding to reactive forms

In reactive forms, custom validators are fairly simple to add. All you have to do is pass the function directly to the `FormControl`.

reactive/hero-form-reactive.component.ts (validator functions)

```
this.heroForm = new FormGroup({
  'name': new FormControl(this.hero.name, [
    Validators.required,
    Validators.minLength(4),
```

```
    forbiddenNameValidator(/bob/i) // <-- Here's how you pass in the custom
    validator.
  ],
  'alterEgo': new FormControl(this.hero.alterEgo),
  'power': new FormControl(this.hero.power, Validators.required)
});
```

Adding to template-driven forms

In template-driven forms, you don't have direct access to the `FormControl` instance, so you can't pass the validator in like you can for reactive forms. Instead, you need to add a directive to the template.

The corresponding `ForbiddenValidatorDirective` serves as a wrapper around the `forbiddenNameValidator`.

Angular recognizes the directive's role in the validation process because the directive registers itself with the `NG_VALIDATORS` provider, a provider with an extensible collection of validators.

shared/forbidden-name.directive.ts (providers)

```
providers: [{provide: NG_VALIDATORS, useExisting: ForbiddenValidatorDirective,
  multi: true}]]
```

The directive class then implements the `Validator` interface, so that it can easily integrate with Angular forms. Here is the rest of the directive to help you get an idea of how it all comes together:

shared/forbidden-name.directive.ts (directive)

```
1. @Directive({
  2.   selector: '[forbiddenName]',
```

```
3. providers: [{provide: NG_VALIDATORS, useExisting:
  ForbiddenValidatorDirective, multi: true}]
4. })
5. export class ForbiddenValidatorDirective implements Validator {
6.   @Input() forbiddenName: string;
7.
8.   validate(control: AbstractControl): {[key: string]: any} {
9.     return this.forbiddenName ? forbiddenNameValidator(new
  RegExp(this.forbiddenName, 'i'))(control)
10.    : null;
11.  }
12. }
```

Once the `ForbiddenValidatorDirective` is ready, you can simply add its selector, `forbiddenName`, to any input element to activate it. For example:

template/hero-form-template.component.html (forbidden-name-input)

```
<input id="name" name="name" class="form-control"
  required minlength="4" forbiddenName="bob"
  [(ngModel)]="hero.name" #name="ngModel" >
```

You may have noticed that the custom validation directive is instantiated with `useExisting` rather than `useClass`. The registered validator must be *this instance* of the `ForbiddenValidatorDirective` –the instance in the form with its `forbiddenName` property bound to “bob”. If you were to replace `useExisting` with `useClass`, then you’d be registering a new class instance, one that doesn’t have a `forbiddenName`.

Control status CSS classes

Like in AngularJS, Angular automatically mirrors many control properties onto the form control element as CSS classes. You can use these classes to style form control elements according to the state of the form.

The following classes are currently supported:

- `.ng-valid`
- `.ng-invalid`
- `.ng-pending`
- `.ng-pristine`
- `.ng.dirty`
- `.ng-untouched`
- `.ng-touched`

The hero form uses the `.ng-valid` and `.ng-invalid` classes to set the color of each form control's border.

forms.css (status classes)

```
.ng-valid[required], .ng-valid.required {  
    border-left: 5px solid #42A948; /* green */  
}  
  
.ng-invalid:not(form) {  
    border-left: 5px solid #a94442; /* red */  
}
```



You can run the [live example](#) / [download example](#) to see the complete reactive and template-driven example code.

This is the archived documentation for Angular v4. Please visit [angular.io](#) to see documentation for the current version of Angular.

Reactive Forms

[Contents >](#)

[Introduction to Reactive Forms](#)

Reactive forms

Template-driven 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.

Try the [Reactive Forms live-example](#) / [download example](#).

You can also run the [Reactive Forms Demo](#) / [download example](#) version and choose one of the intermediate steps from the "demo picker" at the top.

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 principles, 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.

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 descendants 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.

Setup

Follow the steps in the [Setup guide](#) for creating a new project folder (perhaps called `reactive-forms`) based on the *QuickStart seed*.

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. Create a new `data-model.ts` file in the `app` directory and copy the content below into it.

src/app/data-model.ts

```
export class Hero {
  id = 0;
  name = '';
  addresses: Address[];
}

export class Address {
  street = '';
  city = '';
}
```

```
    state = '';
    zip   = '';
}

export const heroes: Hero[] = [
{
  id: 1,
  name: 'Whirlwind',
  addresses: [
    {street: '123 Main', city: 'Anywhere', state: 'CA', zip: '94801'},
    {street: '456 Maple', city: 'Somewhere', state: 'VA', zip: '23226'},
  ],
},
{
  id: 2,
  name: 'Bombastic',
  addresses: [
    {street: '789 Elm', city: 'Smallville', state: 'OH', zip: '04501'},
  ],
},
{
  id: 3,
  name: 'Magneta',
  addresses: [ ]
},
];
export const states = ['CA', 'MD', 'OH', 'VA'];
```

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.

Create a *reactive forms* component

Make a new file called `hero-detail.component.ts` in the `app` directory and import these symbols:

src/app/hero-detail.component.ts

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

Now enter the `@Component` decorator that specifies the `HeroDetailComponent` metadata:

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

```
@Component({
  selector: 'hero-detail',
  templateUrl: './hero-detail.component.html'
})
```

Next, create an exported `HeroDetailComponent` class with a `FormControl`. `FormControl` is a directive that allows you to create and manage a `FormControl` instance directly.

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

```
export class HeroDetailComponent1 {
  name = new FormControl();
}
```

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.

Create the template

Now create the component's template, `src/app/hero-detail.component.html`, with the following markup.

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

```
<h2>Hero Detail</h2>
<h3><i>Just a FormControl</i></h3>
<label class="center-block">Name:
  <input class="form-control" [FormControl]="name">
</label>
```

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.

Import the *ReactiveFormsModule*

The HeroDetailComponent template uses `formControlName` directive from the `ReactiveFormsModule`.

In this sample, you declare the `HeroDetailComponent` in the `AppModule`. Therefore, do the following three things in `app.module.ts`:

1. Use a JavaScript `import` statement to access the `ReactiveFormsModule` and the `HeroDetailComponent`.
2. Add `ReactiveFormsModule` to the `AppModule`'s `imports` list.
3. Add `HeroDetailComponent` to the declarations array.

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

```
import { NgModule }           from '@angular/core';
import { BrowserModule }      from '@angular/platform-browser';
import { ReactiveFormsModule } from '@angular/forms'; // <-- #1 import module

import { AppComponent }        from './app.component';
import { HeroDetailComponent } from './hero-detail.component'; // <-- #1 import component

@NgModule({
  imports: [
    BrowserModule,
    ReactiveFormsModule // <-- #2 add to @NgModule imports
  ],
  declarations: [
    AppComponent,
    HeroDetailComponent
  ]
})
```

```
declarations: [
  AppComponent,
  HeroDetailComponent, // <-- #3 declare app component
],
bootstrap: [ AppComponent ]
})
export class AppModule { }
```

Display the *HeroDetailComponent*

Revise the `AppComponent` template so it displays the `HeroDetailComponent`.

src/app/app.component.ts

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

@Component({
  selector: 'my-app',
  template: `
    <div class="container">
      <h1>Reactive Forms</h1>
      <hero-detail></hero-detail>
    </div>`
})
export class AppComponent { }
```



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 stylesheets* to the head of `index.html`:

index.html

```
<link rel="stylesheet"  
      href="https://unpkg.com/bootstrap@3.3.7/dist/css/bootstrap.min.css">
```

Now that everything is wired up, the browser should display something like this:

Hero Detail

Just a FormControl

Add a FormGroup

Usually, if you have multiple *FormControl*s, 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`:

src/app/hero-detail.component.ts

```
import { Component }             from '@angular/core';
import { FormControl, FormGroup } from '@angular/forms';
```



In the class, wrap the `FormControl` in a `FormGroup` called `heroForm` as follows:

src/app/hero-detail.component.ts

```
export class HeroDetailComponent2 {
  heroForm = new FormGroup ({
    name: new FormControl()
  });
}
```



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.

src/app/hero-detail.component.html

```
<h2>Hero Detail</h2>
<h3><i>FormControl in a FormGroup</i></h3>
<form [formGroup]="heroForm" novalidate>
  <div class="form-group">
    <label class="center-block">Name:</label>
    <input class="form-control" formControlName="name">
  </div>
</form>
```

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?

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`:

src/app/hero-detail.component.html

```
<p>Form value: {{ heroForm.value | json }}</p>
<p>Form status: {{ heroForm.status | json }}</p>
```



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 keystrokes 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.

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`:

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

```
import { Component }           from '@angular/core';
import { FormBuilder, FormGroup } from '@angular/forms';
```

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:

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

```
export class HeroDetailComponent3 {
  heroForm: FormGroup; // <--- heroForm is of type FormGroup
```

```
constructor(private fb: FormBuilder) { // <--- inject FormBuilder
  this.createForm();
}

createForm() {
  this.heroForm = this.fb.group({
    name: '', // <--- the FormControl called "name"
  });
}
```

`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.

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.

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

```
import { Component }          from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
```

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`.

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

```
this.heroForm = this.fb.group({
  name: ['', 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.

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

```
<p>Form value: {{ heroForm.value | json }}</p>
<p>Form status: {{ heroForm.status | json }}</p>
```

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 diagnostic 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`.

`src/app/hero-detail.component.ts (excerpt)`



```
import { Component }                      from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';

import { states } from './data-model';
```

Declare the `states` property and add some address `FormControls` to the `heroForm` as follows.

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

```
export class HeroDetailComponent4 {
  heroForm: FormGroup;
  states = states;

  constructor(private fb: FormBuilder) {
    this.createForm();
  }

  createForm() {
    this.heroForm = this.fb.group({
      name: ['', Validators.required ],
      street: '',
      city: '',
      state: '',
      zip: '',
      power: '',
      sidekick: ''
    });
  }
}
```

Then add corresponding markup in `hero-detail.component.html` within the `form` element.

src/app/hero-detail.component.html

```
<h2>Hero Detail</h2>
<h3><i>A FormGroup with multiple FormControl</i></h3>
<form [formGroup]="heroForm" novalidate>
  <div class="form-group">
    <label class="center-block">Name:</label>
    <input class="form-control" formControlName="name">
  </div>
  <div class="form-group">
    <label class="center-block">Street:</label>
    <input class="form-control" formControlName="street">
  </div>
  <div class="form-group">
    <label class="center-block">City:</label>
    <input class="form-control" formControlName="city">
  </div>
  <div class="form-group">
    <label class="center-block">State:</label>
    <select class="form-control" formControlName="state">
      <option *ngFor="let state of states" [value]="state">{{state}}</option>
    </select>
  </div>
  <div class="form-group">
    <label class="center-block">Zip Code:</label>
  </div>
```

```
<input class="form-control" formControlName="zip">
</label>
</div>
<div class="form-group radio">
  <h4>Super power:</h4>
  <label class="center-block"><input type="radio" formControlName="power"
value="flight">Flight</label>
  <label class="center-block"><input type="radio" formControlName="power"
value="x-ray vision">X-ray vision</label>
  <label class="center-block"><input type="radio" formControlName="power"
value="strength">Strength</label>
</div>
<div class="checkbox">
  <label class="center-block">
    <input type="checkbox" formControlName="sidekick">I have a sidekick.
  </label>
</div>
</form>

<p>Form value: {{ heroForm.value | json }}</p>
```

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](#).

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`.

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

```
export class HeroDetailComponent5 {
  heroForm: FormGroup;
  states = states;

  constructor(private fb: FormBuilder) {
    this.createForm();
  }
}
```

```
}

createForm() {
  this.heroForm = this.fb.group({ // <-- the parent FormGroup
    name: ['', Validators.required ],
    address: this.fb.group({ // <-- the child FormGroup
      street: '',
      city: '',
      state: '',
      zip: ''
    }),
    power: '',
    sidekick: ''
  });
}
}
```

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:

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

```
<div formGroupName="address" class="well well-lg">
  <h4>Secret Lair</h4>
```

```
<div class="form-group">
  <label class="center-block">Street:
    <input class="form-control" formControlName="street">
  </label>
</div>
<div class="form-group">
  <label class="center-block">City:
    <input class="form-control" formControlName="city">
  </label>
</div>
<div class="form-group">
  <label class="center-block">State:
    <select class="form-control" formControlName="state">
      <option *ngFor="let state of states" [value]="state">{{state}}</option>
    </select>
  </label>
</div>
<div class="form-group">
  <label class="center-block">Zip Code:
    <input class="form-control" formControlName="zip">
  </label>
</div>
</div>
```

After these changes, the JSON output in the browser shows the revised *form mode*/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.

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:

src/app/hero-detail.component.html

```
<p>Name value: {{ heroForm.get('name').value }}</p>
```

To get the state of a `FormControl` that's inside a `FormGroup`, use dot notation to path to the control.

src/app/hero-detail.component.html

```
<p>Street value: {{ heroForm.get('address.street').value}}</p>
```

You can use this technique to display *any* property of a `FormControl` such as one of the following:

Property	Description
<code>myControl.value</code>	the value of a <code>FormControl</code> .
<code>myControl.status</code>	the validity of a <code>FormControl</code> . Possible values: <code>VALID</code> , <code>INVALID</code> , <code>PENDING</code> , or <code>DISABLED</code> .
<code>myControl.pristine</code>	<code>true</code> if the user has <i>not</i> changed the value in the UI. Its opposite is <code>myControl.dirty</code> .
<code>myControl.unouched</code>	<code>true</code> if the control user has not yet entered the HTML control and triggered its blur event. Its opposite is <code>myControl.touched</code> .

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](#).

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`:

src/app/data-model.ts (classes)

```
export class Hero {  
  id = 0;  
  name = '';  
  addresses: Address[];  
}  
  
export class Address {  
  street = '';  
  city = '';  
  state = '';  
  zip = '';  
}
```

Here, again, is the component's `FormGroup` definition.

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

```
this.heroForm = this.fb.group({
  name: ['', Validators.required ],
  address: this.fb.group({
    street: '',
    city: '',
    state: '',
    zip: ''
  }),
  power: '',
  sidekick: ''
});
```



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:

src/app/hero-detail-7.component.ts

```
this.heroForm = this.fb.group({
  name: ['', Validators.required ],
```



```
address: this.fb.group(new Address(), // <-- a FormGroup with a new address
power: '',
sidekick: ''
});
```

Also be sure to update the import from `data-model` so you can reference the `Hero` and `Address` classes:

src/app/hero-detail-7.component.ts

```
import { Address, Hero, states } from './data-model';
```

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`.

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

```
this.heroForm.setValue({
name: this.hero.name,
address: this.hero.addresses[0] || new Address()
});
```

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:

src/app/hero-detail-7.component.ts

```
address: this.hero.addresses[0] || new Address()
```

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.

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

```
this.heroForm.patchValue({  
  name: this.hero.name  
});
```

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 mode*/values. But *when* do you set them? The answer depends upon when the component gets the *data mode*/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.

hero-list.component.html (simplified)

```
<nav>
  <a *ngFor="let hero of heroes | async" (click)="select(hero)">{{hero.name}}</a>
</nav>

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

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`.

src/app/hero-detail.component.ts (core imports)

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

Add the `hero` input property.

src/app/hero-detail-6.component.ts

```
@Input() hero: Hero;
```

Add the `ngOnChanges` method to the class as follows:

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

```
ngOnChanges() {
  this.heroForm.setValue({
    name: this.hero.name,
    address: this.hero.addresses[0] || new Address()
  });
}
```

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.

src/app/hero-detail-7.component.ts

```
this.heroForm.reset();
```

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:

src/app/hero-detail.component.ts (ngOnchanges - revised)

```
ngOnChanges() {  
  this.heroForm.reset({  
    name: this.hero.name,  
    address: this.hero.addresses[0] || new Address()  
  });  
}
```



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:

Select a hero:

Refresh

Whirlwind

Bombastic

Magneta

Hero Detail

Editing: Magneta

Name:

Magneta

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, create 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.

Use `FormArray` to present an array of `FormGroup`s

So far, you've seen `FormControls` and `FormGroup`s. A `FormGroup` is a named object whose property values are `FormControls` and other `FormGroup`s.

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* `FormGroup`s.

To get access to the `FormArray` class, import it into `hero-detail.component.ts`:

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

```
import { Component, Input, OnChanges }           from '@angular/core';
import { FormArray, FormBuilder, FormGroup, Validators } from '@angular/forms';

import { Address, Hero, states } from './data-model';
```

To *work* with a `FormArray` you do the following:

1. Define the items (`FormControls` or `FormGroup`s) 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`.

src/app/hero-detail-7.component.ts

```
this.heroForm = this.fb.group({
  name: ['', Validators.required ],
  address: this.fb.group(new Address()), // <-- a FormGroup with a new address
  power: '',
  sidekick: ''
});
```

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:

src/app/hero-detail-8.component.ts

```
this.heroForm = this.fb.group({
  name: ['', Validators.required ],
  secretLairs: this.fb.array([]), // <-- secretLairs as an empty FormArray
  power: '',
  sidekick: ''
});
```

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 `FormGroup`s.

src/app/hero-detail-8.component.ts

```
setAddresses(addresses: Address[]) {
  const addressFGs = addresses.map(address => this.fb.group(address));
  const addressFormArray = this.fb.array(addressFGs);
  this.heroForm.setControl('secretLairs', addressFormArray);
}
```

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 `FormGroup`s, not `Address`s.

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.

src/app/hero-detail.component.ts (secretLayers property)

```
get secretLairs(): FormArray {
  return this.heroForm.get('secretLairs') as FormArray;
}
```



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` `FormGroup`s.

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:

src/app/hero-detail.component.html (*ngFor)

```
<div formArrayName="secretLairs" class="well well-lg">
  <div *ngFor="let address of secretLairs.controls; let i=index" [formGroupName]="i"
>
  <!-- The repeated address template -->
</div>
</div>
```

Here's the complete template for the *secret lairs* section:

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

```
1. <div formArrayName="secretLairs" class="well well-lg">
2.   <div *ngFor="let address of secretLairs.controls; let i=index"
3.     [formGroupName]="i" >
4.       <!-- The repeated address template -->
5.       <h4>Address #{{i + 1}}</h4>
6.       <div style="margin-left: 1em;">
7.         <div class="form-group">
8.           <label class="center-block">Street:
9.             <input class="form-control" formControlName="street">
10.            </label>
11.        </div>
12.        <div class="form-group">
13.          <label class="center-block">City:
14.            <input class="form-control" formControlName="city">
15.          </label>
16.        </div>
<div class="form-group">
```

```
17.      <label class="center-block">State:
18.          <select class="form-control" formControlName="state">
19.              <option *ngFor="let state of states" [value]="state">{{state}}
20.          </option>
21.      </select>
22.  </label>
23. </div>
24. <div class="form-group">
25.     <label class="center-block">Zip Code:
26.         <input class="form-control" formControlName="zip">
27.     </label>
28. </div>
29. <br>
30. <!-- End of the repeated address template -->
31. </div>
32. </div>
```

Add a new lair to the *FormArray*

Add an `addLair` method that gets the `secretLairs` `FormArray` and appends a new `address` `FormGroup` to it.

src/app/hero-detail.component.ts (addLair method)

```
addLair() {
    this.secretLairs.push(this.fb.group(new Address()));
}
```

Place a button on the form so the user can add a new *secret lair* and wire it to the component's `addLair` method.

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

```
<button (click)="addLair()" type="button">Add a Secret Lair</button>
```



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.

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`.

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

```
nameChangeLog: string[] = [];

logNameChange() {
  const nameControl = this.heroForm.get('name');
  nameControl.valueChanges.forEach(
    (value: string) => this.nameChangeLog.push(value)
  );
}
```

Call it in the constructor, after creating the form.

src/app/hero-detail-8.component.ts

```
constructor(private fb: FormBuilder) {  
  this.createForm();  
  this.logNameChange();  
}  
}
```



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:

src/app/hero-detail.component.html (Name change log)

```
<h4>Name change log</h4>  
<div *ngFor="let name of nameChangeLog">{{name}}</div>
```



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.

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:

Refresh

Whirlwind

Bombastic

Magneta

Hero Detail

Editing: Whirlwind

Save

Revert

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 mode*/to a save method on the injected `HeroService`.

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

```
onSubmit() {
  this.hero = this.prepareSaveHero();
  this.heroService.updateHero(this.hero).subscribe(/* error handling */);
  this.ngOnChanges();
}
```

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.

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

```
prepareSaveHero(): Hero {
  const formModel = this.heroForm.value;

  // deep copy of form model lairs
  const secretLairsDeepCopy: Address[] = formModel.secretLairs.map(
    (address: Address) => Object.assign({}, address)
  );

  // return new `Hero` object containing a combination of original hero value(s)
  // and deep copies of changed form model values
  const saveHero: Hero = {
    id: this.hero.id,
    name: formModel.name as string,
    // addresses: formModel.secretLairs // <-- bad!
    addresses: secretLairsDeepCopy
  };
  return saveHero;
}
```

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`.

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

```
revert() { this.ngOnChanges(); }
```

Buttons

Add the "Save" and "Revert" buttons near the top of the component's template:

src/app/hero-detail.component.html (Save and Revert buttons)

```
<form [formGroup]="heroForm" (ngSubmit)="onSubmit()" novalidate>  
  <div style="margin-bottom: 1em">  
    <button type="submit"  
      [disabled]="heroForm.pristine" class="btn btn-success">Save</button>  
    &nbsp;  
    <button type="reset" (click)="revert()">
```

```

[disabled]="heroForm.pristine" class="btn btn-danger">Revert</button>
</div>

<!-- Hero Detail Controls -->
<div class="form-group radio">
  <h4>Super power:</h4>
  <label class="center-block"><input type="radio" formControlName="power"
value="flight">Flight</label>
  <label class="center-block"><input type="radio" formControlName="power"
value="x-ray vision">X-ray vision</label>
  <label class="center-block"><input type="radio" formControlName="power"
value="strength">Strength</label>
</div>
<div class="checkbox">
  <label class="center-block">
    <input type="checkbox" formControlName="sidekick">I have a sidekick.
  </label>
</div>
</form>

```

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 [Reactive Forms \(final\) in Plunker](#) / [download example](#).

Conclusion

This page covered:

- How to create a reactive form component and its corresponding template.
- How to use `FormBuilder` to simplify coding a reactive form.
- Grouping `FormControl`s.
- 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.

The key files of the final version are as follows:

< src/app/app.component.ts src/app/app.module.ts src/app/hero-detail.component.ts src/ >

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

@Component({
  selector: 'my-app',
  template: `
    <div class="container">
      <h1>Reactive Forms</h1>
      <hero-list></hero-list>
    </div>
  `)
export class AppComponent { }
```

You can download the complete source for all steps in this guide from the [Reactive Forms Demo](#) / [download example](#) live example.

This is the archived documentation for Angular v4. Please visit [angular.io](#) to see documentation for the current version of Angular.

Dynamic Forms

[Contents >](#)

Bootstrap

Question model

Question form components

•••

Building handcrafted forms can be costly and time-consuming, especially if you need a great number of them, they're similar to each other, and they change frequently to meet rapidly changing business and regulatory requirements.

It may be more economical to create the forms dynamically, based on metadata that describes the business object model.

This cookbook shows you how to use `formGroup` to dynamically render a simple form with different control types and validation. It's a primitive start. It might evolve to support a much richer variety of questions, more graceful rendering, and superior user experience. All such greatness has humble beginnings.

The example in this cookbook is a dynamic form to build an online application experience for heroes seeking employment. The agency is constantly tinkering with the application process. You can create the forms on the fly *without changing the application code*.

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

Bootstrap

Start by creating an `NgModule` called `AppModule`.

This cookbook uses [reactive forms](#).

Reactive forms belongs to a different `NgModule` called `ReactiveFormsModule`, so in order to access any reactive forms directives, you have to import `ReactiveFormsModule` from the `@angular/forms` library.

Bootstrap the `AppModule` in `main.ts`.

app.module.ts

main.ts

```
1. import { BrowserModule }          from '@angular/platform-browser';
2. import { ReactiveFormsModule }    from '@angular/forms';
3. import { NgModule }               from '@angular/core';
4.
5. import { AppComponent }           from './app.component';
6. import { DynamicFormComponent }   from './dynamic-form.component';
7. import { DynamicFormQuestionComponent } from './dynamic-form-
   question.component';
8.
9. @NgModule({
10.   imports: [ BrowserModule, ReactiveFormsModule ],
11.   declarations: [ AppComponent, DynamicFormComponent,
   DynamicFormQuestionComponent ],
12.   bootstrap: [ AppComponent ]
13. })
14. export class AppModule {
15.   constructor() {
```

```
16. }
17. }
```

Question model

The next step is to define an object model that can describe all scenarios needed by the form functionality. The hero application process involves a form with a lot of questions. The *question* is the most fundamental object in the model.

The following `QuestionBase` is a fundamental question class.

src/app/question-base.ts

```
1. export class QuestionBase<T>{
2.   value: T;
3.   key: string;
4.   label: string;
5.   required: boolean;
6.   order: number;
7.   controlType: string;
8.
9.   constructor(options: {
10.     value?: T,
11.     key?: string,
12.     label?: string,
13.     required?: boolean,
14.     order?: number,
15.     controlType?: string
16.   } = {}) {
```

```
17.     this.value = options.value;
18.     this.key = options.key || '';
19.     this.label = options.label || '';
20.     this.required = !options.required;
21.     this.order = options.order === undefined ? 1 : options.order;
22.     this.controlType = options.controlType || '';
23.   }
24. }
```

From this base you can derive two new classes in `TextboxQuestion` and `DropdownQuestion` that represent textbox and dropdown questions. The idea is that the form will be bound to specific question types and render the appropriate controls dynamically.

`TextboxQuestion` supports multiple HTML5 types such as text, email, and url via the `type` property.

src/app/question-textbox.ts

```
import { QuestionBase } from './question-base';

export class TextboxQuestion extends QuestionBase<string> {
  controlType = 'textbox';
  type: string;

  constructor(options: {} = {}) {
    super(options);
    this.type = options['type'] || '';
  }
}
```

`DropdownQuestion` presents a list of choices in a select box.

src/app/question-dropdown.ts

```
import { QuestionBase } from './question-base';

export class DropdownQuestion extends QuestionBase<string> {
  controlType = 'dropdown';
  options: {key: string, value: string}[] = [];

  constructor(options: {} = {}) {
    super(options);
    this.options = options['options'] || [];
  }
}
```



Next is `QuestionControlService`, a simple service for transforming the questions to a `FormGroup`. In a nutshell, the form group consumes the metadata from the question model and allows you to specify default values and validation rules.

src/app/question-control.service.ts

```
import { Injectable } from '@angular/core';
import { FormControl, FormGroup, Validators } from '@angular/forms';

import { QuestionBase } from './question-base';

@Injectable()
export class QuestionControlService {
  constructor() { }

  toFormGroup(questions: QuestionBase<any>[] ) {
```



```
let group: any = {};  
  
questions.forEach(question => {  
  group[question.key] = question.required ? new FormControl(question.value ||  
'', Validators.required)  
    : new FormControl(question.value ||  
'');  
});  
return new FormGroup(group);  
}  
}
```

Question form components

Now that you have defined the complete model you are ready to create components to represent the dynamic form.

`DynamicFormComponent` is the entry point and the main container for the form.

`dynamic-form.component.html` `dynamic-form.component.ts`

```
1. <div>  
2.   <form (ngSubmit)="onSubmit()" [formGroup]="form">  
3.  
4.   <div *ngFor="let question of questions" class="form-row">  
5.     <df-question [question]="question" [form]="form"></df-question>  
6.   </div>  
7.
```

```
8.   <div class="form-row">
9.     <button type="submit" [disabled]="!form.valid">Save</button>
10.    </div>
11.  </form>
12.
13.  <div *ngIf="payLoad" class="form-row">
14.    <strong>Saved the following values</strong><br>{{payLoad}}
15.  </div>
16. </div>
```

It presents a list of questions, each bound to a `<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.

[dynamic-form-question.component.html](#) [dynamic-form-question.component.ts](#)

```
1. <div [formGroup]="form">
2.   <label [attr.for]="question.key">{{question.label}}</label>
3.
4.   <div [ngSwitch]="question.controlType">
5.
6.     <input *ngSwitchCase="'textbox'" [formControlName]="question.key"
7.             [id]="question.key" [type]="question.type">
8.
9.     <select [id]="question.key" *ngSwitchCase="'dropdown'"
10.        [formControlName]="question.key">
11.       <option *ngFor="let opt of question.options" [value]="opt.key">
12.         {{opt.value}}</option>
```

```
11.   </select>
12.
13.   </div>
14.
15.   <div class="errorMessage" *ngIf="!isValid">{{question.label}} is
    required</div>
16. </div>
```

Notice this component can present any type of question in your model. You only have two types of questions at this point but you can imagine many more. The `ngSwitch` determines which type of question to display.

In both components you're relying on Angular's `formGroup` to connect the template HTML to the underlying control objects, populated from the question model with display and validation rules.

`formControlName` and `formGroup` are directives defined in `ReactiveFormsModule`. The templates can access these directives directly since you imported `ReactiveFormsModule` from `AppModule`.

Questionnaire data

`DynamicFormComponent` expects the list of questions in the form of an array bound to `@Input()` `questions`.

The set of questions you've defined for the job application is returned from the `QuestionService`. In a real app you'd retrieve these questions from storage.

The key point is that you control the hero job application questions entirely through the objects returned from `QuestionService`. Questionnaire maintenance is a simple matter of adding, updating, and removing objects from the `questions` array.

src/app/question.service.ts

```
1. import { Injectable }      from '@angular/core';
2.
3. import { DropdownQuestion } from './question-dropdown';
4. import { QuestionBase }   from './question-base';
5. import { TextboxQuestion } from './question-textbox';
6.
7. @Injectable()
8. export class QuestionService {
9.
10. // Todo: get from a remote source of question metadata
11. // Todo: make asynchronous
12. getQuestions() {
13.
14.   let questions: QuestionBase<any>[] = [
15.
16.     new DropdownQuestion({
17.       key: 'brave',
18.       label: 'Bravery Rating',
19.       options: [
20.         {key: 'solid',  value: 'Solid'},
21.         {key: 'great',  value: 'Great'},
22.         {key: 'good',   value: 'Good'},
23.         {key: 'unproven', value: 'Unproven'}
24.       ],
25.       order: 3
26.     }),
27.
28.     new TextboxQuestion({
29.       key: 'firstName',
30.       label: 'First name',
```



```
31.         value: 'Bombasto',
32.         required: true,
33.         order: 1
34.     )),
35.
36.     new TextboxQuestion({
37.         key: 'emailAddress',
38.         label: 'Email',
39.         type: 'email',
40.         order: 2
41.     })
42. ];
43.
44.     return questions.sort((a, b) => a.order - b.order);
45. }
46. }
```

Finally, display an instance of the form in the `AppComponent` shell.

app.component.ts

```
1. import { Component }      from '@angular/core';
2.
3. import { QuestionService } from './question.service';
4.
5. @Component({
6.     selector: 'my-app',
7.     template: `
8.         <div>
9.             <h2>Job Application for Heroes</h2>
```

```
10.      <dynamic-form [questions]="questions"></dynamic-form>
11.    </div>
12.    `,
13.    providers: [QuestionService]
14.  ))
15. export class AppComponent {
16.   questions: any[];
17.
18.   constructor(service: QuestionService) {
19.     this.questions = service.getQuestions();
20.   }
21. }
```

Dynamic Template

Although in this example you're modelling a job application for heroes, there are no references to any specific hero question outside the objects returned by `QuestionService`.

This is very important since it allows you to repurpose the components for any type of survey as long as it's compatible with the *question* object model. The key is the dynamic data binding of metadata used to render the form without making any hardcoded assumptions about specific questions. In addition to control metadata, you are also adding validation dynamically.

The *Save* button is disabled until the form is in a valid state. When the form is valid, you can click *Save* and the app renders the current form values as JSON. This proves that any user input is bound back to the data model. Saving and retrieving the data is an exercise for another time.

The final form looks like this:

Job Application for Heroes

First name

Email

Bravery Rating

Solid 

[Back to top](#)

This is the archived documentation for Angular v4. Please visit [angular.io](#) to see documentation for the current version of Angular.

Bootstrapping

[Contents >](#)

The *imports* array

The *declarations* array

The *bootstrap* array

•••

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

The [setup](#) instructions produce a new project with the following minimal `AppModule`. You'll evolve this module as your application grows.

src/app/app.module.ts

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

@NgModule({
  imports:      [ BrowserModule ],
```



```
    declarations: [ AppComponent ],
    bootstrap:   [ AppComponent ]
)
export class AppModule { }
```

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.

- *imports* – the `BrowserModule` that this and every application needs to run in a browser.
- *declarations* – the application's lone component, which is also ...
- *bootstrap* – the *root* component that Angular creates and inserts into the `index.html` host web page.

The [NgModules](#) guide dives deeply into the details of NgModules. All you need to know at the moment is a few basics about these three properties.

The *imports* array

NgModules are a way to consolidate features that belong together into discrete units. Many features of Angular itself are organized as NgModules. HTTP services are in the `HttpModule`. The router is in the `RouterModule`. Eventually you may create a feature module.

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 and cookbook 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* NgModules—all of them classes decorated with `@NgModule`—that the application needs to function properly.

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 `NgModule` 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`; *not* NgModule classes, *not* service classes,

not model classes.

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.

Bootstrap in *main.ts*

There are many ways to bootstrap an application. The variations depend upon how you want to compile the application and where you want to run it.

In the beginning, you will compile the application dynamically with the *Just-in-Time (JIT)* compiler and you'll run it in a browser. You can learn about other options later.

The recommended place to bootstrap a JIT-compiled browser application is in a separate file in the `src` folder named `src/main.ts`

`src/main.ts`

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app/app.module';

platformBrowserDynamic().bootstrapModule(AppModule);
```

This code creates a browser platform for dynamic (JIT) 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 `my-app` so Angular looks for a `<my-app>` tag in the `index.html` like this one ...

setup/src/index.html

```
<my-app><!-- content managed by Angular --></my-app>
```

... and displays the `AppComponent` there.

This file is very stable. Once you've set it up, you may never change it again.

More about NgModules

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 [NgModules](#) guide.

This is the archived documentation for Angular v4. Please visit [angular.io](#) to see documentation for the current version of Angular.

NgModules

[Contents >](#)

Angular modularity

The root *AppModule*

Bootstrapping in *main.ts*

•••

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 run module 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.

Before reading this page, read the [The Root Module](#) page, 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 "Tour of Heroes" theme. Here's an index to live examples at key moments in the evolution of the sample:

- [A minimal NgModule app](#) / [download example](#)
- [The first contact module](#) / [download example](#)
- [The revised contact module](#) / [download example](#)
- [Just before adding SharedModule](#) / [download example](#)
- [The final version](#) / [download example](#)

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.

Angular modularity

Modules are a great way to organize an application and extend it with capabilities from external libraries.

Many Angular libraries are modules (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.

Modules 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.

Modules 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. The metadata do the following:

- Declare which components, directives, and pipes belong to the module.
- Make some of those classes public so that other component templates can use them.
- Import other modules with the components, directives, and pipes needed by the components in *this* module.
- Provide services at the application level that any application component can use.

Every Angular app has at least one module class, the *root module*. You bootstrap that module to launch the application.

The root module is all you need in a simple application with a few components. As the app grows, you refactor the root module into *feature modules* that represent collections of related functionality. You then import these modules into the root module.

Later in this page, you'll read about this process. For now, you'll start with the root module.

The root *AppModule*

Every Angular app has a *root module* class. By convention, the *root module* class is called `AppModule` and it exists in a file named `app.module.ts`.

The `AppModule` from the QuickStart seed on the [Setup](#) page is as minimal as possible:

src/app/app.module.ts (minimal)

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

@NgModule({
  imports:      [ BrowserModule ],
```

```
    declarations: [ AppComponent ],
    bootstrap:   [ AppComponent ]
  })
export class AppModule { }
```

The `@NgModule` decorator defines the metadata for the module. This page takes an intuitive approach to understanding the metadata and fills in details as it progresses.

The 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 module's component templates.

The `declarations` list identifies the application's only component, the *root component*, the top of the app's rather bare component tree.

The example `AppComponent` simply displays a data-bound title:

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

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

@Component({
  selector: 'my-app',
  template: '<h1>{{title}}</h1>',
})
export class AppComponent {
  title = 'Minimal NgModule';
}
```

Lastly, the `@NgModule.bootstrap` property identifies this `AppComponent` as the *bootstrap component*. When Angular launches the app, it places the HTML rendering of `AppComponent` in the DOM, inside the `<my-app>` element tags of the `index.html`.

Bootstrapping in `main.ts`

You launch the application by bootstrapping the `AppModule` in the `main.ts` file.

Angular offers a variety of bootstrapping options targeting multiple platforms. This page describes two options, both targeting the browser.

Compile just-in-time (JIT)

In the first, *dynamic* option, the [Angular compiler](#) compiles the application in the browser and then launches the app.

src/main.ts (dynamic)

```
// The browser platform with a compiler
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

// The app module
import { AppModule } from './app/app.module';

// Compile and launch the module
platformBrowserDynamic().bootstrapModule(AppModule);
```

The samples in this page demonstrate the dynamic bootstrapping approach.

Angular 2 Example - Minimal NgModule

Project

app

- app.component.0.ts
- app.module.0.ts
- main.0.ts
- index.html
- README.md
- styles.css
- systemjs.config.js
- tsconfig.json

Preview

Minimal
NgModule



Click to run

Plunker © (2016) is created by Geoffrey Goodman [G](#) [Twitter](#)

You can also [download this example](#).

Compile ahead-of-time (AOT)

Consider the static alternative which can produce a much smaller application that launches faster, especially on mobile devices and high latency networks.

In the `static` option, the Angular compiler runs ahead of time as part of the build process, producing a collection of class factories in their own files. Among them is the `AppModuleNgFactory`.

The syntax for bootstrapping the pre-compiled `AppModuleNgFactory` is similar to the dynamic version that bootstraps the `AppModule` class.

src/main.ts (static)

```
// The browser platform without a compiler
import { platformBrowser } from '@angular/platform-browser';

// The app module factory produced by the static offline compiler
import { AppModuleNgFactory } from './app/app.module.ngfactory';

// Launch with the app module factory.
platformBrowser().bootstrapModuleFactory(AppModuleNgFactory);
```

Because the entire application was pre-compiled, Angular doesn't ship the Angular compiler to the browser and doesn't compile in the browser.

The application code downloaded to the browser is much smaller than the dynamic equivalent and it's ready to execute immediately. The performance boost can be significant.

Both the JIT and AOT compilers generate an `AppModuleNgFactory` class from the same `AppModule` source code. The JIT compiler creates that factory class on the fly, in memory, in the browser. The AOT compiler outputs the factory to a physical file that is imported here in the static version of `main.ts`.

In general, the `AppModule` should neither know nor care how it is bootstrapped.

Although the `AppModule` evolves as the app grows, the bootstrap code in `main.ts` doesn't change. This is the last time you'll look at `main.ts`.

Declare directives and components

As the app evolves, the first addition is a `HighlightDirective`, an [attribute directive](#) that sets the background color of the attached element.

src/app/highlight.directive.ts

```
import { Directive, ElementRef } from '@angular/core';

@Directive({ selector: '[highlight]' })
/** Highlight the attached element in gold */
export class HighlightDirective {
  constructor(el: ElementRef) {
    el.nativeElement.style.backgroundColor = 'gold';
    console.log(
      `AppRoot highlight called for ${el.nativeElement.tagName}`);
  }
}
```



Update the `AppComponent` template to attach the directive to the title:

src/app/app.component.ts

```
template: '<h1 highlight>{{title}}</h1>'
```



If you ran the app now, Angular wouldn't recognize the `highlight` attribute and would ignore it. You must declare the directive in `AppModule`.

Import the `HighlightDirective` class and add it to the module's `declarations` like this:

src/app/app.module.ts

```
declarations: [  
  AppComponent,  
  HighlightDirective,  
,
```



Refactor the title into its own `TitleComponent`. The component's template binds to the component's `title` and `subtitle` properties like this:

src/app/title.component.html

```
<h1 highlight>{{title}} {{subtitle}}</h1>
```



src/app/title.component.ts

```
import { Component, Input } from '@angular/core';  
  
@Component({  
  selector: 'app-title',  
  templateUrl: './title.component.html',  
})  
export class TitleComponent {  
  @Input() subtitle = '';  
  title = 'NgModules';  
}
```



Rewrite the `AppComponent` to display the new `TitleComponent` in the `<app-title>` element, using an input binding to set the `subtitle`.

src/app/app.component.ts (v1)

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

@Component({
  selector: 'my-app',
  template: '<app-title [subtitle]="subtitle"></app-title>'
})
export class AppComponent {
  subtitle = '(v1)';
}
```



Angular won't recognize the `<app-title>` tag until you declare it in `AppModule`. Import the `TitleComponent` class and add it to the module's `declarations`:

src/app/app.module.ts

```
declarations: [
  AppComponent,
  HighlightDirective,
  TitleComponent,
],
```



Service providers

Modules are a great way to provide services for all of the module's components.

The [Dependency Injection](#) page describes the Angular hierarchical dependency-injection system and how to configure that system with `providers` at different levels of the application's component tree.

A module can add providers to the application's root dependency injector, making those services available everywhere in the application.

Many applications capture information about the currently logged-in user and make that information accessible through a user service. This sample application has a dummy implementation of such a `UserService`.

src/app/user.service.ts

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

@Injectable()
/** Dummy version of an authenticated user service */
export class UserService {
  userName = 'Sherlock Holmes';
}
```

The sample application should display a welcome message to the logged-in user just below the application title. Update the `TitleComponent` template to show the welcome message below the application title.

src/app/title.component.html

```
<h1 highlight>{{title}} {{subtitle}}</h1>
<p *ngIf="user">
  <i>Welcome, {{user}}</i>
<p>
```

Update the `TitleComponent` class with a constructor that injects the `UserService` and sets the component's `user` property from the service.

src/app/title.component.ts

```
import { Component, Input } from '@angular/core';
import { UserService } from './user.service';

@Component({
  selector: 'app-title',
  templateUrl: './title.component.html',
})
export class TitleComponent {
  @Input() subtitle = '';
  title = 'NgModules';
  user = '';

  constructor(userService: UserService) {
    this.user = userService.userName;
  }
}
```

You've defined and used the service. Now to *provide* it for all components to use, add it to a `providers` property in the `AppModule` metadata:

src/app/app.module.ts (providers)

```
providers: [ UserService ],
```

Import supporting modules

In the revised `TitleComponent`, an `*ngIf` directive guards the message. There is no message if there is no user.

src/app/title.component.html (ngIf)

```
<p *ngIf="user">  
  <i>Welcome, {{user}}</i>  
<p>
```



Although `AppModule` doesn't declare `NgIf`, the application still compiles and runs. How can that be? The Angular compiler should either ignore or complain about unrecognized HTML.

Angular does recognize `NgIf` because you imported it earlier. The initial version of `AppModule` imports `BrowserModule`.

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

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



Importing `BrowserModule` made all of its public components, directives, and pipes visible to the component templates in `AppModule`.

More accurately, `NgIf` is declared in `CommonModule` from `@angular/common`.

`CommonModule` contributes many of the common directives that applications need, including `ngIf` and `ngFor`.

`BrowserModule` imports `CommonModule` and [re-exports](#) it. The net effect is that an importer of `BrowserModule` gets `CommonModule` directives automatically.

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 modules before

you can use their directives.

To illustrate this point, you'll extend the sample app with `ContactComponent`, a form component that imports form support from the Angular `FormsModule`.

Add the `_ContactComponent_`

[Angular forms](#) are a great way to manage user data entry.

The `ContactComponent` presents a "contact editor," implemented with Angular forms in the [template-driven form](#) style.

Angular form styles

You can write Angular form components in template-driven or [reactive](#) style.

The following sample imports the `FormsModule` from `@angular/forms` because the `ContactComponent` is written in *template-driven* style. Modules with components written in the *reactive* style import the `ReactiveFormsModule`.

The `ContactComponent` selector matches an element named `<app-contact>`. Add an element with that name to the `AppComponent` template, just below the `<app-title>`:

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

```
template: `<app-title [subtitle]="subtitle"></app-title>
<app-contact></app-contact>
`
```

Form components are often complex. The `ContactComponent` has its own `ContactService` and [custom pipe](#) (called `Awesome`), and an alternative version of the `HighlightDirective`.

To make it manageable, place all contact-related material in an `src/app/contact` folder and break the component into three constituent HTML, TypeScript, and css files:

< src/app/contact/contact.component.html src/app/contact/contact.component.ts src/app/co >

```
1. <h2>Contact of {{user_name}}</h2>
2. <div *ngIf="msg" class="msg">{{msg}}</div>
3.
4. <form *ngIf="contacts" (ngSubmit)="onSubmit()" #contactForm="ngForm">
5.   <h3 highlight>{{ contact.name | awesome }}</h3>
6.   <div class="form-group">
7.     <label for="name">Name</label>
8.     <input type="text" class="form-control" required
9.       [(ngModel)]="contact.name"
10.      name="name" #name="ngModel" >
11.      <div [hidden]="name.valid" class="alert alert-danger">
12.        Name is required
13.      </div>
14.    </div>
15.    <br>
16.    <button type="submit" class="btn btn-default"
17.      [disabled]="!contactForm.form.valid">Save</button>
18.    <button type="button" class="btn" (click)="next()"
19.      [disabled]="!contactForm.form.valid">Next Contact</button>
20.    <button type="button" class="btn" (click)="newContact()>New
21.      Contact</button>
22.  </form>
```

In the middle of the component template, notice the two-way data binding `[(ngModel)]`. `ngModel` is the selector for the `NgModel` directive.

Although `NgModel` is an Angular directive, the *Angular compiler* won't recognize it for the following reasons:

- `AppModule` doesn't declare `NgModel`.
- `NgModel` wasn't imported via `BrowserModule`.

Even if Angular somehow recognized `ngModel`, `ContactComponent` wouldn't behave like an Angular form because form features such as validation aren't yet available.

Import the `FormsModule`

Add the `FormsModule` to the `AppModule` metadata's `imports` list.

src/app/app.module.ts

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

Now `[(ngModel)]` binding will work and the user input will be validated by Angular forms, once you declare the new component, pipe, and directive.

Do not add `NgModel` —or the `FORMS_DIRECTIVES`—to the `AppModule` metadata's declarations.

These directives belong to the `FormsModule`.

*Components, directives, and pipes belong to *one module only*.*

Never re-declare classes that belong to another module.

Declare the contact component, directive, and pipe

The application won't compile until you declare the contact component, directive, and pipe. Update the declarations in the `AppModule` accordingly:

src/app/app.module.ts (declarations)

```
declarations: [
  AppComponent,
  HighlightDirective,
  TitleComponent,
  AwesomePipe,
  ContactComponent,
  ContactHighlightDirective
],
```

There are two directives with the same name, both called `HighlightDirective`.

To work around this, create an alias for the contact version using the `as` JavaScript import keyword.

src/app/app.module.1b.ts

```
import {
  HighlightDirective as ContactHighlightDirective
} from './contact/highlight.directive';
```

This solves the immediate issue of referencing both directive *types* in the same file but leaves another issue unresolved. You'll learn more about that issue later in this page, in [Resolve directive conflicts](#).

Provide the `_ContactService_`

The `'ContactComponent'` displays contacts retrieved by the `'ContactService'`, which Angular injects into its constructor.

You have to provide that service somewhere. The `ContactComponent` could provide it, but then the service would be scoped to this component only. You want to share this service with other contact-related components that you'll surely add later.

In this app, add `ContactService` to the `AppModule` metadata's `providers` list:

src/app/app.module.ts (providers)

```
providers: [ ContactService, UserService ],
```



Now you can inject `ContactService` (like `UserService`) into any component in the application.

Application-scoped providers

The `'ContactService'` provider is `_application_-scoped` because Angular registers a module's `'providers'` with the application's `*root injector*`.

Architecturally, the `ContactService` belongs to the Contact business domain. Classes in other domains don't need the `ContactService` and shouldn't inject it.

You might expect Angular to offer a `module`-scoping mechanism to enforce this design. It doesn't. `NgModule` instances, unlike components, don't have their own injectors so they can't have their own

provider scopes.

This omission is intentional. NgModules are designed primarily to extend an application, to enrich the entire app with the module's capabilities.

In practice, service scoping is rarely an issue. Non-contact components can't accidentally inject the `ContactService`. To inject `ContactService`, you must first import its *type*. Only Contact components should import the `ContactService` type.

Read more in the [How do I restrict service scope to a module?](#) section of the [NgModule FAQs](#) page.

Run the app

Everything is in place to run the application with its contact editor.

The app file structure looks like this:

```
app
  app.component.ts
  app.module.ts
  highlight.directive.ts
  title.component.(html|ts)
  user.service.ts
  contact
    awesome.pipe.ts
    contact.component.(css|html|ts)
    contact.service.ts
```

Try the example:

Angular 2 Example - Contact NgModule v.1

Project

app

- contact
- app.component.1b.ts
- app.module.1b.ts
- highlight.directive.ts
- main.1b.ts
- title.component.html
- title.component.ts
- user.service.ts

index.html

README.md

styles.css

systemjs.config.js

tsconfig.json

Preview

Angular Modules (v1)

Welcome, Sherlock Holmes

Contact of Sherlock

Click to run

Awesome Sam Spade

Name

Save Next Contact New Contact

Plunker © (2016) is created by Geoffrey Goodman [GitHub](#) [Twitter](#)

You can also [download this example](#).

Resolve directive conflicts

An issue arose [earlier](#) when you declared the contact's `HighlightDirective` because you already had a `HighlightDirective` class at the application level.

The selectors of the two directives both highlight the attached element with a different color.

`src/app/highlight.directive.ts` `src/app/contact/highlight.directive.ts`

```
import { Directive, ElementRef } from '@angular/core';

@Directive({ selector: '[highlight]' })
/** Highlight the attached element in gold */
export class HighlightDirective {
  constructor(el: ElementRef) {
    el.nativeElement.style.backgroundColor = 'gold';
    console.log(
      `AppRoot highlight called for ${el.nativeElement.tagName}`);
  }
}
```



Both directives are declared in this module so both directives are active.

When the two directives compete to color the same element, the directive that's declared later wins because its DOM changes overwrite the first. In this case, the contact's `HighlightDirective` makes the application title text blue when it should stay gold.

The issue is that two different classes are trying to do the same thing.

It's OK to import the same directive class multiple times. Angular removes duplicate classes and only registers one of them.

But from Angular's perspective, two different classes, defined in different files, that have the same name are not duplicates. Angular keeps both directives and they take turns modifying the same HTML element.

At least the app still compiles. If you define two different component classes with the same selector specifying the same element tag, the compiler reports an error. It can't insert two components in the same DOM location.

To eliminate component and directive conflicts, create feature modules that insulate the declarations in one module from the declarations in another.

Feature modules

This application isn't big yet, but it's already experiencing structural issues.

- The root `AppModule` grows larger with each new application class.
- There are conflicting directives. The `HighlightDirective` in the contact re-colors the work done by the `HighlightDirective` declared in `AppModule`. Also, it 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.

You can resolve these issues with *feature modules*.

A feature module is a class adorned by the `@NgModule` decorator and its metadata, just like a root module. Feature module metadata have the same properties as the metadata for a root module.

The root module and the feature module share the same execution context. They share the same dependency injector, which means the services in one module are available to all.

The modules 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 implementation from other modules.

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.

While you can do everything within the root module, 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 modules through the services it provides and the components, directives, and pipes that it shares.

In the next section, you'll carve the contact functionality out of the root module and into a dedicated feature module.

Make *Contact* a feature module

It's easy to refactor the contact material into a contact feature module.

1. Create the `ContactModule` in the `src/app/contact` folder.
2. Move the contact material from `AppModule` to `ContactModule`.
3. Replace the imported `BrowserModule` with `CommonModule`.
4. Import the `ContactModule` into the `AppModule`.

`AppModule` is the only existing class that changes. But you do add one new file.

Add the *ContactModule*

Here's the new `ContactModule` :

src/app/contact/contact.module.ts

```
1. import { NgModule }           from '@angular/core';
2. import { CommonModule }       from '@angular/common';
3. import { FormsModule }        from '@angular/forms';
4.
5. import { AwesomePipe }        from './awesome.pipe';
6.
7. import
8.     { ContactComponent }    from './contact.component';
9. import { ContactService }     from './contact.service';
10. import { HighlightDirective } from './highlight.directive';
11.
12. @NgModule({
13.   imports:      [ CommonModule, FormsModule ],
14.   declarations: [ ContactComponent, HighlightDirective, AwesomePipe ],
15.   exports:       [ ContactComponent ],
16.   providers:    [ ContactService ]
17. })
18. export class ContactModule { }
```

You copy from `AppModule` the contact-related import statements and `@NgModule` properties that concern the contact, and paste them into `ContactModule`.

You *import* the `FormsModule` because the contact component needs it.

Modules don't inherit access to the components, directives, or pipes that are declared in other modules. What `AppModule` imports is irrelevant to `ContactModule` and vice versa. Before `ContactComponent` can bind with `[(ngModel)]`, its `ContactModule` must import `FormsModule`.

You also replaced `BrowserModule` by `CommonModule`, for reasons explained in the [Should I import BrowserModule or CommonModule?](#) section of the [NgModule FAQs](#) page.

You *declare* the contact component, directive, and pipe in the module `declarations`.

You *export* the `ContactComponent` so other modules that import the `ContactModule` can include it in their component templates.

All other declared contact classes are private by default. The `AwesomePipe` and `HighlightDirective` are hidden from the rest of the application. The `HighlightDirective` can no longer color the `AppComponent` title text.

Refactor the `AppModule`

Return to the `AppModule` and remove everything specific to the contact feature set.

- Delete the contact import statements.
- Delete the contact declarations and contact providers.
- Delete the `FormsModule` from the `imports` list (`AppComponent` doesn't need it).

Leave only the classes required at the application root level.

Then import the `ContactModule` so the app can continue to display the exported `ContactComponent`.

Here's the refactored version of the `AppModule` along with the previous version.

[src/app/app.module.ts \(v2\)](#)

[src/app/app.module.ts \(v1\)](#)

```
1. import { NgModule }           from '@angular/core';
2. import { BrowserModule }     from '@angular/platform-browser';
3.
4. /* App Root */
5. import
6.      { AppComponent }        from './app.component';
7. import { HighlightDirective } from './highlight.directive';
8. import { TitleComponent }    from './title.component';
9. import { UserService }       from './user.service';
10.
11. /* Contact Imports */
12. import
13.      { ContactModule }      from './contact/contact.module';
14.
15. @NgModule({
16.   imports:      [ BrowserModule, ContactModule ],
17.   declarations: [ AppComponent, HighlightDirective, TitleComponent ],
18.   providers:    [ UserService ],
19.   bootstrap:   [ AppComponent ],
20. })
21. export class AppModule { }
```



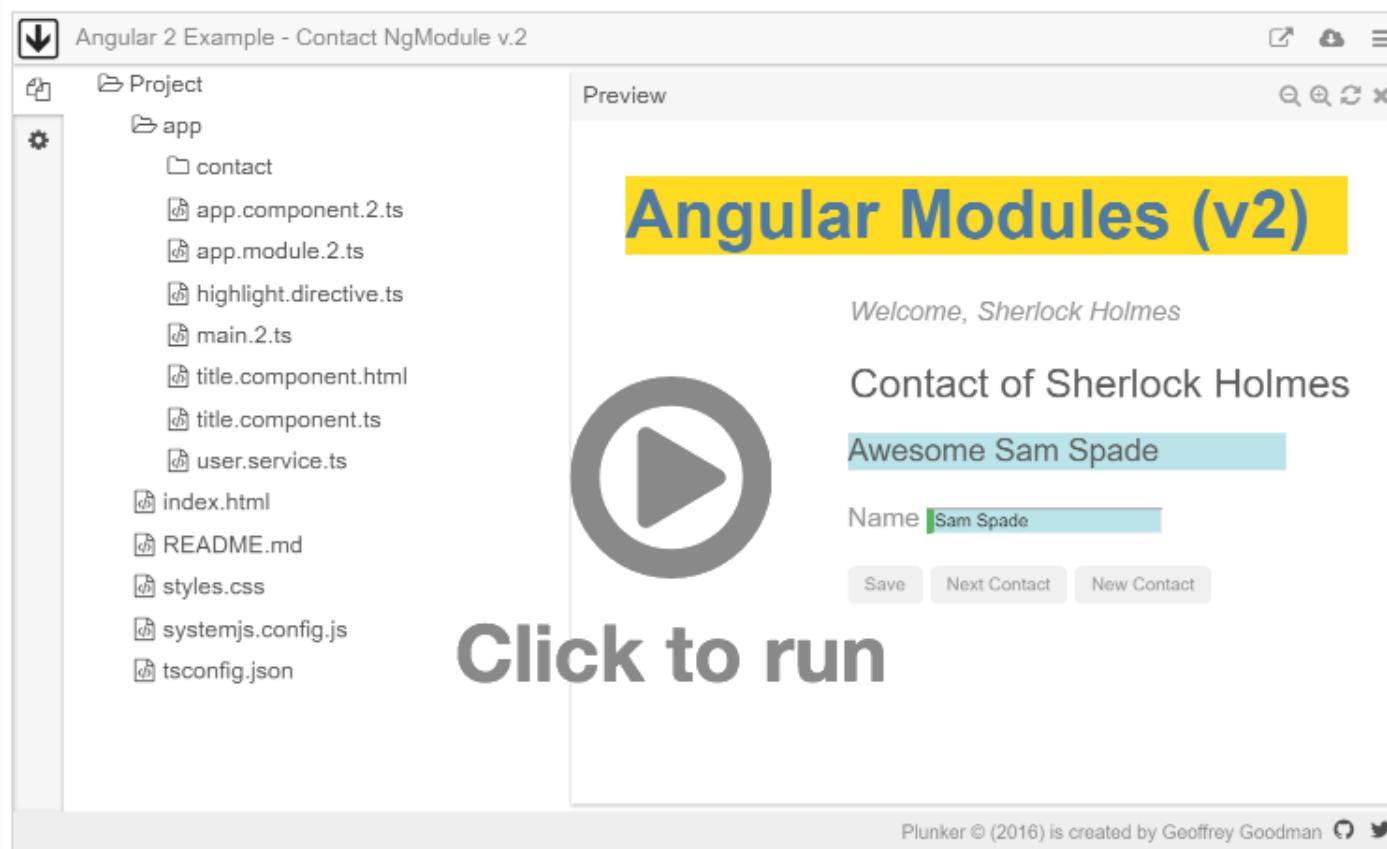
Improvements

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 modules.

- It's simpler:
 - Fewer import statements.
 - No `FormsModule` import.
 - No contact-specific declarations.
 - No `ContactService` provider.
 - No `HighlightDirective` conflict.

Try this `ContactModule` version of the sample.



You can also [download this example](#).

Lazy-loading modules with the router

The Heroic Staffing Agency sample app has evolved. It has two more modules, one for managing the heroes on staff and another for matching crises to the heroes. Both modules are in the early stages of development. Their specifics aren't important to the story and this page doesn't discuss every line of code.

Examine and download the complete source for this version from the [live example](#) / [download example](#)

Some facets of the current application merit discussion are as follows:

- The app has three feature modules: Contact, Hero, and Crisis.
- 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.

The new `AppComponent` template has a title, three links, and a `<router-outlet>`.

src/app/app.component.ts (v3 - Template)

```
template: `

<app-title [subtitle]="subtitle"></app-title>

<nav>

  <a routerLink="contact" routerLinkActive="active">Contact</a>
  <a routerLink="crisis"  routerLinkActive="active">Crisis Center</a>
  <a routerLink="heroes" routerLinkActive="active">Heroes</a>

</nav>

<router-outlet></router-outlet>

`
```

The `<app-contact>` element is gone; you're routing to the *Contact* page now.

The `AppModule` has changed modestly:

src/app/app.module.ts (v3)

```
1. import { NgModule }           from '@angular/core';
2. import { BrowserModule }     from '@angular/platform-browser';
3.
4. /* App Root */
5. import { AppComponent }       from './app.component.3';
6. import { HighlightDirective } from './highlight.directive';
7. import { TitleComponent }    from './title.component';
8. import { UserService }       from './user.service';
9.
10. /* Feature Modules */
11. import { ContactModule }    from './contact/contact.module.3';
12.
13. /* Routing Module */
14. import { AppRoutingModule }   from './app-routing.module.3';
15.
16. @NgModule({
17.   imports: [
18.     BrowserModule,
19.     ContactModule,
20.     AppRoutingModule
21.   ],
22.   providers: [ UserService ],
23.   declarations: [ AppComponent, HighlightDirective, TitleComponent ],
24.   bootstrap: [ AppComponent ]
25. })
```

```
26. export class AppModule { }
```

Some file names bear a `.3` extension that indicates a difference with prior or future versions. The significant differences will be explained in due course.

The module still imports `ContactModule` so that its routes and components are mounted when the app starts.

The module does *not* import `HeroModule` or `CrisisModule`. They'll be fetched and mounted asynchronously when the user navigates to one of their routes.

The significant change from version 2 is the addition of the `AppRoutingModule` to the module `imports`. The `AppRoutingModule` is a [routing module](#) that handles the app's routing concerns.

App routing

src/app/app-routing.module.ts

```
import { NgModule }           from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

export const routes: Routes = [
  { path: '', redirectTo: 'contact', pathMatch: 'full' },
  { path: 'crisis', loadChildren: 'app/crisis/crisis.module#CrisisModule' },
  { path: 'heroes', loadChildren: 'app/hero/hero.module#HeroModule' }
];

@NgModule({
```

```
imports: [RouterModule.forRoot(routes)],
exports: [RouterModule]
})
export class AppRoutingModule {}
```

The router is the subject of the [Routing & Navigation](#) page, so this section skips many of the details and concentrates on the intersection of NgModules and routing.

The `app-routing.module.ts` file 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 here. It's defined in the *Contact* feature's own routing module, `contact-routing.module.ts`. It's standard practice for feature modules with routing components to define their own routes. You'll get to that file in a moment.

The remaining two routes use lazy loading syntax to tell the router where to find the modules:

src/app/app-routing.module.ts

```
{ path: 'crisis', loadChildren: 'app/crisis/crisis.module#CrisisModule' },
{ path: 'heroes', loadChildren: 'app/hero/hero.module#HeroModule' }
```



A lazy-loaded module location is a *string*, not a *type*. In this app, the string identifies both the module *file* and the module *class*, the latter separated from the former by a `#`.

RouterModule.forRoot

The `forRoot` static class method of the `RouterModule` with the provided configuration and added to the `imports` array provides the routing concerns for the module.

src/app/app-routing.module.ts

```
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule {}
```



The returned `AppRoutingModule` class is a `Routing Module` containing both the `RouterModule` directives and the dependency-injection providers that produce a configured `Router`.

This `AppRoutingModule` is intended for the app *root* module only.

Never call `RouterModule.forRoot` in a feature-routing module.

Back in the root `AppModule`, add the `AppRoutingModule` to its `imports` list, and the app is ready to navigate.

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

```
imports: [
  BrowserModule,
  ContactModule,
  AppRoutingModule
],
```



Routing to a feature module

The `src/app/contact` folder holds a new file, `contact-routing.module.ts`. It defines the `contact` route mentioned earlier and provides a `ContactRoutingModule` as follows:

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

```
@NgModule({
  imports: [RouterModule.forChild([
    { path: 'contact', component: ContactComponent }
  ]),
  exports: [RouterModule]
})
export class ContactRoutingModule {}
```



This time you pass the route list to the `forChild` method of the `RouterModule`. The route list is only responsible for providing additional routes and is intended for feature modules.

Always call `RouterModule.forChild` in a feature-routing module.

`forRoot` and `forChild` are conventional names for methods that deliver different `import` values to root and feature modules. Angular doesn't recognize them but Angular developers do.

Follow this convention if you write a similar module that has both shared `declarables` and services.

`ContactModule` has changed in two small but important ways.

`src/app/contact/contact.module.3.ts` `src/app/contact/contact.module.2.ts`

```
@NgModule({
  imports:      [ CommonModule, FormsModule, ContactRoutingModule ],
  declarations: [ ContactComponent, HighlightDirective, AwesomePipe ],
  providers:    [ ContactService ]
})
export class ContactModule { }
```

- It imports the `ContactRoutingModule` object from `contact-routing.module.ts`.
- It no longer exports `ContactComponent`.

Now that you navigate to `ContactComponent` with the router, there's no reason to make it public. Also, `ContactComponent` doesn't need a selector. No template will ever again reference this `ContactComponent`. It's gone from the [AppComponent template](#).

Lazy-loaded routing to a module

The lazy-loaded `HeroModule` and `CrisisModule` follow the same principles as any feature module. They don't look different from the eagerly loaded `ContactModule`.

The `HeroModule` is a bit more complex than the `CrisisModule`, which makes it a more interesting and useful example. Its file structure is as follows:

```
hero
  └── hero-detail.component.ts
```

```
hero-list.component.ts  
hero.component.ts  
hero.module.ts  
hero-routing.module.ts  
hero.service.ts  
highlight.directive.ts
```

This is the child routing scenario familiar to readers of the [Child routing component](#) section of the [Routing & Navigation](#) page. The `HeroComponent` is the feature's top component and routing host. Its template has a `<router-outlet>` that displays either a list of heroes (`HeroList`) or an editor of a selected hero (`HeroDetail`). Both components delegate to the `HeroService` to fetch and save data.

Yet another `HighlightDirective` colors elements in yet a different shade. In the next section, [Shared modules](#), you'll resolve the repetition and inconsistencies.

The `HeroModule` is a feature module like any other.

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

```
@NgModule({  
  imports: [ CommonModule, FormsModule, HeroRoutingModule ],  
  declarations: [  
    HeroComponent, HeroDetailComponent, HeroListComponent,  
    HighlightDirective  
  ]  
})  
export class HeroModule { }
```

It imports the `FormsModule` because the `HeroDetailComponent` template binds with `[(ngModel)]`. It imports the `HeroRoutingModule` from `hero-routing.module.ts` just as `ContactModule` and `CrisisModule` do.

The `CrisisModule` is much the same.

The screenshot shows the Angular 2 Example - NgModule v.3 project in Plunker. The left sidebar displays the project structure:

- Project
- app
 - contact
 - crisis
 - hero
 - app.component.3.ts
 - app.module.3.ts
 - app.routing.3.ts
 - highlight.directive.ts
 - main.3.ts
 - title.component.html
 - title.component.ts
 - user.service.ts
- index.html
- README.md
- styles.css
- systemjs.config.js
- tsconfig.json

The preview window shows the application's interface:

- A yellow header bar with the text "Angular Modules (v3)".
- A large circular play button in the center.
- The text "Welcome, Sherlock Holmes" below the play button.
- A navigation bar with three buttons: "Contact" (highlighted in blue), "Crisis Center", and "Heroes".
- The text "Contact of Sherlock Holmes" below the navigation bar.
- A teal bar with the text "Awesome Sam Spade".
- A contact form with the name "Sam Spade" entered in the "Name" field.
- Buttons for "Save", "Next Contact", and "New Contact".

At the bottom of the preview window, it says "Plunker © (2016) is created by Geoffrey Goodman" with social media links for GitHub and Twitter.

You can also [download this example](#).

Shared modules

The app is shaping up. But it carries three different versions of the `HighlightDirective`. And the many files cluttering the app folder level could be better organized.

Add a `SharedModule` to hold the common components, directives, and pipes and share them with the modules that need them.

1. Create an `src/app/shared` folder.
2. Move the `AwesomePipe` and `HighlightDirective` from `src/app/contact` to `src/app/shared` .
3. Delete the `HighlightDirective` classes from `src/app/` and `src/app/hero` .
4. Create a `SharedModule` class to own the shared material.
5. Update other feature modules to import `SharedModule` .

Here is the `SharedModule` :

src/app/src/app/shared/shared.module.ts

```
1. import { NgModule }           from '@angular/core';
2. import { CommonModule }       from '@angular/common';
3. import { FormsModule }        from '@angular/forms';
4.
5. import { AwesomePipe }        from './awesome.pipe';
6. import { HighlightDirective } from './highlight.directive';
7.
8. @NgModule({
9.   imports:      [ CommonModule ],
10.  declarations: [ AwesomePipe, HighlightDirective ],
11.  exports:      [ AwesomePipe, HighlightDirective,
12.                  CommonModule, FormsModule ]
13. })
14. export class SharedModule { }
```

Note the following:

- It imports the `CommonModule` because its component needs common directives.

- It declares and exports the utility pipe, directive, and component classes as expected.
- It re-exports the `CommonModule` and `FormsModule`

Re-exporting other modules

If you review the application, you may notice that many components requiring `SharedModule` directives also use `NgIf` and `NgFor` from `CommonModule` and bind to component properties with `[(ngModel)]`, a directive in the `FormsModule`. Modules that declare these components would have to import `CommonModule`, `FormsModule`, and `SharedModule`.

You can reduce the repetition by having `SharedModule` re-export `CommonModule` and `FormsModule` so that importers of `SharedModule` get `CommonModule` and `FormsModule` for free.

As it happens, the components declared by `SharedModule` itself don't bind with `[(ngModel)]`. Technically, there is no need for `SharedModule` to import `FormsModule`.

`SharedModule` can still export `FormsModule` without listing it among its `imports`.

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 modules.

The `TitleComponent` is used only once by the `AppComponent`. There's no point in sharing it.

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 module 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 module 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 module that imports that shared module makes its own copy of the service.

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 an `src/app/core` folder.
 - Move the `UserService` and `TitleComponent` from `src/app/` to `src/app/core`.
 - Create a `CoreModule` class to own the core material.
 - Update the `AppRoot` module to import `CoreModule`.

Most of this work is familiar. The interesting part is the `CoreModule`.

`src/app/src/app/core/core.module.ts`

```
1. import {
2.   ModuleWithProviders, NgModule,
3.   Optional, SkipSelf }      from '@angular/core';
4.
5. import { CommonModule }      from '@angular/common';
6.
7. import { TitleComponent }    from './title.component';
8. import { UserService }      from './user.service';
9. @NgModule({
10.   imports:      [ CommonModule ],
11.   declarations: [ TitleComponent ],
12.   exports:      [ TitleComponent ],
13.   providers:    [ UserService ]
14. })
15. export class CoreModule {
16. }
```

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 module owns it and 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 module 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.

Cleanup

Having refactored to a `CoreModule` and a `SharedModule`, it's time to clean up the other modules.

A trimmer `AppModule`

Here is the updated `AppModule` paired with version 3 for comparison:

`src/app/app.module.ts (v4)` `src/app/app.module.ts (v3)`

```
1. import { NgModule }      from '@angular/core';
2. import { BrowserModule } from '@angular/platform-browser';
3.
4. /* App Root */
5. import { AppComponent }  from './app.component';
6.
7. /* Feature Modules */
8. import { ContactModule } from './contact/contact.module';
9. import { CoreModule }    from './core/core.module';
10.
11. /* Routing Module */
12. import { AppRoutingModule } from './app-routing.module';
13.
14. @NgModule({
15.   imports: [
16.     BrowserModule,
17.     ContactModule,
18.     CoreModule,
19.     AppRoutingModule
20.   ],
21.   declarations: [ AppComponent ],
22.   bootstrap:   [ AppComponent ]
23. })
24. export class AppModule { }
```

`AppModule` now has the following qualities:

- A little smaller because many `src/app/root` classes have moved to other modules.
- Stable because you'll add future components and providers to other modules, not this one.
- Delegated to imported modules rather than doing work.
- Focused on its main task, orchestrating the app as a whole.

A trimmer *ContactModule*

Here is the new `ContactModule` paired with the prior version:

`src/app/contact/contact.module.ts (v4)` `src/app/contact/contact.module.ts (v3)`

```
1. import { NgModule }           from '@angular/core';
2. import { SharedModule }       from '../shared/shared.module';
3.
4. import { ContactComponent }   from './contact.component';
5. import { ContactService }     from './contact.service';
6. import { ContactRoutingModule } from './contact-routing.module';
7.
8. @NgModule({
9.   imports:      [ SharedModule, ContactRoutingModule ],
10.  declarations: [ ContactComponent ],
11.  providers:    [ ContactService ]
12. })
13. export class ContactModule { }
```

Notice the following:

- The `AwesomePipe` and `HighlightDirective` are gone.
 - The imports include `SharedModule` instead of `CommonModule` and `FormsModule` .
 - The new version is leaner and cleaner.
-

Configure core services with `CoreModule.forRoot`

A module 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 modules.

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.

src/app/core/user.service.ts (constructor)

```
constructor(@Optional() config: UserServiceConfig) {
```



```
if (config) { this._userName = config.userName; }  
}
```

Here's `CoreModule.forRoot` that takes a `UserServiceConfig` object:

src/app/core/core.module.ts (forRoot)

```
static forRoot(config: UserServiceConfig): ModuleWithProviders {  
  return {  
    ngModule: CoreModule,  
    providers: [  
      {provide: UserServiceConfig, useValue: config}  
    ]  
  };  
}
```

Lastly, call it within the `imports` list of the `AppModule`.

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

```
imports: [  
  BrowserModule,  
  ContactModule,  
  CoreModule.forRoot({userName: 'Miss Marple'}),  
  AppRoutingModule  
],
```

The app displays "Miss Marple" as the user instead of the default "Sherlock Holmes".

Call `forRoot` only in the root application module, `AppModule`. Calling it in any other module, particularly in a lazy-loaded module, 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.

Prevent reimport of the `CoreModule`

Only the root `AppModule` should import the `CoreModule`. **Bad things happen** if a lazy-loaded module 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.

src/app/core/core.module.ts

```
constructor (@Optional() @SkipSelf() parentModule: CoreModule) {  
  if (parentModule) {  
    throw new Error(  
      'CoreModule is already loaded. Import it in the AppModule only');  
  }  
}
```

The constructor tells Angular to inject the `CoreModule` into itself. 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 module such as `HeroModule` (try it).

Angular creates a lazy-loaded module with its own injector, a *child* of the root injector. `@SkipSelf` causes Angular to look for a `CoreModule` in the parent injector, which this time is the root injector. Of course it finds the instance imported by the root `AppModule`. Now `parentModule` exists and the constructor throws the error.

Conclusion

You made it! You can examine and download the complete source for this final version from the live example.

Angular 2 Example - NgModule Final

Project

app

- contact
- core
- crisis
- hero
- shared

app.component.ts

app.module.ts

app.routing.ts

main.ts

index.html

README.md

styles.css

systemjs.config.js

tsconfig.json

Preview

Angular Modules (Final)

Welcome, Miss Marple

Contact Crisis Center Heroes

Contact of Miss Marple

Awesome Sam Spade

Name

Save Next Contact New Contact

Plunker © (2016) is created by Geoffrey Goodman [GitHub](#) [Twitter](#)

You can also [download this 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.

This is the archived documentation for Angular v4. Please visit [angular.io](#) to see documentation for the current version of Angular.

NgModule FAQs

[Contents >](#)

[What classes should I add to *declarations*?](#)

[What is a *declarable*?](#)

[What classes should I *not* add to *declarations*?](#)

•••

NgModules help organize an application into cohesive blocks of functionality.

The [NgModules](#) page guides you from the most elementary `@NgModule` 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](#) page.

What classes should I add to *declarations*?

Add [declarable](#) classes—components, directives, and pipes—to a `declarations` list.

Declare these classes in *exactly one* module of the application. Declare them in *this* module if they *belong* to this module.

What is a *declarable*?

Declarables are the class types—components, directives, and pipes—that you can add to a module's `declarations` list. They're the *only* classes that you can add to `declarations`.

What classes should I *not* add to *declarations*?

Add only [declarable](#) classes to a module's `declarations` list.

Do *not* declare the following:

- A class that's already declared in another module, whether an app module, `@NgModule`, or third-party module.
 - An array of directives imported from another module. For example, don't declare `FORMS_DIRECTIVES` from `@angular/forms`.
 - Module classes.
 - Service classes.
 - Non-Angular classes and objects, such as strings, numbers, functions, entity models, configurations, business logic, and helper classes.
-

Why list the same component in multiple *NgModule* properties?

`AppComponent` is often listed in both `declarations` and `bootstrap`. You might see `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 app 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.
-

What does "Can't bind to 'x' since it isn't a known property of 'y'" mean?

This error usually means that you haven't declared the directive "x" or haven't imported the module to which "x" belongs.

For example, if "x" is `ngModel`, you probably haven't imported the `FormsModule` from `@angular/forms`.

Perhaps you declared "x" in an application sub-module but forgot to export it? The "x" class isn't visible to other modules until you add it to the `exports` list.

What should I import?

Import modules 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 module that **re-exports** it.

Import `FormsModule` from `@angular/forms` if your components have `[(ngModel)]` two-way binding expressions.

Import *shared* and *feature* modules when this module's components incorporate their components, directives, and pipes.

Import only `BrowserModule` in the root `AppModule`.

Should I import `BrowserModule` or `CommonModule`?

The *root application module* (`AppModule`) of almost every browser application should import `BrowserModule` from `@angular/platform-browser`.

`BrowserModule` provides services that are essential to launch and run a browser app.

`BrowserModule` also re-exports `CommonModule` from `@angular/common`, which means that components in the `AppModule` module also have access to the Angular directives every app needs, such as `NgIf` and `NgFor`.

Do not import `BrowserModule` in any other module. *Feature modules* and *lazy-loaded modules* should import `CommonModule` instead. They need the common directives. They don't need to re-install the app-wide providers.

`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.

What if I import the same module twice?

That's not a problem. When three modules all import Module 'A', Angular evaluates Module 'A' once, the first time it encounters it, and doesn't do so again.

That's true at whatever level `A` appears in a hierarchy of imported modules. 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 modules with circular references, so don't let Module 'A' import Module 'B', which imports Module 'A'.

What should I export?

Export [declarable](#) classes that components in *other* modules 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 module.

You *can* export any declarable class—components, directives, and pipes—whether it's declared in this module or in an imported module.

You *can* re-export entire imported modules, which effectively re-exports all of their exported classes. A module can even export a module that it doesn't import.

What should I *not* export?

Don't export the following:

- Private components, directives, and pipes that you need only within components declared in this module. If you don't want another module 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. Its only purpose is to add http service providers to the application as a whole.
-

Can I re-export classes and modules?

Absolutely.

Modules are a great way to selectively aggregate classes from other modules and re-export them in a consolidated, convenience module.

A module can re-export entire modules, which effectively re-exports all of their exported classes. Angular's own `BrowserModule` exports a couple of modules like this:

```
exports: [CommonModule, ApplicationModule]
```



A module can export a combination of its own declarations, selected imported classes, and imported modules.

Don't bother re-exporting pure service modules. Pure service modules don't export [declarable](#) classes that another module could use. For example, there's no point in re-exporting `HttpModule` because it doesn't export anything. Its only purpose is to add http service providers to the application as a whole.

What is the `forRoot` method?

The `forRoot` static method is a convention that makes it easy for developers to configure 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 module, `AppModule`. Importing it in any other module, particularly in a lazy-loaded module, is contrary to the intent and will likely produce a runtime error.

`RouterModule` also offers a `forChild` static method for configuring the routes of lazy-loaded modules.

`forRoot` and `forChild` are conventional names for methods that configure services in root and feature modules respectively.

Angular doesn't recognize these names but Angular developers do. Follow this convention when you write similar modules with configurable service providers.

Why is a service provided in a feature module visible everywhere?

Providers listed in the `@NgModule.providers` of a bootstrapped module have *application scope*. Adding a service provider to `@NgModule.providers` effectively publishes the service to the entire application.

When you import a module, 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 module imports is a primary goal of the NgModule system. Merging module 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`.

Why is a service provided in a *lazy-loaded* module visible only to that module?

Unlike providers of the modules loaded at launch, providers of lazy-loaded modules are *module-scoped*.

When the Angular router lazy-loads a module, it creates a new execution context. That [context has its own injector](#), which is a direct child of the application injector.

The router adds the lazy module's providers and the providers of its imported modules to this child injector.

These providers are insulated from changes to application providers with the same lookup token. When the router creates a component within the lazy-loaded context, Angular prefers service instances created from these providers to the service instances of the application root injector.

What if two modules provide the same service?

When two imported modules, loaded at the same time, list a provider with the same token, the second module's provider "wins". That's because both providers are added to the same injector.

When Angular looks to inject a service for that token, it creates and delivers the instance created by the second provider.

Every class that injects this service gets the instance created by the second provider. Even classes declared within the first module get the instance created by the second provider.

If Module A provides a service for token 'X' and imports a module B that also provides a service for token 'X', then Module A's service definition "wins".

The service provided by the root `AppModule` takes precedence over services provided by imported modules. The `AppModule` always wins.

How do I restrict service scope to a module?

When a module is loaded at application launch, its `@NgModule.providers` have *application-wide scope*; that is, they are available for injection throughout the application.

Imported providers are easily replaced by providers from another imported module. Such replacement might be by design. It could be unintentional and have adverse consequences.

As a general rule, import modules with providers *exactly once*, preferably in the application's *root module*. That's also usually the best place to configure, wrap, and override them.

Suppose a module requires a customized `HttpBackend` that adds a special header for all Http requests. If another module elsewhere in the application also customizes `HttpBackend` or merely imports the

`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 modules do to `HttpBackend`.

Be sure to create module components as children of this module's top component.

You can embed the child components in the top component's template. Alternatively, make the top component a routing host by giving it a `<router-outlet>`. Define child routes and let the router load module components into that outlet.

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 modules](#) to registering in components.

Discussion

Angular registers all startup module 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 NgModule page sample applications, 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".

Should I add other providers to a module or a component?

In general, prefer registering feature-specific providers in modules (`@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`.

Why is it bad if `SharedModule` provides a service to a lazy-loaded module?

This question is addressed in the [Why UserService isn't shared](#) section of the [NgModules](#) page, which discusses the importance of keeping providers out of the `SharedModule`.

Suppose the `UserService` was listed in the module's `providers` (which it isn't). Suppose every module 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 module 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](#) / [download 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 goes bonkers as the Angular creates a new `UserService` instance each time.

Why does lazy loading create a child injector?

Angular adds `@NgModule.providers` to the application root injector, unless the module is lazy loaded. For a lazy-loaded module, Angular creates a *child injector* and adds the module's providers to the child injector.

This means that a module 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 modules?

The answer is grounded in a fundamental characteristic of the Angular dependency-injection system. An injector can add providers *until it's first used*. Once an injector starts creating and delivering services, its

provider list is frozen; no new providers are allowed.

When an application starts, Angular first configures the root injector with the providers of all eagerly loaded modules *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 a module. Angular must add the lazy-loaded module's providers to an injector somewhere. It can't add them to the app root injector because that injector is closed to new providers. So Angular creates a new child injector for the lazy-loaded module context.

How can I tell if a module or service was previously loaded?

Some modules 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 from the NgModules page.

src/app/core/core.module.ts (Constructor)

```
constructor (@Optional() @SkipSelf() parentModule: CoreModule) {
  if (parentModule) {
    throw new Error(
      'CoreModule is already loaded. Import it in the AppModule only');
}
```

```
}
```

What is an *entry component*?

An entry component is any component that Angular loads *imperatively* by type.

A component loaded *declaratively* via its selector is *not* an entry component.

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 module'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 module'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.

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.

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.

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.

What kinds of modules should I have and how should I use them?

Every app is different. Developers have various levels of experience and comfort with the available choices. Some suggestions and guidelines appear to have wide appeal.

The following is preliminary guidance based on early experience using NgModules in a few applications. Read with appropriate caution and reflection.

SharedModule

Create a `SharedModule` with the components, directives, and pipes that you use everywhere in your app. This module should consist entirely of `declarations` , most of them exported.

The `SharedModule` may re-export other [widget modules](#), such as `CommonModule` , `FormsModule` , and modules with the UI controls that you use most widely.

The `SharedModule` should *not* have `providers` for reasons [explained previously](#). Nor should any of its imported or re-exported modules have `providers` . 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 modules 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
Domain	<p>Domain feature modules deliver a user experience <i>dedicated to a particular application domain</i> like editing a customer or placing an order.</p> <p>They typically have a top component that acts as the feature root. Private, supporting sub-components descend from it.</p> <p>Domain feature modules consist mostly of <i>declarations</i>. Only the top component is exported.</p>

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 [Make `Contact` a feature module](#) section of the [NgModules](#) page, before routing is introduced.

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 module. 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 eager loaded routed feature module must be imported by another module 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](#). When they do, the lifetime of the provided services should be the same as the lifetime of the module.

Don't provide application-wide singleton services in a routed feature module or in a module that the routed module imports.

Routing

A [routing module](#) provides *routing configuration* for another module.

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*?](#) on the [Routing & Navigation](#) page.

Service	<p>Service modules <i>provide utility services</i> such as data access and messaging. Ideally, they consist entirely of <i>providers</i> and have no <i>declarations</i>. The <code>CoreModule</code> and Angular's <code>HttpModule</code> are good examples.</p> <p>Service Modules should <i>only</i> be imported by the root <code>AppModule</code>.</p> <p>Do <i>not</i> import service modules in other feature modules. If you deviate from this guideline, know what you're doing and why.</p>
---------	--

Widget	<p>A widget module makes <i>components, directives, and pipes</i> available to external modules. <code>CommonModule</code> and <code>SharedModule</code> are widget modules. Many third-party UI component libraries are widget modules.</p> <p>A widget module should consist entirely of <i>declarations</i>, most of them exported.</p>
--------	--

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 modules 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
--------	-----	------	-----	---------	--------------------------------

What's the difference between Angular 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` also has `imports` and `exports` and they serve a similar purpose.

You *import* other NgModules so you can use their exported classes in component templates. You *export* this NgModule's classes so they can be imported and used by components of *other* modules.

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 module's classes in the `@NgModule.declarations` list.
- An NgModule can only export the [declarable classes](#) it owns or imports from other modules. It doesn't declare or export any other kind of class.

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 module nor are they scoped to the declared classes. They are available *everywhere*.

Here's an *NgModule* class with imports, exports, and declarations.

ngmodule/src/app/contact/contact.module.ts

```
@NgModule({  
  imports:      [ CommonModule, FormsModule ],  
  declarations: [ ContactComponent, HighlightDirective, AwesomePipe ],  
  exports:       [ ContactComponent ],  
  providers:    [ ContactService ]  
})  
export class ContactModule { }
```

Of course you use *JavaScript* modules to write NgModules as seen in the complete `contact.module.ts` file:

src/app/contact/contact.module.ts

```
import { NgModule }           from '@angular/core';
import { CommonModule }       from '@angular/common';
import { FormsModule }        from '@angular/forms';

import { AwesomePipe }        from './awesome.pipe';

import
  { ContactComponent }      from './contact.component';
import { ContactService }    from './contact.service';
import { HighlightDirective } from './highlight.directive';

@NgModule({
  imports:      [ CommonModule, FormsModule ],
  declarations: [ ContactComponent, HighlightDirective, AwesomePipe ],
  exports:      [ ContactComponent ],
  providers:    [ ContactService ]
})
export class ContactModule { }
```



How does Angular find components, directives, and pipes in a template?
What is a ***template reference***?

The [Angular compiler](#) looks inside component templates for other components, directives, and pipes. When it finds one, that's a "template reference".

The Angular compiler finds a component or directive in a template when it can match the *selector* of that component or directive to some HTML in that template.

The compiler finds a pipe if the pipe's *name* appears within the pipe syntax of the template HTML.

Angular only matches selectors and pipe names for classes that are declared by this module or exported by a module that this module imports.

What is the Angular compiler?

The Angular compiler converts the application code you write into highly performant JavaScript code. The `@NgModule` metadata 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 modules.

NgModule API

The following table summarizes the `NgModule` metadata properties.

Property	Description
----------	-------------

declarations

A list of [declarable](#) classes, the *component*, *directive*, and *pipe* classes that *belong to this module*.

These declared classes are visible within the module but invisible to components in a different module unless they are *exported* from this module and the other module *imports* this one.

Components, directives, and pipes must belong to *exactly* one module. The compiler emits an error if you try to declare the same class in more than one module.

Do not re-declare a class imported from another module.

providers

A list of dependency-injection providers.

Angular registers these providers with the root injector of the module's execution context. That's the application's root injector for all modules loaded when the application starts.

Angular can inject one of these provider services into any component in the application. If this module or any module loaded at launch provides the

`HeroService` , Angular can inject the same `HeroService` instance into any app component.

A lazy-loaded module 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 module 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 modules continue to receive the instance created for the application root.

imports

A list of supporting modules.

Specifically, the list of modules whose exported components, directives, or pipes are referenced by the component templates declared in this module. A component template can [reference](#) another component, directive, or pipe when the referenced class is declared in this module or the class was imported from another module.

A component can use the `NgIf` and `NgFor` directives only because its parent module 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 modules. 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 module can use.

Exported declarations are the module's *public API*. A component in another module can [reference](#) *this* module's `HeroComponent` if it imports this module and this module exports `HeroComponent`.

Declarations are private by default. If this module does *not* export `HeroComponent`, no other module can see it.

Importing a module does *not* automatically re-export the imported module's imports. Module 'B' can't use `ngIf` just because it imported module 'A' which imported `CommonModule`. Module 'B' must import `CommonModule` itself.

A module can list another module among its `exports`, in which case all of that module's public components, directives, and pipes are exported.

[Re-export](#) makes module transitivity explicit. If Module 'A' re-exports `CommonModule` and Module 'B' imports Module 'A', Module 'B' components can use `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* in a reachable component template.

Most developers never set this property. The [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 `<router-outlet>`.

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.

This is the archived documentation for Angular v4. Please visit [angular.io](#) to see documentation for the current version of Angular.

Dependency Injection

[Contents >](#)

[Why dependency injection?](#)

[Angular dependency injection](#)

[Configuring the injector](#)

•••

Dependency injection is an important application design pattern. Angular has its own dependency injection framework, and you really can't build an Angular application without it. It's used so widely that almost everyone just calls it *DI*.

This page covers what DI is, why it's so useful, and [how to use it](#) in an Angular app.

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

Why dependency injection?

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

src/app/car/car.ts (without DI)

```
1. export class Car {
2.
3.   public engine: Engine;
4.   public tires: Tires;
5.   public description = 'No DI';
6.
7.   constructor() {
8.     this.engine = new Engine();
9.     this.tires = new Tires();
10.  }
11.
12. // Method using the engine and tires
13. drive() {
14.   return `${this.description} car with ` +
15.     `${this.engine.cylinders} cylinders and ${this.tires.make} tires.`;
16. }
17. }
```



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?

That's super easy. Change the `Car` constructor to a version with DI:

[src/app/car/car.ts \(excerpt with DI\)](#) [src/app/car/car.ts \(excerpt without DI\)](#)

```
public description = 'DI';

constructor(public engine: Engine, public tires: Tires) { }
```

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.

```
// Simple car with 4 cylinders and Flintstone tires.  
let car = new Car(new Engine(), new Tires());
```



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:

```
class Engine2 {  
    constructor(public cylinders: number) {}  
}  
  
// Super car with 12 cylinders and Flintstone tires.  
let bigCylinders = 12;  
let car = new Car(new Engine2(bigCylinders), new Tires());
```



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:

```
class MockEngine extends Engine { cylinders = 8; }
class MockTires  extends Tires  { make = 'YokoGoodStone'; }

// Test car with 8 cylinders and YokoGoodStone tires.
let car = new Car(new MockEngine(), new MockTires());
```

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:

src/app/car/car-factory.ts

```
1. import { Engine, Tires, Car } from './car';
2.
3. // BAD pattern!
4. export class CarFactory {
5.   createCar() {
6.     let car = new Car(this.createEngine(), this.createTires());
7.     car.description = 'Factory';
8.     return car;
}
```

```
9.    }
10.
11.   createEngine() {
12.     return new Engine();
13.   }
14.
15.   createTires() {
16.     return new Tires();
17.   }
18. }
```

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.

src/app/car/car-injector.ts

```
let car = injector.get(Car);
```

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, read on to see how it is implemented in Angular.

Angular dependency injection

Angular ships with its own dependency injection framework. This framework can also be used as a standalone module by other applications and frameworks.

To see what it can do when building components in Angular, start with a simplified version of the `HeroesComponent` that from the [The Tour of Heroes](#).

< src/app/heroes/heroes.component.ts src/app/heroes/hero-list.component.ts src/app/heroes/hero-detail.component.ts >

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

@Component({
  selector: 'my-heroes',
  template: `
    <h2>Heroes</h2>
    <hero-list></hero-list>
  `
})
export class HeroesComponent { }
```

The `HeroesComponent` is the root component of the *Heroes* feature area. It governs all the child components of this area. This stripped down version has only one child, `HeroListComponent`, which displays a list of heroes.

Right now `HeroListComponent` gets heroes from `HEROES`, an in-memory collection defined in another 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 want to get your heroes data from a remote server, you'll have to change the implementation of `heroes` and fix every other use of the `HEROES` mock data.

It's better to make a service that hides how the app gets hero data.

Given that the service is a [separate concern](#), consider writing the service code in its own file.

See [this note](#) for details.

The following `HeroService` exposes a `getHeroes` method that returns the same mock data as before, but none of its consumers need to know that.

src/app/heroes/hero.service.ts

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

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

@Injectable()
export class HeroService {
  getHeroes() { return HEROES; }
}
```



The `@Injectable()` decorator above the service class is covered [shortly](#).

Of course, this isn't a real service. If the app were actually getting data from a remote server, the API would have to be asynchronous, perhaps returning a [Promise](#). You'd also have to rewrite the way components consume the service. This is important in general, but not in this example.

A service is nothing more than a class in Angular. It remains nothing more than a class until you register it with an Angular injector.

Configuring the injector

You don't have to create an Angular injector. Angular creates an application-wide injector for you during the bootstrap process.

src/main.ts (bootstrap)

```
platformBrowserDynamic().bootstrapModule(AppModule);
```

You do have to configure the injector by registering the providers that create the services the application requires. This guide explains what [providers](#) are later.

You can either register a provider within an [NgModule](#) or in application components.

Registering providers in an *NgModule*

Here's the `AppModule` that registers two providers, `UserService` and an `APP_CONFIG` provider, in its `providers` array.

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

```
@NgModule({
```

```
imports: [
  BrowserModule
],
declarations: [
  AppComponent,
  CarComponent,
  HeroesComponent,
/* . . . */
],
providers: [
  UserService,
  { provide: APP_CONFIG, useValue: HERO_DI_CONFIG }
],
bootstrap: [ AppComponent ]
})
export class AppModule {}
```

Because the `HeroService` is used *only* within the `HeroesComponent` and its subcomponents, the top-level `HeroesComponent` is the ideal place to register it.

Registering providers in a component

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

src/app/heroes/heroes.component.ts

```
import { Component }      from '@angular/core';
import { HeroService }    from './hero.service';
```

```
@Component({
  selector: 'my-heroes',
  providers: [HeroService],
  template: `
    <h2>Heroes</h2>
    <hero-list></hero-list>
  `
})
export class HeroesComponent { }
```

When to use *NgModule* versus an application component

On the one hand, a provider in an `NgModule` is registered in the root injector. That means that every provider registered within an `NgModule` will be accessible in the *entire application*.

On the other hand, a provider registered in an application component is available only on that component and all its children.

Here, the `APP_CONFIG` service needs to be available all across the application, so it's registered in the `AppModule` `@NgModule providers` array. But since the `HeroService` is only used within the *Heroes* feature area and nowhere else, it makes sense to register it in the `HeroesComponent`.

Also see "Should I add app-wide providers to the root `AppModule` or the root `AppComponent`?" in the [NgModule FAQ](#).

Preparing the *HeroListComponent* for injection

The `HeroListComponent` should get heroes from the injected `HeroService`. Per the dependency injection pattern, the component must ask for the service in its constructor, [as discussed earlier](#). It's a small change:

`src/app/heroes/hero-list.component (with DI)` `src/app/heroes/hero-list.component (without DI)`

```
1. import { Component } from '@angular/core';
2.
3. import { Hero } from './hero';
4. import { HeroService } from './hero.service';
5.
6. @Component({
7.   selector: 'hero-list',
8.   template: `
9.     <div *ngFor="let hero of heroes">
10.       {{hero.id}} - {{hero.name}}
11.     </div>
12.   `
13. })
14. export class HeroListComponent {
15.   heroes: Hero[];
16.
17.   constructor(heroService: HeroService) {
18.     this.heroes = heroService.getHeroes();
19.   }
20. }
```



Focus on the constructor

Adding a parameter to the constructor isn't all that's happening here.

src/app/heroes/hero-list.component.ts

```
constructor(heroService: HeroService) {  
  this.heroes = heroService.getHeroes();  
}
```



Note that the constructor parameter has the type `HeroService`, and that the `HeroListComponent` class has an `@Component` decorator (scroll up to confirm that fact). Also recall that the parent component (`HeroesComponent`) has `providers` information for `HeroService`.

The constructor parameter type, the `@Component` decorator, and the parent's `providers` information combine to tell the Angular injector to inject an instance of `HeroService` whenever it creates a new `HeroListComponent`.

Implicit injector creation

You saw how to use an injector to create a new `Car` earlier in this guide. You *could* create such an injector explicitly:

src/app/car/car-injector.ts

```
injector = ReflectiveInjector.resolveAndCreate([Car, Engine, Tires]);  
let car = injector.get(Car);
```



You won't find code like that in the Tour of Heroes or any of the other documentation samples. You *could* write code that [explicitly creates an injector](#) if you *had* to, but it's not always the best choice. Angular takes care of creating and calling injectors when it creates components for you—whether through HTML markup,

as in `<hero-list></hero-list>`, or after navigating to a component with the [router](#). If you let Angular do its job, you'll enjoy the benefits of automated dependency injection.

Singleton services

Dependencies are singletons within the scope of an injector. In this guide's example, a single `HeroService` instance is shared among the `HeroesComponent` and its `HeroListComponent` children.

However, Angular DI is a hierarchical injection system, which means that nested injectors can create their own service instances. For more information, see [Hierarchical Injectors](#).

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:

src/app/test.component.ts

```
let expectedHeroes = [{name: 'A'}, {name: 'B'}]
let mockService = <HeroService> {getHeroes: () => expectedHeroes }

it('should have heroes when HeroListComponent created', () => {
  let hlc = new HeroListComponent(mockService);
  expect(hlc.heroes.length).toEqual(expectedHeroes.length);
});
```

Learn more in [Testing](#).

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 revision compared to the original.

`src/app/heroes/hero.service (v2)` `src/app/heroes/hero.service (v1)`

```
1. import { Injectable } from '@angular/core';
2.
3. import { HEROES }      from './mock-heroes';
4. import { Logger }       from '../logger.service';
5.
6. @Injectable()
7. export class HeroService {
8.
9.   constructor(private logger: Logger) { }
10.
11.  getHeroes() {
12.    this.logger.log('Getting heroes ...');
13.    return HEROES;
14.  }
15. }
```

The constructor now asks for an injected instance of a `Logger` and stores it in a private property called `logger`. You call that property within the `getHeroes()` method when anyone asks for heroes.

Why `@Injectable()`?

`@Injectable()` marks a class as available to an injector for instantiation. Generally speaking, an injector reports an error when trying to instantiate a class that is not marked as `@Injectable()`.

As it happens, you could have omitted `@Injectable()` from the first version of `HeroService` because it had no injected parameters. But you must have it now that the service has an injected dependency. You need it because Angular requires constructor parameter metadata in order to inject a `Logger`.

SUGGESTION: ADD `@INJECTABLE()` TO EVERY SERVICE CLASS

Consider adding `@Injectable()` to every service class, even those that don't have dependencies and, therefore, do not technically require it. Here's why:

- **Future proofing:** No need to remember `@Injectable()` when you add a dependency later.
- **Consistency:** All services follow the same rules, and you don't have to wonder why a decorator is missing.

Injectors are also responsible for instantiating components like `HeroesComponent`. So why doesn't `HeroesComponent` have `@Injectable()`?

You *can* add it if you really want to. It isn't necessary because the `HeroesComponent` is already marked with `@Component`, and this decorator class (like `@Directive` and `@Pipe`, which you learn about later) is a

subtype of `@Injectable()`. It is in fact `@Injectable()` decorators that identify a class as a target for instantiation by an injector.

At runtime, injectors can read class metadata in the transpiled JavaScript code and use the constructor parameter type information to determine what things to inject.

Not every JavaScript class has metadata. The TypeScript compiler discards metadata by default. If the `emitDecoratorMetadata` compiler option is true (as it should be in the `tsconfig.json`), the compiler adds the metadata to the generated JavaScript for *every class with at least one decorator*.

While any decorator will trigger this effect, mark the service class with the `@Injectable()` decorator to make the intent clear.

ALWAYS INCLUDE THE PARENTHESES

Always write `@Injectable()`, not just `@Injectable`. The application will fail mysteriously if you forget the parentheses.

Creating and registering a logger service

Inject a logger into `HeroService` in two steps:

1. Create the logger service.
2. Register it with the application.

The logger service is quite simple:

`src/app/logger.service.ts`

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

@Injectable()
export class Logger {
  logs: string[] = [] // capture logs for testing

  log(message: string) {
    this.logs.push(message);
    console.log(message);
  }
}
```

You're likely to need the same logger service everywhere in your application, so put it in the project's `app` folder and register it in the `providers` array of the application module, `AppModule`.

src/app/providers.component.ts (excerpt)

```
providers: [Logger]
```

If you forget to register the logger, Angular throws an exception when it first looks for the logger:

```
EXCEPTION: No provider for Logger! (HeroListComponent -> HeroService -> Logger)
```

That's Angular telling you that the dependency injector couldn't find the *provider* for the logger. It needed that provider to create a `Logger` to inject into a new `HeroService`, which it needed to create and inject into a new `HeroListComponent`.

The chain of creations started with the `Logger` provider. *Providers* are the subject of the next section.

Injector providers

A 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 and other services.

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

Earlier you registered the `Logger` service in the `providers` array of the metadata for the `AppModule` like this:

src/app/providers.component.ts

```
providers: [Logger]
```

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`.

The *Provider* class and *provide* object literal

You wrote the `providers` array like this:

src/app/providers.component.ts

```
providers: [Logger]
```

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

src/app/providers.component.ts

```
[{ provide: Logger, useClass: Logger }]
```



The first is the `token` that serves as the key for both locating a dependency value and registering the provider.

The second is 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.

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`.

src/app/providers.component.ts

```
[{ provide: Logger, useClass: BetterLogger }]
```



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.

src/app/providers.component.ts

```
@Injectable()
class EvenBetterLogger extends Logger {
  constructor(private userService: UserService) { super(); }

  log(message: string) {
    let name = this.userService.user.name;
    super.log(`Message to ${name}: ${message}`);
  }
}
```

Configure it like `BetterLogger`.

src/app/providers.component.ts

```
[ UserService,
  { provide: Logger, useClass: EvenBetterLogger }]
```

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`.

src/app/providers.component.ts

```
[ NewLogger,  
  // Not aliased! Creates two instances of 'NewLogger'  
  { provide: OldLogger, useClass: NewLogger}]
```

The solution: alias with the `useExisting` option.

```
[ NewLogger,  
  // Alias OldLogger w/ reference to NewLogger  
  { provide: OldLogger, useExisting: NewLogger}]
```

Value providers

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

src/app/providers.component.ts

```
// An object in the shape of the logger service  
let silentLogger = {  
  logs: ['Silent logger says "Shhhh!". Provided via "useValue"'],  
  log: () => {}  
};
```

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

```
[{ provide: Logger, useValue: silentLogger }]
```

See more `useValue` examples in the [Non-class dependencies](#) and [InjectionToken](#) sections.

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.

src/app/heroes/hero.service.ts (excerpt)

```
constructor(  
  private logger: Logger,  
  private isAuthorized: boolean) { }  
  
getHeroes() {  
  let auth = this.isAuthorized ? 'authorized' : 'unauthorized';  
  this.logger.log(`Getting heroes for ${auth} user.');
```

```
    return HEROES.filter(hero => this.isAuthorized || !hero.isSecret);  
}
```

You can inject 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:

src/app/heroes/hero.service.provider.ts (excerpt)

```
let heroServiceFactory = (logger: Logger, userService: UserService) => {  
  return new HeroService(logger, userService.user.isAuthorized);  
};
```

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:

src/app/heroes/hero.service.provider.ts (excerpt)

```
export let heroServiceProvider =  
{ provide: HeroService,  
  useFactory: heroServiceFactory,  
  deps: [Logger, UserService]  
};
```

The `useFactory` field tells Angular that the provider is a factory function whose implementation is the `heroServiceFactory`.

The `deps` property is an array of [provider tokens](#). The `Logger` and `UserService` classes serve as tokens for their own class providers. The injector resolves these tokens and injects the corresponding services into the matching factory function parameters.

Notice that you captured the factory provider in an exported variable, `heroServiceProvider`. This extra step makes the factory provider reusable. You can 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:

src/app/heroes/heroes.component (v3) src/app/heroes/heroes.component (v2)

```
1. import { Component }           from '@angular/core';
2.
3. import { heroServiceProvider } from './hero.service.provider';
4.
5. @Component({
6.   selector: 'my-heroes',
7.   template: `
8.     <h2>Heroes</h2>
9.     <hero-list></hero-list>
10.    `,
11.   providers: [heroServiceProvider]
12. })
13. export class HeroesComponent { }
```

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:

src/app/injector.component.ts

```
heroService: HeroService;
```

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:

src/app/heroes/hero-list.component.ts

```
constructor(heroService: HeroService)
```

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

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:

src/app/app-config.ts (excerpt)

```
export interface AppConfig {  
    apiEndpoint: string;  
    title: string;  
}  
  
export const HERO_DI_CONFIG: AppConfig = {  
    apiEndpoint: 'api.heroes.com',  
    title: 'Dependency Injection'  
};
```

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 has an interface, `AppConfig`. Unfortunately, you cannot use a TypeScript interface as a token:

src/app/providers.component.ts

```
// FAIL! Can't use interface as provider token
```

```
[{ provide: AppConfig, useValue: HERO_DI_CONFIG })]
```

src/app/providers.component.ts

```
// FAIL! Can't inject using the interface as the parameter type  
constructor(private config: AppConfig){ }
```



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.

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:

src/app/app.config.ts

```
import { InjectionToken } from '@angular/core';  
  
export let APP_CONFIG = new InjectionToken<AppConfig>('app.config');
```



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

Register the dependency provider using the `InjectionToken` object:

src/app/providers.component.ts

```
providers: [{ provide: APP_CONFIG, useValue: HERO_DI_CONFIG }]
```



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

src/app/app.component.ts

```
constructor(@Inject(APP_CONFIG) config: AppConfig) {
  this.title = config.title;
}
```



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

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

src/app/app.module.ts (ngmodule-providers)

```
providers: [
  UserService,
  { provide: APP_CONFIG, useValue: HERO_DI_CONFIG }
],
```



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()`:

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

```
constructor(@Optional() private logger: Logger) {
  if (this.logger) {
    this.logger.log(some_message);
  }
}
```

When using `@Optional()`, your code must be prepared for a null value. If you don't register a `logger` 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](#).

Appendix: Working with injectors directly

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

src/app/injector.component.ts

```
1. @Component({
2.   selector: 'my-injectors',
3.   template: `
4.     <h2>Other Injections</h2>
5.     <div id="car">{{car.drive()}}</div>
6.     <div id="hero">{{hero.name}}</div>
7.     <div id="rodent">{{rodent}}</div>
8.   `,
9.   providers: [Car, Engine, Tires, heroServiceProvider, Logger]
10. })
11. export class InjectorComponent implements OnInit {
12.   car: Car;
13.
14.   heroService: HeroService;
15.   hero: Hero;
16.
17.   constructor(private injector: Injector) { }
18.
19.   ngOnInit() {
20.     this.car = this.injector.get(Car);
21.     this.heroService = this.injector.get(HeroService);
22.     this.hero = this.heroService.getHeroes()[0];
23.   }
24.
25.   get rodent() {
26.     let rousDontExist = `R.O.U.S.'s? I don't think they exist!`;
27.     return this.injector.get(ROUS, rousDontExist);
28.   }

```

29. }

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](#).

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.

Appendix: Why have 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](#). But why flirt with trouble? Avoid the problem altogether by defining components and services in separate files.

This is the archived documentation for Angular v4. Please visit [angular.io](#) to see documentation for the current version of Angular.

Hierarchical Dependency Injectors

[Contents >](#)

[The injector tree](#)

[Injector bubbling](#)

[Re-providing a service at different levels](#)

•••

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

☞ The injector tree

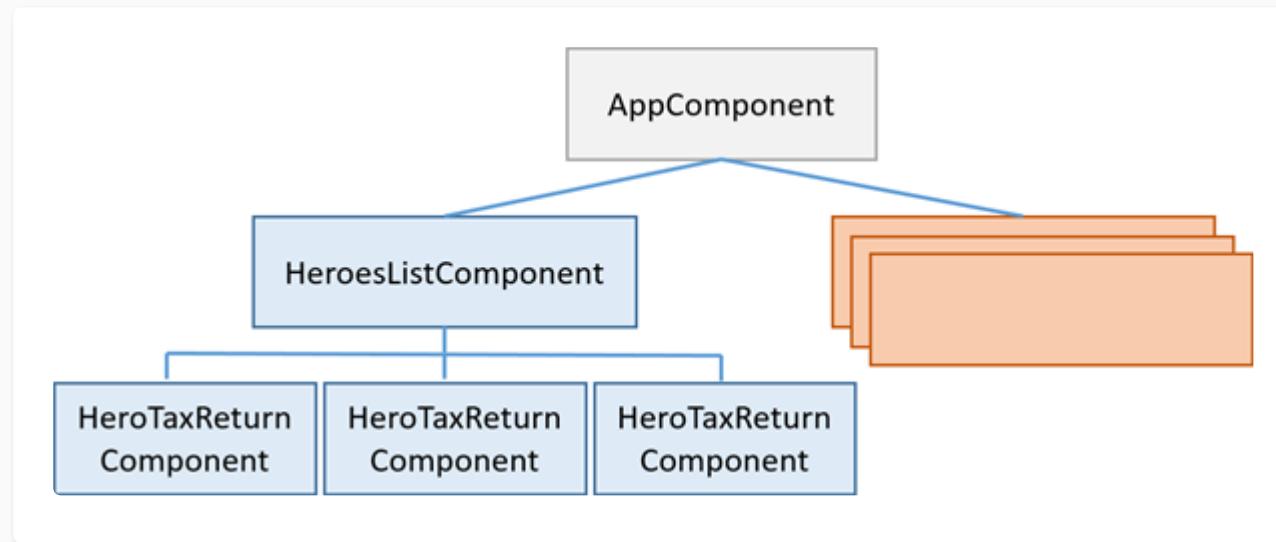
In the [Dependency Injection](#) guide, you learned how to configure a dependency injector and how to retrieve dependencies where you need them.

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:

src/app/villains-list.component.ts (metadata)

```
@Component({
  selector: 'villains-list',
  templateUrl: './villains-list.component.html',
  providers: [ VillainsService ]
})
```

By providing `VillainsService` in the `VillainsListComponent` metadata and nowhere else, the service becomes available only in the `VillainsListComponent` and its sub-component tree. It's still a singleton, but it's a singleton that exists solely in the *villain* domain.

Now you know that a hero component can't access it. You've reduced your exposure to error.

Scenario: multiple edit sessions

Many applications allow users to work on several open tasks at the same time. For example, in a tax preparation application, the preparer could be working on several tax returns, switching from one to the other throughout the day.

This guide demonstrates that scenario with an example in the Tour of Heroes theme. Imagine an outer `HeroListComponent` that displays a list of super heroes.

To open a hero's tax return, the preparer clicks on a hero name, which opens a component for editing that return. Each selected hero tax return opens in its own component and multiple returns can be open at the same time.

Each tax return component has the following characteristics:

- Is its own tax return editing session.
- Can change a tax return without affecting a return in another component.
- Has the ability to save the changes to its tax return or cancel them.

Hero Tax Returns

- RubberMan 
- Tornado

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.

src/app/hero-tax-return.service.ts

```
1. import { Injectable } from '@angular/core';
```



```
2. import { HeroTaxReturn } from './hero';
3. import { HeroesService } from './heroes.service';
4.
5. @Injectable()
6. export class HeroTaxReturnService {
7.   private currentTaxReturn: HeroTaxReturn;
8.   private originalTaxReturn: HeroTaxReturn;
9.
10.  constructor(private heroService: HeroesService) { }
11.
12.  set taxReturn (htr: HeroTaxReturn) {
13.    this.originalTaxReturn = htr;
14.    this.currentTaxReturn = htr.clone();
15.  }
16.
17.  get taxReturn (): HeroTaxReturn {
18.    return this.currentTaxReturn;
19.  }
20.
21.  restoreTaxReturn() {
22.    this.taxReturn = this.originalTaxReturn;
23.  }
24.
25.  saveTaxReturn() {
26.    this.taxReturn = this.currentTaxReturn;
27.    this.heroService.saveTaxReturn(this.currentTaxReturn).subscribe();
28.  }
29. }
```

Here is the `HeroTaxReturnComponent` that makes use of it.

src/app/hero-tax-return.component.ts

```
1. import { Component, EventEmitter, Input, Output } from '@angular/core';
2. import { HeroTaxReturn }          from './hero';
3. import { HeroTaxReturnService }   from './hero-tax-return.service';
4.
5. @Component({
6.   selector: 'hero-tax-return',
7.   templateUrl: './hero-tax-return.component.html',
8.   styleUrls: [ './hero-tax-return.component.css' ],
9.   providers: [ HeroTaxReturnService ]
10. })
11. export class HeroTaxReturnComponent {
12.   message = '';
13.   @Output() close = new EventEmitter<void>();
14.
15.   get taxReturn(): HeroTaxReturn {
16.     return this.heroTaxReturnService.taxReturn;
17.   }
18.   @Input()
19.   set taxReturn (htr: HeroTaxReturn) {
20.     this.heroTaxReturnService.taxReturn = htr;
21.   }
22.
23.   constructor(private heroTaxReturnService: HeroTaxReturnService) { }
24.
25.   onCanceled()  {
26.     this.flashMessage('Canceled');
27.     this.heroTaxReturnService.restoreTaxReturn();
```

```
28.  };
29.
30.  onClose() { this.close.emit(); };
31.
32.  onSaved() {
33.    this.flashMessage('Saved');
34.    this.heroTaxReturnService.saveTaxReturn();
35.  }
36.
37.  flashMessage(msg: string) {
38.    this.message = msg;
39.    setTimeout(() => this.message = '', 500);
40.  }
41. }
```

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.

src/app/hero-tax-return.component.ts (providers)

```
providers: [ HeroTaxReturnService ]
```

The `HeroTaxReturnComponent` has its own provider of the `HeroTaxReturnService`. Recall that every component *instance* has its own injector. Providing the service at the component level ensures that *every* instance of the component gets its own, private instance of the service. 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 [live example](#) / [download example](#).

Scenario: specialized providers

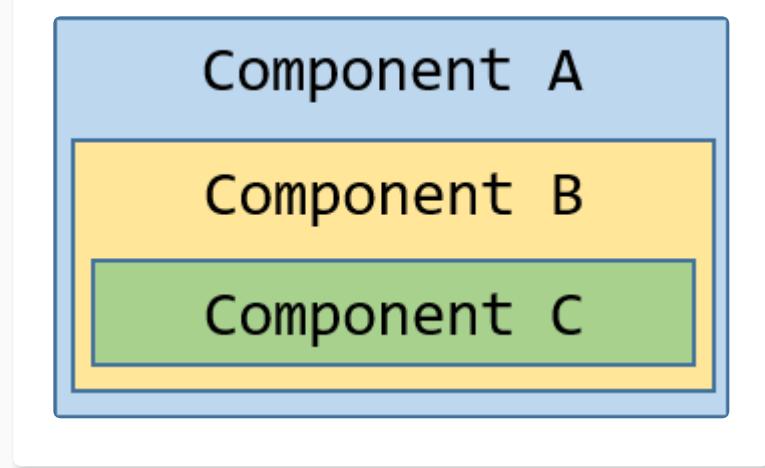
Another reason to re-provide a service is to substitute a *more specialized* implementation of that service, deeper in the component tree.

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.

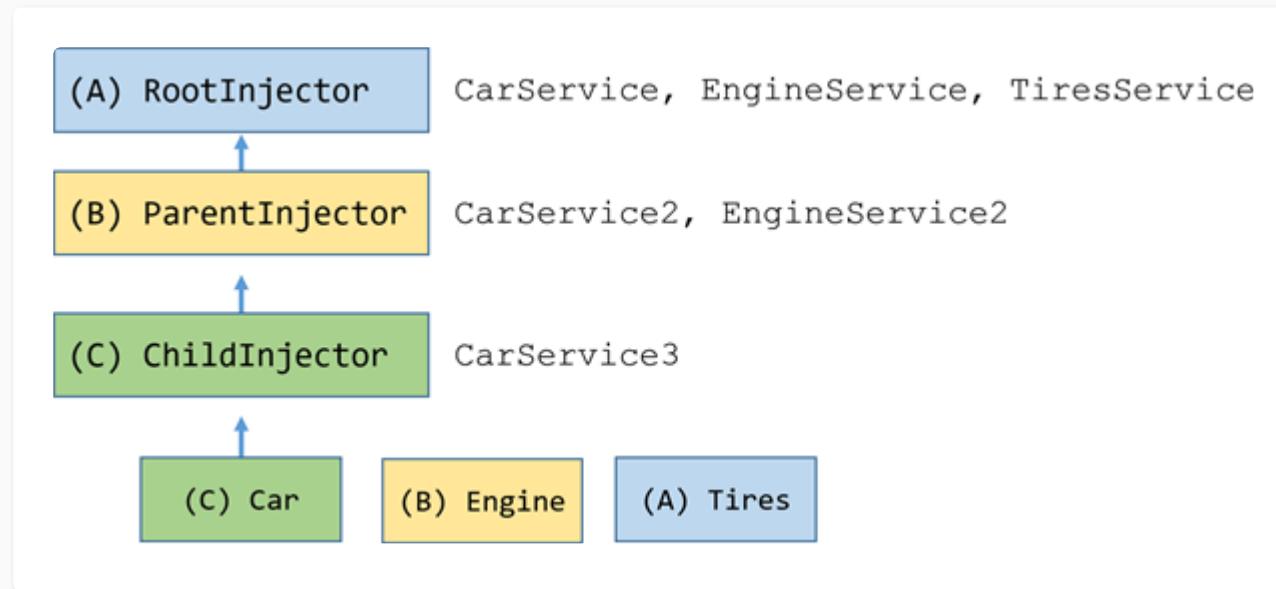
Then you create a child component (B) that defines its own, *specialized* providers for `CarService` and `EngineService` that have special capabilities suitable for whatever is going on in component (B).

Component (B) is the parent of another component (C) that defines its own, even *more specialized* provider for `CarService`.



Behind the scenes, each component sets up its own injector with zero, one, or more providers defined for that component itself.

When you resolve an instance of `Car` at the deepest component (C), its injector produces an instance of `Car` resolved by injector (C) with an `Engine` resolved by injector (B) and `Tires` resolved by the root injector (A).



The code for this *cars* scenario is in the `car.components.ts` and `car.services.ts` files of the sample which you can review and download from the [live example / download example](#).

This is the archived documentation for Angular v4. Please visit [angular.io](#) to see documentation for the current version of Angular.

Dependency Injection

[Contents >](#)

[Application-wide dependencies](#)

[External module configuration](#)

[@Injectable\(\) and nested service dependencies](#)

•••

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

See the [live example](#) / [download example](#) of the code in this cookbook.

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.

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

```
import { LoggerService }      from './logger.service';
import { UserContextService } from './user-context.service';
import { UserService }        from './user.service';

@Component({
  selector: 'my-app',
  templateUrl: './app.component.html',
  providers: [ LoggerService, UserContextService, UserService ]
})
export class AppComponent {
  /* . . . */
}
```

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.

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

src/app/hero-bios.component.ts (component constructor injection)

```
constructor(logger: LoggerService) {
  logger.logInfo('Creating HeroBiosComponent');
}
```

src/app/user-context.service.ts (service constructor injection)

```
constructor(private userService: UserService, private loggerService: LoggerService)  
{  
}
```



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`.

src/app/app.module.ts (providers)

```
providers: [  
  { provide: LocationStrategy, useClass: HashLocationStrategy }  
]
```



`@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`.

src/app/app.component.ts

```
constructor(logger: LoggerService, public userContext: UserContextService) {  
  userContext.loadUser(this.userId);  
  logger.logInfo('AppComponent initialized');  
}
```

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

user-context.service.ts (injection)

```
@Injectable()  
export class UserContextService {  
  constructor(private userService: UserService, private loggerService:  
    LoggerService) {  
  }  
}
```

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

@Injectable()

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

user-context.service.ts (@Injectable)

```
@Injectable()  
export class UserContextService {  
}
```

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.

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:

src/app/sorted-heroes.component.ts (HeroesBaseComponent excerpt)

```
@Component({
  selector: 'unsorted-heroes',
  template: `<div *ngFor="let hero of heroes">{{hero.name}}</div>`,
  providers: [HeroService]
})
export class HeroesBaseComponent implements OnInit {
  constructor(private heroService: HeroService) { }
}
```

When Angular creates 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.

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.

Imagine a `HeroBiosComponent` that presents three instances of the `HeroBioComponent`.

ap/hero-bios.component.ts

```
@Component({
  selector: 'hero-bios',
  template: `
    <hero-bio [heroId]="1"></hero-bio>
    <hero-bio [heroId]="2"></hero-bio>
    <hero-bio [heroId]="3"></hero-bio>`,
  providers: [HeroService]
})
export class HeroBiosComponent {
```

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.

src/app/hero-cache.service.ts

```
1. @Injectable()
2. export class HeroCacheService {
3.   hero: Hero;
4.   constructor(private heroService: HeroService) {}
5.
6.   fetchCachedHero(id: number) {
7.     if (!this.hero) {
8.       this.hero = this.heroService.getHeroById(id);
9.     }
10.    return this.hero;
11.  }
12. }
```

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.

src/app/hero-bio.component.ts

```
1. @Component({
2.   selector: 'hero-bio',
3.   template: `
4.     <h4>{{hero.name}}</h4>
5.     <ng-content></ng-content>
```

```
6.      <textarea cols="25" [(ngModel)]="hero.description"></textarea>`,  
7.      providers: [HeroCacheService]  
8.    })  
9.  
10.   export class HeroBioComponent implements OnInit {  
11.     @Input() heroId: number;  
12.  
13.     constructor(private heroCache: HeroCacheService) { }  
14.  
15.     ngOnInit() { this.heroCache.fetchCachedHero(this.heroId); }  
16.  
17.     get hero() { return this.heroCache.hero; }  
18.   }
```

The parent `HeroBiosComponent` binds a value to 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](#) / [download example](#) and confirm that the three `HeroBioComponent` instances have their own cached hero data.

Hero Bios

RubberMan

Hero of many talents

Magma

Hero of all trades

Mr. Nice

The name says it all

Qualify dependency lookup with `@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.

Demonstration

The `HeroBiosAndContactsComponent` is a revision of the `HeroBiosComponent` that you looked at [above](#).

src/app/hero-bios.component.ts (HeroBiosAndContactsComponent)

```
1. @Component({
2.   selector: 'hero-bios-and-contacts',
3.   template: `
4.     <hero-bio [heroId]="1"> <hero-contact></hero-contact> </hero-bio>
5.     <hero-bio [heroId]="2"> <hero-contact></hero-contact> </hero-bio>
6.     <hero-bio [heroId]="3"> <hero-contact></hero-contact> </hero-bio>`,
7.   providers: [HeroService]
8. })
9. export class HeroBiosAndContactsComponent {
10.   constructor(logger: LoggerService) {
11.     logger.logInfo('Creating HeroBiosAndContactsComponent');
12.   }
13. }
```

Focus on the template:

dependency-injection-in-action/src/app/hero-bios.component.ts

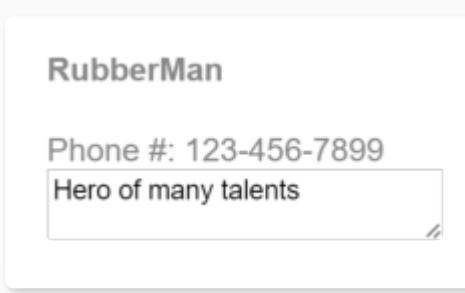
```
template: `<hero-bio [heroId]="1"> <hero-contact></hero-contact> </hero-bio>
<hero-bio [heroId]="2"> <hero-contact></hero-contact> </hero-bio>
<hero-bio [heroId]="3"> <hero-contact></hero-contact> </hero-bio>`,
```

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:

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

```
template: `<h4>{{hero.name}}</h4>
<ng-content></ng-content>
<textarea cols="25" [(ngModel)]="hero.description"></textarea>`,
```

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:

src/app/hero-contact.component.ts

```
1. @Component({
```

```
2.   selector: 'hero-contact',
3.   template: `
4.     <div>Phone #: {{phoneNumber}}
5.     <span *ngIf="hasLogger">!!!</span></div>
6.   )
7. export class HeroContactComponent {
8.
9.   hasLogger = false;
10.
11.  constructor(
12.    @Host() // limit to the host component's instance of the
13.    HeroCacheService
14.    private heroCache: HeroCacheService,
15.    @Host() // limit search for logger; hides the application-wide
16.    logger
17.    @Optional() // ok if the logger doesn't exist
18.    private loggerService: LoggerService
19.  ) {
20.    if (loggerService) {
21.      this.hasLogger = true;
22.      loggerService.logInfo('HeroContactComponent can log!');
23.    }
24.
25.    get phoneNumber() { return this.heroCache.hero.phone; }
26.
27.  }
```

Focus on the constructor parameters:

src/app/hero-contact.component.ts

```
@Host() // limit to the host component's instance of the HeroCacheService  
private heroCache: HeroCacheService,  
  
@Host() // limit search for logger; hides the application-wide logger  
@Optional() // ok if the logger doesn't exist  
private loggerService: LoggerService
```



The `@Host()` function decorating the `heroCache` 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)
```

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.

src/app/highlight.directive.ts

```
1. import { Directive, ElementRef, HostListener, Input } from '@angular/core'; 
2.
3. @Directive({
4.   selector: '[myHighlight]'
5. })
6. export class HighlightDirective {
7.
8.   @Input('myHighlight') highlightColor: string;
9.
10.  private el: HTMLElement;
11.
12.  constructor(el: ElementRef) {
13.    this.el = el.nativeElement;
14.  }
15.
16.  @HostListener('mouseenter') onMouseEnter() {
17.    this.highlight(this.highlightColor || 'cyan');
```

```
18.  }
19.
20. @HostListener('mouseleave') onMouseLeave() {
21.   this.highlight(null);
22. }
23.
24. private highlight(color: string) {
25.   this.el.style.backgroundColor = color;
26. }
27. }
```

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

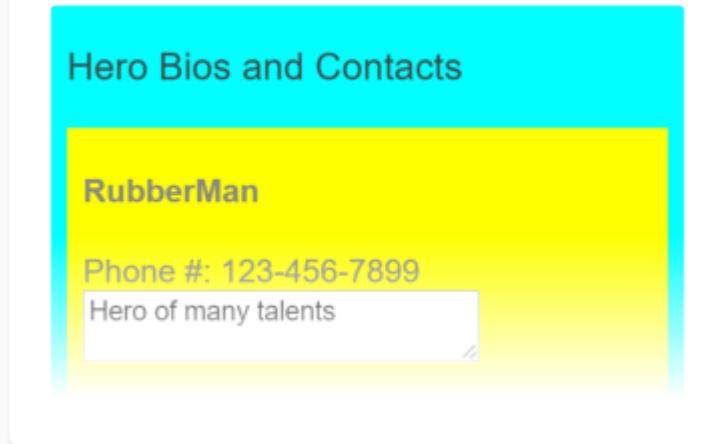
Angular sets the constructor's `el` parameter to the injected `ElementRef`, which is a wrapper around that DOM element. Its `nativeElement` property exposes the DOM element for the directive to manipulate.

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.

src/app/app.component.html (highlight)

```
<div id="highlight" class="di-component" myHighlight>
  <h3>Hero Bios and Contacts</h3>
  <div myHighlight="yellow">
    <hero-bios-and-contacts></hero-bios-and-contacts>
  </div>
</div>
```

The following image shows the effect of mousing over the `<hero-bios-and-contacts>` tag.



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:

src/app/hero-bios.component.ts (component constructor injection)

```
constructor(logger: LoggerService) {  
  logger.logInfo('Creating HeroBiosComponent');  
}
```

Angular asks the injector for the service associated with 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](#).

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:

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

```
providers: [ LoggerService, UserContextService, UserService ]
```

Defining providers

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

src/app/hero-bios.component.ts (class provider)

```
providers: [ HeroService ]
```

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.

hero-of-the-month.component.ts

```
1. import { Component, Inject } from '@angular/core';
2.
3. import { DateLoggerService } from './date-logger.service';
4. import { Hero } from './hero';
5. import { HeroService } from './hero.service';
6. import { LoggerService } from './logger.service';
7. import { MinimalLogger } from './minimal-logger.service';
8. import { RUNNERS_UP,
9.         runnersUpFactory } from './runners-up';
10.
11. @Component({
12.   selector: 'hero-of-the-month',
```

```

13.   templateUrl: './hero-of-the-month.component.html',
14.   providers: [
15.     { provide: Hero,           useValue: someHero },
16.     { provide: TITLE,         useValue: 'Hero of the Month' },
17.     { provide: HeroService,   useClass: HeroService },
18.     { provide: LoggerService, useClass: DateLoggerService },
19.     { provide: MinimalLogger, useExisting: LoggerService },
20.     { provide: RUNNERS_UP,    useFactory: runnersUpFactory(2), deps: [Hero,
21.       HeroService] }
22.   ],
23.   export class HeroOfTheMonthComponent {
24.     logs: string[] = [];
25.
26.     constructor(
27.       logger: MinimalLogger,
28.       public heroOfTheMonth: Hero,
29.       @Inject(RUNNERS_UP) public runnersUp: string,
30.       @Inject(TITLE) public title: string)
31.     {
32.       this.logs = logger.logs;
33.       logger.logInfo('starting up');
34.     }
35.   }

```

The `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.

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:

dependency-injection-in-action/src/app/hero-of-the-month.component.ts

```
{ provide: Hero,           useValue:    someHero },
{ provide: TITLE,          useValue:    'Hero of the Month' },
```



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 particularly helpful when the dependency is a simple value like a string, a number, or a function.

The value of a *value provider* must be defined *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:

dependency-injection-in-action/src/app/hero-of-the-month.component.ts

```
const someHero = new Hero(42, 'Magma', 'Had a great month!', '555-555-5555');
```



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

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`:

dependency-injection-in-action/src/app/hero-of-the-month.component.ts

```
{ provide: HeroService, useClass: HeroService },
{ provide: LoggerService, useClass: DateLoggerService },
```

The first provider is the *de-sugared*, expanded form of the most typical case in which the class to be created (`HeroService`) is also the provider's dependency injection token. 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:

src/app/date-logger.service.ts

```
@Injectable()
export class DateLoggerService extends LoggerService
{
  logInfo(msg: any) { super.logInfo(stamp(msg)); }
  logDebug(msg: any) { super.logInfo(stamp(msg)); }
  logError(msg: any) { super.logError(stamp(msg)); }

  function stamp(msg: any) { return msg + ' at ' + new Date(); }
}
```



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*.

dependency-injection-in-action/src/app/hero-of-the-month.component.ts

```
{ provide: MinimalLogger, useExisting: LoggerService },
```



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:

src/app/minimal-logger.service.ts

```
// Class used as a "narrowing" interface that exposes a minimal logger
// Other members of the actual implementation are invisible
export abstract class MinimalLogger {
  logs: string[];
  logInfo: (msg: string) => void;
}
```

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

src/app/hero-of-the-month.component.ts (minimal version)

```
@Component({
  selector: 'hero-of-the-month',
  templateUrl: './hero-of-the-month.component.html',
  // Todo: move this aliasing, `useExisting` provider to the AppModule
  providers: [{ provide: MinimalLogger, useExisting: LoggerService }]
})
export class HeroOfTheMonthComponent {
  logs: string[] = [];
  constructor(logger: MinimalLogger) {
    logger.logInfo('starting up');
  }
}
```

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

```
this.logs = logger.logs;
logger.logInfo('sta ⚡ logInfo (method) MinimalLogger.logInfo(msg: string): void
logs
```

Behind the scenes, Angular 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)
```

useFactory—the factory provider

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

dependency-injection-in-action/src/app/hero-of-the-month.component.ts

```
{ provide: RUNNERS_UP,      useFactory: runnersUpFactory(2), deps: [Hero,
HeroService] }
```

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.

runners-up.ts (excerpt)

```
export function runnersUpFactory(take: number) {
  return (winner: Hero, heroService: HeroService): string => {
    /* ... */
  };
}
```



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 [live example](#) / [download example](#) for the full source code.

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.

class-interface

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

dependency-injection-in-action/src/app/hero-of-the-month.component.ts

```
{ provide: MinimalLogger, useExisting: LoggerService },
```



The `MinimalLogger` is an abstract class.

dependency-injection-in-action/src/app/minimal-logger.service.ts

```
// Class used as a "narrowing" interface that exposes a minimal logger
// Other members of the actual implementation are invisible
export abstract class MinimalLogger {
  logs: string[];
  logInfo: (msg: string) => void;
}
```



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:

dependency-injection-in-action/src/app/minimal-logger.service.ts

```
var MinimalLogger = (function () {
  function MinimalLogger() {}
  return MinimalLogger;
}());
exports("MinimalLogger", MinimalLogger);
```

Notice that it doesn't have 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.

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.

dependency-injection-in-action/src/app/hero-of-the-month.component.ts

```
{ provide: TITLE,           useValue: 'Hero of the Month' },
{ provide: RUNNERS_UP,     useFactory: runnersUpFactory(2), deps: [Hero,
HeroService] }
```

You created the `TITLE` token like this:

dependency-injection-in-action/src/app/hero-of-the-month.component.ts

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

export const TITLE = new InjectionToken<string>('title');
```

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

Inject into a derived class

Take care when writing a component that inherits from another component. If the base component has injected dependencies, you must re-provide and re-inject them in the derived class and then pass them down to the base class through the constructor.

In this contrived example, `SortedHeroesComponent` inherits from `HeroesBaseComponent` to display a *sorted* list of heroes.



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.

src/app/sorted-heroes.component.ts (HeroesBaseComponent)

```
1. @Component({
2.   selector: 'unsorted-heroes',
3.   template: `<div *ngFor="let hero of heroes">{{hero.name}}</div>`,
4.   providers: [HeroService]
5. })
6. export class HeroesBaseComponent implements OnInit {
```

```
7.  constructor(private heroService: HeroService) { }
8.
9.  heroes: Array<Hero>;
10.
11. ngOnInit() {
12.   this.heroes = this.heroService.getAllHeroes();
13.   this.afterGetHeroes();
14. }
15.
16. // Post-process heroes in derived class override.
17. protected afterGetHeroes(){}
18.
19. }
```

Keep constructors simple. 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.

src/app/sorted-heroes.component.ts (SortedHeroesComponent)

```
1. @Component({
2.   selector: 'sorted-heroes',
```

```
3.   template: `<div *ngFor="let hero of heroes">{{hero.name}}</div>`,  
4.   providers: [HeroService]  
5. })  
6. export class SortedHeroesComponent extends HeroesBaseComponent {  
7.   constructor(heroService: HeroService) {  
8.     super(heroService);  
9.   }  
10.  
11.  protected afterGetHeroes() {  
12.    this.heroes = this.heroes.sort((h1, h2) => {  
13.      return h1.name < h2.name ? -1 :  
14.        (h1.name > h2.name ? 1 : 0);  
15.    });  
16.  }  
17. }
```

Now take note of the `afterGetHeroes()` method. Your first instinct might have been to create an `ngOnInit` method in `SortedHeroesComponent` and do the sorting there. But Angular calls the *derived* class's `ngOnInit` *before* calling the base class's `ngOnInit` so you'd be sorting the heroes array *before they arrived*. That produces a nasty error.

Overriding the base class's `afterGetHeroes()` method solves the problem.

These complications argue for *avoiding component inheritance*.

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.

Find a parent component of known type

You use standard class injection to acquire a parent component whose type you know.

In the following example, the parent `AlexComponent` has several children including a `CathyComponent`:

parent-finder.component.ts (AlexComponent v.1)

```
@Component({
  selector: 'alex',
  template: `
    <div class="a">
      <h3>{{name}}</h3>
      <cathy></cathy>
      <craig></craig>
      <carol></carol>
    </div>`,
})
export class AlexComponent extends Base
{
  name= 'Alex';
```

```
}
```

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

parent-finder.component.ts (CathyComponent)

```
@Component({
  selector: 'cathy',
  template: `
    <div class="c">
      <h3>Cathy</h3>
      {{alex ? 'Found' : 'Did not find'}} Alex via the component class.<br>
    </div>`
})
export class CathyComponent {
  constructor( @Optional() public alex: AlexComponent ) { }
}
```



Notice that even though the `@Optional` qualifier is there for safety, the [live example](#) / [download example](#) confirms that the `alex` parameter is set.

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`.

parent-finder.component.ts (Alex class signature)

```
export class AlexComponent extends Base
```

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

parent-finder.component.ts (CraigComponent)

```
@Component({
  selector: 'craig',
  template: `
    <div class="c">
      <h3>Craig</h3>
      {{alex ? 'Found' : 'Did not find'}} Alex via the base class.
    </div>`
})
```

```
export class CraigComponent {  
  constructor( @Optional() public alex: Base ) { }  
}
```

Unfortunately, this does not work. The [live example](#) / [download example](#) confirms that the `alex` parameter is null. *You cannot inject a parent by its base class.*

Find a parent by its class-interface

You can find a parent component with a [class-interface](#).

The parent must cooperate by providing an *alias* to itself in the name of a *class-interface* token.

Recall that Angular always adds a component instance to its own injector; that's why you could inject *Alex* into *Cathy* [earlier](#).

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

parent-finder.component.ts (AlexComponent providers)

```
providers: [{ provide: Parent, useExisting: forwardRef(() => AlexComponent) }],
```



`Parent` is the provider's *class-interface* token. The `forwardRef` breaks the circular reference you just created by having the `AlexComponent` refer to itself.

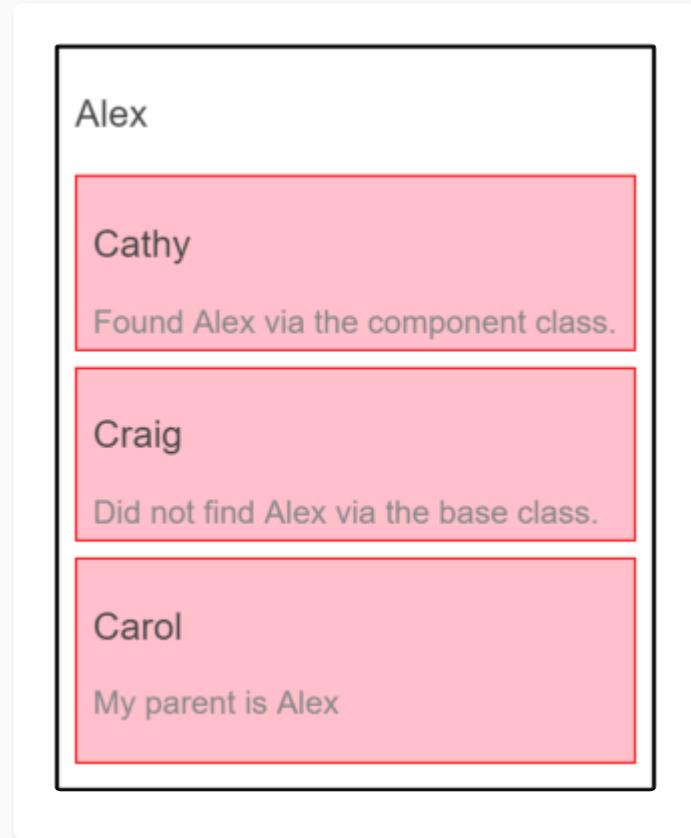
Carol, the third of *Alex*'s child components, injects the parent into its `parent` parameter, the same way you've done it before:

parent-finder.component.ts (CarolComponent class)

```
export class CarolComponent {  
  name= 'Carol';  
  constructor( @Optional() public parent: Parent ) { }  
}
```



Here's *Alex* and family in action:



Find 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*:

parent-finder.component.ts (BarryComponent)

```
const templateB = `

<div class="b">
  <div>
    <h3>{{name}}</h3>
    <p>My parent is {{parent?.name}}</p>
  </div>
  <carol></carol>
  <chris></chris>
</div>`;

@Component({
  selector: 'barry',
  template: templateB,
  providers: [{ provide: Parent, useExisting: forwardRef(() => BarryComponent) }]
})
export class BarryComponent implements Parent {
  name = 'Barry';
  constructor( @SkipSelf() @Optional() public parent: Parent ) { }
}
```

Barry's `providers` array looks just like *Alex*'s. If you're going to keep writing *alias providers* like this you should create a [helper function](#).

For now, focus on *Barry*'s constructor:

Barry's constructor

Carol's constructor

```
constructor( @SkipSelf() @Optional() public parent: Parent ) { }
```



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

`@SkipSelf` is essential for two reasons:

1. It tells the injector to start its search for a `Parent` dependency in a component *above* itself, which *is* what parent means.
2. Angular throws a cyclic dependency error if you omit the `@SkipSelf` decorator.

```
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

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*.

parent-finder.component.ts (Parent class-interface)

```
export abstract class Parent { name: string; }
```

The `Parent` *class-interface* defines a `name` property with a type declaration but *no implementation*. The `name` property is the only member of a parent component that a child component can call. Such a narrow interface helps decouple the child component class from its parent components.

A component that could serve as a parent *should* implement the *class-interface* as the `AliceComponent` does:

parent-finder.component.ts (AliceComponent class signature)

```
export class AliceComponent implements Parent
```

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

parent-finder.component.ts (AlexComponent class signature)

```
export class AlexComponent extends Base
```

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()` helper function

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

dependency-injection-in-action/src/app/parent-finder.component.ts

```
providers: [{ provide: Parent, useExisting: forwardRef(() => AlexComponent) }],
```

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

dependency-injection-in-action/src/app/parent-finder.component.ts

```
// Helper method to provide the current component instance in the name of a
'parentType'.
const provideParent =
  (component: any) => {
  return { provide: Parent, useExisting: forwardRef(() => component) };
};
```

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

dependency-injection-in-action/src/app/parent-finder.component.ts

```
providers: [ provideParent(AliceComponent) ]
```

You can do better. The current version of the helper function can only alias the `Parent` *class-interface*. The application might have a variety of parent types, each with its own *class-interface* token.

Here's a revised version that defaults to `parent` but also accepts an optional second parameter for a different parent *class-interface*.

dependency-injection-in-action/src/app/parent-finder.component.ts

```
// Helper method to provide the current component instance in the name of a
// `parentType`.
// The `parentType` defaults to `Parent` when omitting the second parameter.
const provideParent =
  (component: any, parentType?: any) => {
  return { provide: parentType || Parent, useExisting: forwardRef(() => component)
};

}
```

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

dependency-injection-in-action/src/app/parent-finder.component.ts

```
providers: [ provideParent(BethComponent, DifferentParent) ]
```

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` :

parent-finder.component.ts (AlexComponent providers)

```
providers: [{ provide: Parent, useExisting: forwardRef(() => AlexComponent) }],
```

This is the archived documentation for Angular v4. Please visit [angular.io](#) to see documentation for the current version of Angular.

HttpClient

[Contents >](#)

[Setup: installing the module](#)

[Making a request for JSON data](#)

[Typechecking the response](#)

•••

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.

```
1. // app.module.ts:
```



```
2.  
3. import {NgModule} from '@angular/core';  
4. import {BrowserModule} from '@angular/platform-browser';  
5.  
6. // Import HttpClientModule from @angular/common/http  
7. import {HttpClientModule} from '@angular/common/http';  
8.  
9. @NgModule({  
10.   imports: [  
11.     BrowserModule,  
12.     // Include it under 'imports' in your application module  
13.     // after BrowserModule.  
14.     HttpClientModule,  
15.   ],  
16. })  
17. export class AppModule {}
```

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.

```
1. @Component(...)  
2. export class MyComponent implements OnInit {  
3.  
4.   results: string[];  
5.  
6.   // Inject HttpClient into your component or service.  
7.   constructor(private http: HttpClient) {}  
8.  
9.   ngOnInit(): void {  
10.     // Make the HTTP request:  
11.     this.http.get('/api/items').subscribe(data => {  
12.       // Read the result field from the JSON response.  
13.       this.results = data['results'];  
14.     });  
15.   }  
16. }
```



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.

```
1. http
2.   .get<ItemsResponse>('/api/items')
3.   .subscribe(
4.     data => {...},
5.     (err: HttpErrorResponse) => {
6.       if (err.error instanceof Error) {
7.         // A client-side or network error occurred. Handle it accordingly.
8.         console.log('An error occurred:', err.error.message);
9.       } else {
10.         // The backend returned an unsuccessful response code.
11.         // The response body may contain clues as to what went wrong,
12.         console.log(`Backend returned code ${err.status}, body was:
${err.error}`);
13.       }
14.     }
15.   );
```

`.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('/textfield.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:

```
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:

```
1. import {Injectable} from '@angular/core';  
2. import {HttpEvent, HttpInterceptor, HttpHandler, HttpRequest} from  
   '@angular/common/http';  
3.  
4. @Injectable()  
5. export class AuthInterceptor implements HttpInterceptor {
```

```
6.  constructor(private auth: AuthService) {}
7.
8.  intercept(req: HttpRequest<any>, next: HttpHandler):
   Observable<HttpEvent<any>> {
9.    // Get the auth header from the service.
10.   const authHeader = this.auth.getAuthorizationHeader();
11.   // Clone the request to add the new header.
12.   const authReq = req.clone({headers: req.headers.set('Authorization',
13.     authHeader)}));
14.   // Pass on the cloned request instead of the original request.
15.   return next.handle(authReq);
16. }
```

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:

```
1. import 'rxjs/add/operator/do';
```



```
2.  
3. export class TimingInterceptor implements HttpInterceptor {  
4.   constructor(private auth: AuthService) {}  
5.  
6.   intercept(req: HttpRequest<any>, next: HttpHandler):  
    Observable<HttpEvent<any>> {  
7.     const started = Date.now();  
8.     return next  
9.       .handle(req)  
10.      .do(event => {  
11.        if (event instanceof HttpResponse) {  
12.          const elapsed = Date.now() - started;  
13.          console.log(`Request for ${req.urlWithParams} took ${elapsed}  
ms.`);  
14.        }  
15.      });  
16.    }  
17. }
```

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.

```
1. @Injectable()
2. export class CachingInterceptor implements HttpInterceptor {
3.   constructor(private cache: HttpCache) {}
4.
5.   intercept(req: HttpRequest<any>, next: HttpHandler):
6.     Observable<HttpEvent<any>> {
7.     // Before doing anything, it's important to only cache GET requests.
8.     // Skip this interceptor if the request method isn't GET.
9.     if (req.method !== 'GET') {
10.       return next.handle(req);
11.     }
12.     // First, check the cache to see if this request exists.
13.     const cachedResponse = this.cache.get(req);
14.     if (cachedResponse) {
15.       // A cached response exists. Serve it instead of forwarding
16.       // the request to the next handler.
17.       return Observable.of(cachedResponse);
18.     }
}
```

```
19.  
20.    // No cached response exists. Go to the network, and cache  
21.    // the response when it arrives.  
22.    return next.handle(req).do(event => {  
23.        // Remember, there may be other events besides just the response.  
24.        if (event instanceof HttpResponse) {  
25.            // Update the cache.  
26.            this.cache.put(req, event);  
27.        }  
28.    });  
29.}  
30.}
```

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.

```
1. intercept(req: HttpRequest<any>, next: HttpHandler):  
   Observable<HttpEvent<any>> {  
2.     // Still skip non-GET requests.  
3.     if (req.method !== 'GET') {  
4.       return next.handle(req);  
5.     }  
6.  
7.     // This will be an Observable of the cached value if there is one,  
8.     // or an empty Observable otherwise. It starts out empty.  
9.     let maybeCachedResponse: Observable<HttpEvent<any>> = Observable.empty();
```

```
10.  
11. // Check the cache.  
12. const cachedResponse = this.cache.get(req);  
13. if (cachedResponse) {  
14.   maybeCachedResponse = Observable.of(cachedResponse);  
15. }  
16.  
17. // Create an Observable (but don't subscribe) that represents making  
18. // the network request and caching the value.  
19. const networkResponse = next.handle(req).do(event => {  
20.   // Just like before, check for the HttpResponse event and cache it.  
21.   if (event instanceof HttpResponse) {  
22.     this.cache.put(req, event);  
23.   }  
24. });  
25.  
26. // Now, combine the two and send the cached response first (if there is  
27. // one), and the network response second.  
28. return Observable.concat(maybeCachedResponse, networkResponse);  
29. }
```

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`

```
1. it('expects a GET request', inject([HttpClient, HttpTestingController], 
  (http: HttpClient, httpMock: HttpTestingController) => {
2.   // Make an HTTP GET request, and expect that it return an object
3.   // of the form {name: 'Test Data'}.
4.   http
5.     .get('/data')
6.     .subscribe(data => expect(data['name']).toEqual('Test Data'));
7.
8.   // At this point, the request is pending, and no response has been
9.   // sent. The next step is to expect that the request happened.
10.  const req = httpMock.expectOne('/data');
```

```
11.  
12. // If no request with that URL was made, or if multiple requests match,  
13. // expectOne() would throw. However this test makes only one request to  
14. // this URL, so it will match and return a mock request. The mock request  
15. // can be used to deliver a response or make assertions against the  
16. // request. In this case, the test asserts that the request is a GET.  
17. expect(req.request.method).toEqual('GET');  
18.  
19. // Next, fulfill the request by transmitting a response.  
20. req.flush({name: 'Test Data'});  
21.  
22. // Finally, assert that there are no outstanding requests.  
23. httpMock.verify();  
24. }));
```

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());
```



This is the archived documentation for Angular v4. Please visit [angular.io](#) to see documentation for the current version of Angular.

Routing & Navigation

[Contents >](#)

[Overview](#)

[The Basics](#)

<base href>

•••

The Angular Router enables navigation from one [view](#) to the next as users perform application tasks.

This guide covers the router's primary features, illustrating them through the evolution of a small application that you can [run live in the browser](#) / [download example](#).

Overview

The browser is a familiar model of application navigation:

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

The Angular Router ("the router") borrows from this model. It can interpret a browser URL as an instruction to navigate to a client-generated view. It can pass optional parameters along to the supporting view component that help it decide what specific content to present. You can bind the router to links on a page and it will navigate to the appropriate application view when the user clicks a link. You can navigate imperatively when the user clicks a button, selects from a drop box, or in response to some other stimulus from any source. And the router logs activity in the browser's history journal so the back and forward buttons work as well.

The Basics

This guide proceeds in phases, marked by milestones, starting from a simple two-pager and building toward a modular, multi-view design with child routes.

An introduction to a few core router concepts will help orient you to the details that follow.

`<base href>`

Most routing applications should add a `<base>` element to the `index.html` as the first child in the `<head>` tag to tell the router how to compose navigation URLs.

If the `app` folder is the application root, as it is for the sample application, set the `href` value *exactly* as shown here.

src/index.html (base-href)

```
<base href="/">
```



Router imports

The Angular Router is an optional service that presents a particular component view for a given URL. It is not part of the Angular core. It is in its own library package, `@angular/router`. Import what you need from it as you would from any other Angular package.

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

```
import { RouterModule, Routes } from '@angular/router';
```



You'll learn about more options in the [details below](#).

Configuration

A routed Angular application has one singleton instance of the `Router` service. When the browser's URL changes, that router looks for a corresponding `Route` from which it can determine the component to display.

A router has no routes until you configure it. The following example creates four route definitions, configures the router via the `RouterModule.forRoot` method, and adds the result to the `AppModule`'s `imports` array.

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

```
const appRoutes: Routes = [
  { path: 'crisis-center', component: CrisisListComponent },
  { path: 'hero/:id',      component: HeroDetailComponent },
  {
    path: 'heroes',
```



```

        component: HeroListComponent,
        data: { title: 'Heroes List' }
    },
    { path: '',
      redirectTo: '/heroes',
      pathMatch: 'full'
    },
    { path: '**', component: PageNotFoundComponent }
];
}

@NgModule({
  imports: [
    RouterModule.forRoot(
      appRoutes,
      { enableTracing: true } // <-- debugging purposes only
    )
    // other imports here
  ],
  ...
})
export class AppModule { }

```

The `appRoutes` array of *routes* describes how to navigate. Pass it to the `RouterModule.forRoot` method in the module `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.

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 -->
```



Router links

Now you have routes configured and a place to render them, but how do you navigate? The URL could arrive directly from the browser address bar. But most of the time you navigate as a result of some user action such as the click of an anchor tag.

Consider the following template:

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

```
template: `
  <h1>Angular Router</h1>
  <nav>
    <a routerLink="/crisis-center" routerLinkActive="active">Crisis Center</a>
    <a routerLink="/heroes" routerLinkActive="active">Heroes</a>
  </nav>
  <router-outlet></router-outlet>
`
```



The `RouterLink` directives on the anchor tags give the router control over those elements. The navigation paths are fixed, so you can assign a string to the `routerLink` (a "one-time" binding).

Had the navigation path been more dynamic, you could have bound to a template expression that returned an array of route link parameters (the *link parameters array*). The router resolves that array into a complete URL.

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.

Router state

After the end of each successful navigation lifecycle, the router builds a tree of `ActivatedRoute` objects that make up the current state of the router. You can access the current `RouterState` from anywhere in the application using the `Router` service and the `routerState` property.

Each `ActivatedRoute` in the `RouterState` provides methods to traverse up and down the route tree to get information from parent, child and sibling routes.

Activated route

The route path and parameters are available through an injected router service called the `ActivatedRoute`. It has a great deal of useful information including:

Property	Description
<code>url</code>	An <code>Observable</code> of the route path(s), represented as an array of strings for each part of the route path.
<code>data</code>	An <code>Observable</code> that contains the <code>data</code> object provided for the route. Also contains any resolved values from the <code>resolve</code> guard.
<code>paramMap</code>	An <code>Observable</code> that contains a <code>map</code> of the required and <code>optional parameters</code> specific to the route. The map supports retrieving single and multiple values from

the same parameter.

`queryParamMap` An `Observable` that contains a `map` of the [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](#) 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](#).

`firstChild` Contains the first `ActivatedRoute` in the list of this route's child routes.

`children` Contains all the [child routes](#) activated under the current route.

Two older properties are still available. They are less capable than their replacements, discouraged, and may be deprecated in a future Angular version.

`params` – An `Observable` that contains the required and [optional parameters](#) specific to the route.
Use `paramMap` instead.

`queryParams` – An `Observable` that contains the [query parameters](#) available to all routes. Use `queryParamMap` instead.

Router events

During each navigation, the `Router` emits navigation events through the `Router.events` property. These events range from when the navigation starts and ends to many points in between. The full list of navigation events is displayed in the table below.

Router Event	Description
<code>NavigationStart</code>	An event triggered when navigation starts.
<code>RoutesRecognized</code>	An event triggered when the Router parses the URL and the routes are recognized.
<code>RouteConfigLoadStart</code>	An event triggered before the <code>Router</code> lazy loads a route configuration.
<code>RouteConfigLoadEnd</code>	An event triggered after a route has been lazy loaded.
<code>NavigationEnd</code>	An event triggered when navigation ends successfully.
<code>NavigationCancel</code>	An event triggered when navigation is canceled. This is due to a Route Guard returning <code>false</code> during navigation.

NavigationError

An [event](#) 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.

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
<code>Router</code>	Displays the application component for the active URL. Manages navigation from one component to the next.
<code>RouterModule</code>	A separate NgModule that provides the necessary service providers and directives for navigating through application views.
<code>Routes</code>	Defines an array of <code>Routes</code> , each mapping a URL path to a component.
<code>Route</code>	Defines how the router should navigate to a component based on a URL pattern. Most routes consist of a path and a component type.
<code>RouterOutlet</code>	The directive (<code><router-outlet></code>) 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 <code>routerLink</code> directive that is bound to a <i>string</i> or a <i>link parameters array</i> triggers a navigation.
RouterLinkActive	The directive for adding/removing classes from an HTML element when an associated <code>routerLink</code> 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 <code>RouterLink</code> or pass the array as an argument to the <code>Router.navigate</code> method.
Routing component	An Angular component with a <code>RouterOutlet</code> that displays views based on router navigations.

The sample application

This guide describes development of a multi-page routed sample application. Along the way, it highlights design decisions and describes key features of the router such as:

- Organizing the application features into modules.
- Navigating to a component (*Heroes* link to "Heroes List").
- Including a route parameter (passing the Hero `id` while routing to the "Hero Detail").
- Child routes (the *Crisis Center* has its own routes).
- The `CanActivate` guard (checking route access).
- The `CanActivateChild` guard (checking child route access).
- The `CanDeactivate` guard (ask permission to discard unsaved changes).
- The `Resolve` guard (pre-fetching route data).
- Lazy loading feature modules.
- The `CanLoad` guard (check before loading feature module assets).

The guide proceeds as a sequence of milestones as if you were building the app step-by-step. But, it is not a tutorial and it glosses over details of Angular application construction that are more thoroughly covered elsewhere in the documentation.

The full source for the final version of the app can be seen and downloaded from the [live example](#) / [download example](#).

The sample application in action

Imagine an application that helps the *Hero Employment Agency* run its business. Heroes need work and the agency finds crises for them to solve.

The application has three main feature areas:

1. A *Crisis Center* for maintaining the list of crises for assignment to heroes.
2. A *Heroes* area for maintaining the list of heroes employed by the agency.
3. An *Admin* area to manage the list of crises and heroes.

Try it by clicking on this [live example link](#) / [download example](#).

Once the app warms up, you'll see a row of navigation buttons and the *Heroes* view with its list of heroes.

The screenshot shows a web application interface titled "HEROES". At the top, there is a navigation bar with four items: "Crisis Center", "Heroes" (which is highlighted in blue), "Admin", and "Login". Below the navigation bar, the word "HEROES" is displayed in bold capital letters. A list of heroes is shown in a table format, each row containing an ID number in a blue box and a hero name in a grey box. The heroes listed are: Mr. Nice (ID 11), Narco (ID 12), Bombasto (ID 13), Celeritas (ID 14), Magneta (ID 15), and RubberMan (ID 16).

ID	Name
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.

The screenshot shows a web application interface titled "HEROES". At the top, there is a navigation bar with four items: "Crisis Center", "Heroes" (which is highlighted in blue), "Admin", and "Login". Below the navigation bar, the word "HEROES" is displayed in bold capital letters. A hero named "Magneta" is selected, and the details are shown in a form. The ID is listed as "Id: 15" and the name is listed as "Name: Magneta" with a text input field containing "Magneta". A "Back" button is located at the bottom left of the form.

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.

The screenshot shows a web application interface titled "CRISIS CENTER". At the top, there is a navigation bar with four items: "Crisis Center" (which is highlighted in blue), "Heroes", "Admin", and "Login". Below the navigation bar, the title "CRISIS CENTER" is displayed in bold capital letters. Underneath the title is a list of four crisis entries, each enclosed in a light gray box with a dark gray border. The entries are numbered 1 through 4 and list the following crisis names: "Dragon Burning Cities", "Sky Rains Great White Sharks", "Giant Asteroid Heading For Earth", and "Procrastinators Meeting Delayed Again". At the bottom of the page, there is a welcome message: "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.

This screenshot shows the same "CRISIS CENTER" application after a crisis has been edited. The navigation bar at the top remains the same. The title "CRISIS CENTER" is followed by the same list of four crises. However, the third crisis entry, which was originally "Giant Asteroid Heading For Earth", has now been changed to "GIGANTIC Asteroid Heading For Earth". Below the crisis list, there is a child view containing the crisis detail. It shows the crisis ID as "Id: 3" and the new name "GIGANTIC Asteroid Heading For Earth" in a text input field. There are also "Save" and "Cancel" buttons at the bottom of this view.

Unlike *Hero Detail*, which updates as you type, *Crisis Detail* changes are temporary until you either save or discard them by pressing the "Save" or "Cancel" buttons. Both buttons navigate back to the *Crisis Center* and its list of crises.

Do not click either button yet. Click the browser back button or the "Heroes" link instead.

Up pops a dialog box.



You can say "OK" and lose your changes or click "Cancel" and continue editing.

Behind this behavior is the router's `CanDeactivate` guard. The guard gives you a chance to clean-up or ask the user's permission before navigating away from the current view.

The `Admin` and `Login` buttons illustrate other router capabilities to be covered later in the guide. This short introduction will do for now.

Proceed to the first application milestone.

Milestone 1: Getting started with the router

Begin with a simple version of the app that navigates between two empty views.

Component Router

Crisis Center Heroes

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](#) Appendix, learn why HTML5 style is preferred, how to adjust its behavior, and how to switch to the older hash (#) style, if necessary.

You must add a [`<base href>` element](#) to the app's `index.html` for `pushState` routing to work. The browser uses the `<base href>` value to prefix *relative* URLs when referencing CSS files, scripts, and images.

Add the `<base>` element just after the `<head>` tag. If the `app` folder is the application root, as it is for this application, set the `href` value in `index.html` *exactly* as shown here.

`src/index.html (base-href)`

```
<base href="/">
```



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 `<base href...>` with a script that writes the `<base>` tag on the fly.

```
<script>document.write('<base href="' + document.location + '" />');</script>
```



You only need this trick for the live example, not production code.

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.

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.

src/app/app.module.ts (first-config)

```
import { NgModule }           from '@angular/core';
import { BrowserModule }     from '@angular/platform-browser';
import { FormsModule }        from '@angular/forms';
import { RouterModule, Routes } from '@angular/router';

import { AppComponent }       from './app.component';
import { CrisisListComponent } from './crisis-list.component';
import { HeroListComponent }  from './hero-list.component';
```

```
const appRoutes: Routes = [
  { path: 'crisis-center', component: CrisisListComponent },
  { path: 'heroes', component: HeroListComponent },
];

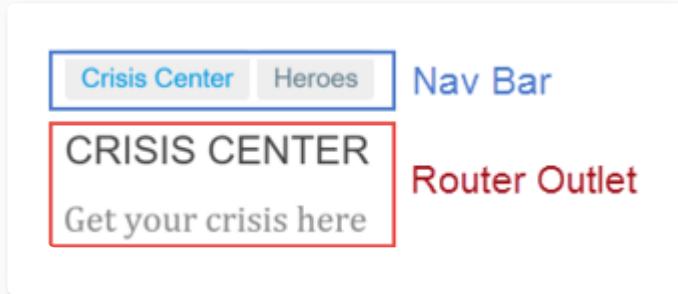
@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    RouterModule.forRoot(
      appRoutes,
      { enableTracing: true } // <-- debugging purposes only
    )
  ],
  declarations: [
    AppComponent,
    HeroListComponent,
    CrisisListComponent,
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule {}
```

Adding the configured `RouterModule` to the `AppModule` is sufficient for simple route configurations. As the application grows, you'll want to refactor the routing configuration into a separate file and create a [Routing Module](#), a special type of Service Module dedicated to the purpose of routing in feature modules.

Providing the `RouterModule` in the `AppModule` makes the Router available everywhere in the application.

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:



The corresponding component template looks like this:

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

```
template: `

<h1>Angular Router</h1>

<nav>
  <a routerLink="/crisis-center" routerLinkActive="active">Crisis Center</a>
  <a routerLink="/heroes" routerLinkActive="active">Heroes</a>
</nav>
<router-outlet></router-outlet>
`
```

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 `<router-outlet>` element to the DOM and subsequently inserts the navigated view element immediately *after* the `<router-outlet>`.

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](#).

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.

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:

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

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

@Component({
  selector: 'my-app',
  template: `
    <h1>Angular Router</h1>
    <nav>
      <a routerLink="/crisis-center" routerLinkActive="active">Crisis Center</a>
      <a routerLink="/heroes" routerLinkActive="active">Heroes</a>
    </nav>
    <router-outlet></router-outlet>
```

```
    })  
export class AppComponent { }
```

Wildcard route

You've created two routes in the app so far, one to `/crisis-center` and the other to `/heroes`. Any other URL causes the router to throw an error and crash the app.

Add a wildcard route to intercept invalid URLs and handle them gracefully. A *wildcard route* has a path consisting of two asterisks. It matches *every* URL. The router will select *this* route if it can't match a route earlier in the configuration. A wildcard route can navigate to a custom "404 Not Found" component or [redirect](#) to an existing route.

The router selects the route with a [first match wins](#) strategy. Wildcard routes are the least specific routes in the route configuration. Be sure it is the *last* route in the configuration.

To test this feature, add a button with a `RouterLink` to the `HeroListComponent` template and set the link to `"/sidekicks"`.

src/app/hero-list.component.ts (excerpt)

```
import { Component } from '@angular/core';  
  
@Component({  
  template: `  
    <h2>HEROES</h2>  
    <p>Get your heroes here</p>
```

```
<button routerLink="/sidekicks">Go to sidekicks</button>
`

})
export class HeroListComponent { }
```

The application will fail if the user clicks that button because you haven't defined a `"/sidekicks"` route yet.

Instead of adding the `"/sidekicks"` route, define a `wildcard` route instead and have it navigate to a simple `PageNotFoundComponent`.

src/app/app.module.ts (wildcard)

```
{ path: '**', component: PageNotFoundComponent }
```

Create the `PageNotFoundComponent` to display when users visit invalid URLs.

src/app/not-found.component.ts (404 component)

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

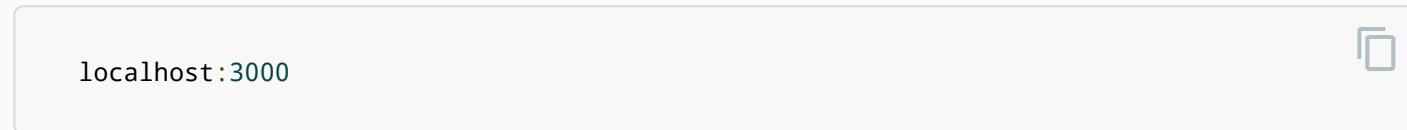
@Component({
  template: '<h2>Page not found</h2>'
})
export class PageNotFoundComponent {}
```

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.

The *default* route to heroes

When the application launches, the initial URL in the browser bar is something like:



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.

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.

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

```
const appRoutes: Routes = [
  { path: 'crisis-center', component: CrisisListComponent },
  { path: 'heroes', component: HeroListComponent },
```

```
{ path: '', redirectTo: '/heroes', pathMatch: 'full' },
{ path: '**', component: PageNotFoundComponent }
];
```

A redirect route requires a `pathMatch` property to tell the router how to match a URL to the path of a route. The router throws an error if you don't. In this app, the router should select the route to the `HeroListComponent` only when the *entire URL* matches `''`, so set the `pathMatch` value to `'full'`.

Technically, `pathMatch = 'full'` results in a route hit when the *remaining*, unmatched segments of the URL match `''`. In this example, the redirect is in a top level route so the *remaining URL* and the *entire URL* are the same thing.

The other possible `pathMatch` value is `'prefix'` which tells the router to match the redirect route when the *remaining URL begins* with the redirect route's *prefix* path.

Don't do that here. If the `pathMatch` value were `'prefix'`, *every URL* would match `''`.

Try setting it to `'prefix'` then click the `Go to sidekicks` button. Remember that's a bad URL and you should see the "Page not found" page. Instead, you're still on the "Heroes" page. Enter a bad URL in the browser address bar. You're instantly re-routed to `/heroes`. *Every URL*, good or bad, that falls through to *this* route definition will be a match.

The default route should redirect to the `HeroListComponent` *only when the entire url is ''*.

Remember to restore the redirect to `pathMatch = 'full'`.

Learn more in Victor Savkin's [post on redirects](#).

Basics wrap up

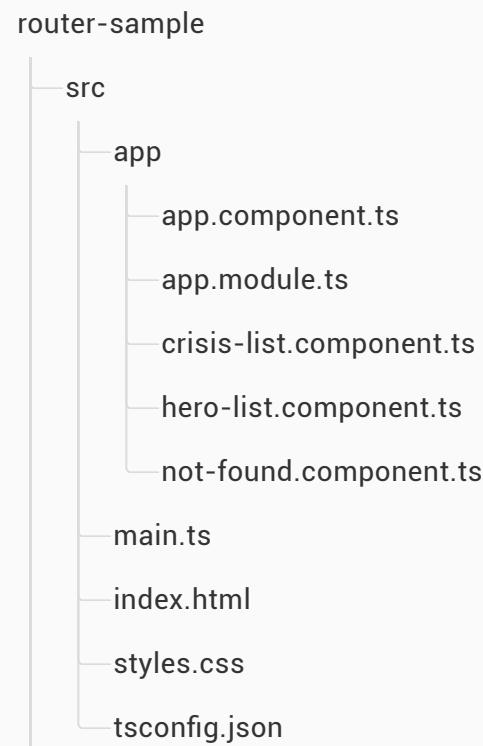
You've got a very basic navigating app, one that can switch between two views when the user clicks a link.

You've learned how to do the following:

- Load the router library.
- Add a nav bar to the shell template with anchor tags, `routerLink` and `routerLinkActive` directives.
- Add a `router-outlet` to the shell template where views will be displayed.
- Configure the router module with `RouterModule.forRoot`.
- Set the router to compose HTML5 browser URLs.
- handle invalid routes with a `wildcard` route.
- navigate to the default route when the app launches with an empty path.

The 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:



node_modules ...

package.json

Here are the files discussed in this milestone.



app.component.ts

app.module.ts

main.ts

hero-list.component.ts



```
1. import { Component } from '@angular/core';
2.
3. @Component({
4.   selector: 'my-app',
5.   template: `
6.     <h1>Angular Router</h1>
7.     <nav>
8.       <a routerLink="/crisis-center" routerLinkActive="active">Crisis
9.         Center</a>
10.        <a routerLink="/heroes" routerLinkActive="active">Heroes</a>
11.      </nav>
12.      <router-outlet></router-outlet>
13.    `
14.  export class AppComponent {}
```



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](#).

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.

```
src/app/app-routing.module.ts
```



```
1. import { NgModule }           from '@angular/core';
2. import { RouterModule, Routes } from '@angular/router';
3.
4. import { CrisisListComponent } from './crisis-list.component';
5. import { HeroListComponent }  from './hero-list.component';
6. import { PageNotFoundComponent } from './not-found.component';
7.
8. const appRoutes: Routes = [
9.   { path: 'crisis-center', component: CrisisListComponent },
10.  { path: 'heroes',        component: HeroListComponent },
11.  { path: '',             redirectTo: '/heroes', pathMatch: 'full' },
12.  { path: '**',           component: PageNotFoundComponent }
13. ];
14.
15. @NgModule({
16.   imports: [
17.     RouterModule.forRoot(
18.       appRoutes,
19.       { enableTracing: true } // <-- debugging purposes only
20.     )
21.   ],
22.   exports: [
23.     RouterModule
24.   ]
25. })
26. export class AppRoutingModule {}
```

Next, update the `app.module.ts` file, first importing the newly created `AppRoutingModule` from `app-routing.module.ts`, then replacing `RouterModule.forRoot` in the `imports` array with the `AppRoutingModule`.

src/app/app.module.ts

```
1. import { NgModule }      from '@angular/core';
2. import { BrowserModule } from '@angular/platform-browser';
3. import { FormsModule }   from '@angular/forms';
4.
5. import { AppComponent }   from './app.component';
6. import { AppRoutingModule } from './app-routing.module';
7.
8. import { CrisisListComponent } from './crisis-list.component';
9. import { HeroListComponent }  from './hero-list.component';
10. import { PageNotFoundComponent } from './not-found.component';
11.
12. @NgModule({
13.   imports: [
14.     BrowserModule,
15.     FormsModule,
16.     AppRoutingModule
17.   ],
18.   declarations: [
19.     AppComponent,
20.     HeroListComponent,
21.     CrisisListComponent,
22.     PageNotFoundComponent
23.   ],
24.   bootstrap: [ AppComponent ]
25. })
26. export class AppModule { }
```

Later in this guide you will create [multiple routing modules](#) and discover that you must import those routing modules [in the correct order](#).

The application continues to work just the same, and you can use `AppRoutingModule` as the central place to maintain future routing configuration.

Do you need a *Routing Module*?

The *Routing Module* replaces the routing configuration in the root or feature module. Either configure routes in the Routing Module or within the module itself but not in both.

The Routing Module is a design choice whose value is most obvious when the configuration is complex and includes specialized guard and resolver services. It can seem like overkill when the actual configuration is dead simple.

Some developers skip the Routing Module (for example, `AppRoutingModule`) when the configuration is simple and merge the routing configuration directly into the companion module (for example, `AppModule`).

Choose one pattern or the other and follow that pattern consistently.

Most developers should always implement a Routing Module for the sake of consistency. It keeps the code clean when configuration becomes complex. It makes testing the feature module easier. Its existence calls attention to the fact that a module is routed. It is where developers expect to find and expand routing configuration.

Milestone 3: Heroes feature

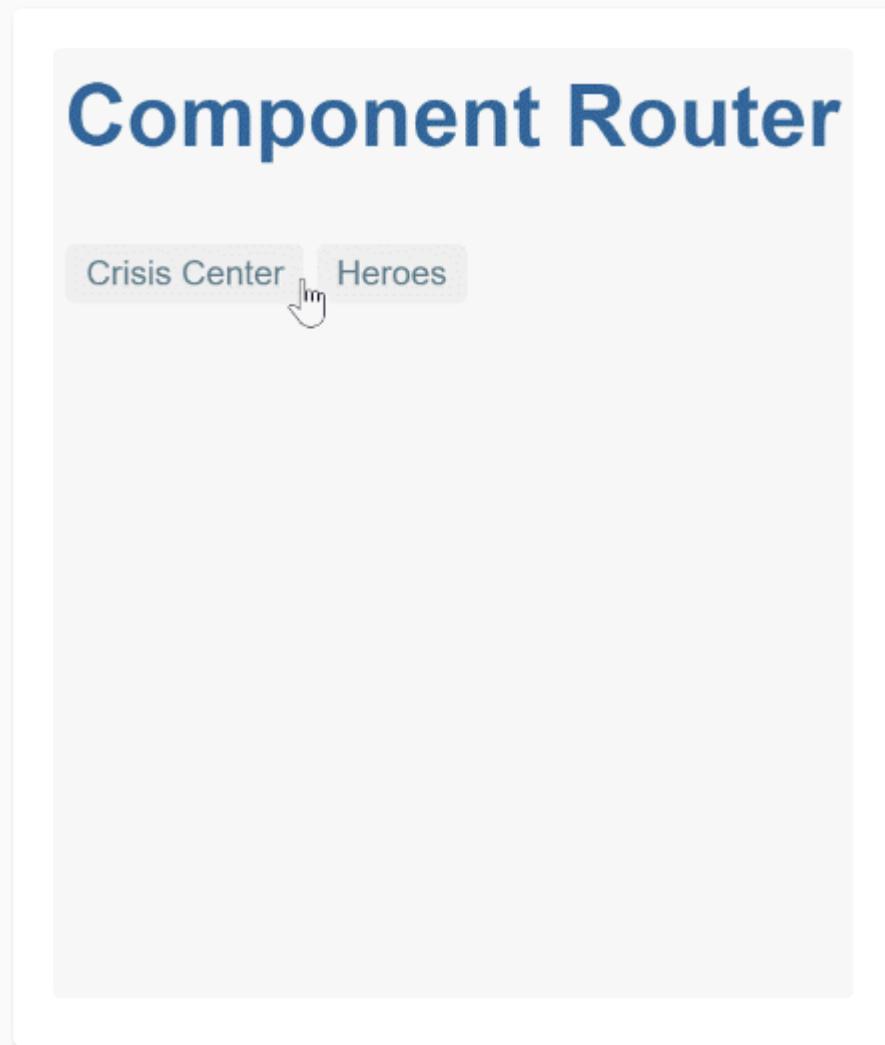
You've seen how to navigate using the `RouterLink` directive. Now you'll learn the following:

- Organize the app and routes into *feature areas* using modules.
- Navigate imperatively from one component to another.

- Pass required and optional information in route parameters.

This example recreates the heroes feature in the "Services" episode of the [Tour of Heroes tutorial](#), and you'll be copying much of the code from the [Tour of Heroes: Services example code / download example](#).

Here's how the user will experience this version of the app:



A typical application has multiple *feature areas*, each dedicated to a particular business purpose.

While you could continue to add files to the `src/app/` folder, that is unrealistic and ultimately not maintainable. Most developers prefer to put each feature area in its own folder.

You are about to break up the app into different *feature modules*, each with its own concerns. Then you'll import into the main module and navigate among them.

Add heroes functionality

Follow these steps:

- Create 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 / download example](#).
- 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:

src/app/heroes/heroes.module.ts (pre-routing)

```
1. import { NgModule }      from '@angular/core';
2. import { CommonModule }  from '@angular/common';
3. import { FormsModule }   from '@angular/forms';
4.
5. import { HeroListComponent } from './hero-list.component';
6. import { HeroDetailComponent } from './hero-detail.component';
```

```
7.  
8. import { HeroService } from './hero.service';  
9.  
10. @NgModule({  
11.   imports: [  
12.     CommonModule,  
13.     FormsModule,  
14.   ],  
15.   declarations: [  
16.     HeroListComponent,  
17.     HeroDetailComponent  
18.   ],  
19.   providers: [ HeroService ]  
20. })  
21. export class HeroesModule {}
```

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
```

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.

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`.

src/app/heroes/heroes-routing.module.ts

```
1. import { NgModule }           from '@angular/core';
2. import { RouterModule, Routes } from '@angular/router';
3.
4. import { HeroListComponent }   from './hero-list.component';
5. import { HeroDetailComponent } from './hero-detail.component';
6.
7. const heroesRoutes: Routes = [
8.   { path: 'heroes', component: HeroListComponent },
9.   { path: 'hero/:id', component: HeroDetailComponent }
10. ];
11.
12. @NgModule({
13.   imports: [
14.     RouterModule.forChild(heroesRoutes)
15.   ],

```

```
16.   exports: [
17.     RouterModule
18.   ]
19. }
20. export class HeroRoutingModule { }
```

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 `NgModule` 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.

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:

src/app/heroes/heroes.module.ts

```
1. import { NgModule }      from '@angular/core';
2. import { CommonModule }  from '@angular/common';
3. import { FormsModule }   from '@angular/forms';
4.
5. import { HeroListComponent }  from './hero-list.component';
6. import { HeroDetailComponent } from './hero-detail.component';
7.
8. import { HeroService } from './hero.service';
9.
10. import { HeroRoutingModule } from './heroes-routing.module';
11.
12. @NgModule({
13.   imports: [
14.     CommonModule,
15.     FormsModule,
16.     HeroRoutingModule
17.   ],
18.   declarations: [
19.     HeroListComponent,
20.     HeroDetailComponent
21.   ],
22.   providers: [ HeroService ]
```

```
23. })
24. export class HeroesModule {}
```

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.

src/app/app-routing.module.ts (v2)

```
import { NgModule }           from '@angular/core';
import { RouterModule, Routes } from '@angular/router';

import { CrisisListComponent } from './crisis-list.component';
// import { HeroListComponent } from './hero-list.component'; // <-- delete this line
import { PageNotFoundComponent } from './not-found.component';

const appRoutes: Routes = [
  { path: 'crisis-center', component: CrisisListComponent },
  // { path: 'heroes',      component: HeroListComponent }, // <-- delete this line
  { path: '',    redirectTo: '/heroes', pathMatch: 'full' },
```

```
    { path: '**', component: PageNotFoundComponent }
];

@NgModule({
  imports: [
    RouterModule.forRoot(
      appRoutes,
      { enableTracing: true } // <-- debugging purposes only
    )
  ],
  exports: [
    RouterModule
  ]
})
export class AppRoutingModule {}
```

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:

src/app/app.module.ts

```
1. import { NgModule }      from '@angular/core';
2. import { BrowserModule } from '@angular/platform-browser';
3. import { FormsModule }   from '@angular/forms';
4.
5. import { AppComponent }   from './app.component';
6. import { AppRoutingModule } from './app-routing.module';
7. import { HeroesModule }   from './heroes/heroes.module';
8.
9. import { CrisisListComponent } from './crisis-list.component';
10. import { PageNotFoundComponent } from './not-found.component';
11.
12. @NgModule({
13.   imports: [
14.     BrowserModule,
15.     FormsModule,
16.     HeroesModule,
17.     AppRoutingModule
18.   ],
19.   declarations: [
20.     AppComponent,
21.     CrisisListComponent,
22.     PageNotFoundComponent
23.   ],
24.   bootstrap: [ AppComponent ]
25. })
26. export class AppModule { }
```

Module import order matters

Look at the module `imports` array. Notice that the `AppRoutingModule` is *last*. Most importantly, it comes *after* the `HeroesModule`.

src/app/app.module.ts (module-imports)

```
imports: [  
  BrowserModule,  
  FormsModule,  
  HeroesModule,  
  AppRoutingModule  
,
```



The order of route configuration matters. The router accepts the first route that matches a navigation request path.

When all routes were in one `AppRoutingModule`, you put the default and **wildcard** routes last, after the `/heroes` route, so that the router had a chance to match a URL to the `/heroes` route *before* hitting the wildcard route and navigating to "Page not found".

The routes are no longer in one file. They are distributed across two modules, `AppRoutingModule` and `HeroesRoutingModule`.

Each routing module augments the route configuration *in the order of import*. If you list `AppRoutingModule` first, the wildcard route will be registered *before* the hero routes. The wildcard route – which matches *every* URL – will intercept the attempt to navigate to a hero route.

Reverse the routing modules and see for yourself that a click of the heroes link results in "Page not found". Learn about inspecting the runtime router configuration [below](#).

Route definition with a parameter

Return to the `HeroesRoutingModule` and look at the route definitions again. The route to `HeroDetailComponent` has a twist.

src/app/heroes/heroes-routing.module.ts (excerpt)

```
{ path: 'hero/:id', component: HeroDetailComponent }
```



Notice the `:id` token in the path. That creates a slot in the path for a Route Parameter. In this case, the router will insert the `id` of a hero into that slot.

If you tell the router to navigate to the detail component and display "Magneta", you expect a hero id to appear in the browser URL like this:

```
localhost: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.

Setting the route parameters in the list view

After navigating to the `HeroDetailComponent`, you expect to see the details of the selected hero. You need two pieces of information: the routing path to the component and the hero's `id`.

Accordingly, the *link parameters array* has *two* items: the routing *path* and a *route parameter* that specifies the `id` of the selected hero.

src/app/heroes/hero-list.component.ts (link-parameters-array)

```
['/hero', hero.id] // { 15 }
```



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.

Activated Route in action

Import the `Router`, `ActivatedRoute`, and `ParamMap` tokens from the router package.

src/app/heroes/hero-detail.component.ts (activated route)

```
import { Router, ActivatedRoute, ParamMap } from '@angular/router';
```



Import the `switchMap` operator because you need it later to process the `Observable` route parameters.

src/app/heroes/hero-detail.component.ts (switchMap operator import)

```
import 'rxjs/add/operator/switchMap';
```



As usual, you write a constructor that asks Angular to inject services that the component requires and reference them as private variables.

src/app/heroes/hero-detail.component.ts (constructor)

```
constructor(  
  private route: ActivatedRoute,  
  private router: Router,  
  private service: HeroService  
) {}
```



Later, in the `ngOnInit` method, you use the `ActivatedRoute` service to retrieve the parameters for the route, pull the hero `id` from the parameters and retrieve the hero to display.

src/app/heroes/hero-detail.component.ts (ngOnInit)

```
ngOnInit() {  
  this.hero$ = this.route.paramMap  
    .switchMap((params: ParamMap) =>  
      this.service.getHero(params.get('id')));  
}
```



The `paramMap` processing is a bit tricky. When the map changes, you `get()` the `id` parameter from the changed parameters.

Then you tell the `HeroService` to fetch the hero with that `id` and return the result of the `HeroService` request.

You might think to use the RxJS `map` operator. But the `HeroService` returns an `Observable<Hero>`. So you flatten the `Observable` with the `switchMap` operator instead.

The `switchMap` operator also cancels previous in-flight requests. If the user re-navigates to this route with a new `id` while the `HeroService` is still retrieving the old `id`, `switchMap` discards that old request and returns the hero for the new `id`.

The observable `Subscription` will be handled by the `AsyncPipe` and the component's `hero` property will be (re)set with the retrieved hero.

ParamMap API

The `ParamMap` API is inspired by the [URLSearchParams interface](#). It provides methods to handle parameter access for both route parameters (`paramMap`) and query parameters (`queryParamMap`).

Member	Description
<code>has(name)</code>	Returns <code>true</code> if the parameter name is in the map of parameters.
<code>get(name)</code>	Returns the parameter name value (a <code>string</code>) if present, or <code>null</code> if the parameter name is not in the map. Returns the <i>first</i> element if the parameter value is actually an array of values.
<code>getAll(name)</code>	Returns a <code>string array</code> of the parameter name value if found, or an empty array if the parameter name value is not in the map. Use <code>getAll</code> when a single

parameter could have multiple values.

keys

Returns a `string array` of all parameter names in the map.

Observable `paramMap` and component reuse

In this example, you retrieve the route parameter map from an `Observable`. That implies that the route parameter map can change during the lifetime of this component.

They might. By default, the router re-uses a component instance when it re-navigates to the same component type without visiting a different component first. The route parameters could change each time.

Suppose a parent component navigation bar had "forward" and "back" buttons that scrolled through the list of heroes. Each click navigated imperatively to the `HeroDetailComponent` with the next or previous `id`.

You don't want the router to remove the current `HeroDetailComponent` instance from the DOM only to re-create it for the next `id`. That could be visibly jarring. Better to simply re-use the same component instance and update the parameter.

Unfortunately, `ngOnInit` is only called once per component instantiation. You need a way to detect when the route parameters change from *within the same instance*. The observable `paramMap` property handles that beautifully.

When subscribing to an observable in a component, you almost always arrange to unsubscribe when the component is destroyed.

There are a few exceptional observables where this is not necessary. The `ActivatedRoute` observables are among the exceptions.

The `ActivatedRoute` and its observables are insulated from the `Router` itself. The `Router` destroys a routed component when it is no longer needed and the injected `ActivatedRoute` dies

with it.

Feel free to unsubscribe anyway. It is harmless and never a bad practice.

Snapshot: the *no-observable* alternative

This application won't re-use the `HeroDetailComponent`. The user always returns to the hero list to select another hero to view. There's no way to navigate from one hero detail to another hero detail without visiting the list component in between. Therefore, the router creates a new `HeroDetailComponent` instance every time.

When you know for certain that a `HeroDetailComponent` instance will *never, never, ever* be re-used, you can simplify the code with the *snapshot*.

The `route.snapshot` provides the initial value of the route parameter map. You can access the parameters directly without subscribing or adding observable operators. It's much simpler to write and read:

src/app/heroes/hero-detail.component.ts (ngOnInit snapshot)

```
ngOnInit() {  
  let id = this.route.snapshot.paramMap.get('id');  
  
  this.hero$ = this.service.getHero(id);  
}
```

Remember: you only get the *initial* value of the parameter map with this technique. Stick with the observable `paramMap` approach if there's even a chance that the router could re-use the component. This sample stays with the observable `paramMap` strategy just in case.

Navigating back to the list component

The `HeroDetailComponent` has a "Back" button wired to its `gotoHeroes` method that navigates imperatively back to the `HeroListComponent`.

The router `navigate` method takes the same one-item *link parameters array* that you can bind to a `[routerLink]` directive. It holds the *path to the HeroListComponent*:

src/app/heroes/hero-detail.component.ts (excerpt)

```
gotoHeroes() {
  this.router.navigate(['/heroes']);
}
```

Route Parameters: Required or optional?

Use *route parameters* to specify a *required* parameter value *within* the route URL as you do when navigating to the `HeroDetailComponent` in order to view the hero with *id* 15:

localhost: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.

Heroes list: optionally selecting a hero

When navigating to the `HeroDetailComponent` you specified the *required* `id` of the hero-to-edit in the *route parameter* and made it the second item of the *link parameters array*.

src/app/heroes/hero-list.component.ts (link-parameters-array)

```
[ '/hero', hero.id ] // { 15 }
```



The router embedded the `id` value in the navigation URL because you had defined it as a route parameter with an `:id` placeholder token in the route `path`:

src/app/heroes/heroes-routing.module.ts (hero-detail-route)

```
{ path: 'hero/:id', component: HeroDetailComponent }
```



When the user clicks the back button, the `HeroDetailComponent` constructs another *link parameters array* which it uses to navigate back to the `HeroListComponent`.

src/app/heroes/hero-detail.component.ts (gotoHeroes)

```
gotoHeroes() {
  this.router.navigate(['/heroes']);
}
```



This array lacks a route parameter because you had no reason to send information to the `HeroListComponent`.

Now you have a reason. You'd like to send the id of the current hero with the navigation request so that 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:

src/app/heroes/hero-detail.component.ts (go to heroes)

```
gotoHeroes(hero: Hero) {  
  let heroId = hero ? hero.id : null;  
  // Pass along the hero id if available  
  // so that the HeroList component can select that hero.  
  // Include a junk 'foo' property for fun.  
  this.router.navigate(['/heroes', { id: heroId, foo: 'foo' }]);  
}
```



The application still works. Clicking "back" returns to the hero list view.

Look at the browser address bar.

It should look something like this, depending on where you run it:

```
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](#) by the founder of the web, Tim Berners-Lee.

Although matrix notation never made it into the HTML standard, it is legal and it became popular among browser routing systems as a way to isolate parameters belonging to parent and child routes.

The Router is such a system and provides support for the matrix notation across browsers.

The syntax may seem strange to you but users are unlikely to notice or care as long as the URL can be emailed and pasted into a browser address bar as this one can.

Route parameters in the *ActivatedRoute* service

The list of heroes is unchanged. No hero row is highlighted.

The [live example](#) / [download example](#) *does* highlight the selected row because it demonstrates the final state of the application which includes the steps you're *about* to cover. At the moment this guide is describing the state of affairs *prior* to those steps.

The `HeroListComponent` isn't expecting any parameters at all and wouldn't know what to do with them. You can change that.

Previously, when navigating from the `HeroListComponent` to the `HeroDetailComponent`, you subscribed to the route parameter map `Observable` and made it available to the `HeroDetailComponent` in the `ActivatedRoute` service. You injected that service in the constructor of the `HeroDetailComponent`.

This time you'll be navigating in the opposite direction, from the `HeroDetailComponent` to the `HeroListComponent`.

First you extend the router import statement to include the `ActivatedRoute` service symbol:

src/app/heroes/hero-list.component.ts (import)

```
import { ActivatedRoute, ParamMap } from '@angular/router';
```

Import the `switchMap` operator to perform an operation on the `Observable` of route parameter map.

src/app/heroes/hero-list.component.ts (rxjs imports)

```
import 'rxjs/add/operator/switchMap';
import { Observable } from 'rxjs/Observable';
```



Then you inject the `ActivatedRoute` in the `HeroListComponent` constructor.

src/app/heroes/hero-list.component.ts (constructor and ngOnInit)

```
export class HeroListComponent implements OnInit {
  heroes$: Observable<Hero[]>;
  private selectedId: number;

  constructor(
    private service: HeroService,
    private route: ActivatedRoute
  ) {}

  ngOnInit() {
    this.heroes$ = this.route.paramMap
      .switchMap((params: ParamMap) => {
        // (+) before `params.get()` turns the string into a number
        this.selectedId = +params.get('id');
        return this.service.getHeroes();
      });
  }
}
```



The `ActivatedRoute.paramMap` property is an `Observable` map of route parameters. The `paramMap` emits a new map of values that includes `id` when the user navigates to the component. In `ngOnInit` you subscribe to those values, set the `selectedId`, and get the heroes.

Update the template with a [class binding](#). The binding adds the `selected` CSS class when the comparison returns `true` and removes it when `false`. Look for it within the repeated `` tag as shown here:

src/app/heroes/hero-list.component.ts (template)

```
template: `
<h2>HEROES</h2>
<ul class="items">
  <li *ngFor="let hero of heroes$ | async"
      [class.selected]="hero.id === selectedId">
    <a [routerLink]=["/hero", hero.id]>
      <span class="badge">{{ hero.id }}</span>{{ hero.name }}
    </a>
  </li>
</ul>

<button routerLink="/sidekicks">Go to sidekicks</button>
`
```

When the user navigates from the heroes list to the "Magneta" hero and back, "Magneta" appears selected:

14 Celeritas

15 Magneta

16 RubberMan

The optional `foo` route parameter is harmless and continues to be ignored.

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`:

src/app/app.module.ts (animations-module)

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

@NgModule({
  imports: [
    BrowserAnimationsModule
```

Create an `animations.ts` file in the root `src/app/` folder. The contents look like this:

src/app/animations.ts (excerpt)

```
import { animate, AnimationEntryMetadata, state, style, transition, trigger } from
```

```
'@angular/core';

// Component transition animations
export const slideInDownAnimation: AnimationEntryMetadata =
  trigger('routeAnimation', [
    state('*', {
      style({
        opacity: 1,
        transform: 'translateX(0)'
      })
    },
    transition(':enter', [
      style({
        opacity: 0,
        transform: 'translateX(-100%)'
      ),
      animate('0.2s ease-in')
    ]),
    transition(':leave', [
      animate('0.5s ease-out', style({
        opacity: 0,
        transform: 'translateY(100%)'
      }))
    ])
  ]);

```

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.

src/app/heroes/hero-detail.component.ts (host bindings)

```
@HostBinding('@routeAnimation') routeAnimation = true;  
@HostBinding('style.display') display = 'block';  
@HostBinding('style.position') position = 'absolute';
```

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.

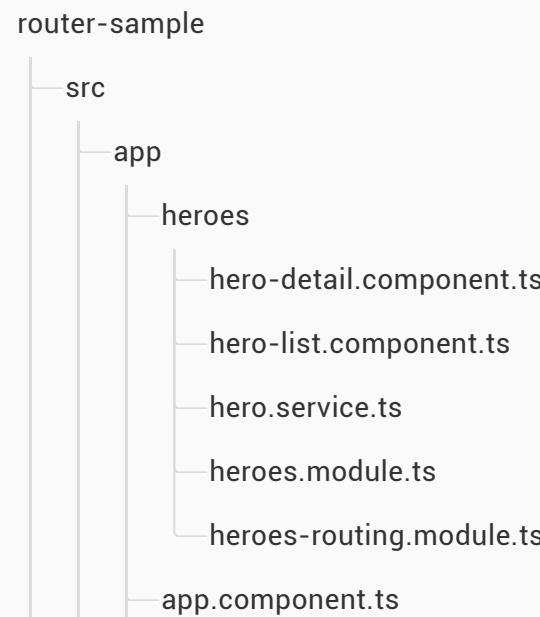
Applying route animations to individual components works for a simple demo, but in a real life app, it is better to animate routes based on *route paths*.

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:



```
|- 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.



app.component.ts

app.module.ts

app-routing.module.ts

hero-list.component.ts



```
1. import { Component } from '@angular/core';  
2.  
3. @Component({  
4.   selector: 'my-app',  
5.   template: `  
6.     <h1>Angular Router</h1>  
7.     <nav>  
8.       <a routerLink="/crisis-center" routerLinkActive="active">Crisis  
Center</a>  
9.       <a routerLink="/heroes" routerLinkActive="active">Heroes</a>  
10.      </nav>  
11.      <router-outlet></router-outlet>
```



```
12.  
13. })  
14. export class AppComponent { }
```

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:

src/app/crisis-center/crisis.service.ts (mock-crises)

```
import 'rxjs/add/observable/of';  
import 'rxjs/add/operator/map';  
import { BehaviorSubject } from 'rxjs/BehaviorSubject';  
  
export class Crisis {  
  constructor(public id: number, public name: string) {}  
}  
  
const CRISES = [  
  new Crisis(1, 'Dragon Burning Cities'),  
  new Crisis(2, 'Sky Rains Great White Sharks'),
```

```
new Crisis(3, 'Giant Asteroid Heading For Earth'),  
new Crisis(4, 'Procrastinators Meeting Delayed Again'),  
];
```

The resulting crisis center is a foundation for introducing a new concept—child routing. You can leave *Heroes* in its current state as a contrast with the *Crisis Center* and decide later if the differences are worthwhile.

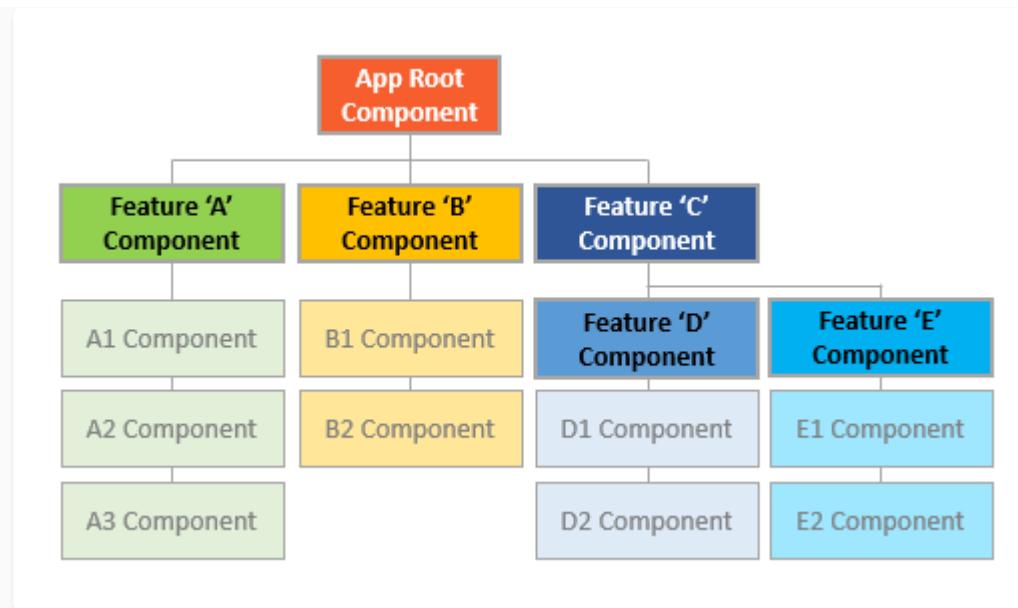
In keeping with the [Separation of Concerns principle](#), changes to the *Crisis Center* won't affect the `AppModule` or any other feature's component.

A crisis center with child routes

This section shows you how to organize the crisis center to conform to the following recommended pattern for Angular applications:

- Each feature area resides in its own folder.
- Each feature has its own Angular feature module.
- Each area has its own area root component.
- Each area root component has its own router outlet and child routes.
- Feature area routes rarely (if ever) cross with routes of other features.

If your app had many feature areas, the app component trees might look like this:



Child routing component

Add the following `crisis-center.component.ts` to the `crisis-center` folder:

src/app/crisis-center/crisis-center.component.ts

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

@Component({
  template: `
    <h2>CRISIS CENTER</h2>
    <router-outlet></router-outlet>
  `
})
export class CrisisCenterComponent {}
```

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.

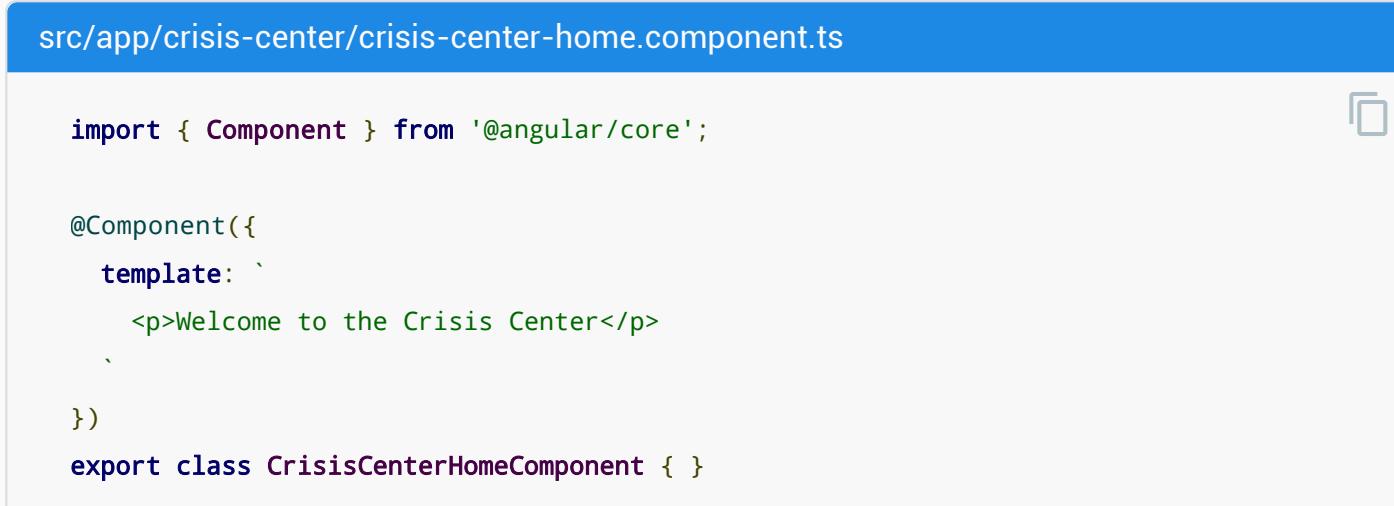
Like most shells, the `CrisisCenterComponent` class is very simple, simpler even than `AppComponent` : it has no business logic, and its template has no links, just a title and `<router-outlet>` for the crisis center child views.

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.

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.

```
src/app/crisis-center/crisis-center-home.component.ts
```



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

@Component({
  template: `
    <p>Welcome to the Crisis Center</p>
  `
})
export class CrisisCenter HomeComponent {}
```

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.

src/app/crisis-center/crisis-center-routing.module.ts (Routes)

```
const crisisCenterRoutes: Routes = [
  {
    path: 'crisis-center',
    component: CrisisCenterComponent,
    children: [
      {
        path: '',
        component: CrisisListComponent,
        children: [
          {
            path: ':id',
            component: CrisisDetailComponent
          },
          {
            path: '',
            component: CrisisCenter HomeComponent
          }
        ]
      }
    ]
  }
];
```

Notice that the parent `crisis-center` route has a `children` property with a single route containing the `CrisisListComponent`. The `CrisisListComponent` route also has a `children` array with two routes.

These two routes navigate to the crisis center child components, `CrisisCenter HomeComponent` 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 `CrisisCenter HomeComponent`, the full URL is `/crisis-center` (`/crisis-center + '' + ''`).
- To navigate to the `CrisisDetail Component` for a crisis with `id=2`, the full URL is `/crisis-center/2` (`/crisis-center + '' + '/2'`).

The absolute URL for the latter example, including the `localhost` origin, is

```
localhost:3000/crisis-center/2
```

Here's the complete `crisis-center-routing.module.ts` file with its imports.

```
src/app/crisis-center/crisis-center-routing.module.ts (excerpt)
```

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

```
import { RouterModule, Routes } from '@angular/router';

import { CrisisCenter HomeComponent } from './crisis-center-home.component';
import { CrisisListComponent }      from './crisis-list.component';
import { CrisisCenterComponent }   from './crisis-center.component';
import { CrisisDetailComponent }   from './crisis-detail.component';

const crisisCenterRoutes: Routes = [
  {
    path: 'crisis-center',
    component: CrisisCenterComponent,
    children: [
      {
        path: '',
        component: CrisisListComponent,
        children: [
          {
            path: ':id',
            component: CrisisDetailComponent
          },
          {
            path: '',
            component: CrisisCenter HomeComponent
          }
        ]
      }
    ]
  }
];
```

```
@NgModule({
  imports: [
    RouterModule.forChild(crisisCenterRoutes)
  ],
  exports: [
    RouterModule
  ]
})
export class CrisisCenterRoutingModule { }
```

Import crisis center module into the *AppModule* routes

As with the `HeroesModule`, you must add the `CrisisCenterModule` to the `imports` array of the `AppModule` before the `AppRoutingModule`:

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

```
import { NgModule }      from '@angular/core';
import { CommonModule }  from '@angular/common';
import { FormsModule }   from '@angular/forms';

import { AppComponent }      from './app.component';
import { PageNotFoundComponent } from './not-found.component';

import { AppRoutingModule }  from './app-routing.module';
import { HeroesModule }     from './heroes/heroes.module';
import { CrisisCenterModule } from './crisis-center/crisis-center.module';

import { DialogService }    from './dialog.service';
```

```
@NgModule({
  imports: [
    CommonModule,
    FormsModule,
    HeroesModule,
    CrisisCenterModule,
    AppRoutingModule
  ],
  declarations: [
    AppComponent,
    PageNotFoundComponent
  ],
  providers: [
    DialogService
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

Remove the initial crisis center route from the `app-routing.module.ts`. The feature routes are now provided by the `HeroesModule` and the `CrisisCenter` modules.

The `app-routing.module.ts` file retains the top-level application routes such as the default and wildcard routes.

src/app/app-routing.module.ts (v3)

```
import { NgModule }             from '@angular/core';
import { RouterModule, Routes } from '@angular/router';

import { ComposeMessageComponent } from './compose-message.component';
```

```
import { PageNotFoundComponent } from './not-found.component';

const appRoutes: Routes = [
  { path: '', redirectTo: '/heroes', pathMatch: 'full' },
  { path: '**', component: PageNotFoundComponent }
];

@NgModule({
  imports: [
    RouterModule.forRoot(
      appRoutes,
      { enableTracing: true } // <-- debugging purposes only
    )
  ],
  exports: [
    RouterModule
  ]
})
export class AppRoutingModule {}
```

Relative navigation

While building out the crisis center feature, you navigated to the crisis detail route using an absolute path that begins with a *slash*.

The router matches such *absolute* paths to routes starting from the top of the route configuration.

You could continue to use absolute paths like this to navigate inside the *Crisis Center* feature, but that pins the links to the parent routing structure. If you changed the parent `/crisis-center` path, you would have to change the link parameters array.

You can free the links from this dependency by defining paths that are relative to the current URL segment. Navigation *within* the feature area remains intact even if you change the parent route path to the feature.

Here's an example:

The router supports directory-like syntax in a *link parameters list* to help guide route name lookup:

`./` or no leading slash is relative to the current level.

`../` to go up one level in the route path.

You can combine relative navigation syntax with an ancestor path. If you must navigate to a sibling route, you could use the `../<sibling>` convention to go up one level, then over and down the sibling route path.

To navigate a relative path with the `Router.navigate` method, you must supply the `ActivatedRoute` to give the router knowledge of where you are in the current route tree.

After the *link parameters array*, add an object with a `relativeTo` property set to the `ActivatedRoute`. The router then calculates the target URL based on the active route's location.

Always specify the complete *absolute* path when calling router's `navigateByUrl` method.

Navigate to crisis list with a relative URL

You've already injected the `ActivatedRoute` that you need to compose the relative navigation path.

When using a `RouterLink` to navigate instead of the `Router` service, you'd use the *same* link parameters array, but you wouldn't provide the object with the `relativeTo` property. The `ActivatedRoute` is implicit

in a `RouterLink` directive.

Update the `gotoCrises` method of the `CrisisDetailComponent` to navigate back to the *Crisis Center* list using relative path navigation.

src/app/crisis-center/crisis-detail.component.ts (relative navigation)

```
// Relative navigation back to the crises
this.router.navigate(['../'], { id: crisisId, foo: 'foo' }), { relativeTo: this.route
});
```



Notice that the path goes up a level using the `../` syntax. If the current crisis `id` is `3`, the resulting path back to the crisis list is `/crisis-center/;id=3;foo=foo`.

Displaying multiple routes in named outlets

You decide to give users a way to contact the crisis center. When a user clicks a "Contact" button, you want to display a message in a popup view.

The popup should stay open, even when switching between pages in the application, until the user closes it by sending the message or canceling. Clearly you can't put the popup in the same outlet as the other pages.

Until now, you've defined a single outlet and you've nested child routes under that outlet to group routes together. The router only supports one primary *unnamed* outlet per template.

A template can also have any number of *named* outlets. Each named outlet has its own set of routes with their own components. Multiple outlets can be displaying different content, determined by different routes, all at the same time.

Add an outlet named "popup" in the `AppComponent`, directly below the unnamed outlet.

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

```
<router-outlet></router-outlet>
<router-outlet name="popup"></router-outlet>
```



That's where a popup will go, once you learn how to route a popup component to it.

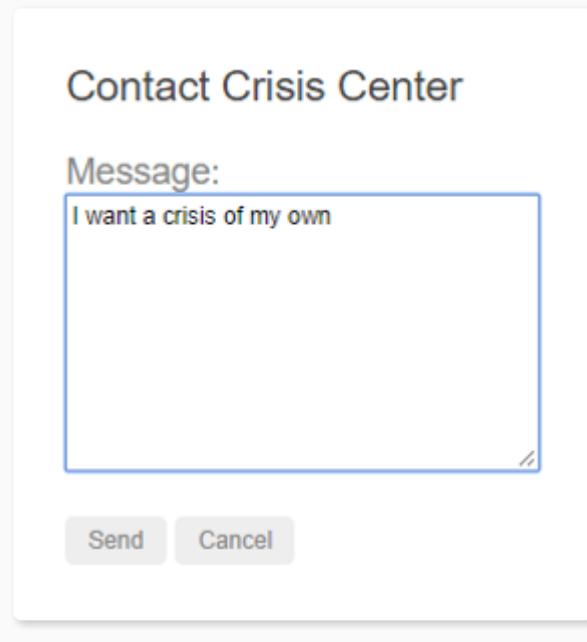
Secondary routes

Named outlets are the targets of *secondary routes*.

Secondary routes look like primary routes and you configure them the same way. They differ in a few key respects.

- They are independent of each other.
- They work in combination with other routes.
- They are displayed in named outlets.

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".



Here's the component and its template:

src/app/compose-message.component.ts src/app/compose-message.component.html

```
1. import { Component, HostBinding } from '@angular/core';
2. import { Router } from '@angular/router';
3.
4. import { slideInDownAnimation } from './animations';
5.
6. @Component({
7.   templateUrl: './compose-message.component.html',
8.   styles: [ ':host { position: relative; bottom: 10%; }' ],
9.   animations: [ slideInDownAnimation ]
10. })
11. export class ComposeMessageComponent {
12.   @HostBinding('@routeAnimation') routeAnimation = true;
13.   @HostBinding('style.display') display = 'block';
14.   @HostBinding('style.position') position = 'absolute';
15.
16.   details: string;
17.   sending = false;
18.
19.   constructor(private router: Router) {}
20.
21.   send() {
22.     this.sending = true;
23.     this.details = 'Sending Message...';
24.
25.     setTimeout(() => {
```

```
26.      this.sending = false;
27.      this.closePopup();
28.    }, 1000);
29.  }
30.
31.  cancel() {
32.    this.closePopup();
33.  }
34.
35.  closePopup() {
36.    // Providing a `null` value to the named outlet
37.    // clears the contents of the named outlet
38.    this.router.navigate([{ outlets: { popup: null }}]);
39.  }
40.}
```

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`.

Add a secondary route

Open the `AppRoutingModule` and add a new `compose` route to the `appRoutes`.

src/app/app-routing.module.ts (compose route)

```
{  
  path: 'compose',  
  component: ComposeMessageComponent,  
  outlet: 'popup'  
},
```



The `path` and `component` properties should be familiar. There's a new property, `outlet`, set to `'popup'`. This route now targets the `popup` outlet and the `ComposeMessageComponent` will display there.

The user needs a way to open the popup. Open the `AppComponent` and add a "Contact" link.

src/app/app.component.ts (contact-link)

```
<a [routerLink]="[{ outlets: { popup: ['compose'] } }]>Contact</a>
```



Although the `compose` route is pinned to the "popup" outlet, that's not sufficient for wiring the route to a `RouterLink` directive. You have to specify the named outlet in a *link parameters array* and bind it to the `RouterLink` with a property binding.

The *link parameters array* contains an object with a single `outlets` property whose value is another object keyed by one (or more) outlet names. In this case there is only the "popup" outlet property and its value is another *link parameters array* that specifies the `compose` route.

You are in effect saying, *when the user clicks this link, display the component associated with the compose route in the 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.

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.

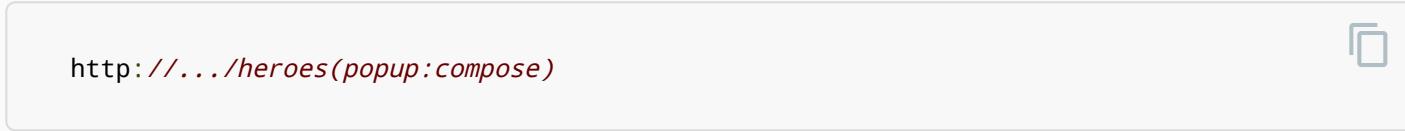


http://.../crisis-center(*popup:compose*)

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.

Clearing secondary routes

As you've learned, a component in an outlet persists until you navigate away to a new component.

Secondary outlets are no different in this regard.

Each secondary outlet has its own navigation, independent of the navigation driving the primary outlet.

Changing a current route that displays in the primary outlet has no effect on the popup outlet. That's why the popup stays visible as you navigate among the crises and heroes.

Clicking the "send" or "cancel" buttons *does* clear the popup view. To see how, look at the `closePopup()` method again:

src/app/compose-message.component.ts (closePopup)

```
closePopup() {  
  // Providing a `null` value to the named outlet  
  // clears the contents of the named outlet  
  this.router.navigate([{ outlets: { popup: null }}]);  
}
```

It navigates imperatively with the `Router.navigate()` method, passing in a *link parameters array*.

Like the array bound to the `Contact` `RouterLink` in the `AppComponent`, this one includes an object with an `outlets` property. The `outlets` property value is another object with outlet names for keys. The only named outlet is `'popup'`.

This time, the value of `'popup'` is `null`. That's not a route, but it is a legitimate value. Setting the `popup RouterOutlet` to `null` clears the outlet and removes the secondary popup route from the current URL.

Milestone 5: Route guards

At the moment, *any* user can navigate *anywhere* in the application *anytime*. That's not always the right thing to do.

- Perhaps the user is not authorized to navigate to the target component.
- Maybe the user must login (*authenticate*) first.
- Maybe you should fetch some data before you display the target component.
- You might want to save pending changes before leaving a component.
- You might ask the user if it's OK to discard pending changes rather than save them.

You 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.

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.

< src/app/admin/admin-dashboard.component.ts src/app/admin/admin.component.ts src/ap >

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

@Component({
  template: `
    <p>Dashboard</p>
  `,
})
export class AdminDashboardComponent { }
```

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:

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

```
const adminRoutes: Routes = [
  {
    path: 'admin',
    component: AdminComponent,
    children: [
      {
        path: '',
        children: [
          { path: 'crises', component: ManageCrisesComponent },
          { path: 'heroes', component: ManageHeroesComponent },
          { path: '', component: AdminDashboardComponent }
        ]
      }
    ]
  }
];

@NgModule({
  imports: [
    RouterModule.forChild(adminRoutes)
```

```
  ],
  exports: [
    RouterModule
  ]
})
export class AdminRoutingModule {}
```

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.

src/app/app.module.ts (admin module)

```
import { NgModule }      from '@angular/core';
import { CommonModule }  from '@angular/common';
import { FormsModule }   from '@angular/forms';

import { AppComponent }      from './app.component';
import { PageNotFoundComponent } from './not-found.component';

import { AppRoutingModule }   from './app-routing.module';
import { HeroesModule }      from './heroes/heroes.module';
```

```
import { CrisisCenterModule }      from './crisis-center/crisis-center.module';
import { AdminModule }             from './admin/admin.module';

import { DialogService }          from './dialog.service';

@NgModule({
  imports: [
    CommonModule,
    FormsModule,
    HeroesModule,
    CrisisCenterModule,
    AdminModule,
    AppRoutingModule
  ],
  declarations: [
    AppComponent,
    PageNotFoundComponent
  ],
  providers: [
    DialogService
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule {}
```

Add an "Admin" link to the `AppComponent` shell so that users can get to this feature.

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

`template:` 

```
<h1 class="title">Angular Router</h1>
<nav>
  <a routerLink="/crisis-center" routerLinkActive="active">Crisis Center</a>
  <a routerLink="/heroes" routerLinkActive="active">Heroes</a>
  <a routerLink="/admin" routerLinkActive="active">Admin</a>
  <a [routerLink]="[{ outlets: { popup: ['compose'] } }]>Contact</a>
</nav>
<router-outlet></router-outlet>
<router-outlet name="popup"></router-outlet>
`
```

Guard the admin feature

Currently every route within the *Crisis Center* is open to everyone. The new *admin* feature should be accessible only to authenticated users.

You could hide the link until the user logs in. But that's tricky and difficult to maintain.

Instead you'll write a `canActivate()` guard method to redirect anonymous users to the login page when they try to enter the admin area.

This is a general purpose guard—you can imagine other features that require authenticated users—so you 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:

src/app/auth-guard.service.ts (excerpt)

```
import { Injectable }      from '@angular/core';
import { CanActivate }    from '@angular/router';
```

```
@Injectable()
export class AuthGuard implements CanActivate {
  canActivate() {
    console.log('AuthGuard#canActivate called');
    return true;
  }
}
```

Next, open `admin-routing.module.ts`, import the `AuthGuard` class, and update the admin route with a `canActivate` guard property that references it:

src/app/admin/admin-routing.module.ts (guarded admin route)

```
import { AuthGuard } from '../auth-guard.service';

const adminRoutes: Routes = [
  {
    path: 'admin',
    component: AdminComponent,
    canActivate: [AuthGuard],
    children: [
      {
        path: '',
        children: [
          { path: 'crises', component: ManageCrisesComponent },
          { path: 'heroes', component: ManageHeroesComponent },
          { path: '', component: AdminDashboardComponent }
        ],
      }
    ]
}
```

```
}

];

@NgModule({
  imports: [
    RouterModule.forChild(adminRoutes)
  ],
  exports: [
    RouterModule
  ]
})
export class AdminRoutingModule {}
```

The admin feature is now protected by the guard, albeit protected poorly.

Teach *AuthGuard* to authenticate

Make the `AuthGuard` at least pretend to authenticate.

The `AuthGuard` should call an application service that can login a user and retain information about the current user. Here's a demo `AuthService` :

src/app/auth.service.ts (excerpt)

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

import { Observable } from 'rxjs/Observable';
import 'rxjs/add/observable/of';
import 'rxjs/add/operator/do';
import 'rxjs/add/operator/delay';
```

```
@Injectable()
export class AuthService {
  isLoggedIn = false;

  // store the URL so we can redirect after logging in
  redirectUrl: string;

  login(): Observable<boolean> {
    return Observable.of(true).delay(1000).do(val => this.isLoggedIn = true);
  }

  logout(): void {
    this.isLoggedIn = false;
  }
}
```

Although it doesn't actually log in, it has what you need for this discussion. It has an `isLoggedIn` flag to tell you whether the user is authenticated. Its `login` method simulates an API call to an external service by returning an Observable that resolves successfully after a short pause. The `redirectUrl` property will store the attempted URL so you can navigate to it after authenticating.

Revise the `AuthGuard` to call it.

src/app/auth-guard.service.ts (v2)

```
import { Injectable }      from '@angular/core';
import {
  CanActivate, Router,
  ActivatedRouteSnapshot,
  RouterStateSnapshot
}                      from '@angular/router';
```

```
import { AuthService }      from './auth.service';

@Injectable()
export class AuthGuard implements CanActivate {
  constructor(private authService: AuthService, private router: Router) {}

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean {
    let url: string = state.url;

    return this.checkLogin(url);
  }

  checkLogin(url: string): boolean {
    if (this.authService.isLoggedIn) { return true; }

    // Store the attempted URL for redirecting
    this.authService.redirectUrl = url;

    // Navigate to the login page with extras
    this.router.navigate(['/login']);
    return false;
  }
}
```

Notice that you *inject* the `AuthService` and the `Router` in the constructor. You haven't provided the `AuthService` yet but it's good to know that you can inject helpful services into routing guards.

This guard returns a synchronous boolean result. If the user is logged in, it returns true and the navigation continues.

The `ActivatedRouteSnapshot` contains the *future* route that will be activated and the `RouterStateSnapshot` contains the *future* `RouterState` of the application, should you pass through the guard check.

If the user is not logged in, you store the attempted URL the user came from using the `RouterStateSnapshot.url` and tell the router to navigate to a login page—a page you haven't created yet. This secondary navigation automatically cancels the current navigation; `checkLogin()` returns `false` just to be clear about that.

Add the *LoginComponent*

You need a `LoginComponent` for the user to log in to the app. After logging in, you'll redirect to the stored URL if available, or use the default URL. There is nothing new about this component or the way you wire it into the router configuration.

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.

`src/app/app.module.ts` `src/app/login.component.ts` `src/app/login-routing.module.ts`

```
1. import { NgModule }      from '@angular/core';
2. import { BrowserModule } from '@angular/platform-browser';
3. import { FormsModule }   from '@angular/forms';
4. import { BrowserAnimationsModule } from '@angular/platform-
   browser/animations';
5.
6. import { Router } from '@angular/router';
7.
8. import { AppComponent }      from './app.component';
9. import { AppRoutingModule } from './app-routing.module';
```

```
10.  
11. import { HeroesModule }          from './heroes/heroes.module';  
12. import { ComposeMessageComponent } from './compose-message.component';  
13. import { LoginRoutingModule }     from './login-routing.module';  
14. import { LoginComponent }        from './login.component';  
15. import { PageNotFoundComponent }  from './not-found.component';  
16.  
17. import { DialogService }         from './dialog.service';  
18.  
19. @NgModule({  
20.   imports: [  
21.     BrowserModule,  
22.     FormsModule,  
23.     HeroesModule,  
24.     LoginRoutingModule,  
25.     AppRoutingModule,  
26.     BrowserAnimationsModule  
27.   ],  
28.   declarations: [  
29.     AppComponent,  
30.     ComposeMessageComponent,  
31.     LoginComponent,  
32.     PageNotFoundComponent  
33.   ],  
34.   providers: [  
35.     DialogService  
36.   ],  
37.   bootstrap: [ AppComponent ]  
38. })  
39. export class AppModule {
```

```
40. // Diagnostic only: inspect router configuration
41. constructor(router: Router) {
42.   console.log('Routes: ', JSON.stringify(router.config, undefined, 2));
43. }
44. }
```

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](#).

`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`:

src/app/auth-guard.service.ts (excerpt)

```
import { Injectable }      from '@angular/core';
import {
  CanActivate, Router,
  ActivatedRouteSnapshot,
  RouterStateSnapshot,
  CanActivateChild
}
      from '@angular/router';
import { AuthService }    from './auth.service';

@Injectable()
export class AuthGuard implements CanActivate, CanActivateChild {
  constructor(private authService: AuthService, private router: Router) {}

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean {
    let url: string = state.url;

    return this.checkLogin(url);
  }

  canActivateChild(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):
boolean {
    return this.canActivate(route, state);
  }

  /* . . . */
}
```

Add the same `AuthGuard` to the `component-less` admin route to protect all other child routes at one time instead of adding the `AuthGuard` to each route individually.

src/app/admin/admin-routing.module.ts (excerpt)

```
const adminRoutes: Routes = [
  {
    path: 'admin',
    component: AdminComponent,
    canActivate: [AuthGuard],
    children: [
      {
        path: '',
        canActivateChild: [AuthGuard],
        children: [
          { path: 'crises', component: ManageCrisesComponent },
          { path: 'heroes', component: ManageHeroesComponent },
          { path: '', component: AdminDashboardComponent }
        ]
      }
    ]
  }
];

@NgModule({
  imports: [
    RouterModule.forChild(adminRoutes)
  ],
  exports: [
    RouterModule
  ]
})
export class AdminRoutingModule {}
```

`CanDeactivate`: handling unsaved changes

Back in the "Heroes" workflow, the app accepts every change to a hero immediately without hesitation or validation.

In the real world, you might have to accumulate the user's changes. You might have to validate across fields. You might have to validate on the server. You might have to hold changes in a pending state until the user confirms them *as a group* or cancels and reverts all changes.

What do you do about unapproved, unsaved changes when the user navigates away? You can't just leave and risk losing the user's changes; that would be a terrible experience.

It's better to pause and let the user decide what to do. If the user cancels, you'll stay put and allow more changes. If the user approves, the app can save.

You still might delay navigation until the save succeeds. If you let the user move to the next screen immediately and the save were to fail (perhaps the data are ruled invalid), you would lose the context of the error.

You can't block while waiting for the server—that's not possible in a browser. You need to stop the navigation while you wait, asynchronously, for the server to return with its answer.

You need the `CanDeactivate` guard.

Cancel and save

The sample application doesn't talk to a server. Fortunately, you have another way to demonstrate an asynchronous router hook.

Users update crisis information in the `CrisisDetailComponent`. Unlike the `HeroDetailComponent`, the user changes do not update the crisis entity immediately. Instead, the app updates the entity when the user presses the *Save* button and discards the changes when the user presses the *Cancel* button.

Both buttons navigate back to the crisis list after save or cancel.

src/app/crisis-center/crisis-detail.component.ts (cancel and save methods)

```
cancel() {  
  this.gotoCrises();  
}  
  
save() {  
  this.crisis.name = this.editName;  
  this.gotoCrises();  
}
```



What if the user tries to navigate away without saving or canceling? The user could push the browser back button or click the heroes link. Both actions trigger a navigation. Should the app save or cancel automatically?

This demo does neither. Instead, it asks the user to make that choice explicitly in a confirmation dialog box that *waits asynchronously for the user's answer*.

You could wait for the user's answer with synchronous, blocking code. The app will be more responsive—and can do other work—by waiting for the user's answer asynchronously. Waiting for the user asynchronously is like waiting for the server asynchronously.

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`).

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.

src/app/can-deactivate-guard.service.ts

```
1. import { Injectable }      from '@angular/core';
2. import { CanDeactivate }   from '@angular/router';
3. import { Observable }      from 'rxjs/Observable';
4.
5. export interface CanComponentDeactivate {
6.   canDeactivate: () => Observable<boolean> | Promise<boolean> | boolean;
7. }
8.
9. @Injectable()
10. export class CanDeactivateGuard implements
11.   CanDeactivate<CanComponentDeactivate> {
12.   canDeactivate(component: CanComponentDeactivate) {
13.     return component.canDeactivate ? component.canDeactivate() : true;
14.   }
15. }
```

Alternatively, you could make a component-specific `CanDeactivate` guard for the `CrisisDetailComponent`. The `canDeactivate()` method provides you with the current instance of the component, the current `ActivatedRoute`, and `RouterStateSnapshot` in case you needed to access some external information. This would be useful if you only wanted to use this guard for this component and needed to get the component's properties or confirm whether the router should allow navigation away from it.

src/app/can-deactivate-guard.service.ts (component-specific)

```
import { Injectable }          from '@angular/core';
import { Observable }         from 'rxjs/Observable';
import { CanDeactivate,
         ActivatedRouteSnapshot,
         RouterStateSnapshot } from '@angular/router';

import { CrisisDetailComponent } from './crisis-center/crisis-detail.component';

@Injectable()
export class CanDeactivateGuard implements CanDeactivate<CrisisDetailComponent> {

  canDeactivate(
    component: CrisisDetailComponent,
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ): Observable<boolean> | boolean {
    // Get the Crisis Center ID
    console.log(route.paramMap.get('id'));

    // Get the current URL
    console.log(state.url);

    // Allow synchronous navigation ('true') if no crisis or the crisis is unchanged
    if (!component.crisis || component.crisis.name === component.editName) {
      return true;
    }
    // Otherwise ask the user with the dialog service and return its
    // observable which resolves to true or false when the user decides
  }
}
```

```
        return component.dialogService.confirm('Discard changes?');
    }
}
```

Looking back at the `CrisisDetailComponent`, it implements the confirmation workflow for unsaved changes.

src/app/crisis-center/crisis-detail.component.ts (excerpt)

```
canDeactivate(): Observable<boolean> | boolean {
    // Allow synchronous navigation ('true') if no crisis or the crisis is unchanged
    if (!this.crisis || this.crisis.name === this.editName) {
        return true;
    }
    // Otherwise ask the user with the dialog service and return its
    // observable which resolves to true or false when the user decides
    return this.dialogService.confirm('Discard changes?');
}
```

Notice that the `canDeactivate()` method *can* return synchronously; it returns `true` immediately if there is no crisis or there are no pending changes. But it can also return a `Promise` or an `Observable` and the router will wait for that to resolve to truthy (`navigate`) or falsy (`stay put`).

Add the `Guard` to the crisis detail route in `crisis-center-routing.module.ts` using the `canDeactivate` array property.

src/app/crisis-center/crisis-center-routing.module.ts (can deactivate guard)

```
import { NgModule }             from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
```

```
import { CrisisCenter HomeComponent } from './crisis-center-home.component';
import { CrisisListComponent }      from './crisis-list.component';
import { CrisisCenterComponent }   from './crisis-center.component';
import { CrisisDetailComponent }   from './crisis-detail.component';

import { CanDeactivateGuard }     from '../can-deactivate-guard.service';

const crisisCenterRoutes: Routes = [
  {
    path: '',
    redirectTo: '/crisis-center',
    pathMatch: 'full'
  },
  {
    path: 'crisis-center',
    component: CrisisCenterComponent,
    children: [
      {
        path: '',
        component: CrisisListComponent,
        children: [
          {
            path: ':id',
            component: CrisisDetailComponent,
            canDeactivate: [CanDeactivateGuard]
          },
          {
            path: '',
            component: CrisisCenter HomeComponent
          }
        ]
      }
    ]
  }
]
```

```
        }
    ]
}
];
}

@NgModule({
  imports: [
    RouterModule.forChild(crisisCenterRoutes)
  ],
  exports: [
    RouterModule
  ]
})
export class CrisisCenterRoutingModule { }
```

Add the `Guard` to the main `AppRoutingModule` `providers` array so the `Router` can inject it during the navigation process.

```
1. import { NgModule }             from '@angular/core';
2. import { RouterModule, Routes } from '@angular/router';
3.
4. import { ComposeMessageComponent } from './compose-message.component';
5. import { CanDeactivateGuard }     from './can-deactivate-guard.service';
6. import { PageNotFoundComponent } from './not-found.component';
7.
8. const appRoutes: Routes = [
9.   {
10.     path: 'compose',
```

```
11.     component: ComposeMessageComponent,
12.     outlet: 'popup'
13.   },
14.   { path: '', redirectTo: '/heroes', pathMatch: 'full' },
15.   { path: '**', component: PageNotFoundComponent }
16. ];
17.
18. @NgModule({
19.   imports: [
20.     RouterModule.forRoot(
21.       appRoutes,
22.       { enableTracing: true } // <-- debugging purposes only
23.     )
24.   ],
25.   exports: [
26.     RouterModule
27.   ],
28.   providers: [
29.     CanDeactivateGuard
30.   ]
31. })
32. export class AppRoutingModule {}
```

Now you have given the user a safeguard against unsaved changes.

Resolve: pre-fetching component data

In the `Hero Detail` and `Crisis Detail`, the app waited until the route was activated to fetch the respective hero or crisis.

This worked well, but there's a better way. If you were using a real world API, there might be some delay before the data to display is returned from the server. You don't want to display a blank component while waiting for the data.

It's preferable to pre-fetch data from the server so it's ready the moment the route is activated. This also allows you to handle errors before routing to the component. There's no point in navigating to a crisis detail for an `id` that doesn't have a record. It'd be better to send the user back to the `Crisis List` that shows only valid crisis centers.

In summary, you want to delay rendering the routed component until all necessary data have been fetched.

You need a *resolver*.

Fetch data before navigating

At the moment, the `CrisisDetailComponent` retrieves the selected crisis. If the crisis is not found, it navigates back to the crisis list view.

The experience might be better if all of this were handled first, before the route is activated. A `CrisisDetailResolver` service could retrieve a `Crisis` or navigate away if the `Crisis` does not exist *before* activating the route and creating the `CrisisDetailComponent`.

Create the `crisis-detail-resolver.service.ts` file within the `Crisis Center` feature area.

src/app/crisis-center/crisis-detail-resolver.service.ts

```
1. import 'rxjs/add/operator/map';
2. import 'rxjs/add/operator/take';
3. import { Injectable }           from '@angular/core';
4. import { Observable }          from 'rxjs/Observable';
5. import { Router, Resolve, RouterStateSnapshot,
6.           ActivatedRouteSnapshot } from '@angular/router';
```

```
7.  
8. import { Crisis, CrisisService } from './crisis.service';  
9.  
10. @Injectable()  
11. export class CrisisDetailResolver implements Resolve<Crisis> {  
12.   constructor(private cs: CrisisService, private router: Router) {}  
13.  
14.   resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):  
    Observable<Crisis> {  
15.     let id = route.paramMap.get('id');  
16.  
17.     return this.cs.getCrisis(id).take(1).map(crisis => {  
18.       if (crisis) {  
19.         return crisis;  
20.       } else { // id not found  
21.         this.router.navigate(['/crisis-center']);  
22.         return null;  
23.       }  
24.     });  
25.   }  
26. }
```

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.

src/app/crisis-center/crisis-center-routing.module.ts (resolver)

```
import { CrisisDetailResolver } from './crisis-detail-resolver.service';

@NgModule({
  imports: [
    RouterModule.forChild(crisisCenterRoutes)
  ],
  exports: [
    RouterModule
  ],
  providers: [
    CrisisDetailResolver
  ]
})
export class CrisisCenterRoutingModule { }
```

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.

src/app/crisis-center/crisis-detail.component.ts (ngOnInit v2)

```
ngOnInit() {
  this.route.data
    .subscribe((data: { crisis: Crisis }) => {
      this.editName = data.crisis.name;
      this.crisis = data.crisis;
    });
}
```

Two critical points

1. The router's `Resolve` interface is optional. The `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.

< app.component.ts crisis-center-home.component.ts crisis-center.component.ts crisis >

```
1. import { Component } from '@angular/core';
2.
3. @Component({
```

```
4.   selector: 'my-app',
5.   template: `
6.     <h1 class="title">Angular Router</h1>
7.     <nav>
8.       <a routerLink="/crisis-center" routerLinkActive="active">Crisis
9.         Center</a>
10.        <a routerLink="/heroes" routerLinkActive="active">Heroes</a>
11.        <a routerLink="/admin" routerLinkActive="active">Admin</a>
12.        <a routerLink="/login" routerLinkActive="active">Login</a>
13.        <a [routerLink]="[{ outlets: { popup: ['compose'] } }]">Contact</a>
14.      </nav>
15.      <router-outlet></router-outlet>
16.      <router-outlet name="popup"></router-outlet>
17.    `
18.  export class AppComponent {
19. }
```

auth-guard.service.ts can-deactivate-guard.service.ts

```
1. import { Injectable }           from '@angular/core';
2. import {
3.   CanActivate, Router,
4.   ActivatedRouteSnapshot,
5.   RouterStateSnapshot,
6.   CanActivateChild
7. }                               from '@angular/router';
8. import { AuthService }          from './auth.service';
9.
```

```
10. @Injectable()
11. export class AuthGuard implements CanActivate, CanActivateChild {
12.   constructor(private authService: AuthService, private router: Router) {}
13.
14.   canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):
15.     boolean {
16.     let url: string = state.url;
17.
18.     return this.checkLogin(url);
19.
20.   canActivateChild(route: ActivatedRouteSnapshot, state:
21.     RouterStateSnapshot): boolean {
22.     return this.canActivate(route, state);
23.
24.   checkLogin(url: string): boolean {
25.     if (this.authService.isLoggedIn) { return true; }
26.
27.     // Store the attempted URL for redirecting
28.     this.authService.redirectUrl = url;
29.
30.     // Navigate to the login page
31.     this.router.navigate(['/login']);
32.     return false;
33.   }
34. }
```

Query parameters and fragments

In the [route parameters](#) example, you only dealt with parameters specific to the route, but what if you wanted optional parameters available to all routes? This is where query parameters come into play.

[Fragments](#) refer to certain elements on the page identified with an `id` attribute.

Update the `AuthGuard` to provide a `session_id` query that will remain after navigating to another route.

Add an `anchor` element so you can jump to a certain point on the page.

Add the `NavigationExtras` object to the `router.navigate` method that navigates you to the `/login` route.

src/app/auth-guard.service.ts (v3)

```
import { Injectable }           from '@angular/core';
import {
  CanActivate, Router,
  ActivatedRouteSnapshot,
  RouterStateSnapshot,
  CanActivateChild,
  NavigationExtras
}
           from '@angular/router';
import { AuthService }         from './auth.service';

@Injectable()
export class AuthGuard implements CanActivate, CanActivateChild {
  constructor(private authService: AuthService, private router: Router) {}

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean {
    let url: string = state.url;

    return this.checkLogin(url);
  }

  canActivateChild(childRoute: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean {
    let url: string = state.url;

    return this.checkLogin(url);
  }

  checkLogin(url: string): boolean {
    if (this.authService.isLoggedIn) return true;

    this.router.navigate(['/login'], { queryParams: { returnUrl: url } });
    return false;
  }
}
```

```
}

canActivateChild(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):
boolean {
    return this.canActivate(route, state);
}

checkLogin(url: string): boolean {
    if (this.authService.isLoggedIn) { return true; }

    // Store the attempted URL for redirecting
    this.authService.redirectUrl = url;

    // Create a dummy session id
    let sessionId = 123456789;

    // Set our navigation extras object
    // that contains our global query params and fragment
    let navigationExtras: NavigationExtras = {
        queryParams: { 'session_id': sessionId },
        fragment: 'anchor'
    };

    // Navigate to the login page with extras
    this.router.navigate(['/login'], navigationExtras);
    return false;
}
}
```

You can also preserve query parameters and fragments across navigations without having to provide them again when navigating. In the `LoginComponent`, you'll add an *object* as the second argument in the `router.navigate` function and provide the `queryParamsHandling` and `preserveFragment` to pass along the current query parameters and fragment to the next route.

src/app/login.component.ts (preserve)

```
// Set our navigation extras object
// that passes on our global query params and fragment
let navigationExtras: NavigationExtras = {
  queryParamsHandling: 'preserve',
  preserveFragment: true
};

// Redirect the user
this.router.navigate([redirect], navigationExtras);
```

The `queryParamsHandling` feature also provides a `merge` option, which will preserve and combine the current query parameters with any provided query parameters when navigating.

Since you'll be navigating to the `Admin Dashboard` route after logging in, you'll update it to handle the query parameters and fragment.

src/app/admin/admin-dashboard.component.ts (v2)

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from '@angular/router';
import { Observable } from 'rxjs/Observable';
```

```
import 'rxjs/add/operator/map';

@Component({
  template: `
    <p>Dashboard</p>

    <p>Session ID: {{ sessionId | async }}</p>
    <a id="anchor"></a>
    <p>Token: {{ token | async }}</p>
  `
})
export class AdminDashboardComponent implements OnInit {
  sessionId: Observable<string>;
  token: Observable<string>;

  constructor(private route: ActivatedRoute) {}

  ngOnInit() {
    // Capture the session ID if available
    this.sessionId = this.route
      .queryParamMap
      .map(params => params.get('session_id') || 'None');

    // Capture the fragment if available
    this.token = this.route
      .fragment
      .map(fragment => fragment || 'None');
  }
}
```

Query parameters and *fragments* are also available through the `ActivatedRoute` service. Just like *route parameters*, the query parameters and fragments are provided as an `Observable`. The updated *Crisis Admin* component feeds the `Observable` directly into the template using the `AsyncPipe`.

Now, you can click on the *Admin* button, which takes you to the *Login* page with the provided `queryParamMap` and `fragment`. After you click the login button, notice that you have been redirected to the `Admin Dashboard` page with the query parameters and fragment still intact in the address bar.

You can use these persistent bits of information for things that need to be provided across pages like authentication tokens or session ids.

The `query params` and `fragment` can also be preserved using a `RouterLink` with the `queryParamsHandling` and `preserveFragment` bindings respectively.

Milestone 6: Asynchronous routing

As you've worked through the milestones, the application has naturally gotten larger. As you continue to build out feature areas, the overall application size will continue to grow. At some point you'll reach a tipping point where the application takes 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.

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`.

app-routing.module.ts (load children)

```
{
  path: 'admin',
  loadChildren: 'app/admin/admin.module#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.

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:

src/app/auth-guard.service.ts (CanLoad guard)

```
canLoad(route: Route): boolean {
  let url = `/${route.path}`;

  return this.checkLogin(url);
}
```



The router sets the `canLoad()` method's `route` parameter to the intended destination URL. The `checkLogin()` method redirects to that URL once the user has logged in.

Now import the `AuthGuard` into the `AppRoutingModule` and add the `AuthGuard` to the `canLoad` array property for the `admin` route. The completed admin route looks like this:

app-routing.module.ts (lazy admin route)

```
{
  path: 'admin',
  loadChildren: 'app/admin/admin.module#AdminModule',
  canLoad: [AuthGuard]
},
```



Preloading: background loading of feature areas

You've learned how to load modules on-demand. You can also load modules asynchronously with *preloading*.

This may seem like what the app has been doing all along. Not quite. The `AppModule` is loaded when the application starts; that's *eager* loading. Now the `AdminModule` loads only when the user clicks on a link; that's *lazy* loading.

Preloading is something in between. Consider the *Crisis Center*. It isn't the first view that a user sees. By default, the *Heroes* are the first view. For the smallest initial payload and fastest launch time, you should eagerly load the `AppModule` and the `HeroesModule`.

You could lazy load the *Crisis Center*. But you're almost certain that the user will visit the *Crisis Center* within minutes of launching the app. Ideally, the app would launch with just the `AppModule` and the `HeroesModule` loaded and then, almost immediately, load the `CrisisCenterModule` in the background. By the time the user navigates to the *Crisis Center*, its module will have been loaded and ready to go.

That's *preloading*.

How preloading works

After each *successful*/navigation, the router looks in its configuration for an unloaded module that it can preload. Whether it preloads a module, and which modules it preloads, depends upon the *preload strategy*.

The `Router` offers two preloading strategies out of the box:

- No preloading at all which is the default. Lazy loaded feature areas are still loaded on demand.
- Preloading of all lazy loaded feature areas.

Out of the box, the router either never preloads, or preloads every lazy load module. The `Router` also supports [custom preloading strategies](#) for fine control over which modules to preload and when.

In this next section, you'll update the `CrisisCenterModule` to load lazily by default and use the `PreloadAllModules` strategy to load it (and *all other* lazy loaded modules) as soon as possible.

Lazy load the *crisis center*

Update the route configuration to lazy load the `CrisisCenterModule`. Take the same steps you used to configure `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*:

`app.module.ts` `app-routing.module.ts` `crisis-center-routing.module.ts`

```
1. import { NgModule }      from '@angular/core';
2. import { BrowserModule } from '@angular/platform-browser';
3. import { FormsModule }   from '@angular/forms';
4. import { BrowserAnimationsModule } from '@angular/platform-
   browser/animations';
5.
6. import { Router } from '@angular/router';
7.
8. import { AppComponent }      from './app.component';
9. import { AppRoutingModule }   from './app-routing.module';
10.
11. import { HeroesModule }     from './heroes/heroes.module';
12. import { ComposeMessageComponent } from './compose-message.component';
13. import { LoginRoutingModule }   from './login-routing.module';
14. import { LoginComponent }    from './login.component';
15. import { PageNotFoundComponent } from './not-found.component';
16.
17. import { DialogService }    from './dialog.service';
18.
19. @NgModule({
20.   imports: [
21.     BrowserModule,
22.     FormsModule,
```

```
23.   HeroesModule,
24.   LoginRoutingModule,
25.   AppRoutingModule,
26.   BrowserAnimationsModule
27. ],
28. declarations: [
29.   AppComponent,
30.   ComposeMessageComponent,
31.   LoginComponent,
32.   PageNotFoundComponent
33. ],
34. providers: [
35.   DialogService
36. ],
37. bootstrap: [ AppComponent ]
38. })
39. export class AppModule {
40.   // Diagnostic only: inspect router configuration
41.   constructor(router: Router) {
42.     console.log('Routes: ', JSON.stringify(router.config, undefined, 2));
43.   }
44. }
```

You could try this now and confirm that the `CrisisCenterModule` loads after you click the "Crisis Center" button.

To enable preloading of all lazy loaded modules, import the `PreloadAllModules` token from the Angular router package.

The second argument in the `RouterModule.forRoot` method takes an object for additional configuration options. The `preloadingStrategy` is one of those options. Add the `PreloadAllModules` token to the `forRoot` call:

src/app/app-routing.module.ts (preload all)

```
RouterModule.forRoot(  
  appRoutes,  
  {  
    enableTracing: true, // <-- debugging purposes only  
    preloadingStrategy: PreloadAllModules  
  }  
)
```



This tells the `Router` preloader to immediately load *all* lazy loaded routes (routes with a `loadChildren` property).

When you visit `http://localhost: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.

CanLoad blocks preload

The `PreloadAllModules` strategy does not load feature areas protected by a `CanLoad` guard. This is by design.

You added a `CanLoad` guard to the route in the `AdminModule` a few steps back to block loading of that module until the user is authorized. That `CanLoad` guard takes precedence over the preload strategy.

If you want to preload a module *and* guard against unauthorized access, drop the `canLoad()` guard method and rely on the `canActivate()` guard alone.

Custom Preloading Strategy

Preloading every lazy loaded modules works well in many situations, but it isn't always the right choice, especially on mobile devices and over low bandwidth connections. You may choose to preload only certain feature modules, based on user metrics and other business and technical factors.

You can control what and how the router preloads with a custom preloading strategy.

In this section, you'll add a custom strategy that *only* preloads routes whose `data.preload` flag is set to `true`. Recall that you can add anything to the `data` property of a route.

Set the `data.preload` flag in the `crisis-center` route in the `AppRoutingModule`.

src/app/app-routing.module.ts (route data preload)

```
{
  path: 'crisis-center',
  loadChildren: 'app/crisis-center/crisis-center.module#CrisisCenterModule',
  data: { preload: true }
},
```

Add a new file to the project called `selective-preloading-strategy.ts` and define a `SelectivePreloadingStrategy` service class as follows:

src/app/selective-preloading-strategy.ts (excerpt)

```
import 'rxjs/add/observable/of';
import { Injectable } from '@angular/core';
import { PreloadingStrategy, Route } from '@angular/router';
import { Observable } from 'rxjs/Observable';
```

```
@Injectable()
export class SelectivePreloadingStrategy implements PreloadingStrategy {
  preloadedModules: string[] = [];

  preload(route: Route, load: () => Observable<any>): Observable<any> {
    if (route.data && route.data['preload']) {
      // add the route path to the preloaded module array
      this.preloadedModules.push(route.path);

      // log the route path to the console
      console.log('Preloaded: ' + route.path);

      return load();
    } else {
      return Observable.of(null);
    }
  }
}
```

`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.

src/app/admin/admin-dashboard.component.ts (preloaded modules)

```
import { Component, OnInit }      from '@angular/core';
import { ActivatedRoute }        from '@angular/router';
import { Observable }           from 'rxjs/Observable';

import { SelectivePreloadingStrategy } from '../selective-preloading-strategy';

import 'rxjs/add/operator/map';
```

```
@Component({
  template: `
    <p>Dashboard</p>

    <p>Session ID: {{ sessionId | async }}</p>
    <a id="anchor"></a>
    <p>Token: {{ token | async }}</p>

    Preloaded Modules
    <ul>
      <li *ngFor="let module of modules">{{ module }}</li>
    </ul>
  `

})
export class AdminDashboardComponent implements OnInit {
  sessionId: Observable<string>;
  token: Observable<string>;
  modules: string[];

  constructor(
    private route: ActivatedRoute,
    private preloadStrategy: SelectivePreloadingStrategy
  ) {
    this.modules = preloadStrategy.preloadedModules;
  }

  ngOnInit() {
    // Capture the session ID if available
    this.sessionId = this.route
      .queryParamMap
```

```
.map(params => params.get('session_id') || 'None');

// Capture the fragment if available
this.token = this.route
  .fragment
  .map(fragment => fragment || 'None');
}

}
```

Once the application loads the initial route, the `CrisisCenterModule` is preloaded. Verify this by logging in to the `Admin` feature area and noting that the `crisis-center` is listed in the `Preloaded Modules`. It's also logged to the browser's console.

Migrating URLs with Redirects

You've setup the routes for navigating around your application. You've used navigation imperatively and declaratively to many different routes. But like any application, requirements change over time. You've setup links and navigation to `/heroes` and `/hero/:id` from the `HeroListComponent` and `HeroDetailComponent` components. If there was a requirement that links to `heroes` become `superheroes`, you still want the previous URLs to navigate correctly. You also don't want to go and update every link in your application, so redirects makes refactoring routes trivial.

Changing `/heroes` to `/superheroes`

Let's take the `Hero` routes and migrate them to new URLs. The `Router` checks for redirects in your configuration before navigating, so each redirect is triggered when needed. To support this change, you'll add redirects from the old routes to the new routes in the `heroes-routing.module`.

`src/app/heroes/heroes-routing.module.ts (heroes redirects)`

```
import { NgModule }           from '@angular/core';
import { RouterModule, Routes } from '@angular/router';

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

const heroesRoutes: Routes = [
  { path: 'heroes', redirectTo: '/superheroes' },
  { path: 'hero/:id', redirectTo: '/superhero/:id' },
  { path: 'superheroes', component: HeroListComponent },
  { path: 'superhero/:id', component: HeroDetailComponent }
];

@NgModule({
  imports: [
    RouterModule.forChild(heroesRoutes)
  ],
  exports: [
    RouterModule
  ]
})
export class HeroRoutingModule { }
```

You'll notice two different types of redirects. The first change is from `/heroes` to `/superheroes` without any parameters. This is a straightforward redirect, unlike the change from `/hero/:id` to `/superhero/:id`, which includes the `:id` route parameter. Router redirects also use powerful pattern matching, so the `Router` inspects the URL and replaces route parameters in the `path` with their appropriate destination. Previously, you navigated to a URL such as `/hero/15` with a route parameter `id` of `15`.

The Router also supports [query parameters](#) and the [fragment](#) when using redirects.

- When using absolute redirects, the Router will use the query parameters and the fragment from the redirectTo in the route config.
- When using relative redirects, the Router use the query params and the fragment from the source URL.

Before updating the `app-routing.module.ts`, you'll need to consider an important rule. Currently, our empty path route redirects to `/heroes`, which redirects to `/superheroes`. This *won't* work and is by design as the Router handles redirects once at each level of routing configuration. This prevents chaining of redirects, which can lead to endless redirect loops.

So instead, you'll update the empty path route in `app-routing.module.ts` to redirect to `/superheroes`.

src/app/app-routing.module.ts (superheroes redirect)

```
import { NgModule }           from '@angular/core';
import { RouterModule, Routes } from '@angular/router';

import { ComposeMessageComponent } from './compose-message.component';
import { PageNotFoundComponent } from './not-found.component';

import { CanDeactivateGuard } from './can-deactivate-guard.service';
import { AuthGuard }           from './auth-guard.service';
import { SelectivePreloadingStrategy } from './selective-preloading-strategy';

const appRoutes: Routes = [
  {
    path: 'compose',
    component: ComposeMessageComponent,
  },
  {
    path: '',
    pathMatch: 'full',
    redirectTo: '/superheroes',
  },
]
```

```
        outlet: 'popup'
    },
{
    path: 'admin',
    loadChildren: 'app/admin/admin.module#AdminModule',
    canLoad: [AuthGuard]
},
{
    path: 'crisis-center',
    loadChildren: 'app/crisis-center/crisis-center.module#CrisisCenterModule',
    data: { preload: true }
},
{ path: '', redirectTo: '/superheroes', pathMatch: 'full' },
{ path: '**', component: PageNotFoundComponent }
];

@NgModule({
    imports: [
        RouterModule.forRoot(
            appRoutes,
            {
                enableTracing: true, // <-- debugging purposes only
                preloadingStrategy: SelectivePreloadingStrategy,
            }
        )
    ],
    exports: [
        RouterModule
    ],
})
```

```
        providers: [
          CanDeactivateGuard,
          SelectivePreloadingStrategy
        ]
      })
export class AppRoutingModule { }
```

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`.

src/app/app.component.ts (superheroes active routerLink)

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

@Component({
  selector: 'my-app',
  template: `
    <h1 class="title">Angular Router</h1>
    <nav>
      <a routerLink="/crisis-center" routerLinkActive="active">Crisis Center</a>
      <a routerLink="/superheroes" routerLinkActive="active">Heroes</a>
      <a routerLink="/admin" routerLinkActive="active">Admin</a>
      <a routerLink="/login" routerLinkActive="active">Login</a>
      <a [routerLink]="[{ outlets: { popup: ['compose'] } }]>Contact</a>
    </nav>
    <router-outlet></router-outlet>
    <router-outlet name="popup"></router-outlet>
  `
})
```

```
export class AppComponent {  
}
```

With the redirects setup, all previous routes now point to their new destinations and both URLs still function as intended.

Inspect the router's configuration

You put a lot of effort into configuring the router in several routing module files and were careful to list them [in the proper order](#). Are routes actually evaluated as you planned? How is the router really configured?

You can inspect the router's current configuration any time by injecting it and examining its `config` property. For example, update the `AppModule` as follows and look in the browser console window to see the finished route configuration.

src/app/app.module.ts (inspect the router config)

```
import { Router } from '@angular/router';  
  
export class AppModule {  
    // Diagnostic only: inspect router configuration  
    constructor(router: Router) {  
        console.log('Routes: ', JSON.stringify(router.config, undefined, 2));  
    }  
}
```



Wrap up and final app

You've covered a lot of ground in this guide and the application is too big to reprint here. Please visit the [Router Sample in Plunker](#) / [download example](#) where you can download the final source code.

Appendices

The balance of this guide is a set of appendices that elaborate some of the points you covered quickly above.

The appendix material isn't essential. Continued reading is for the curious.

Appendix: link parameters array

A link parameters array holds the following ingredients for router navigation:

- The *path* of the route to the destination component.
- Required and optional route parameters that go into the route URL.

You can bind the `RouterLink` directive to such an array like this:

src/app/app.component.ts (h-anchor)

```
<a [routerLink]=["/heroes"]>Heroes</a>
```

You've written a two element array when specifying a route parameter like this:

src/app/heroes/hero-list.component.ts (nav-to-detail)

```
<a [routerLink]=["/hero", hero.id]>
  <span class="badge">{{ hero.id }}</span>{{ hero.name }}
</a>
```

You can provide optional route parameters in an object like this:

src/app/app.component.ts (cc-query-params)

```
<a [routerLink]=["/crisis-center", { foo: 'foo' }]>Crisis Center</a>
```



These three examples cover the need for an app with one level routing. The moment you add a child router, such as the crisis center, you create new link array possibilities.

Recall that you specified a default child route for the crisis center so this simple `RouterLink` is fine.

src/app/app.component.ts (cc-anchor-w-default)

```
<a [routerLink]=["/crisis-center"]>Crisis Center</a>
```



Parse it out.

- The first item in the array identifies the parent route (`/crisis-center`).
- There are no parameters for this parent route so you're done with it.
- There is no default for the child route so you need to pick one.
- You're navigating to the `CrisisListComponent`, whose route path is `/`, but you don't need to explicitly add the slash.
- Voilà! `['/crisis-center']`.

Take it a step further. Consider the following router link that navigates from the root of the application down to the *Dragon Crisis*:

src/app/app.component.ts (Dragon-anchor)

```
<a [routerLink]=["/crisis-center", 1]>Dragon Crisis</a>
```



- The first item in the array identifies the parent route (`/crisis-center`).
- There are no parameters for this parent route so you're done with it.
- The second item identifies the child route details about a particular crisis (`/:id`).
- The details child route requires an `id` route parameter.
- You added the `id` of the *Dragon Crisis* as the second item in the array (`1`).
- The resulting path is `/crisis-center/1`.

If you wanted to, you could redefine the `AppComponent` template with *Crisis Center* routes exclusively:

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

```
template: `

<h1 class="title">Angular Router</h1>
<nav>
  <a [routerLink]=["'/crisis-center']>Crisis Center</a>
  <a [routerLink]=["'/crisis-center/1', { foo: 'foo' }]>Dragon Crisis</a>
  <a [routerLink]=["'/crisis-center/2']>Shark Crisis</a>
</nav>
<router-outlet></router-outlet>
`
```

In sum, you can write applications with one, two or more levels of routing. The link parameters array affords the flexibility to represent any routing depth and any legal sequence of route paths, (required) router parameters, and (optional) route parameter objects.

Appendix: *LocationStrategy* and browser URL styles

When the router navigates to a new component view, it updates the browser's location and history with a URL for that view. This is a strictly local URL. The browser shouldn't send this URL to the server and should not reload the page.

Modern HTML5 browsers support `history.pushState`, a technique that changes a browser's location and history without triggering a server page request. The router can compose a "natural" URL that is indistinguishable from one that would otherwise require a page load.

Here's the *Crisis Center* URL in this "HTML5 pushState" style:

```
localhost:3002/crisis-center/
```



Older browsers send page requests to the server when the location URL changes *unless* the change occurs after a "#" (called the "hash"). Routers can take advantage of this exception by composing in-application route URLs with hashes. Here's a "hash URL" that routes to the *Crisis Center*.

```
localhost:3002/src/#/crisis-center/
```



The router supports both styles with two `LocationStrategy` providers:

1. `PathLocationStrategy` —the default "HTML5 pushState" style.
2. `HashLocationStrategy` —the "hash URL" style.

The `RouterModule.forRoot` function sets the `LocationStrategy` to the `PathLocationStrategy`, making it the default strategy. You can switch to the `HashLocationStrategy` with an override during the bootstrapping process if you prefer it.

Learn about providers and the bootstrap process in the [Dependency Injection guide](#).

Which strategy is best?

You must choose a strategy and you need to make the right call early in the project. It won't be easy to change later once the application is in production and there are lots of application URL references in the wild.

Almost all Angular projects should use the default HTML5 style. It produces URLs that are easier for users to understand. And it preserves the option to do *server-side rendering* later.

Rendering critical pages on the server is a technique that can greatly improve perceived responsiveness when the app first loads. An app that would otherwise take ten or more seconds to start could be rendered on the server and delivered to the user's device in less than a second.

This option is only available if application URLs look like normal web URLs without hashes (#) in the middle.

Stick with the default unless you have a compelling reason to resort to hash routes.

HTML5 URLs and the *<base href>*

While the router uses the [HTML5 pushState](#) style by default, you *must* configure that strategy with a base href.

The preferred way to configure the strategy is to add a [*<base href> element*](#) tag in the `<head>` of the `index.html`.

```
src/index.html (base-href)
<base href="/">
```

Without that tag, the browser may not be able to load resources (images, CSS, scripts) when "deep linking" into the app. Bad things could happen when someone pastes an application link into the browser's address bar or clicks such a link in an email.

Some developers may not be able to add the `<base>` element, perhaps because they don't have access to `<head>` or the `index.html`.

Those developers may still use HTML5 URLs by taking two remedial steps:

1. Provide the router with an appropriate `APP_BASE_HREF` value.
2. Use *root URLs* for all web resources: CSS, images, scripts, and template HTML files.

HashLocationStrategy

You can go old-school with the `HashLocationStrategy` by providing the `useHash: true` in an object as the second argument of the `RouterModule.forRoot` in the `AppModule`.

src/app/app.module.ts (hash URL strategy)

```
import { NgModule }           from '@angular/core';
import { BrowserModule }     from '@angular/platform-browser';
import { FormsModule }        from '@angular/forms';
import { Routes, RouterModule } from '@angular/router';

import { AppComponent }       from './app.component';
import { PageNotFoundComponent } from './not-found.component';

const routes: Routes = [
];

@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    RouterModule.forRoot(routes, { useHash: true }) // ...#/crisis-center/
  ],
  declarations: [
]
```

```
  AppComponent,  
  PageNotFoundComponent  
,  
 providers: [  
  
,  
 bootstrap: [ AppComponent ]  
})  
export class AppModule { }
```


This is the archived documentation for Angular v4. Please visit [angular.io](#) to see documentation for the current version of Angular.

Testing

[Contents >](#)

[Live examples](#)

[Introduction to Angular Testing](#)

[Tools and technologies](#)

•••

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.

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 / download example](#) .
- [The first component spec with inline template / download example](#) .
- [A component spec with external template / download example](#) .
- [The QuickStart seed's AppComponent spec / download example](#) .
- [The sample application to be tested / download example](#) .

- All specs that test the sample application / [download example](#) .
 - A grab bag of additional specs / [download example](#) .
-

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.

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 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 <i>within</i> the Angular environment.

Karma

The [karma test runner](#) 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.

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](#).

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.

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`.

```
src/app/1st.spec.ts
```

```
describe('1st tests', () => {  
  it('true is true', () => expect(true).toBe(true));  
});
```



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

DEBUG

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.

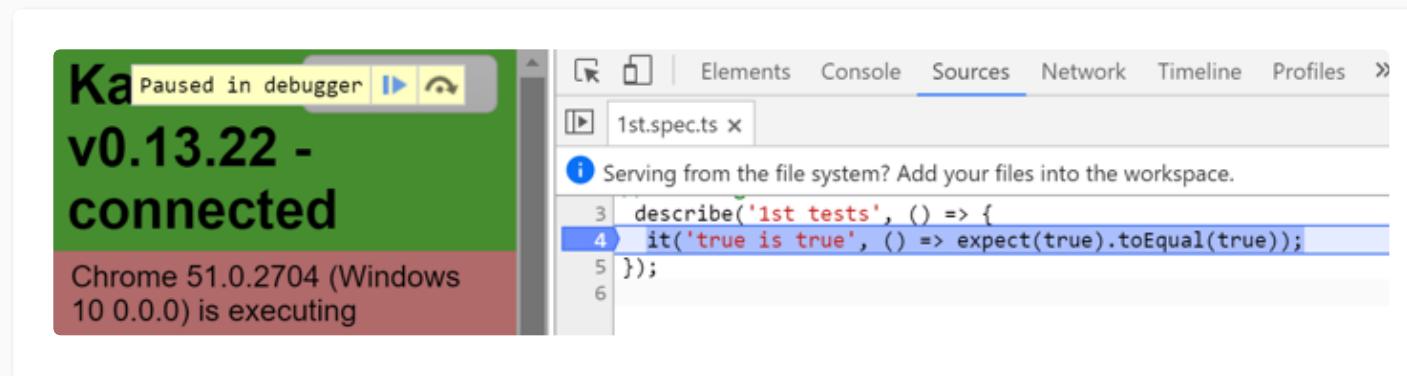
Restore the expectation from `false` back to `true`. Both processes detect the change, re-run, and karma reports complete success.

The console log can be quite long. Keep your eye on the last line. When all is well, it reads
SUCCESS .

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.



Try the live example

You can also try this test as a [First spec](#) / [download example](#) in plunker. All of the tests in this guide are available as [live examples](#).

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.

`src/app/banner-inline.component.ts`

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

@Component({
  selector: 'app-banner',
  template: '<h1>{{title}}</h1>'
})
export class BannerComponent {
  title = 'Test Tour of Heroes';
}
```

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.

src/app/banner-inline.component.spec.ts (imports)

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { By } from '@angular/platform-browser';
import { DebugElement } from '@angular/core';

import { BannerComponent } from './banner-inline.component';
```

Here's the `describe` and the `beforeEach` that precedes the tests:

src/app/banner-inline.component.spec.ts (beforeEach)

```
describe('BannerComponent (inline template)', () => {

  let comp: BannerComponent;
  let fixture: ComponentFixture<BannerComponent>;
  let de: DebugElement;
  let el: HTMLElement;

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [ BannerComponent ], // declare the test component
    });

    fixture = TestBed.createComponent(BannerComponent);

    comp = fixture.componentInstance; // BannerComponent test instance
```

```
// query for the title <h1> by CSS element selector
de = fixture.debugElement.query(By.css('h1'));
el = de.nativeElement;
});
});
```

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](#) 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.

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, nor `configureTestingModule` nor any of the `override...` methods. If you try, `TestBed` throws an error.

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.

The tests

Jasmine runs the `beforeEach` function before each of these tests

src/app/banner-inline.component.spec.ts (tests)

```
it('should display original title', () => {
  fixture.detectChanges();
  expect(el.textContent).toContain(comp.title);
});

it('should display a different test title', () => {
  comp.title = 'Test Title';
  fixture.detectChanges();
  expect(el.textContent).toContain('Test Title');
});
```

These tests ask the `DebugElement` for the native HTML element to satisfy their expectations.

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:

src/app/banner-inline.component.spec.ts (no detectChanges)

```
it('no title in the DOM until manually call `detectChanges``', () => {
  expect(el.textContent).toEqual('');
});
```

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*.

Try the live example

Take a moment to explore this component spec as a [Spec for component with inline template](#) / [download example](#) and lock in these fundamentals of component unit testing.

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:

src/app/banner.component.detect-changes.spec.ts (import)

```
import { ComponentFixtureAutoDetect } from '@angular/core/testing';
```

Then add it to the `providers` array of the testing module configuration:

src/app/banner.component.detect-changes.spec.ts (AutoDetect)

```
TestBed.configureTestingModule({
  declarations: [ BannerComponent ],
  providers: [
    { provide: ComponentFixtureAutoDetect, useValue: true }
  ]
})
```

Here are three tests that illustrate how automatic change detection works.

src/app/banner.component.detect-changes.spec.ts (AutoDetect Tests)

```
it('should display original title', () => {
  // Hooray! No `fixture.detectChanges()` needed
  expect(el.textContent).toContain(comp.title);
```

```
});

it('should still see original title after comp.title change', () => {
  const oldTitle = comp.title;
  comp.title = 'Test Title';
  // Displayed title is old because Angular didn't hear the change :(
  expect(el.textContent).toContain(oldTitle);
});

it('should display updated title after detectChanges', () => {
  comp.title = 'Test Title';
  fixture.detectChanges(); // detect changes explicitly
  expect(el.textContent).toContain(comp.title);
});
```

The first test shows the benefit of automatic change detection.

The second and third test reveal an important limitation. The Angular testing environment does *not* know that the test changed the component's `title`. The `ComponentFixtureAutoDetect` service responds to *asynchronous activities* such as promise resolution, timers, and DOM events. But a direct, synchronous update of the component property is invisible. The test must call `fixture.detectChanges()` manually to trigger another cycle of change detection.

Rather than wonder when the test fixture will or won't perform change detection, the samples in this guide *always call* `detectChanges()` *explicitly*. There is no harm in calling `detectChanges()` more often than is strictly necessary.

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.

src/app/banner.component.ts

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

@Component({
  selector: 'app-banner',
  templateUrl: './banner.component.html',
  styleUrls: ['./banner.component.css']
})
export class BannerComponent {
  title = 'Test Tour of Heroes';
}
```

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.

src/app/banner.component.spec.ts (first `beforeEach`)

```
// async beforeEach
beforeEach(async(() => {
  TestBed.configureTestingModule({
    declarations: [ BannerComponent ], // declare the test component
  })
  .compileComponents(); // compile template and css
}));
```

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.

src/app/banner.component.detect-changes.spec.ts

```
import { async } from '@angular/core/testing';
```

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.

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, nor `configureTestingModule` nor any of the `override...` methods. The `TestBed` throws an error if you try.

The second synchronous `beforeEach`

A *synchronous* `beforeEach` containing the remaining setup steps follows the asynchronous `beforeEach`.

`src/app/banner.component.spec.ts (second beforeEach)`

```
// synchronous beforeEach
beforeEach(() => {
  fixture = TestBed.createComponent(BannerComponent);

  comp = fixture.componentInstance; // BannerComponent test instance

  // query for the title <h1> by CSS element selector
  de = fixture.debugElement.query(By.css('h1'));
  el = de.nativeElement;
});

});
```



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.

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.

Try the live example

Take a moment to explore this component spec as a [Spec for component with external template / download example](#).

The [Quickstart seed](#) provides a similar test of its `AppComponent` as you can see in [this QuickStart seed spec / download example](#). 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 refactor the template into a separate file. The tests in this guide only call `compileComponents` when necessary.

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`:

src/app/welcome.component.ts

```
import { Component, OnInit } from '@angular/core';
import { UserService } from './model';

@Component({
  selector: 'app-welcome',
  template: '<h3 class="welcome" ><i>{{welcome}}</i></h3>'
})
export class WelcomeComponent implements OnInit {
  welcome = '-- not initialized yet --';
  constructor(private userService: UserService) { }

  ngOnInit(): void {
    this.userService.user$.subscribe(user => {
      this.welcome = `Welcome ${user.name}!`;
    });
  }
}
```

```
    this.welcome = this.userService.isLoggedIn ?  
      'Welcome, ' + this.userService.user.name :  
      'Please log in.';  
  }  
}  
}
```

The `WelcomeComponent` has decision logic that interacts with the service, logic that makes this component worth testing. Here's the testing module configuration for the spec file,

`src/app/welcome.component.spec.ts` :

```
src/app/welcome.component.spec.ts  
  
 TestBed.configureTestingModule({  
   declarations: [ WelcomeComponent ],  
   // providers: [ UserService ] // NO! Don't provide the real service!  
   // Provide a test-double instead  
   providers: [ {provide: UserService, useValue: userServiceStub} ]  
});
```

This time, in addition to declaring the *component-under-test*, the configuration adds a `UserService` provider to the `providers` list. But not the real `UserService`.

Provide service test doubles

A *component-under-test* doesn't have to be injected with real services. In fact, it is usually better if they are test doubles (stubs, fakes, spies, or mocks). The purpose of the spec is to test the component, not the service, and real services can be trouble.

Injecting the real `UserService` could be a nightmare. The real service might ask the user for login credentials and attempt to reach an authentication server. These behaviors can be hard to intercept. It is far

easier and safer to create and register a test double in place of the real `UserService`.

This particular test suite supplies a minimal `UserService` stub that satisfies the needs of the `WelcomeComponent` and its tests:

src/app/welcome.component.spec.ts

```
userServiceStub = {  
  isLoggedIn: true,  
  user: { name: 'Test User' }  
};
```



Get injected services

The tests need access to the (stub) `UserService` injected into the `WelcomeComponent`.

Angular has a hierarchical injection system. There can be injectors at multiple levels, from the root injector created by the `TestBed` down through the component tree.

The safest way to get the injected service, the way that *always works*, is to get it from the injector of the *component-under-test*. The component injector is a property of the fixture's `DebugElement`.

WelcomeComponent's injector

```
// UserService actually injected into the component  
userService = fixture.debugElement.injector.get(UserService);
```



TestBed.get

You *may* also be able to get the service from the root injector via `TestBed.get`. This is easier to remember and less verbose. But it only works when Angular injects the component with the service instance in the test's root injector. 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:

TestBed injector

```
// UserService from the root injector
userService = TestBed.get(UserService);
```

The `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](#), which explains why you must get the service from the component's injector instead.

Always get the service from an injector

Do *not* reference the `userServiceStub` object that's provided to the testing module in the body of your test. It does not work! The `userService` instance injected into the component is a completely *different* object, a clone of the provided `userServiceStub`.

src/app/welcome.component.spec.ts

```
it('stub object and injected UserService should not be the same', () => {
  expect(userServiceStub === userService).toBe(false);

  // Changing the stub object has no effect on the injected service
```

```
    userServiceStub.isLoggedIn = false;
    expect(userService.isLoggedIn).toBe(true);
});
```

Final setup and tests

Here's the complete `beforeEach` using `TestBed.get`:

src/app/welcome.component.spec.ts

```
beforeEach(() => {
  // stub UserService for test purposes
  userServiceStub = {
    isLoggedIn: true,
    user: { name: 'Test User' }
  };

  TestBed.configureTestingModule({
    declarations: [ WelcomeComponent ],
    providers:    [ {provide: UserService, useValue: userServiceStub} ]
  });

  fixture = TestBed.createComponent(WelcomeComponent);
  comp    = fixture.componentInstance;

  // UserService from the root injector
  userService = TestBed.get(UserService);

  // get the "welcome" element by CSS selector (e.g., by class name)
```

```
        de = fixture.debugElement.query(By.css('.welcome'));
        el = de.nativeElement;
    });
}
```

And here are some tests:

src/app/welcome.component.spec.ts

```
it('should welcome the user', () => {
    fixture.detectChanges();
    const content = el.textContent;
    expect(content).toContain('Welcome', '"Welcome ..."]');
    expect(content).toContain('Test User', 'expected name');
});

it('should welcome "Bubba"', () => {
    userService.user.name = 'Bubba'; // welcome message hasn't been shown yet
    fixture.detectChanges();
    expect(el.textContent).toContain('Bubba');
});

it('should request login if not logged in', () => {
    userService.isLoggedIn = false; // welcome message hasn't been shown yet
    fixture.detectChanges();
    const content = el.textContent;
    expect(content).not.toContain('Welcome', 'not welcomed');
    expect(content).toMatch(/log in/i, '"log in"');
});
```

The first is a sanity test; it confirms that the stubbed `UserService` is called and working.

The second parameter to the Jasmine matcher (e.g., 'expected name') is an optional 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.

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`.

src/app/shared/twain.component.ts

```
@Component({
  selector: 'twain-quote',
  template: '<p class="twain"><i>{{quote}}</i></p>'
})
export class TwainComponent implements OnInit {
  intervalId: number;
  quote = '...';
```

```
constructor(private twainService: TwainService) { }

ngOnInit(): void {
  this.twainService.getQuote().then(quote => this.quote = quote);
}

}
```

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:

src/app/shared/twain.component.spec.ts (setup)

```
beforeEach(() => {
  TestBed.configureTestingModule({
    declarations: [ TwainComponent ],
    providers:    [ TwainService ],
  });

  fixture = TestBed.createComponent(TwainComponent);
  comp    = fixture.componentInstance;

  // TwainService actually injected into the component
  twainService = fixture.debugElement.injector.get(TwainService);

  // Setup spy on the `getQuote` method
  spy = spyOn(twainService, 'getQuote')
    .and.returnValue(Promise.resolve(testQuote));
```

```
// Get the Twain quote element by CSS selector (e.g., by class name)
de = fixture.debugElement.query(By.css('.twain'));
el = de.nativeElement;
});
```

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.

src/app/shared/twain.component.spec.ts

```
spy = spyOn(twainService, 'getQuote')
    .and.returnValue(Promise.resolve(testQuote));
```

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](#).

Here are the tests with commentary to follow:

src/app/shared/twain.component.spec.ts (tests)

```
1. it('should not show quote before OnInit', () => {
2.   expect(el.textContent).toBe('', 'nothing displayed');
3.   expect(spy.calls.any()).toBe(false, 'getQuote not yet called');
4. });
5.
6. it('should still not show quote after component initialized', () => {
7.   fixture.detectChanges();
8.   // getQuote service is async => still has not returned with quote
9.   expect(el.textContent).toBe('...', 'no quote yet');
10.  expect(spy.calls.any()).toBe(true, 'getQuote called');
11. });
12.
13. it('should show quote after getQuote promise (async)', async(() => {
14.   fixture.detectChanges();
15.
16.   fixture.whenStable().then(() => { // wait for async getQuote
17.     fixture.detectChanges();          // update view with quote
18.     expect(el.textContent).toBe(testQuote);
19.   });
20. }));
21.
22. it('should show quote after getQuote promise (fakeAsync)', fakeAsync(() => {
23.   fixture.detectChanges();
24.   tick();                         // wait for async getQuote
25.   fixture.detectChanges(); // update view with quote
26.   expect(el.textContent).toBe(testQuote);
27. }));

```



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*.

The `async` function in *it*

Notice the `async` in the third test.

src/app/shared/twain.component.spec.ts (async test)

```
it('should show quote after getQuote promise (async)', async(() => {
  fixture.detectChanges();

  fixture.whenStable().then(() => { // wait for async getQuote
    fixture.detectChanges();           // update view with quote
    expect(el.textContent).toBe(testQuote);
  });
}));
```

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](#), removes this artifact and affords a more linear coding experience.

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.

The *fakeAsync* function

The fourth test verifies the same component behavior in a different way.

src/app/shared/twain.component.spec.ts (fakeAsync test)

```
it('should show quote after getQuote promise (fakeAsync)', fakeAsync(() => {
  fixture.detectChanges();
```



```
    tick();           // wait for async getQuote
    fixture.detectChanges(); // update view with quote
    expect(el.textContent).toBe(testQuote);
  }));
}
```

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`.

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.

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:

src/app/shared/twain.component.spec.ts (done test)

```
it('should show quote after getQuote promise (done)', (done: any) => {
  fixture.detectChanges();

  // get the spy promise and wait for it to resolve
  spy.calls.mostRecent().returnValue.then(() => {
    fixture.detectChanges() // update view with quote
    expect(el.textContent).toBe(testQuote);
    done();
  });
});
```

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.

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:

src/app/dashboard/dashboard.component.html (excerpt)

```
<dashboard-hero *ngFor="let hero of heroes" class="col-1-4"
  [hero]=hero (selected)="gotoDetail($event)" >
</dashboard-hero>
```

The `DashboardHeroComponent` appears in an `*ngFor` repeater, which sets each component's `hero` input property to the looping value and listens for the component's `selected` event.

Here's the component's definition:

src/app/dashboard/dashboard-hero.component.ts (component)

```
@Component({
  selector:    'dashboard-hero',
  templateUrl: './dashboard-hero.component.html',
  styleUrls: [ './dashboard-hero.component.css' ]
})
export class DashboardHeroComponent {
  @Input() hero: Hero;
  @Output() selected = new EventEmitter<Hero>();
  click() { this.selected.emit(this.hero); }
}
```

While testing a component this simple has little intrinsic value, it's worth knowing how. You can use one of these approaches:

- Test it as used by `DashboardComponent`.
- Test it as a stand-alone component.
- Test it as used by a substitute for `DashboardComponent`.

A quick look at the `DashboardComponent` constructor discourages the first approach:

src/app/dashboard/dashboard.component.ts (constructor)

```
constructor(
  private router: Router,
  private heroService: HeroService) {
```

The `DashboardComponent` depends on the Angular router and the `HeroService`. You'd probably have to replace them both with test doubles, which is a lot of work. The router seems particularly challenging.

The discussion below covers testing components that require the router.

The immediate goal is to test the `DashboardHeroComponent`, not the `DashboardComponent`, so, try the second and third options.

Test `DashboardHeroComponent` stand-alone

Here's the spec file setup.

src/app/dashboard/dashboard-hero.component.spec.ts (setup)

```
// async beforeEach
beforeEach( async(() => {
  TestBed.configureTestingModule({
    declarations: [ DashboardHeroComponent ],
  })
  .compileComponents(); // compile template and css
}));

// synchronous beforeEach
beforeEach(() => {
  fixture = TestBed.createComponent(DashboardHeroComponent);
  comp    = fixture.componentInstance;
  heroEl  = fixture.debugElement.query(By.css('.hero')); // find hero element

  // pretend that it was wired to something that supplied a hero
  expectedHero = new Hero(42, 'Test Name');
  comp.hero = expectedHero;
})
```

```
fixture.detectChanges(); // trigger initial data binding
});
```

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:

src/app/dashboard/dashboard-hero.component.spec.ts (name test)

```
it('should display hero name', () => {
  const expectedPipedName = expectedHero.name.toUpperCase();
  expect(heroEl.nativeElement.textContent).toContain(expectedPipedName);
});
```

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:

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

```
<div (click)="click()" class="hero">
  {{hero.name | uppercase}}
</div>
```

This small test demonstrates how Angular tests can verify a component's visual representation—something not possible with [isolated unit 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:

src/app/dashboard/dashboard-hero.component.spec.ts (click test)

```
it('should raise selected event when clicked', () => {
  let selectedHero: Hero;
  comp.selected.subscribe((hero: Hero) => selectedHero = hero);

  heroEl.triggerEventHandler('click', null);
  expect(selectedHero).toBe(expectedHero);
});
```

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.

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.

src/app/dashboard/dashboard-hero.component.spec.ts

```
heroEl.triggerEventHandler('click', null);
```



The test assumes (correctly in this case) that the runtime event handler—the component's `click()` method—doesn't care about the event object.

Other handlers are less forgiving. For example, the `RouterLink` directive expects an object with a `button` property that identifies which mouse button was pressed. This directive throws an error if the event object doesn't do this correctly.

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:

testing/index.ts (click helper)

```
/** Button events to pass to `DebugElement.triggerEventHandler` for RouterLink event handler */
export const ButtonClickEvents = {
  left: { button: 0 },
  right: { button: 2 }
};

/** Simulate element click. Defaults to mouse left-button click event. */
export function click(el: DebugElement | HTMLElement, eventObj: any =
```



```
ButtonClickEvents.left): void {
  if (el instanceof HTMLElement) {
    el.click();
  } else {
    el.triggerEventHandler('click', eventObj);
  }
}
```

The first parameter is the *element-to-click*. If you wish, you can pass a custom event object as the second parameter. The default is a (partial) [left-button mouse event object](#) accepted by many handlers including the `RouterLink` directive.

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.

src/app/dashboard/dashboard-hero.component.spec.ts (click test revised)

```
it('should raise selected event when clicked', () => {
  let selectedHero: Hero;
  comp.selected.subscribe((hero: Hero) => selectedHero = hero);

  click(heroEl); // triggerEventHandler helper
  expect(selectedHero).toBe(expectedHero);
});
```

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:

src/app/dashboard/dashboard-hero.component.spec.ts (test host)

```
@Component({
  template: `
    <dashboard-hero [hero]="hero" (selected)="onSelected($event)"></dashboard-
    hero>`
})
class TestHostComponent {
  hero = new Hero(42, 'Test Name');
  selectedHero: Hero;
  onSelected(hero: Hero) { this.selectedHero = hero; }
}
```

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:

src/app/dashboard/dashboard-hero.component.spec.ts (test host setup)

```
beforeEach( async(() => {
  TestBed.configureTestingModule({
    declarations: [ DashboardHeroComponent, TestHostComponent ], // declare both
  }).compileComponents();
});

beforeEach(() => {
  // create TestHostComponent instead of DashboardHeroComponent
  fixture = TestBed.createComponent(TestHostComponent);
  testHost = fixture.componentInstance;
  heroEl = fixture.debugElement.query(By.css('.hero')); // find hero
  fixture.detectChanges(); // trigger initial data binding
});
```

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:

src/app/dashboard/dashboard-hero.component.spec.ts (test-host)

```
it('should display hero name', () => {
  const expectedPipedName = testHost.hero.name.toUpperCase();
  expect(heroEl.nativeElement.textContent).toContain(expectedPipedName);
});

it('should raise selected event when clicked', () => {
  click(heroEl);
  // selected hero should be the same data bound hero
  expect(testHost.selectedHero).toBe(testHost.hero);
});
```

Only the selected event test differs. It confirms that the selected `DashboardHeroComponent` hero really does find its way up through the event binding to the host component.

Test a routed component

Testing the actual `DashboardComponent` seemed daunting because it injects the `Router`.

src/app/dashboard/dashboard.component.ts (constructor)

```
constructor(
  private router: Router,
  private heroService: HeroService) {
```

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`

src/app/dashboard/dashboard.component.ts (goToDetail)

```
gotoDetail(hero: Hero) {  
  let url = `/heroes/${hero.id}`;  
  this.router.navigateByUrl(url);  
}
```

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:

src/app/dashboard/dashboard.component.spec.ts (Router Stub)

```
class RouterStub {  
  navigateByUrl(url: string) { return url; }  
}
```

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.

src/app/dashboard/dashboard.component.spec.ts (compile and create)

```
beforeEach( async(() => {  
  TestBed.configureTestingModule({  
    providers: [  
  ]  
})});
```

```

        { provide: HeroService, useClass: FakeHeroService },
        { provide: Router,      useClass: RouterStub }
    ]
})
.compileComponents().then(() => {
  fixture = TestBed.createComponent(DashboardComponent);
  comp = fixture.componentInstance;
});

```

The following test clicks the displayed hero and confirms (with the help of a spy) that `Router.navigateByUrl` is called with the expected url.

src/app/dashboard/dashboard.component.spec.ts (navigate test)

```

it('should tell ROUTER to navigate when hero clicked',
  inject([Router], (router: Router) => { // ...
  
```



```

    const spy = spyOn(router, 'navigateByUrl');

    heroClick(); // trigger click on first inner <div class="hero">

    // args passed to router.navigateByUrl()
    const navArgs = spy.calls.first().args[0];

    // expecting to navigate to id of the component's first hero
    const id = comp.heroes[0].id;
    expect(navArgs).toBe('/heroes/' + id,
      'should nav to HeroDetail for first hero');
  }));

```

The `inject` function

Notice the `inject` function in the second `it` argument.

src/app/dashboard/dashboard.component.spec.ts

```
it('should tell ROUTER to navigate when hero clicked',
  inject([Router], (router: Router) => { // ...
}));
```



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:

Component's injector

```
// UserService actually injected into the component
userService = fixture.debugElement.injector.get(UserService);
```

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`.

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:

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

```
constructor(
  private heroDetailsService: HeroDetailService,
  private route: ActivatedRoute,
```

```
private router: Router) {  
}
```

HeroDetailComponent subscribes to ActivatedRoute.params changes in its ngOnInit method.

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

```
ngOnInit(): void {  
  // get hero when 'id' param changes  
  this.route.paramMap.subscribe(p => this.getHero(p.has('id') && p.get('id')));  
}
```

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 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*.

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`:

testing/router-stubs.ts (ActivatedRouteStub)

```
import { BehaviorSubject } from 'rxjs/BehaviorSubject';
import { convertToParamMap, ParamMap } from '@angular/router';

@Injectable()
export class ActivatedRouteStub {

  // ActivatedRoute.paramMap is Observable
  private subject = new BehaviorSubject(convertToParamMap(this._testParamMap));
  paramMap = this.subject.asObservable();

  // Test parameters
  private _testParamMap: ParamMap;
  get testParamMap() { return this._testParamMap; }
  set testParamMap(params: {}) {
    this._testParamMap = convertToParamMap(params);
    this.subject.next(this._testParamMap);
  }

  // ActivatedRoute.snapshot.paramMap
  get snapshot() {
    return { paramMap: this.testParamMap };
  }
}
```

```
    }  
}
```

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` 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.

Testing with the *Observable* test double

Here's a test demonstrating the component's behavior when the observed `id` refers to an existing hero:

`src/app/hero/hero-detail.component.spec.ts (existing id)`



```
describe('when navigate to existing hero', () => {
  let expectedHero: Hero;

  beforeEach( async(() => {
    expectedHero = firstHero;
    activatedRoute.testParamMap = { id: expectedHero.id };
    createComponent();
  }));

  it('should display that hero\'s name', () => {
    expect(page.nameDisplay.textContent).toBe(expectedHero.name);
  });
});
```

The `createComponent` method and `page` object are discussed [in the next section](#). 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.

src/app/hero/hero-detail.component.spec.ts (bad id)

```
describe('when navigate to non-existent hero id', () => {
  beforeEach( async(() => {
    activatedRoute.testParamMap = { id: 99999 };
    createComponent();
  }));
});
```

```
it('should try to navigate back to hero list', () => {
  expect(page.gotoSpy.calls.any()).toBe(true, 'comp.gotoList called');
  expect(page.navSpy.calls.any()).toBe(true, 'router.navigate called');
});
});
```

While this app doesn't have a route to the `HeroDetailComponent` that omits the `id` parameter, it might add such a route someday. The component should do something reasonable when there is no `id`.

In this implementation, the component should create and display a new hero. New heroes have `id=0` and a blank `name`. This test confirms that the component behaves as expected:

src/app/hero/hero-detail.component.spec.ts (no id)

```
describe('when navigate with no hero id', () => {
  beforeEach(async( createComponent ));

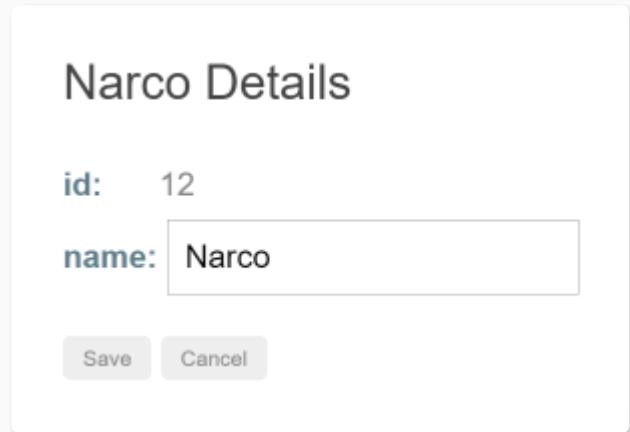
  it('should have hero.id === 0', () => {
    expect(comp.hero.id).toBe(0);
  });

  it('should display empty hero name', () => {
    expect(page.nameDisplay.textContent).toBe('');
  });
});
```

Inspect and download *all* of the guide's application test code with this [live example / download example](#).

Use a *page* object to simplify setup

The `HeroDetailComponent` is a simple view with a title, two hero fields, and two buttons.



A screenshot of a web application component titled "Narco Details". It displays the英雄ID as "id: 12" and the英雄Name as "name: Narco" in a text input field. At the bottom are two buttons: "Save" and "Cancel".

But there's already plenty of template complexity.

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

```
<div *ngIf="hero">
  <h2><span>{{hero.name | titlecase}}</span> Details</h2>
  <div>
    <label>id: </label>{{hero.id}}</div>
  <div>
    <label for="name">name: </label>
    <input id="name" [(ngModel)]="hero.name" placeholder="name" />
  </div>
  <button (click)="save()">Save</button>
  <button (click)="cancel()">Cancel</button>
</div>
```

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 `` 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`

src/app/hero/hero-detail.component.spec.ts (Page)

```
class Page {
  gotoSpy:      jasmine.Spy;
  navSpy:       jasmine.Spy;

  saveBtn:      DebugElement;
  cancelBtn:    DebugElement;
  nameDisplay:  HTMLElement;
  nameInput:    HTMLInputElement;

  constructor() {
    const router = TestBed.get(Router); // get router from root injector
    this.gotoSpy = spyOn(comp, 'gotoList').and.callThrough();
    this.navSpy = spyOn(router, 'navigate');
  }

  /** Add page elements after hero arrives */
}
```

```

addPageElements() {
  if (comp.hero) {
    // have a hero so these elements are now in the DOM
    const buttons = fixture.debugElement.queryAll(By.css('button'));
    this.saveBtn = buttons[0];
    this.cancelBtn = buttons[1];
    this.nameDisplay = fixture.debugElement.query(By.css('span')).nativeElement;
    this.nameInput = fixture.debugElement.query(By.css('input')).nativeElement;
  }
}

```

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.

src/app/hero/hero-detail.component.spec.ts (createComponent)

```

/** Create the HeroDetailComponent, initialize it, set test variables */
function createComponent() {
  fixture = TestBed.createComponent(HeroDetailComponent);
  comp = fixture.componentInstance;
  page = new Page();

  // 1st change detection triggers ngOnInit which gets a hero
  fixture.detectChanges();
  return fixture.whenStable().then(() => {
    // 2nd change detection displays the async-fetched hero
    fixture.detectChanges();
    page.addPageElements();
  });
}

```

```
});  
}  
}
```

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.

src/app/hero/hero-detail.component.spec.ts (selected tests)

```
it('should display that hero\'s name', () => {  
  expect(page.nameDisplay.textContent).toBe(expectedHero.name);  
});  
  
it('should navigate when click cancel', () => {  
  click(page.cancelBtn);  
  expect(page.navSpy.calls.any()).toBe(true, 'router.navigate called');  
});  
  
it('should save when click save but not navigate immediately', () => {  
  // Get service injected into component and spy on its `saveHero` method.  
  // It delegates to fake `HeroService.updateHero` which delivers a safe test  
  // result.  
  const hds = fixture.debugElement.injector.get(HeroDetailsService);  
  const saveSpy = spyOn(hds, 'saveHero').and.callThrough();  
  
  click(page.saveBtn);  
  expect(saveSpy.calls.any()).toBe(true, 'HeroDetailsService.save called');  
  expect(page.navSpy.calls.any()).toBe(false, 'router.navigate not called');
```

```
});

it('should navigate when click save and save resolves', fakeAsync(() => {
  click(page.saveBtn);
  tick(); // wait for async save to complete
  expect(page.navSpy.calls.any()).toBe(true, 'router.navigate called');
}));

it('should convert hero name to Title Case', () => {
  const inputName = 'quick BROWN fox';
  const titleCaseName = 'Quick Brown Fox';

  // simulate user entering new name into the input box
  page.nameInput.value = inputName;

  // dispatch a DOM event so that Angular learns of input value change.
  page.nameInput.dispatchEvent(newEvent('input'));

  // Tell Angular to update the output span through the title pipe
  fixture.detectChanges();

  expect(page.nameDisplay.textContent).toBe(titleCaseName);
});
```

Setup with module imports

Earlier component tests configured the testing module with a few `declarations` like this:

`src/app/dashboard/dashboard-hero.component.spec.ts (config)`

```
// async beforeEach
beforeEach( async(() => {
  TestBed.configureTestingModule({
    declarations: [ DashboardHeroComponent ],
  })
  .compileComponents(); // compile template and css
}));
```

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:

src/app/hero/hero-detail.component.spec.ts (FormsModule setup)

```
beforeEach( async(() => {
  TestBed.configureTestingModule({
    imports:      [ FormsModule ],
    declarations: [ HeroDetailComponent, TitleCasePipe ],
    providers:   [
```

```
    { provide: ActivatedRoute, useValue: activatedRoute },
    { provide: HeroService,    useClass: FakeHeroService },
    { provide: Router,         useClass: RouterStub},
  ]
})
.compileComponents();
}});
```

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:

src/app/hero/hero-detail.component.spec.ts (SharedModule setup)

```
beforeEach( async(() => {
  TestBed.configureTestingModule({
    imports:      [ SharedModule ],
    declarations: [ HeroDetailComponent ],
    providers: [
      { provide: ActivatedRoute, useValue: activatedRoute },
      { provide: HeroService,    useClass: FakeHeroService },
      { provide: Router,         useClass: RouterStub},
    ]
})
.compileComponents();
}});
```

It's a bit tighter and smaller, with fewer import statements (not shown).

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:

src/app/hero/hero-detail.component.spec.ts (HeroModule setup)

```
beforeEach( async(() => {
   TestBed.configureTestingModule({
    imports: [ HeroModule ],
    providers: [
      { provide: ActivatedRoute, useValue: activatedRoute },
      { provide: HeroService, useClass: FakeHeroService },
      { provide: Router, useClass: RouterStub },
    ]
  })
  .compileComponents();
});
```

That's *really* crisp. Only the *test doubles* in the `providers` remain. Even the `HeroDetailComponent` declaration is gone.

In fact, if you try to declare it, Angular 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.

Override a component's providers

The `HeroDetailComponent` provides its own `HeroDetailsService`.

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

```
@Component({
  selector:    'app-hero-detail',
  templateUrl: './hero-detail.component.html',
  styleUrls:  ['./hero-detail.component.css'],
  providers:  [ HeroDetailsService ]
})
export class HeroDetailComponent implements OnInit {
  constructor(
    private heroDetailsService: HeroDetailsService,
    private route: ActivatedRoute,
    private router: Router) {
  }
}
```

It's not possible to stub the component's `HeroDetailsService` 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`.

src/app/hero/hero-detail.service.ts (prototype)

```
@Injectable()
export class HeroDetailService {
  constructor(private heroService: HeroService) { }
  /* . . . */
}
```

The [previous test configuration](#) 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:

src/app/hero/hero-detail.component.spec.ts (Override setup)

```
beforeEach( async(() => {
  TestBed.configureTestingModule({
    imports: [ HeroModule ],
    providers: [
      { provide: ActivatedRoute, useValue: activatedRoute },
      { provide: Router, useClass: RouterStub },
    ]
  })
  // Override component's own provider
  .overrideComponent(HeroDetailComponent, {
    set: {
      providers: [
        { provide: HeroDetailService, useClass: HeroDetailServiceSpy }
      ]
    }
  })
  .compileComponents();
}));
```

Notice that `TestBed.configureTestingModule` no longer provides a (fake) `HeroService` because it's **not needed**.

The *overrideComponent* method

Focus on the `overrideComponent` method.

src/app/hero/hero-detail.component.spec.ts (overrideComponent)

```
.overrideComponent(HeroDetailComponent, {  
  set: {  
    providers: [  
      { provide: HeroDetailsService, useClass: HeroDetailsServiceSpy }  
    ]  
  }  
})
```

It takes two arguments: the component type to override (`HeroDetailComponent`) and an override metadata object. The [override metadata object](#) is a generic defined as follows:

```
type MetadataOverride = {  
  add?: 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[];
```

```
...
```

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:

src/app/hero/hero-detail.component.spec.ts (HeroDetailServiceSpy)

```
class HeroDetailServiceSpy {
  testHero = new Hero(42, 'Test Hero');

  getHero = jasmine.createSpy('getHero').and.callFake(
    () => Promise
      .resolve(true)
      .then(() => Object.assign({}, this.testHero))
  );

  saveHero = jasmine.createSpy('saveHero').and.callFake(
    (hero: Hero) => Promise
      .resolve(true)
      .then(() => Object.assign(this.testHero, hero))
  );
}
```

```
);  
}
```

The override tests

Now the tests can control the component's hero directly by manipulating the spy-stub's `testHero` and confirm that service methods were called.

src/app/hero/hero-detail.component.spec.ts (override tests)

```
let hdsSpy: HeroDetailServiceSpy;  
  
beforeEach( async(() => {  
  createComponent();  
  // get the component's injected HeroDetailsService  
  hdsSpy = fixture.debugElement.injector.get(HeroDetailService) as any;  
}));  
  
it('should have called `getHero`', () => {  
  expect(hdsSpy.getHero.calls.count()).toBe(1, 'getHero called once');  
});  
  
it('should display stub hero\'s name', () => {  
  expect(page.nameDisplay.textContent).toBe(hdsSpy.testHero.name);  
});  
  
it('should save stub hero change', fakeAsync(() => {  
  const origName = hdsSpy.testHero.name;  
  const newName = 'New Name';  
})
```

```
page.nameInput.value = newName;
page.nameInput.dispatchEvent(newEvent('input')); // tell Angular

expect(comp.hero.name).toBe(newName, 'component hero has new name');
expect(hdsSpy.testHero.name).toBe(origName, 'service hero unchanged before save');

click(page.saveBtn);
expect(hdsSpy.saveHero.calls.count()).toBe(1, 'saveHero called once');

tick(); // wait for async save to complete
expect(hdsSpy.testHero.name).toBe(newName, 'service hero has new name after
save');
expect(page.navSpy.calls.any()).toBe(true, 'router.navigate called');
}));
```

More overrides

The `TestBed.overrideComponent` method can be called multiple times for the same or different components. The `TestBed` offers similar `overrideDirective`, `overrideModule`, and `overridePipe` methods for digging into and replacing parts of these other classes.

Explore the options and combinations on your own.

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.

src/app/app.component.html

```
<app-banner></app-banner>  
<app-welcome></app-welcome>  
  
<nav>  
  <a routerLink="/dashboard">Dashboard</a>  
  <a routerLink="/heroes">Heroes</a>  
  <a routerLink="/about">About</a>  
</nav>  
  
<router-outlet></router-outlet>
```



The component class does nothing.

src/app/app.component.ts

```
import { Component } from '@angular/core';  
  
@Component({  
  selector: 'my-app',  
  templateUrl: './app.component.html'  
})  
export class AppComponent { }
```



Unit tests can confirm that the anchors are wired properly without engaging the router. See why this is worth doing [below](#).

Stubbing unneeded components

The test setup should look familiar.

src/app/app.component.spec.ts (Stub Setup)

```
beforeEach( async(() => {
  TestBed.configureTestingModule({
    declarations: [
      AppComponent,
      BannerComponent, WelcomeStubComponent,
      RouterLinkStubDirective, RouterOutletStubComponent
    ]
  })

  .compileComponents()
  .then(() => {
    fixture = TestBed.createComponent(AppComponent);
    comp = fixture.componentInstance;
  });
}));
```

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.

Stubbing the *RouterLink*

The `RouterLinkStubDirective` contributes substantively to the test:

testing/router-stubs.ts (RouterLinkStubDirective)

```
@Directive({
  selector: '[routerLink]',
  host: {
    '(click)': 'onClick()'
  }
})
export class RouterLinkStubDirective {
  @Input('routerLink') linkParams: any;
  navigatedTo: any = null;

  onClick() {
    this.navigatedTo = this.linkParams;
  }
}
```

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.

By.directive and injected directives

A little more setup triggers the initial data binding and gets references to the navigation links:

src/app/app.component.spec.ts (test setup)

```
beforeEach(() => {
  // trigger initial data binding
  fixture.detectChanges();

  // find DebugElements with an attached RouterLinkStubDirective
  linkDes = fixture.debugElement
    .queryAll(By.directive(RouterLinkStubDirective));

  // get the attached link directive instances using the DebugElement injectors
  links = linkDes
    .map(de => de.injector.get(RouterLinkStubDirective) as RouterLinkStubDirective);
});
```

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.

Here are some tests that leverage this setup:

src/app/app.component.spec.ts (selected tests)

```
it('can get RouterLinks from template', () => {
  expect(links.length).toBe(3, 'should have 3 links');
  expect(links[0].linkParams).toBe('/dashboard', '1st link should go to Dashboard');
  expect(links[1].linkParams).toBe('/heroes', '1st link should go to Heroes');
```

```
});

it('can click Heroes link in template', () => {
  const heroesLinkDe = linkDes[1];
  const heroesLink = links[1];

  expect(heroesLink.navigatedTo).toBeNull('link should not have navigated yet');

  heroesLinkDe.triggerEventHandler('click', null);
  fixture.detectChanges();

  expect(heroesLink.navigatedTo).toBe('/heroes');
});
```

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.

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`.

"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.

```
1. import { NO_ERRORS_SCHEMA }           from '@angular/core';
2. import { AppComponent }              from './app.component';
3. import { RouterOutletStubComponent } from '../testing';
4.
5. beforeEach( async(() => {
6.   TestBed.configureTestingModule({
7.     declarations: [ AppComponent, RouterLinkStubDirective ],
8.     schemas:      [ NO_ERRORS_SCHEMA ]
9.   })
10.
11.  .compileComponents()
12.  .then(() => {
13.    fixture = TestBed.createComponent(AppComponent);
14.    comp    = fixture.componentInstance;
15.  });
16. }));

```



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.

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.

src/app/shared/highlight.directive.ts

```
import { Directive, ElementRef, Input, OnChanges } from '@angular/core';

@Directive({ selector: '[highlight]' })
/** Set backgroundColor for the attached element to highlight color
 * and set the element's customProperty to true */
export class HighlightDirective implements OnChanges {

  defaultColor = 'rgb(211, 211, 211)'; // lightgray

  @Input('highlight') bgColor: string;

  constructor(private el: ElementRef) {
    el.nativeElement.style.customProperty = true;
  }

  ngOnChanges() {
    this.el.nativeElement.style.backgroundColor = this.bgColor || this.defaultColor;
  }
}
```

It's used throughout the application, perhaps most simply in the `AboutComponent` :

src/app/about.component.ts

```
import { Component } from '@angular/core';
@Component({
  template: `
    <h2 highlight="skyblue">About</h2>
    <twain-quote></twain-quote>
    <p>All about this sample</p>`
})
export class AboutComponent { }
```

Testing the specific use of the `HighlightDirective` within the `AboutComponent` requires only the techniques explored above (in particular the "Shallow test" approach).

src/app/about.component.spec.ts

```
beforeEach(() => {
  fixture = TestBed.configureTestingModule({
    declarations: [ AboutComponent, HighlightDirective ],
    schemas:      [ NO_ERRORS_SCHEMA ]
  })
  .createComponent(AboutComponent);
  fixture.detectChanges(); // initial binding
});

it('should have skyblue <h2>', () => {
  const de = fixture.debugElement.query(By.css('h2'));
  const bgColor = de.nativeElement.style.backgroundColor;
```

```
    expect(bgColor).toBe('skyblue');
});
```

However, testing a single use case is unlikely to explore the full range of a directive's capabilities. Finding and testing all components that use the directive is tedious, brittle, and almost as unlikely to afford full coverage.

[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.

src/app/shared/highlight.directive.spec.ts (TestComponent)

```
@Component({
  template: `
    <h2 highlight="yellow">Something Yellow</h2>
    <h2 highlight>The Default (Gray)</h2>
    <h2>No Highlight</h2>
    <input #box [highlight]="box.value" value="cyan"/>`
})
class TestComponent { }
```

Something Yellow

The Default (Gray)

No Highlight

cyan

The `<input>` case binds the `HighlightDirective` to the name of a color value in the input box. The initial value is the word "cyan" which should be the background color of the input box.

Here are some tests of this component:

src/app/shared/highlight.directive.spec.ts (selected tests)

```
1. beforeEach(() => {
2.   fixture = TestBed.configureTestingModule({
3.     declarations: [ HighlightDirective, TestComponent ]
4.   })
5.   .createComponent(TestComponent);
6.
7.   fixture.detectChanges(); // initial binding
8.
9.   // all elements with an attached HighlightDirective
10.  des = fixture.debugElement.queryAll(By.directive(HighlightDirective));
11.
12.  // the h2 without the HighlightDirective
```

```
13. bareH2 = fixture.debugElement.query(By.css('h2:not([highlight])'));
14. });
15.
16. // color tests
17. it('should have three highlighted elements', () => {
18.   expect(des.length).toBe(3);
19. });
20.
21. it('should color 1st <h2> background "yellow"', () => {
22.   const bgColor = des[0].nativeElement.style.backgroundColor;
23.   expect(bgColor).toBe('yellow');
24. });
25.
26. it('should color 2nd <h2> background w/ default color', () => {
27.   const dir = des[1].injector.get(HighlightDirective) as HighlightDirective;
28.   const bgColor = des[1].nativeElement.style.backgroundColor;
29.   expect(bgColor).toBe(dir.defaultColor);
30. });
31.
32. it('should bind <input> background to value color', () => {
33.   // easier to work with nativeElement
34.   const input = des[2].nativeElement as HTMLInputElement;
35.   expect(input.style.backgroundColor).toBe('cyan', 'initial
backgroundColor');
36.
37.   // dispatch a DOM event so that Angular responds to the input value
change.
38.   input.value = 'green';
39.   input.dispatchEvent(newEvent('input'));
40.   fixture.detectChanges();
```

```
41.  
42.   expect(input.style.backgroundColor).toBe('green', 'changed  
      backgroundColor');  
43. });  
44.  
45.  
46. it('bare <h2> should not have a customProperty', () => {  
47.   expect(bareH2.properties['customProperty']).toBeUndefined();  
48. });
```

A few techniques are noteworthy:

- The `By.directive` predicate is a great way to get the elements that have this directive *when their element types are unknown*.
- The `:not pseudo-class` in `By.css('h2:not([highlight])')` helps find `<h2>` elements that *do not* have the directive. `By.css('*:not([highlight])')` finds *any* element that does not have the directive.
- `DebugElement.styles` affords access to element styles even in the absence of a real browser, thanks to the `DebugElement` abstraction. But feel free to exploit the `nativeElement` when that seems easier or more clear than the abstraction.
- Angular adds a directive to the injector of the element to which it is applied. The test for the default color uses the injector of the second `<h2>` to get its `HighlightDirective` instance and its `defaultColor`.
- `DebugElement.properties` affords access to the artificial custom property that is set by the directive.

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.

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.

src/app/bag/bag.no-testbed.spec.ts

1. // Straight Jasmine - no imports from Angular test libraries
- 2.



```
3. describe('FancyService without the TestBed', () => {
4.   let service: FancyService;
5.
6.   beforeEach(() => { service = new FancyService(); });
7.
8.   it('#getValue should return real value', () => {
9.     expect(service.getValue()).toBe('real value');
10.    });
11.
12.  it('#getAsyncValue should return async value', (done: DoneFn) => {
13.    service.getAsyncValue().then(value => {
14.      expect(value).toBe('async value');
15.      done();
16.    });
17.  });
18.
19.  it('#getTimeoutValue should return timeout value', (done: DoneFn) => {
20.    service = new FancyService();
21.    service.getTimeoutValue().then(value => {
22.      expect(value).toBe('timeout value');
23.      done();
24.    });
25.  });
26.
27.  it('#getObservableValue should return observable value', (done: DoneFn) =>
28.  {
29.    service.getObservableValue().subscribe(value => {
30.      expect(value).toBe('observable value');
31.      done();
32.    });
33.  });
34.
```

```
32. });
33.
34.});
```

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`.

`src/app/bag/bag.no-testbed.spec.ts` (Isolated) `src/app/bag/bag.spec.ts` (with Angular testing utilities)

```
it('#getTimeoutValue should return timeout value', (done: DoneFn) => {
  service = new FancyService();
  service.getTimeoutValue().then(value => {
    expect(value).toBe('timeout value');
    done();
  });
});
```



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.

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:

src/app/bag/bag.ts

```
@Injectable()
export class DependentService {
  constructor(private dependentService: FancyService) { }
  getValue() { return this.dependentService.getValue(); }
}
```

It delegates its only method, `getValue`, to the injected `FancyService`.

Here are several ways to test it.

src/app/bag/bag.no-testbed.spec.ts

```
1. describe('DependentService without the TestBed', () => {
2.   let service: DependentService;
3.
4.   it('#getValue should return real value by way of the real FancyService',
5.     () => {
6.       service = new DependentService(new FancyService());
7.       expect(service.getValue()).toBe('real value');
8.     });
9.
10.  it('#getValue should return faked value by way of a fakeService', () => {
11.    service = new DependentService(new FakeFancyService());
12.    expect(service.getValue()).toBe('faked value');
```

```
12.  });
13.
14.  it('#getValue should return faked value from a fake object', () => {
15.    const fake = { getValue: () => 'fake value' };
16.    service = new DependentService(fake as FancyService);
17.    expect(service.getValue()).toBe('fake value');
18.  });
19.
20.  it('#getValue should return stubbed value from a FancyService spy', () =>
{
21.    const fancy = new FancyService();
22.    const stubValue = 'stub value';
23.    const spy = spyOn(fancy, 'getValue').and.returnValue(stubValue);
24.    service = new DependentService(fancy);
25.
26.    expect(service.getValue()).toBe(stubValue, 'service returned stub
value');
27.    expect(spy.calls.count()).toBe(1, 'stubbed method was called once');
28.    expect(spy.calls.mostRecent().returnValue).toBe(stubValue);
29.  });
30.});
```

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*.

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.

src/app/shared/title-case.pipe.ts

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({name: 'titlecase', pure: false})
/** Transform to Title Case: uppercase the first letter of the words in a string.*/
export class TitleCasePipe implements PipeTransform {

  transform(input: string): string {
    return input.length === 0 ? '' :
      input.replace(/\w\S*/g, (txt => txt[0].toUpperCase() +
      txt.substr(1).toLowerCase() ));
  }
}
```

Anything that uses a regular expression is worth testing thoroughly. Use simple Jasmine to explore the expected cases and the edge cases.

src/app/shared/title-case.pipe.spec.ts

```
1. describe('TitleCasePipe', () => {
```

```
2. // This pipe is a pure, stateless function so no need for BeforeEach
3. let pipe = new TitleCasePipe();
4.
5. it('transforms "abc" to "Abc"', () => {
6.   expect(pipe.transform('abc')).toBe('Abc');
7. });
8.
9. it('transforms "abc def" to "Abc Def"', () => {
10.   expect(pipe.transform('abc def')).toBe('Abc Def');
11. });
12.
13. // ... more tests ...
14. });
```

Write 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:

src/app/hero/hero-detail.component.spec.ts (pipe test)

```
1. it('should convert hero name to Title Case', () => {
2.   const inputName = 'quick BROWN fox';
3.   const titleCaseName = 'Quick Brown Fox';
4.
5.   // simulate user entering new name into the input box
6.   page.nameInput.value = inputName;
7.
```

```
8. // dispatch a DOM event so that Angular learns of input value change.  
9. page.nameInput.dispatchEvent(newEvent('input'));  
10.  
11. // Tell Angular to update the output span through the title pipe  
12. fixture.detectChanges();  
13.  
14. expect(page.nameDisplay.textContent).toBe(titleCaseName);  
15.});
```

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.

src/app/bag/bag.ts (ButtonComp)

```
@Component({  
  selector: 'button-comp',  
  template: `  
    <button (click)="clicked()">Click me!</button>  
    <span>{{message}}</span>  
  `})  
export class ButtonComponent {  
  isOn = false;  
  clicked() { this.isOn = !this.isOn; }  
  get message() { return `The light is ${this.isOn ? 'On' : 'Off'}`; }  
}
```

The following Angular test demonstrates that clicking a button in the template leads to an update of the on-screen message.

src/app/bag/bag.spec.ts (ButtonComp)

```
it('should support clicking a button', () => {
  const fixture = TestBed.createComponent(ButtonComponent);
  const btn = fixture.debugElement.query(By.css('button'));
  const span = fixture.debugElement.query(By.css('span')).nativeElement;

  fixture.detectChanges();
  expect(span.textContent).toMatch(/is off/i, 'before click');

  click(btn);
  fixture.detectChanges();
  expect(span.textContent).toMatch(/is on/i, 'after click');
});
```

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 ``). 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.

src/app/bag/bag.no-testbed.spec.ts (ButtonComp)

```
describe('ButtonComp', () => {
  let comp: ButtonComponent;
```

```
beforeEach(() => comp = new ButtonComponent());

it('#isOn should be false initially', () => {
  expect(comp.isOn).toBe(false);
});

it('#clicked() should set #isOn to true', () => {
  comp.clicked();
  expect(comp.isOn).toBe(true);
});

it('#clicked() should set #message to "is on"', () => {
  comp.clicked();
  expect(comp.message).toMatch(/is on/i);
});

it('#clicked() should toggle #isOn', () => {
  comp.clicked();
  expect(comp.isOn).toBe(true);
  comp.clicked();
  expect(comp.isOn).toBe(false);
});
});
```

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.

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
<code>async</code>	Runs the body of a test (<code>it</code>) or setup (<code>beforeEach</code>) function within a special <i>async test zone</i> . See discussion above .
<code>fakeAsync</code>	Runs the body of a test (<code>it</code>) within a special <i>fakeAsync test zone</i> , enabling a linear control flow coding style. See discussion above .
<code>tick</code>	Simulates the passage of time and the completion of pending asynchronous activities by flushing both <i>timer</i> and <i>micro-task</i> queues within the <i>fakeAsync test zone</i> . The curious, dedicated reader might enjoy this lengthy blog post, " 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 .

inject	Injects one or more services from the current <code>TestBed</code> injector into a test function. See above .
discardPeriodicTasks	<p>When a <code>fakeAsync</code> test ends with pending timer event <i>tasks</i> (queued <code>setTimeout</code> and <code>setInterval</code> callbacks), the test fails with a clear error message.</p> <p>In general, a test should end with no queued tasks. When pending timer tasks are expected, call <code>discardPeriodicTasks</code> to flush the <i>task</i> queue and avoid the error.</p>
flushMicrotasks	<p>When a <code>fakeAsync</code> test ends with pending <i>micro-tasks</i> such as unresolved promises, the test fails with a clear error message.</p> <p>In general, a test should wait for micro-tasks to finish. When pending microtasks are expected, call <code>flushMicrotasks</code> to flush the <i>micro-task</i> queue and avoid the error.</p>
ComponentFixtureAutoDetect	A provider token for a service that turns on automatic change detection .
get TestBed	Gets the current instance of the <code>TestBed</code> . Usually unnecessary because the static class methods of the <code>TestBed</code> class are typically sufficient. The <code>TestBed</code> instance exposes a few rarely used members that are not available as static methods.

TestBed class summary

The `TestBed` class is one of the principal Angular testing utilities. Its API is quite large and can be overwhelming until you've explored it, a little at a time. Read the early part of this guide first to get the basics before trying to absorb the full API.

The module definition passed to `configureTestingModule` is a subset of the `@NgModule` metadata properties.

```
type TestModuleMetadata = {  
  providers?: any[];  
  declarations?: any[];  
  imports?: any[];  
  schemas?: Array<SchemaMetadata | any[]>;  
};
```

Each override method takes a `MetadataOverride<T>` where `T` is the kind of metadata appropriate to the method, that is, the parameter of an `@NgModule`, `@Component`, `@Directive`, or `@Pipe`.

```
type MetadataOverride = {  
  add?: T;  
  remove?: T;  
  set?: T;  
};
```

The `TestBed` API consists of static class methods that either update or reference a *global* instance of the `TestBed`.

Internally, all static methods cover methods of the current runtime `TestBed` instance, which is also returned by the `get TestBed()` function.

Call `TestBed` methods *within* a `beforeEach()` to ensure a fresh start before each individual test.

Here are the most important static methods, in order of likely utility.

Methods	Description
<code>configureTestingModule</code>	<p>The testing shims (<code>karma-test-shim</code>, <code>browser-test-shim</code>) establish the initial test environment and a default testing module. The default testing module is configured with basic declaratives and some Angular service substitutes that every tester needs.</p> <p>Call <code>configureTestingModule</code> to refine the testing module configuration for a particular set of tests by adding and removing imports, declarations (of components, directives, and pipes), and providers.</p>
<code>compileComponents</code>	<p>Compile the testing module asynchronously after you've finished configuring it. You must call this method if <i>any</i> of the testing module components have a <code>templateUrl</code> or <code>styleUrls</code> because fetching component template and style files is necessarily asynchronous. See above.</p> <p>After calling <code>compileComponents</code>, the <code>TestBed</code> configuration is frozen for the duration of the current spec.</p>
<code>createComponent</code>	<p>Create an instance of a component of type <code>T</code> based on the current <code>TestBed</code> configuration. After calling <code>compileComponent</code>, the <code>TestBed</code> configuration is frozen for the duration of the current spec.</p>
<code>overrideModule</code>	<p>Replace metadata for the given <code>NgModule</code>. Recall that modules can import other modules. The <code>overrideModule</code> method can reach deeply into the current testing module to modify one of these inner modules.</p>

overrideComponent	Replace metadata for the given component class, which could be nested deeply within an inner module.		
overrideDirective	Replace metadata for the given directive class, which could be nested deeply within an inner module.		
overridePipe	Replace metadata for the given pipe class, which could be nested deeply within an inner module.		
get	<p>Retrieve a service from the current <code>TestBed</code> injector.</p> <p>The <code>inject</code> function is often adequate for this purpose. But <code>inject</code> throws an error if it can't provide the service.</p> <p>What if the service is optional?</p> <p>The <code>TestBed.get</code> method takes an optional second parameter, the object to return if Angular can't find the provider (<code>null</code> in this example):</p> <div data-bbox="601 979 1531 1224" style="background-color: #0078D4; color: white; padding: 10px; border-radius: 10px;"><p>src/app/bag/bag.spec.ts</p><div style="background-color: #F0F0F0; padding: 10px; border-radius: 10px; margin-top: 10px;"><p>The code sample is missing for testing/src/app/bag/bag.spec.ts#testbed-get</p></div></div> <p>After calling <code>get</code>, the <code>TestBed</code> configuration is frozen for the duration of the current spec.</p> <tr><td>initTestEnvironment</td><td>Initialize the testing environment for the entire test run.</td></tr>	initTestEnvironment	Initialize the testing environment for the entire test run.
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-<platform_name>/testing/<platform_name>`.

`resetTestEnvironment`

Reset the initial test environment, including the default testing module.

A few of the `TestBed` instance methods are not covered by static `TestBed` *class* methods. These are rarely needed.

The *ComponentFixture*

The `TestBed.createComponent<T>` creates an instance of the component `T` and returns a strongly typed `ComponentFixture` for that component.

The `ComponentFixture` properties and methods provide access to the component, its DOM representation, and aspects of its Angular environment.

ComponentFixture properties

Here are the most important properties for testers, in order of likely utility.

Properties	Description

componentInstance	The instance of the component class created by <code>TestBed.createComponent</code> .
debugElement	The <code>DebugElement</code> associated with the root element of the component. The <code>debugElement</code> provides insight into the component and its DOM element during test and debugging. It's a critical property for testers. The most interesting members are covered below .
nativeElement	The native DOM element at the root of the component.
changeDetectorRef	The <code>ChangeDetectorRef</code> for the component. The <code>ChangeDetectorRef</code> is most valuable when testing a component that has the <code>ChangeDetectionStrategy.OnPush</code> method or the component's change detection is under your programmatic control.

ComponentFixture methods

The *fixture* methods cause Angular to perform certain tasks on the component tree. Call these method to trigger Angular behavior in response to simulated user action.

Here are the most useful methods for testers.

Methods	Description
<code>detectChanges</code>	Trigger a change detection cycle for the component. Call it to initialize the component (it calls <code>ngOnInit</code>) and after your test code, change the component's data bound property values. Angular can't see

that you've changed `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 exception 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](#).

`destroy`

Trigger component destruction.

DebugElement

The `DebugElement` provides crucial insights into the component's DOM representation.

From the test root component's `DebugElement` returned by `fixture.debugElement`, you can walk (and query) the fixture's entire element and component subtrees.

Here are the most useful `DebugElement` members for testers, in approximate order of utility:

Member	Description
<code>nativeElement</code>	The corresponding DOM element in the browser (null for WebWorkers).
<code>query</code>	Calling <code>query(predicate: Predicate<DebugElement>)</code> returns the first <code>DebugElement</code> that matches the <code>predicate</code> at any depth in the subtree.
<code>queryAll</code>	Calling <code>queryAll(predicate: Predicate<DebugElement>)</code> returns all <code>DebugElements</code> that matches the <code>predicate</code> at any depth in subtree.
<code>injector</code>	The host dependency injector. For example, the root element's component instance injector.
<code>componentInstance</code>	The element's own component instance, if it has one.
<code>context</code>	An object that provides parent context for this element. Often an ancestor component instance that governs this element. When an element is repeated within <code>*ngFor</code> , the context is an <code>NgForRow</code> whose <code>\$implicit</code> property is the value of the row instance value. For example, the <code>hero</code> in <code>*ngFor="let hero of heroes"</code> .

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](#).

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.

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":

src/app/bag/bag.spec.ts

```
// Filter for DebugElements with a #content reference
const contentRefs = el.queryAll( de => de.references['content']);
```



The Angular `By` class has three static methods for common predicates:

- `By.all` - return all elements.
- `By.css(selector)` - return elements with matching CSS selectors.
- `By.directive(directive)` - return elements that Angular matched to an instance of the directive class.

src/app/hero/hero-list.component.spec.ts

```
// Can find DebugElement either by css selector or by directive
const h2        = fixture.debugElement.query(By.css('h2'));
const directive = fixture.debugElement.query(By.directive(HighlightDirective));
```



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
<code>karma.conf.js</code>	<p>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.</p> <p>It loads three other setup files: <code>systemjs.config.js</code> <code>systemjs.config.extras.js</code> * <code>karma-test-shim.js</code></p>
<code>karma-test-shim.js</code>	<p>This shim prepares karma specifically for the Angular test environment and launches karma itself. It loads the <code>systemjs.config.js</code> file as part of that process.</p>
<code>systemjs.config.js</code>	<p>SystemJS 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 <code>systemjs.config.js</code> you installed during setup.</p>
<code>systemjs.config.extras.js</code>	<p>An optional file that supplements the SystemJS configuration in <code>systemjs.config.js</code> with configuration for the specific needs of the application itself.</p> <p>A stock <code>systemjs.config.js</code> can't anticipate those needs. You fill the gaps here.</p>

The sample version for this guide adds the model barrel to the `SystemJs packages` configuration.

systemjs.config.extras.js

```
/** App specific SystemJS configuration */
System.config({
  packages: {
    // barrels
    'app/model': {main: 'index.js', defaultExtension:'js'},
    'app/model/testing': {main: 'index.js', defaultExtension:'js'}
  }
});
```



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.

This is the archived documentation for Angular v4. Please visit angular.io to see documentation for the current version of Angular.

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

```
<section *ngIf="showSection">
```

```
import { CommonModule } from '@angular/common';
```

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

	<code>@Directive({ property1: value1, ... })</code>
<code>selector: '.cool-button:not(a)'</code>	<p>Specifies a CSS selector that identifies this directive within a template. Supported selectors include <code>element</code>, <code>[attribute]</code>, <code>.class</code>, and <code>:not()</code>.</p> <p>Does not support parent-child relationship selectors.</p>
<code>providers: [MyService, { provide: ... }]</code>	List of dependency injection providers for this directive and its children.
Component configuration	
	<code>@Component</code> extends <code>@Directive</code> , so the <code>@Directive</code> configuration applies to components as well
<code>moduleId: module.id</code>	If set, the <code>templateUrl</code> and <code>styleUrl</code> are resolved relative to the component.
<code>viewProviders: [MyService, { provide: ... }]</code>	List of dependency injection providers scoped to this component's view.
<code>template: 'Hello {{name}}'</code> <code>templateUrl: 'my-component.html'</code>	Inline template or external template URL of the component's view.
<code>styles: ['.primary {color: red}']</code> <code>styleUrls: ['my-component.css']</code>	List of inline CSS styles or external stylesheet URLs for styling the component's view.

Class field decorators for directives and components

```
@Input() myProperty;
```

```
import { Input, ... } from '@angular/core';
```

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.

<code>ngAfterContentChecked() { ... }</code>	Called after every check of the component's or directive's content.
<code>ngAfterViewInit() { ... }</code>	Called after <code>ngAfterContentInit</code> when the component's view has been initialized. Applies to components only.
<code>ngAfterViewChecked() { ... }</code>	Called after every check of the component's view. Applies to components only.
<code>ngOnDestroy() { ... }</code>	Called once, before the instance is destroyed.

Dependency injection configuration	
<code>{ provide: MyService, useClass: MyMockService }</code>	Sets or overrides the provider for <code>MyService</code> to the <code>MyMockService</code> class.
<code>{ provide: MyService, useFactory: myFactory }</code>	Sets or overrides the provider for <code>MyService</code> to the <code>myFactory</code> factory function.
<code>{ provide: MyValue, useValue: 41 }</code>	Sets or overrides the provider for <code>MyValue</code> to the value <code>41</code> .

Routing and navigation	
	<pre>import { Routes, RouterModule, ... } from '@angular/router';</pre>

```
const routes: Routes = [
{ path: '', component: HomeComponent },
{ path: 'path/:routeParam', component: MyComponent },
{ path: 'staticPath', component: ... },
{ path: '**', component: ... },
{ path: 'oldPath', redirectTo: '/staticPath' },
{ path: ..., component: ..., data: { message: 'Custom' } }
];
]);
```

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

Configures routes for the application. Supports static, parameterized, redirect, and wildcard routes. Also supports custom route data and resolve.

```
<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">
```

```
<a [routerLink]="[ '/path' ]"
routerLinkActive="active">
```

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.

The provided classes are added to the element when the `routerLink` becomes the current active route.

```
class CanActivateGuard implements CanActivate {
canActivate(
```

An interface for defining a class that the router should call first to determine if it should activate this component. Should

```
route: ActivatedRouteSnapshot,  
state: RouterStateSnapshot  
) : Observable<boolean>|Promise<boolean>|boolean { ... }  
}
```

```
{ path: ..., canActivate: [CanActivateGuard] }
```

```
class CanDeactivateGuard implements CanDeactivate<T> {  
canDeactivate(  
component: T,  
route: ActivatedRouteSnapshot,  
state: RouterStateSnapshot  
) : Observable<boolean>|Promise<boolean>|boolean { ... }  
}
```

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

```
class CanActivateChildGuard implements CanActivateChild {  
canActivateChild(  
route: ActivatedRouteSnapshot,  
state: RouterStateSnapshot  
) : Observable<boolean>|Promise<boolean>|boolean { ... }  
}
```

```
{ path: ..., canActivateChild: [CanActivateGuard],  
children: ... }
```

```
class ResolveGuard implements Resolve<T> {
```

return a boolean or an Observable/Promise that resolves to a boolean.

An interface for defining a class that the router should call first to determine if it should deactivate this component after a navigation. Should return a boolean or an Observable/Promise that resolves to a boolean.

An interface for defining a class that the router should call first to determine if it should activate the child route. Should return a boolean or an Observable/Promise that resolves to a boolean.

An interface for defining a class that the router should call

```
resolve(  
  route: ActivatedRouteSnapshot,  
  state: RouterStateSnapshot  
) : Observable<any>|Promise<any>|any { ... }  
}
```

```
{ path: ..., resolve: [ResolveGuard] }
```

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.

