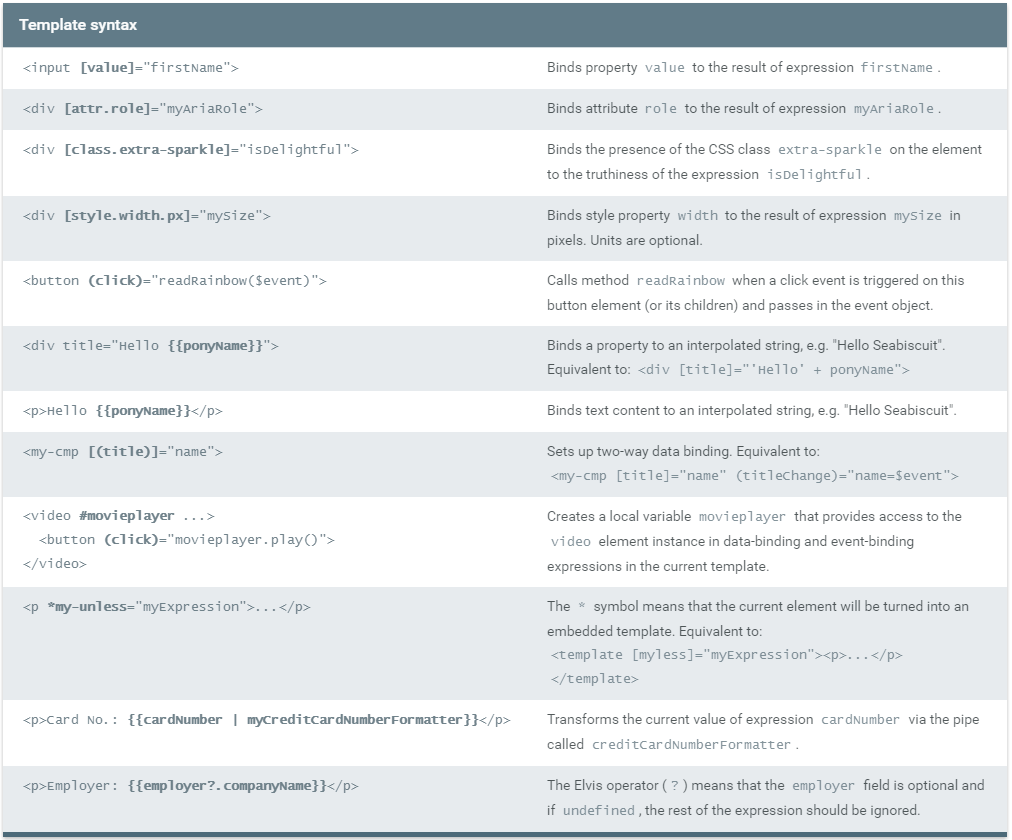
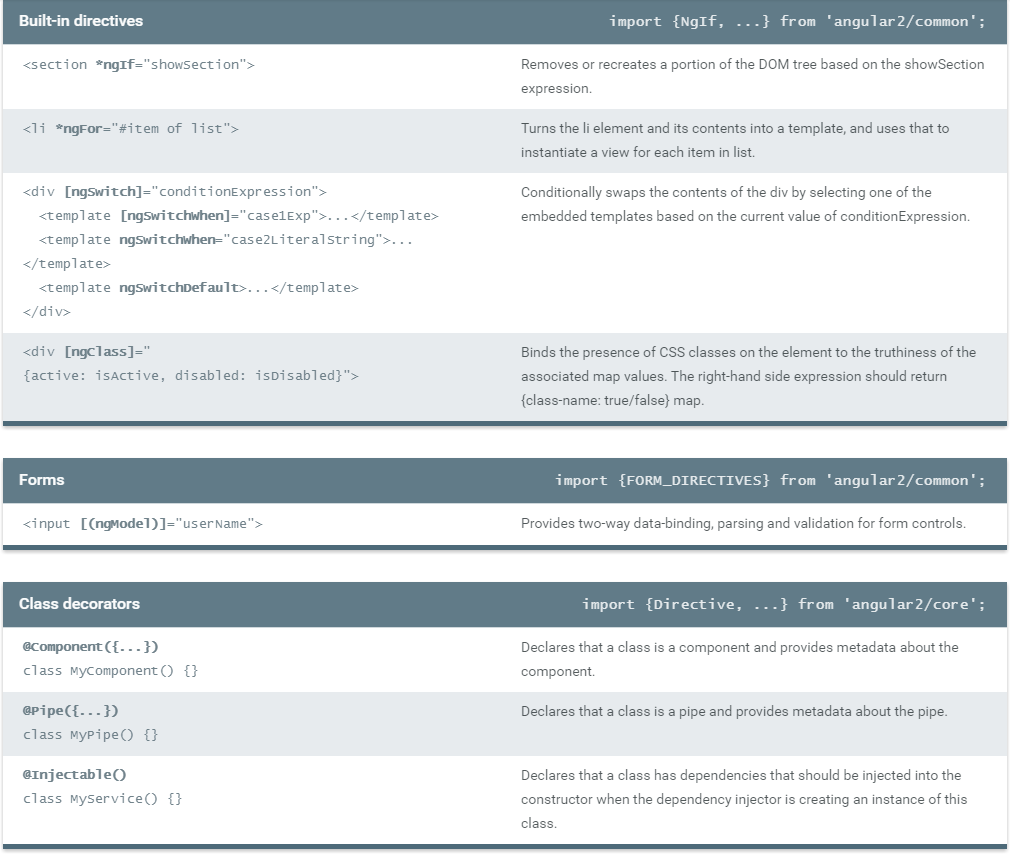
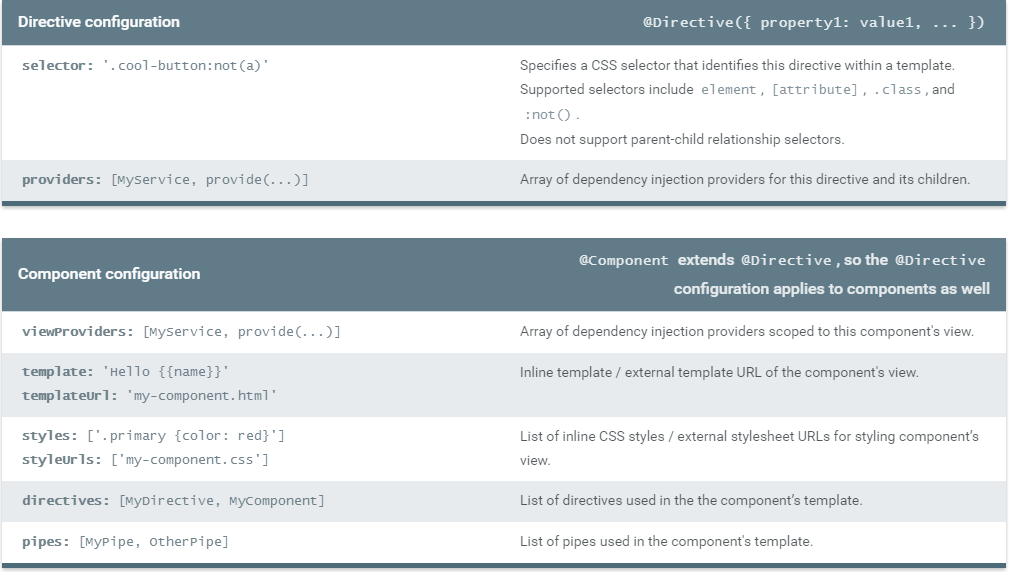
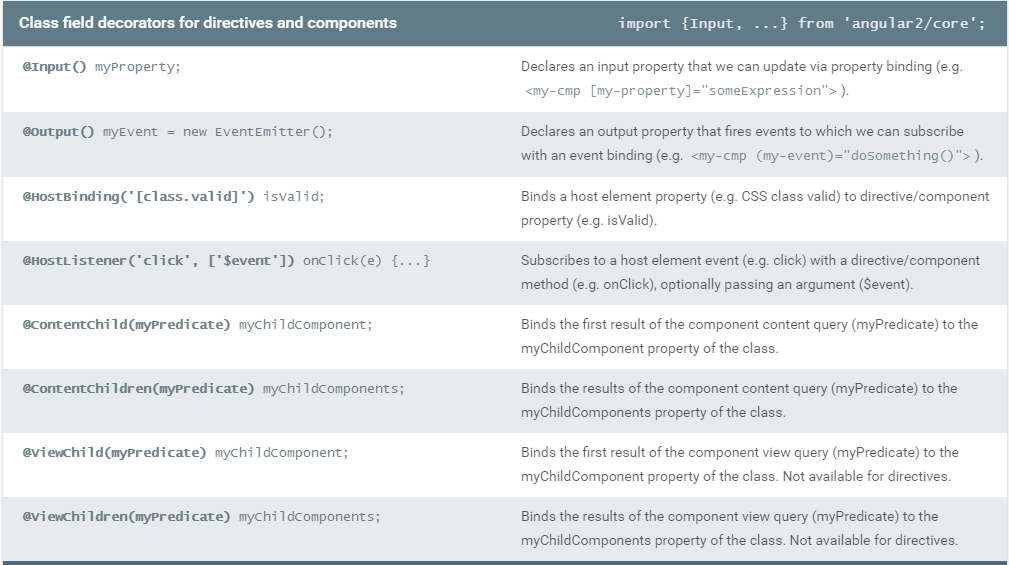
1. Angular for TypeScript Cheat Sheet



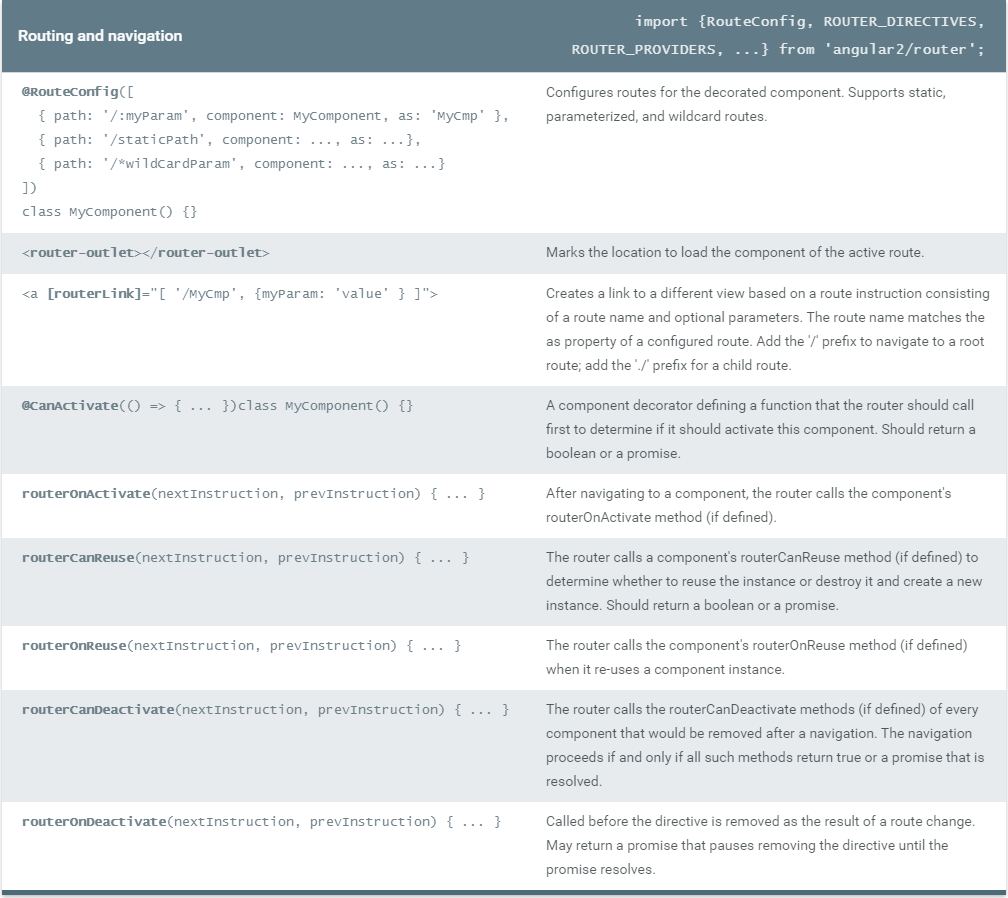












1. Architecture Overview

Angular 2 is a framework to help us build client applications in HTML and JavaScript.

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

We write applications by composing HTML templates with Angularized-markup, writing component classes to manage those templates, adding application logic in services, and handing the top root component to Angular's bootstrapper.

Angular takes over, presenting our application content in a browser and responding to user interactions according to the instructions we provided.

Of course there is more to it than this. We'll learn the details when we dive into the guide chapters. Let's get the big picture first.

## overview

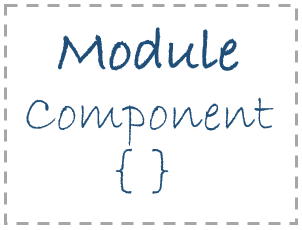
The architecture diagram identifies the eight main building blocks of an Angular 2 application:

1. [Module](https://angular.io/docs/ts/latest/guide/architecture.html#module)
2. [Component](https://angular.io/docs/ts/latest/guide/architecture.html#component)
3. [Template](https://angular.io/docs/ts/latest/guide/architecture.html#template)
4. [Metadata](https://angular.io/docs/ts/latest/guide/architecture.html#metadata)
5. [Data Binding](https://angular.io/docs/ts/latest/guide/architecture.html#data-binding)
6. [Service](https://angular.io/docs/ts/latest/guide/architecture.html#service)
7. [Directive](https://angular.io/docs/ts/latest/guide/architecture.html#directive)
8. [Dependency Injection](https://angular.io/docs/ts/latest/guide/architecture.html#dependency-injection)

Learn these eight and we're on our way.

The code referenced in this chapter is available as a [live example](https://angular.io/resources/live-examples/architecture/ts/plnkr.html).

## The Module

****Angular apps are modular.

In general we assemble our application from many **modules**.

A typical module is a cohesive block of code dedicated to a single purpose. A module **exports** something of value in that code, typically one thing such as a class. 

Modules are optional

We highly recommend modular design. TypeScript has great support for ES2015 module syntax and our chapters assume we're taking a modular approach using that syntax. That's why we list Module among the basic building blocks.

Angular itself doesn't require a modular approach nor this particular syntax. Don't use it if you don't want it. Each chapter has plenty to offer after you steer clear of the import and export statements.

Find setup and organization clues in the JavaScript track (select it from the combo box at the top of this page) which demonstrates Angular 2 development with plain old JavaScript and no module system.

Perhaps the first module we meet is a module that exports a component class. The component is one of the basic Angular blocks, we write a lot of them, and we'll talk about components in the next segment. For the moment it is enough to know that a component class is the kind of thing we'd export from a module.

Most applications have an AppComponent. By convention, we'll find it in a file named app.component.ts. Look inside such a file and we'll see an export statement like this one.

app/app.component.ts (excerpt)

export class AppComponent { }

The export statement tells TypeScript that this is a module whose AppComponent class is public and accessible to other modules of the application.

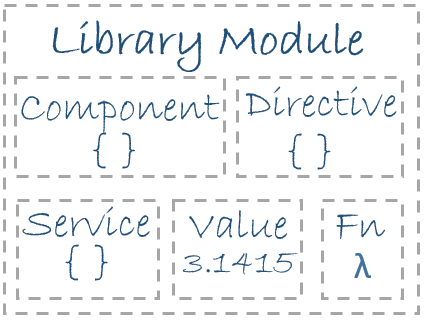
When we need a reference to the AppComponent, we **import** it like this:

app/boot.ts (excerpt)

import {AppComponent} from './app.component';

The import statement tells the system it can get an AppComponent from a module named app.component located in a neighboring file. The **module name** (AKA module id) is often the same as the filename without its extension.

### Library Modules

Some modules are libraries of other modules.

Angular itself ships as a collection of library modules called "barrels". Each Angular library is actually a public facade over several logically related private modules.

The angular2/core library is the primary Angular library module from which we get most of what we need. 

There are other important Angular library modules too such as angular2/common, angular2/router, and angular2/http.

Learn more about how Angular organizes and distributes modules in "[Modules, barrels and bundles](https://github.com/angular/angular/blob/master/modules/angular2/docs/bundles/overview.md)".

We import what we need from an Angular library module in much the same way. For example, we import the Angular **Component function** from the angular2/core module like this:

import {Component} from 'angular2/core';

Compare that syntax to our previous import of AppComponent.

import {AppComponent} from './app.component';

Notice the difference? In the first case, when importing from an Angular library module, the import statement refers to the bare module name,angular2/core, without a path prefix.

When we import from one of our own files, we prefix the module name with the file path. In this example we specify a relative file path (./). That means the source module is in the same folder (./) as the module importing it. We could path up and around the application folder structure if the source module were somewhere else.

We import and export in the ECMAScript 2015 (ES2015) module syntax. Learn more about that syntax [here](http://www.2ality.com/2014/09/es6-modules-final.html) and many other places on the web.

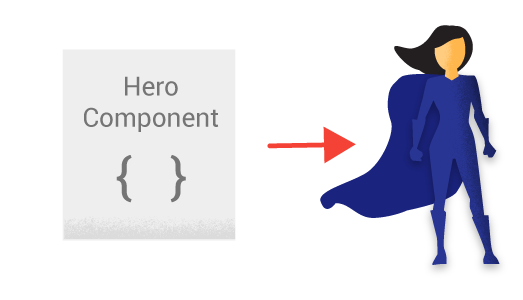
The infrastructure behind module loading and importing is an important subject. But it's a subject outside the scope of this introduction to Angular. While we're focused on our application, import and export is about all we need to know.

The key take aways are:

* Angular apps are composed of modules.
* Modules export things — classes, function, values — that other modules import.
* We prefer to write our application as a collection of modules, each module exporting one thing.

The first module we write will most likely export a component.

## The Component

****A **Component** controls a patch of screen real estate that we could call a view. The shell at the application root with navigation links, that list of heroes, the hero editor ... they're all views controlled by Components.

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

A HeroListComponent, for example, might have a heroes property that returns an array of heroes that it acquired from a service. It might have a selectHero() method that sets a selectedHero property when the user click on a hero from that list. It might be a class like this:

app/hero-list.component.ts

1. export class HeroListComponent implements OnInit {
2. constructor(private \_service: HeroService){ }
3. heroes:Hero[];
4. selectedHero: Hero;
5. ngOnInit(){
6. this.heroes = this.\_service.getHeroes();
7. }
8. selectHero(hero: Hero) { this.selectedHero = hero; }
9. }

Angular creates, updates, and destroys components as the user moves through the application. The developer can take action at each moment in this lifecycle through optional [Lifecycle Hooks](https://angular.io/docs/ts/latest/guide/lifecycle-hooks.html).

We're not showing those hooks in this example but we are making a mental note to find out about them later.

We may wonder who is calling that constructor? Who provides the service parameter? For the moment, have faith that Angular will call the constructor and deliver an appropriate HeroService when we need it.

## The Template

****We 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 much of the time ... and then it gets a bit strange. Here is a template for ourHeroList component.

app/hero-list.component.html

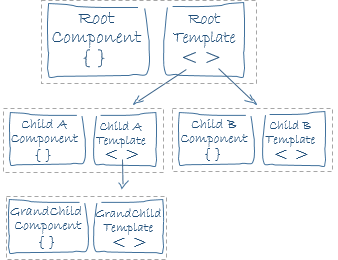
1. <h2>Hero List</h2>
2. <p><i>Pick a hero from the list</i></p>
3. <div \*ngFor="#hero of heroes" (click)="selectHero(hero)">
4. {{hero.name}}
5. </div>
6. <hero-detail \*ngIf="selectedHero" [hero]="selectedHero"></hero-detail>

We recognize <h2> and <div>. But there's other markup that no one told us about in school. What are \*ngFor, {{hero.name}},(click), [hero], and <hero-detail>?

These are examples of Angular's [template syntax](https://angular.io/docs/ts/latest/guide/template-syntax.html). We will grow accustomed to that syntax and may even learn to love it. We'll begin to explain it in a moment.

Before we do, focus attention on the last line. The <hero-detail> tag is a custom element representing the HeroDetailComponent.

The HeroDetailComponent is a different component than the HeroListComponent we'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 theHeroListComponent. The HeroDetailComponent is a **child** of the the HeroListComponent.

Notice how <hero-detail> rests comfortably among the HTML elements we already know. We can mix ... and will mix ... our custom components with native HTML in the same layouts.

And in this manner we can and will compose complex component trees to build out our richly featured application.

## Angular Metadata

**Metadata**Metadata tells Angular how to process a class.

[Looking back](https://angular.io/docs/ts/latest/guide/architecture.html#component-code) at the HeroListComponent, we see that it's just a class. There is no evidence of a framework, no "Angular" in it at all.

In fact, it really is just a class. It's not a component until we tell Angular about it.

We tell Angular that HeroListComponent is a component by attaching **metadata** to the class.

The easy way to attach metadata in TypeScript is with a **decorator**. Here's some metadata for HeroListComponent:

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

1. @Component({
2. selector: 'hero-list',
3. templateUrl: 'app/hero-list.component.html',
4. directives: [HeroDetailComponent],
5. providers: [HeroService]
6. })
7. export class HeroesComponent { ... }

Here we see the @Component decorator which (no surprise) identifies the class immediately below it as a Component class.

A decorator is a function. Decorators often have a configuration parameter. The @Component decorator takes a required configuration object with the information Angular needs to create and present the component and its view.

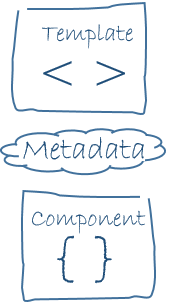
Here we see a few of the possible @Component configuration options:

* selector - a css selector that tells Angular to create and insert an instance of this component where it finds a <hero-list> tag inparent HTML. If the template of the application shell (a Component) contained

<hero-list></hero-list>

Angular inserts an instance of the HeroListComponent view between those tags.

* templateUrl - the address of this component's template which we showed [above](https://angular.io/docs/ts/latest/guide/architecture.html#template).
* directives - an array of the Components or Directives that this template requires. We saw in the last line of our template that we expect Angular to insert a HeroDetailComponent in the space indicated by <hero-detail> tags. Angular will do so only if we mention the HeroDetailComponent in this directives array.
* providers - an array of **dependency injection providers** for services that the component requires. This is one way to tell Angular that our component's constructor requires a HeroService so it can get the list of heroes to display. We'll get to dependency injection in a moment.

The @Component function takes the configuration object and turns it into metadata that it attaches to the component class definition. Angular discovers this metadata at runtime and thus knows how to do "the right thing".

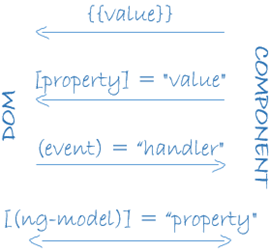
The template, metadata, and component together describe the view.

We apply other metadata decorators in a similar fashion to guide Angular behavior. The @Injectable, @Input, @Output,@RouterConfig are a few of the more popular decorators we'll master as our Angular knowledge grows.

The architectural take-away is that we must add metadata to our code so that Angular knows what to do.

## Data Binding

Without a framework, we 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 the experienced jQuery programmer can attest.



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

There are four forms of data binding syntax. Each form has a direction - to the DOM, from the DOM, or in both directions - as indicated by the arrows in the diagram.

We saw three forms of data binding in our [example](https://angular.io/docs/ts/latest/guide/architecture.html#template) template:

app/hero-list.component (excerpt)

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

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

<div (click)="selectHero(hero)></div>

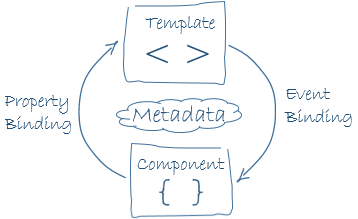
* The "[interpolation](https://angular.io/docs/ts/latest/guide/displaying-data.html#interpolation)" displays the component's hero.name property value within the <div> tags.
* The [hero] [property binding](https://angular.io/docs/ts/latest/guide/template-syntax.html#property-binding) passes the selectedHero from the parent HeroListComponent to the hero property of the childHeroDetailComponent.
* The (click) [event binding](https://angular.io/docs/ts/latest/guide/user-input.html#click) calls the Component's selectHero method when the user clicks on 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 ngModeldirective. We didn't have a two-way binding in the HeroListComponent template; here's an example from the HeroDetailComponenttemplate (not shown):

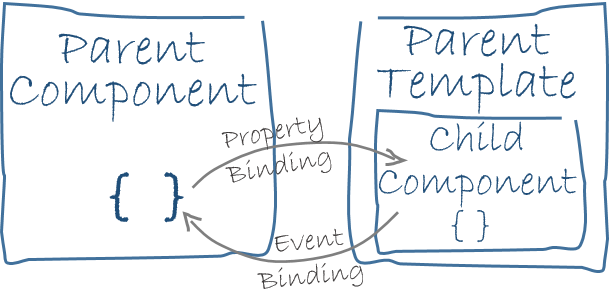
<input [(ngModel)]="hero.name">

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

Angular processes all data bindings once per JavaScript event cycle, depth-first from the root of the application component tree.



We don't know all the details yet but it's clear from these examples that data binding plays an important role in communication between a template and its component ...



... **and** between parent and child components

## The Directive



Our Angular templates are dynamic. When Angular renders them, it transforms the DOM according to the instructions given by a **directive**.

A directive is a class with directive metadata. In TypeScript we'd apply the @Directive decorator to attach metadata to the class.

We already met one form of directive: the component. A component is a directive-with-a-template and the @Component decorator is actually a@Directive decorator extended with template-oriented features.

While the **component is technically a directive**, it is so distinctive and central to Angular applications that we chose to separate the component from the directive in our architectural overview.

There are two other kinds of directives as well that we call "structural" and "attribute" directives.

They tend to appear within an element tag like attributes, 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.

We see two built-in structural directives at play in our [example](https://angular.io/docs/ts/latest/guide/architecture.html#template) template:

<div \*ngFor="#hero of heroes"></div>

<hero-detail \*ngIf="selectedHero"></hero-detail>

* [\*ngFor](https://angular.io/docs/ts/latest/guide/displaying-data.html#ngFor) tells Angular to stamp out one <div> per hero in the heroes list.
* [\*ngIf](https://angular.io/docs/ts/latest/guide/displaying-data.html#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.

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

It modifies the behavior of an existing element (typically an <input>) by setting its display value property and responding to change events.

Angular ships with a small number of other directives that either alter the layout structure (e.g. [ngSwitch](https://angular.io/docs/ts/latest/guide/template-syntax.html#ngSwitch)) or modify aspects of DOM elements and components (e.g. [ngStyle](https://angular.io/docs/ts/latest/guide/template-syntax.html#ngStyle) and [ngClass](https://angular.io/docs/ts/latest/guide/template-syntax.html#ngClass)).

And of course we can write our own directives.

## The Service



"Service" is a broad category encompassing any value, function or feature that our 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 itself has no definition of a service. There is no ServiceBase class.

Yet services are fundamental to any Angular application.

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

app/logger.service.ts (class only)

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 fetches heroes and returns them in a resolved [promise](http://www.html5rocks.com/en/tutorials/es6/promises/). The HeroService depends on the LoggerServiceand another BackendService that handles the server communication grunt work.

app/hero.service.ts (class only)

export class HeroService {

constructor(

private \_backend: BackendService,

private \_logger: Logger) { }

private \_heroes:Hero[] = [];

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.

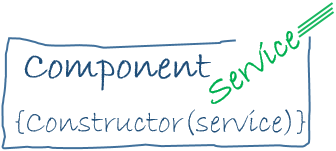
Our components are big consumers of services. They depend upon services to handle most chores. They don't fetch data from the server, they don't validate user input, they don't log directly to the console. They delegate such tasks to services.

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 non-trivial to services.

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

Angular does help us follow these principles ... by making it easy to factor our 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.

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

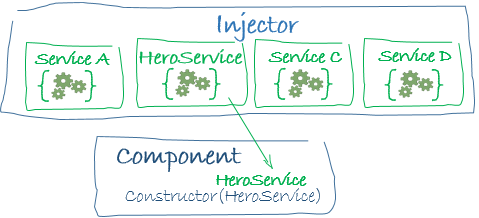
app/hero-list.component (constructor)

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 what we mean by 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, we 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.

We can register providers at any level of the application component tree. We often do so at the root when we bootstrap the application so that the same instance of a service is available everywhere.

app/boot.ts (excerpt)

bootstrap(AppComponent, [BackendService, HeroService, Logger]);

Alternatively, we might register at a component level ...

app/hero-list.component.ts (excerpt)

@Component({

providers: [HeroService]

})

export class HeroesComponent { ... }

... in which case we get a new instance of the service with each new instance of that component.

We've vastly over-simplified dependency injection for this overview. We can learn the full story in the [Dependency Injection](https://angular.io/docs/ts/latest/guide/dependency-injection.html) chapter.

The points to remember are:

* dependency injection is wired into the 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 using a provider.
* a provider is a recipe for creating a service.
* we register providers with injectors.

## Wrap up

We've learned just a bit about the eight main building blocks of an Angular application

1. [Module](https://angular.io/docs/ts/latest/guide/architecture.html#module)
2. [Component](https://angular.io/docs/ts/latest/guide/architecture.html#component)
3. [Template](https://angular.io/docs/ts/latest/guide/architecture.html#template)
4. [Metadata](https://angular.io/docs/ts/latest/guide/architecture.html#metadata)
5. [Data Binding](https://angular.io/docs/ts/latest/guide/architecture.html#data-binding)
6. [Service](https://angular.io/docs/ts/latest/guide/architecture.html#service)
7. [Directive](https://angular.io/docs/ts/latest/guide/architecture.html#directive)
8. [Dependency Injection](https://angular.io/docs/ts/latest/guide/architecture.html#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 we'll need or want to know.

## The Other Stuff

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

**Animations** - A forthcoming animation library makes it easy for developers to animate component behavior without deep knowledge of animation techniques or css.

**Bootstrap** - A method to configure and launch the root application component.

**Change Detection** - Learn how Angular decides that a component property value has changed and when to update the screen. Learn how it uses **zones** to intercept asynchronous activity and run its change detection strategies.

[**Component Router**](https://angular.io/docs/ts/latest/guide/router.html) - With the Component Router service, users can navigate a multi-screen application in a familiar web browsing style using URLs.

**Events** - The DOM raises events. So can components and services. Angular offers mechanisms for publishing and subscribing to events including an implementation of the [RxJS Observable](https://github.com/zenparsing/es-observable) proposal.

[**Forms**](https://angular.io/docs/ts/latest/guide/forms.html) - 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 this Angular HTTP client.

[**Lifecycle Hooks**](https://angular.io/docs/ts/latest/guide/lifecycle-hooks.html) - We can tap into key moments in the lifetime of a component, from its creation to its destruction, by implementing the "Lifecycle Hook" interfaces.

[**Pipes**](https://angular.io/docs/ts/latest/guide/pipes.html) - Services that transform values for display. We can put pipes in our templates to improve the user experience. For example, this currency pipe expression,

price | currency:'USD':true

displays a price of "42.33" as $42.33.

[**Testing**](https://angular.io/docs/ts/latest/testing/index.html) - Angular provides a testing library for "unit testing" our application parts as they interact with the Angular framework.

1. Displaying Data

In Angular, we display data by binding component properties to elements in HTML templates using interpolation and other forms of Property Binding.

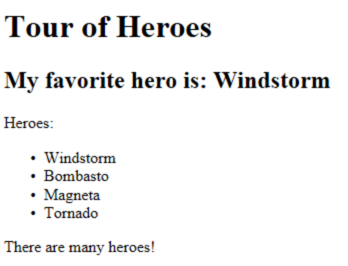
## Displaying Component Properties

We typically display data in Angular by binding controls in an HTML template to properties of an Angular Component.

In this chapter, we'll create a component with a list of heroes. Each hero has a name. We'll display the list of hero names and conditionally show a selected hero in a detail area below the list.

[Live Example](https://angular.io/resources/live-examples/displaying-data/ts/plnkr.html)

Our final UI looks like this:



## Showing component properties with interpolation

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

Let's build a small illustrative example together.

Create a new project folder (displaying-data) and follow the steps in the [QuickStart](https://angular.io/docs/ts/latest/quickstart.html).

Then modify the app.component.ts file by changing the template and the body of the component. When we're done, it should look like this:

app/app.component.ts

1. import {Component} from 'angular2/core';
2. @Component({
3. selector: 'my-app',
4. template: `
5. <h1>{{title}}</h1>
6. <h2>My favorite hero is: {{myHero}}</h2>
7. `
8. })
9. export class AppComponent {
10. title = 'Tour of Heroes';
11. myHero = 'Windstorm';
12. }

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

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

template: `

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

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

`

The template is a multi-line string within ECMAScript 2015 back-tics (`). The back-tick (`) is not the same character as a single quote ('). It has many nice features. The feature we're exploiting is the ability to compose the string over several lines which makes for much more readable HTML.

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

More precisely, the re-display occurs after some kind of asynchronous event related to the view such as a keystroke, a timer completion, or an asynchXHR response. We don't have those in this sample. But then the properties aren't changing on their own either. For the moment we must operate on faith.

Notice that we haven't called **new** to create an instance of the AppComponent class. Angular is creating an instance for us. How?

Notice the CSS selector in the @Component decorator that specifies an element named "my-app". Remember back in QuickStart that we added the <my-app> element to the body of our index.html

<body>

<my-app>loading...</my-app>

</body>

When we bootstrap with the AppComponent class (see boot.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.

We're ready to see changes in a running app by firing up the npm script that both compiles and serves our applications while watching for changes.

npm start

We should see the title and hero name:

Title and Hero

Let's review some of the choices we made and consider alternatives.

## Template inline or template file?

We can store our component's template in one of two places. We can define it "inline" using the template property as we do here. Or we can define the template in a separate HTML file and link to it in the component metadata using the @Component decorator's templateUrlproperty.

We're using the inline style because the template is small and it makes for clearer demonstration. The choice between them is a matter of taste, circumstances, and organization policy.

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

## Constructor or variable initialization?

We initialized our component properties using variable assignment. This is a wonderfully concise and compact technique.

Some folks prefer to declare the properties and initialize them within a constructor like this:

export class AppCtorComponent {

title: string;

myHero: string;

constructor() {

this.title = 'Tour of Heroes';

this.myHero = 'Windstorm';

}

}

That's fine too. The choice between them is a matter of taste and organization policy. We'll adopt the more terse "variable assignment" style in this chapter simply because there will be less code to read.

## Showing an array property with NgFor

We want to display a list of heroes. We begin by adding a mock heroes name array to the component, just above myHero and redefinemyHero to be the first name in the array.

app/app.component.ts (class)

export class AppComponent {

title = 'Tour of Heroes';

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

myHero = this.heroes[0];

}

Now we use the Angular NgFor "repeater" Directive in the template to display each item in the heroes list.

app/app.component.ts (template)

template: `

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

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

<p>Heroes:</p>

<ul>

<li \*ngFor="#hero of heroes">

{{ hero }}

</li>

</ul>

`

Our presentation is the familiar HTML unordered list with <ul> and <li> tags. Let's focus on the <li> tag.

<li \*ngFor="#hero of heroes">

{{ hero }}

</li>

We added a somewhat mysterious \*ngFor to the <li> element. That's the Angular "repeater" directive. It's presence on the <li> tag marks that <li> element (and its children) as the "repeater template".

Don't forget the leading asterisk (\*) in front of \*ngFor. It is an essential part of the syntax. Learn more about this and NgFor in the[Template Syntax](https://angular.io/docs/ts/latest/guide/template-syntax.html#ngFor) chapter.

Notice the #hero in the NgFor double-quoted instruction. The #hero is a "[template local variable](https://angular.io/docs/ts/latest/guide/template-syntax.html#local-vars")" declaration. The (#) prefix declares a local variable name named hero.

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

We happened to give NgFor an array to display. In fact, NgFor can repeat items for any [iterable](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Iteration_protocols) object.

Assuming we're still running under the npm go command, we should see heroes appearing in an unordered list.



## Creating a class for the data

We are defining our data directly inside our component. That's fine for a demo but certainly isn't a best practice. It's not even a good practice. Although we won't do anything about that in this chapter, we'll make a mental note to fix this down the road.

At the moment, we're binding to an array of strings. We do that occasionally in real applications but most of the time we're displaying objects, potentially instances of classes.

Let's turn our array of hero names into an array of Hero objects. For that we'll need a `Hero' class.

Create a new file in the app/ folder called hero.ts with the following short bit of code.

app/hero.ts

export class Hero {

constructor(

public id:number,

public name:string) { }

}

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

If we are new to TypeScript, it may not look like we have properties. But we do. We're taking advantage of a TypeScript short-cut in our declaration of the constructor parameters.

Consider the first parameter:

public id:number,

That brief syntax simultaneously

* declares a constructor parameter and its type
* declare a public property of the same name
* initializes that property with the corresponding argument when we "new" an instance of the class.

## Use the Hero class

Let's redefine the heroes property in our component to return an array of these Heroes and also set the myHero property with the first of these mock heroes.

app.component.ts (excerpt)

heroes = [

new Hero(1, 'Windstorm'),

new Hero(13, 'Bombasto'),

new Hero(15, 'Magneta'),

new Hero(20, 'Tornado')

];

myHero = this.heroes[0];

We'll have to update the template. At the moment it displays the entire hero object which used to be a string value. Let's fix that so we interpolate the hero.name property

app.component.ts (template)

template: `

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

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

<p>Heroes:</p>

<ul>

<li \*ngFor="#hero of heroes">

{{ hero.name }}

</li>

</ul>

`

Our display looks the same but we know how much better it is under the hood.

## Conditional display with NgIf

Sometimes the app should display a view or a portion of a view only under prescribed circumstances.

In our example, we'd like to display a message if we have a large number of heroes ... say more than 3.

The Angular NgIf directive will insert or remove an element based on a truthy/falsey condition. We can see it in action by adding the following paragraph at the bottom of the template:

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

Don't forget the leading asterisk (\*) in front of \*ngIf. It is an essential part of the syntax. Learn more about this and NgIf in the[Template Syntax](https://angular.io/docs/ts/latest/guide/template-syntax.html#ngIf) chapter.

The [template expression](https://angular.io/docs/ts/latest/guide/template-syntax.html#template-expressions) inside the double quotes looks much like JavaScript and it is much like JavaScript. When the component's list of heroes has more than 3 items, Angular adds the paragraph to the DOM and the message appears. If there were 3 or fewer items, Angular omits the paragraph and there is no message.

Angular isn't showing and hiding the message. It is adding and removing the paragraph element from the DOM. That hardly matters here. It would matter a great deal from a performance perspective if we were conditionally including or excluding a big chunk of HTML with many data bindings.

Try it out. We have four items in the array so 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.

Play with it.

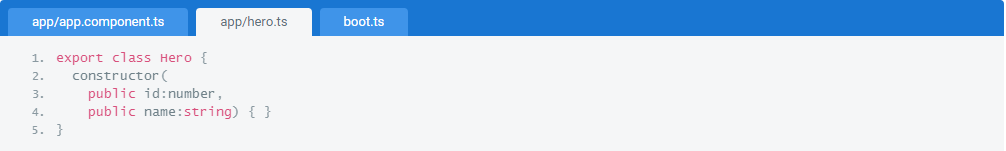
## Summary

Now we know how to

* use **interpolation** with the double curly braces to display a component property,
* use **NgFor** to display a list of items,
* use a TypeScript class to shape the model data for our component and display properties of that model,
* use **NgIf** to conditionally display a chunk of HTML based on a boolean expression.

Our final code:







## User Input

User input triggers DOM events. We listen to those events with EventBindings that funnel updated values back into our components and models.

When the user clicks a link, pushes a button, or types on the keyboard we want to know about it. These user actions all raise DOM events. In this chapter we learn to bind to those events using the Angular Event Binding syntax.

[Live Example](https://angular.io/resources/live-examples/user-input/ts/plnkr.html).

## Binding to User Input Events

We can listen to [any DOM event](https://developer.mozilla.org/en-US/docs/Web/Events) with an [Angular Event Binding](https://angular.io/docs/ts/latest/guide/template-syntax.html#event-binding).

The syntax is simple. We assign a template expression to the DOM event name, surrounded in parentheses. A click Event Binding makes for a quick illustration.

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

The (click) to the left of the equal sign identifies the button's click event as the **target of the binding**. The text within quotes on the right is the "**template expression**" in which we respond to the click event by calling the component's onClickMe method. A [template expression](https://angular.io/docs/ts/latest/guide/template-syntax.html#template-expressions) is a subset of JavaScript with a few added tricks.

When writing a binding we must be aware of a template expression's **execution context**. The identifiers appearing within an expression belong to a specific context object. That object is usually the Angular component that controls the template ... which it definitely is in this case because that snippet of HTML belongs to the following component:

app/click-me.component.ts

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

The onClickMe in the template refers to the onClickMe method of the component. When the user clicks the button, Angular calls the component's onClickMe method.

## Get user input from the $event object

We can bind to all kinds of events. Let's bind to the "keyup" event of an input box and replay what the user types back onto the screen.

This time we'll (1) listen to an event and (2) grab the user's input.

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

1. template: `
2. <input (keyup)="onKey($event)">
3. <p>{{values}}</p>
4. `

Angular makes an event object available in the **$event** variable which we pass to the component's onKey() method. The user data we want is in that variable somewhere.

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

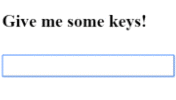
1. export class KeyUpComponent\_v1 {
2. values='';
3. // without strong typing
4. onKey(event:any) {
5. this.values += event.target.value + ' | ';
6. }
7. }

The shape of the $event object is determined by whatever raises the event. The keyup event comes from the DOM so $event must be a[standard DOM event object](https://developer.mozilla.org/en-US/docs/Web/API/Event). The $event.target gives us the [HTMLInputElement](https://developer.mozilla.org/en-US/docs/Web/API/HTMLInputElement) which has a value property and that's where we find our user input data.

We had this in mind when we passed $event to our onKey() component method. This is where we extract the user's input and concatenate it to the previous user data that we're accumulating in the component's values property. We then use [interpolation](https://angular.io/docs/ts/latest/guide/template-syntax.html#interpolation) to display the accumulating values property back on screen.

Enter the letters "abc", backspace to remove them, and we should see:

a | ab | abc | ab | a | |



We cast the $event as an any type which means we've abandoned strong typing to simplify our code. We generally prefer the strong typing that TypeScript affords. We can rewrite the method, casting to HTML DOM objects like this.

app/keyup.components.ts (class v.1 - strongly typed )

1. export class KeyUpComponent\_v1 {
2. values='';
3. // with strong typing
4. onKey(event:KeyboardEvent) {
5. this.values += (<HTMLInputElement>event.target).value + ' | ';
6. }
7. }

Strong typing reveals a serious problem with passing a DOM event into the method: too much awareness of template details, too little separation of concerns.

We'll address this problem in our next try at processing user keystrokes.

## Get user input from a local template variable

There's another way to get the user data without the $event variable.

Angular has a syntax feature called [**local template variables**](https://angular.io/docs/ts/latest/guide/template-syntax.html#local-vars). These variables grant us direct access to an element. We declare a local template variable by preceding an identifier with a hash/pound character (#).

Let's demonstrate with a clever keystroke loopback in an ultra-simple template.

app/loop-back.component.ts

1. @Component({
2. selector: 'loop-back',
3. template:`
4. <input #box (keyup)="0">
5. <p>{{box.value}}</p>
6. `
7. })
8. export class LoopbackComponent { }

We've declared a template local variable named box on the <input> element. The box variable is a reference to the <input> element itself which means we can grab 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 which does nothing.

Type in the input box and watch the display update with each keystroke. Voila!



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

Angular only updates the bindings (and therefore the screen) if we do something in response to asynchronous events such as keystrokes.

That's why we bind the keyup event to an expression that does ... well, nothing. We're binding to the number 0, the shortest expression we can think of. That is all it takes to keep Angular happy. We said it would be clever!

That local template variable is intriguing. It's clearly easier to get to the textbox with that variable than to go through the $event object. Maybe we can re-write our previous "key-up" example using the variable to acquire the user's input. Let's give it a try.

app/keyup.components.ts (v2)

1. @Component({
2. selector: 'key-up2',
3. template: `
4. <input #box (keyup)="onKey(box.value)">
5. <p>{{values}}</p>
6. `
7. })
8. export class KeyUpComponent\_v2 {
9. values='';
10. onKey(value:string) {
11. this.values += value + ' | ';
12. }
13. }

That sure seems easier. An especially nice aspect of this approach is that our component code gets clean data values from the view. It no longer requires knowledge of the $event and its structure.

## Key event filtering (with key.enter)

Perhaps we don't care about every keystroke. We're only interested in the input box value when the user hits the "Enter" key. We'd like to ignore all other keys. When we bind to the (keyup) event, our event handling expression hears every key stroke. We could filter the keys first, examining every $event.keyCode, and update the values property only if the key is "Enter".

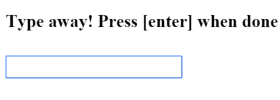
Angular can filter the key events for us. Angular has a special syntax for keyboard events. We can listen for just the "Enter" key by binding to Angular's keyup.enter pseudo-event.

Only then do we update the component's values property ... inside the event expression rather than in the component ... because we can ... even if it is a dubious practice.

app/keyup.components.ts (v3)

1. @Component({
2. selector: 'key-up3',
3. template: `
4. <input #box (keyup.enter)="values=box.value">
5. <p>{{values}}</p>
6. `
7. })
8. export class KeyUpComponent\_v3 {
9. values='';
10. }

Here's how it works.



## On blur

Our previous example won't transfer the current state of the input box if the user mouses away and clicks elsewhere on the page. We only update the component's values property when the user presses "Enter" inside the input box.

Let's fix that by listening to the input box's blur event as well.

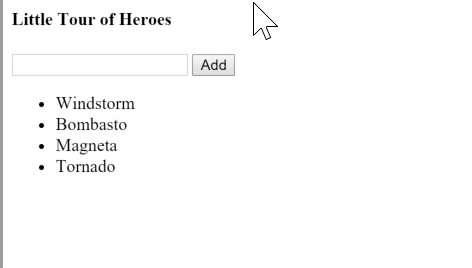
app/keyup.components.ts (v4)

1. @Component({
2. selector: 'key-up4',
3. template: `
4. <input #box
5. (keyup.enter)="values=box.value"
6. (blur)="values=box.value">
7. <p>{{values}}</p>
8. `
9. })
10. export class KeyUpComponent\_v4 {
11. values='';
12. }

## Put it all together

We learned how to [display data](https://angular.io/docs/ts/latest/guide/displaying-data.html) in the previous chapter. We've acquired a small arsenal of event binding techniques in this chapter.

Let's put it all together in a micro-app that can display a list of heroes and add new heroes to that list by typing in the input box and hitting "Enter", clicking "Add", or clicking elsewhere on the page.



Below is the "Little Tour of Heroes" component. We'll call out the highlights after we bask briefly in its minimalist glory.

app/little-tour.component.ts

1. @Component({
2. selector: 'little-tour',
3. template: `
4. <input #newHero
5. (keyup.enter)="addHero(newHero.value)"
6. (blur)="addHero(newHero.value); newHero.value='' ">
7. <button (click)=addHero(newHero.value)>Add</button>
8. <ul><li \*ngFor="#hero of heroes">{{hero}}</li></ul>
9. `
10. })
11. export class LittleTourComponent {
12. heroes=['Windstorm', 'Bombasto', 'Magneta', 'Tornado'];
13. addHero(newHero:string) {
14. if (newHero) {
15. this.heroes.push(newHero);
16. }
17. }
18. }

We've seen almost everything here before. A few things are new or bear repeating.

### newHero template variable

The newHero template variable refers to the <input> element.

We can access newHero from any sibling or child of the <input> element. When the user clicks the button, we don't need a fancy CSS selector to track down the textbox and extract its value.

### Extract the input box value

We could have passed the newHero into the component's addHero() method.

But that would require addHero to pick its way through the <input> DOM element, something we learned to dislike in our first try at a[KeyupComponent](https://angular.io/docs/ts/latest/guide/user-input.html#keyup1).

Instead, we grab the input box value and pass that to addHero(). The component knows nothing about HTML or DOM which is the way we like it.

### Don't let template expressions be complex

We bound (blur) to two JavaScript statements.

We like the first one that calls addHero. We do not like the second one that assigns an empty string to the input box value.

We did it for a good reason. We have to clear the input box after adding the new hero to the list. The component has no way to do that itself — because it has no access to the input box (our design choice).

Although it works, we are rightly wary of JavaScript in HTML. Template expressions are powerful. We're supposed to use them responsibly. Complex JavaScript in HTML is irresponsible.

Should we reconsider our reluctance to pass the input box into the component?

There should be a better third way. And there is as we'll see when we learn about NgModel in the [Forms](https://angular.io/docs/ts/latest/guide/forms.html) chapter.

1. FORMS

## A form creates a cohesive, effective, and compelling data entry experience. An Angular form coordinates a set of data-bound user controls, tracks changes, validates input, and presents errors.

We’ve all used a form to login, submit a help request, place an order, book a flight, schedule a meeting and perform countless other data entry tasks. Forms are the mainstay of business applications.

Any seasoned web developer can slap together an HTML form with all the right tags. It's more challenging to create a cohesive data entry experience that guides the user efficiently and effectively through the workflow behind the form.

That takes design skills that are, to be frank, well out of scope for this chapter.

It also takes framework support for **two-way data binding, change tracking, validation, and error handling** ... which we shall cover in this chapter on Angular forms.

We will build a simple form from scratch, one step at a time. Along the way we'll learn

* How to build an Angular form with a component and template
* The ngModel two-way data binding syntax for reading and writing values to input controls
* The ngControl directive to track the change state and validity of form controls
* The special CSS classes that ngControl adds to form controls and how we can use them to provide strong visual feedback
* How to display validation errors to users and enable/disable form controls
* How to share information across controls with template local variables

[Live Example](https://angular.io/resources/live-examples/forms/ts/plnkr.html)

## Template-Driven Forms

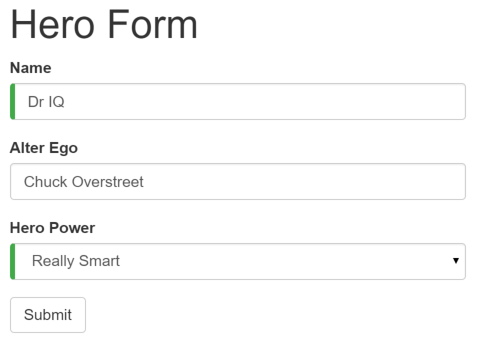
Many of us will build forms by writing templates in the Angular [template syntax](https://angular.io/docs/ts/latest/guide/template-syntax.html) with the form-specific directives and techniques described in this chapter.

That's not the only way to create a form but it's the way we'll cover in this chapter.

We can build almost any form we need with an Angular template — login forms, contact forms ... pretty much any business forms. We 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.

It will be pretty easy because Angular handles many of the repetitive, boiler plate tasks we'd otherwise wrestle with ourselves.

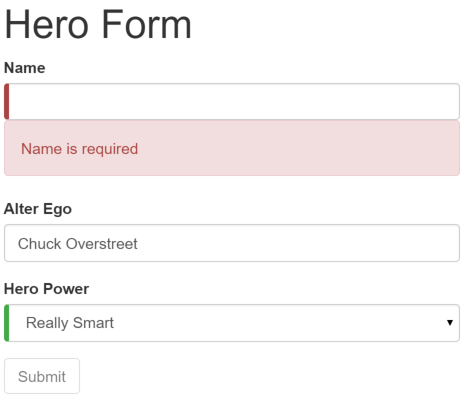
We'll discuss and learn to build the following template-driven form:



Here at the Hero Employment Agency we use this form to maintain personal information about the heroes in our stable. Every hero needs a job. It's our 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 we delete the hero name, the form displays a validation error in an attention grabbing style:



Note that the submit button is disabled and the "required" bar to the left of the input control changed from green to red.

We'll' customize the colors and location of the "required" bar with standard CSS.

We will build this form in the following sequence of 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. Add the **ngModel** directive to each form input control
5. Add the **ngControl** directive to each form input control
6. Add custom CSS to provide visual feedback
7. Show and hide validation error messages
8. Handle form submission with **ngSubmit**
9. Disable the form’s submit button until the form is valid

## Setup

Create a new project folder (angular2-forms) and follow the steps in the [QuickStart](https://angular.io/docs/ts/latest/quickstart.html).

## Create the Hero Model Class

As users enter form data, we capture their changes and update an instance of a model. We can't layout the form until we 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 our Hero class with its three required fields (id, name, power) and one optional field (alterEgo).

Create a new file in the app folder called hero.ts and give it the following class definition:

app/hero.ts

1. export class Hero {
2. constructor(
3. public id: number,
4. public name: string,
5. public power: string,
6. public alterEgo?: string
7. ) { }
8. }

It's an anemic model with few requirements and no behavior. Perfect for our demo.

The TypeScript compiler generates a public field for each public constructor parameter and assigns the parameter’s value to that field automatically when we create new heroes.

The alterEgo is optional and the constructor lets us omit it; note the (?) in alterEgo?.

We can create a new hero like this:

let myHero = new Hero(42, 'SkyDog',

'Fetch any object at any distance', 'Leslie Rollover');

console.log('My hero is called ' + myHero.name); // "My hero is called SkyDog"

## Create a Form component

An Angular form has two parts: an HTML-based template and a code-based Component to handle data and user interactions.

We begin with the Component because it states, in brief, what the Hero editor can do.

Create a new file called hero-form.component.ts and give it the following definition:

app/hero-form.component.ts

1. import {Component} from 'angular2/core';
2. import {NgForm} from 'angular2/common';
3. import { Hero } from './hero';
4. @Component({
5. selector: 'hero-form',
6. templateUrl: 'app/hero-form.component.html'
7. })
8. export class HeroFormComponent {
9. powers = ['Really Smart', 'Super Flexible',
10. 'Super Hot', 'Weather Changer'];
11. model = new Hero(18, 'Dr IQ', this.powers[0], 'Chuck Overstreet');
12. submitted = false;
13. onSubmit() { this.submitted = true; }
14. // TODO: Remove this when we're done
15. get diagnostic() { return JSON.stringify(this.model); }
16. }

There’s nothing special about this component, nothing form-specific, nothing to distinguish it from any component we've written before.

Understanding this component requires only the Angular 2 concepts we’ve learned in previous chapters

1. We import the Component decorator from the Angular library as we usually do.
2. The @Component selector value of "hero-form" means we can drop this form in a parent template with a <hero-form> tag.
3. The templateUrl property points to a separate file for template HTML called hero-form.component.html.
4. We defined dummy data for model and powers as befits a demo. Down the road, we can inject a data service to get and save real data or perhaps expose these properties as [inputs and outputs](https://angular.io/docs/ts/latest/guide/template-syntax.html#inputs-outputs) for binding to a parent component. None of this concerns us now and these future changes won't affect our form.
5. We threw in a diagnostic property at the end to return a JSON representation of our model. It'll help us see what we're doing during our development; we've left ourselves a cleanup note to discard it later.

Why don't we write the template inline in the component file as we often do elsewhere in the Developer Guide?

There is no “right” answer for all occasions. We like inline templates when they are short. Most form templates won't be short. TypeScript and JavaScript files generally aren't the best place to write (or read) large stretches of HTML and few editors are much help with files that have a mix of HTML and code. We also like short files with a clear and obvious purpose like this one.

We made a good choice to put the HTML template elsewhere. We'll write that template in a moment. Before we do, we'll take a step back and revise the app.component.ts to make use of our new HeroFormComponent.

## Revise the app.component.ts

app.component.ts is the application's root component. It will host our new HeroFormComponent.

Replace the contents of the "QuickStart" version with the following:

app/app.component.ts

1. import {Component} from 'angular2/core';
2. import {HeroFormComponent} from './hero-form.component'
3. @Component({
4. selector: 'my-app',
5. template: '<hero-form></hero-form>',
6. directives: [HeroFormComponent]
7. })
8. export class AppComponent { }

There are only three changes:

1. We import the new HeroFormComponent.
2. The template is simply the new element tag identified by the component's selector property.
3. The directives array tells Angular that our template depends upon the HeroFormComponent which is itself a Directive (as are all Components).

## Create an initial HTML Form Template

Create a new template file called hero-form.component.html and give it the following definition:

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" required>
7. </div>
8. <div class="form-group">
9. <label for="alterEgo">Alter Ego</label>
10. <input type="text" class="form-control">
11. </div>
12. <button type="submit" class="btn btn-default">Submit</button>
13. </form>
14. </div>

That is plain old HTML 5. We'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.

We've got a Submit button at the bottom with some classes on it.

**We are not using Angular yet**. There are no bindings. No extra directives. Just layout.

The container,form-group, form-control, and btn classes come from [Twitter Boostrap](http://getbootstrap.com/css/). Purely cosmetic. We're using Bootstrap to gussy up our form. Hey, what's a form without a little style!

ANGULAR FORMS DO NOT 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.

Let's add the stylesheet.

1. Open a terminal window in the application root folder and enter the command:

npm install bootstrap --save

1. Open index.html and add the following link to the <head>.

<link rel="stylesheet" href="node\_modules/bootstrap/dist/css/bootstrap.min.css">

## Add Powers with ****\*ngFor****

Our hero may choose one super power from a fixed list of Agency-approved powers. We maintain that list internally (inHeroFormComponent).

We'll add a select to our form and bind the options to the powers list using NgFor, a technique we might have seen before in the[Displaying Data](https://angular.io/docs/ts/latest/guide/displaying-data.html) chapter.

Add the following HTML immediately below the Alter Ego group.

app/hero-form.component.html (excerpt)

<div class="form-group">

<label for="power">Hero Power</label>

<select class="form-control" required>

<option \*ngFor="#p of powers" [value]="p">{{p}}</option>

</select>

</div>

We are repeating the <options> tag for each power in the list of Powers. The #p local template variable is a different power in each iteration; we display its name using the interpolation syntax with the double-curly-braces.

## Two-way data binding with ****\*ngModel****

Running the app right now would be disappointing.



We don't see hero data because we are not binding to the Hero yet. We know how to do that from earlier chapters. [Displaying Data](https://angular.io/docs/ts/latest/guide/displaying-data.html) taught us Property Binding. [User Input](https://angular.io/docs/ts/latest/guide/user-input.html) showed us how to listen for DOM events with an Event Binding and how to update a component property with the displayed value.

Now we need to display, listen, and extract at the same time.

We could use those techniques again in our form. Instead we'll introduce something new, the NgModel directive, that makes binding our form to the model super-easy.

Find the <input> tag for the "Name" and update it like this

app/hero-form.component.html (excerpt)

<input type="text" class="form-control" required

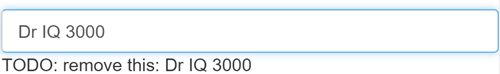
[(ngModel)]="model.name" >

TODO: remove this: {{model.name}}

We appended a diagnostic interpolation after the input tag so we can see what we're doing. We left ourselves a note to throw it way when we're done.

Focus on the binding syntax: [(ngModel)]="...".

If we ran the app right now and started typing in the Name input box, adding and deleting characters, we'd see them appearing and disappearing from the interpolated text. At some point it might look like this.



The diagnostic is evidence that we really are flowing values from the input box to the model and back again. **That's two-way data binding!**

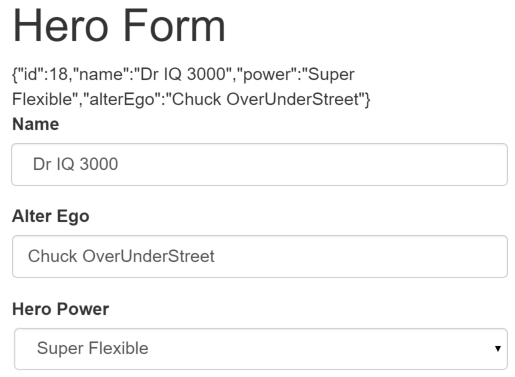
Let's add similar [(ngModel)] bindings to Alter Ego and Hero Power. We'll ditch the input box binding message and add a new binding at the top to the component's diagnostic property. Then we can confirm that two-way data binding works for the entire Hero model.

After revision the core of our form should have three [(ngModel)] bindings that look much like this:

app/hero-form.component.html (excerpt)

1. {{diagnostic}}
2. <div class="form-group">
3. <label for="name">Name</label>
4. <input type="text" class="form-control" required
5. [(ngModel)]="model.name" >
6. </div>
7. <div class="form-group">
8. <label for="alterEgo">Alter Ego</label>
9. <input type="text" class="form-control"
10. [(ngModel)]="model.alterEgo">
11. </div>
12. <div class="form-group">
13. <label for="power">Hero Power</label>
14. <select class="form-control" required
15. [(ngModel)]="model.power" >
16. <option \*ngFor="#p of powers" [value]="p">{{p}}</option>
17. </select>
18. </div>

If we ran the app right now and changed every Hero model property, the form might display like this:



The diagnostic near the top of the form confirms that all of our changes are reflected in the model.

**Delete** the {{diagnostic}} binding at the top as it has served its purpose.

### Inside [(ngModel)]

This section is an optional deep dive into [(ngModel)]. Not interested? Skip ahead!

The punctuation in the binding syntax, **[()]**, is a good clue to what's going on.

In a Property Binding, a value flows from the model to a target property on screen. We identify that target property by surrounding its name in brackets,**[]**. This is a one-way data binding **from the model to the view**.

In an Event Binding, we flow the value from the target property on screen to the model. We identify that target property by surrounding its name in parentheses, **()**. This is a one-way data binding in the opposite direction **from the view to the model**.

No wonder Angular chose to combine the punctuation as **[()]** to signify a two-way data binding and a **flow of data in both directions**.

In fact, we can break the NgModel binding into its two separate modes as we do in this re-write of the "Name" <input> binding:

app/hero-form.component.html (excerpt)

<input type="text" class="form-control" required

[ngModel]="model.name"

(ngModelChange)="model.name = $event" >

TODO: remove this: {{model.name}}

The Property Binding should feel familiar. The Event Binding might seem strange.

The ngModelChange is not an <input> element event. It is actually an event property of the NgModel directive. When Angular sees a binding target in the form [(abc)], it expects the abc directive to have an abc input property and an abc-change output property.

The other oddity is the template expression, model.name = $event. We're used to seeing an $event object coming from a DOM event. ThengModelChange property doesn't produce a DOM event; it's an Angular EventEmitter property that returns the input box value when it fires — which is precisely what we should assign to the model's `name' property.

Nice to know but is it practical? We almost always prefer [(ngModel)]. We might split the binding if we had to do something special in the event handling such as debounce or throttle the key strokes.

Learn more about NgModel and other template syntax in the [Template Syntax](https://angular.io/docs/ts/latest/guide/template-syntax.html) chapter.

## Track change-state and validity with ****ngControl****

A form isn't just about data binding. We'd also like to know the state of the controls on our form. The NgControl directive keeps track of control state for us.

NGCONTROL REQUIRES FORM

The NgControl is one of a family of NgForm directives that can only be applied to a control within a <form> tag.

Our application can ask an NgControl if the user touched the control, if the value changed, or if the value became invalid.

NgControl doesn't just track state; it updates the control with special Angular CSS classes from the set we listed above. We can leverage those class names to change the appearance of the control and make messages appear or disappear.

We'll explore those effects soon. Right now we should **add ngControlto all three form controls**, starting with the Name input box

app/hero-form.component.html (excerpt)

<input type="text" class="form-control" required

[(ngModel)]="model.name"

ngControl="name" >

Be sure to assign a unique name to each ngControl directive.

Angular registers controls under their ngControl names with the NgForm. We didn't add the NgForm directive explicitly but it's here and we'll talk about it [later in this chapter](https://angular.io/docs/ts/latest/guide/forms.html#ngForm).

## Add Custom CSS for Visual Feedback

NgControl doesn't just track state. It updates the control with three classes that reflect the state.

|  |  |  |
| --- | --- | --- |
| State | Class if true | Class if false |
| Control has been visited | ng-touched | ng-untouched |
| Control's value has changed | ng-dirty | ng-pristine |
| Control's value is valid | ng-valid | ng-invalid |

Let's add a temporary [local template variable](https://angular.io/docs/ts/latest/guide/template-syntax.html#local-vars) named **spy** to the "Name" <input> tag and use the spy to display those classes.

app/hero-form.component.html (excerpt)

<input type="text" class="form-control" required

[(ngModel)]="model.name"

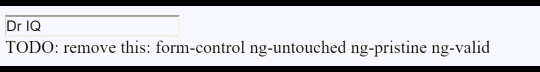
ngControl="name" #spy >

<br>TODO: remove this: {{spy.className}}

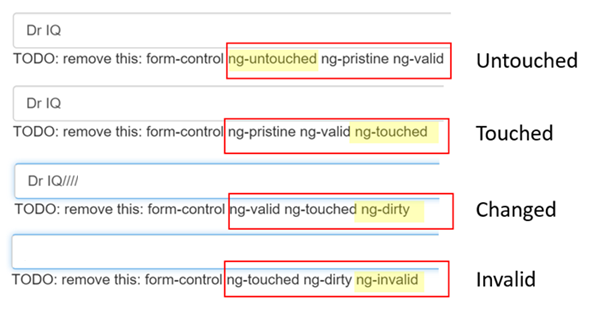
Now run the app and focus on the Name input box. Follow the next four steps precisely

1. Look but don't touched
2. Click in the input box, then click outside the text input box
3. Add slashes to the end of the name
4. Erase the name

The actions and effects are as follows:

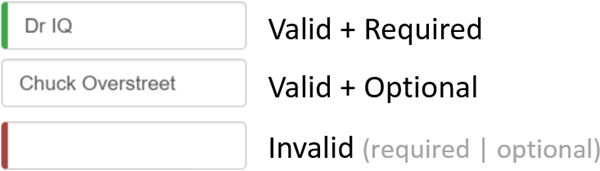


We should be able to see the following four sets of class names and their transitions:



The (ng-valid | ng-invalid) pair are most interesting to us. We want to send a strong visual signal when the data are invalid and we want to mark required fields.

We realize we can do both at the same time with a colored bar on the left of the input box:



We achieve this effect by adding two styles to a new styles.css file that we add to our project as a sibling to index.html.

styles.css

.ng-valid[required] {

border-left: 5px solid #42A948; /\* green \*/

}

.ng-invalid {

border-left: 5px solid #a94442; /\* red \*/

}

These styles select for the two Angular validity classes and the HTML 5 "required" attribute.

We update the <head> of the index.html to include this style sheet.

index.html (excerpt)

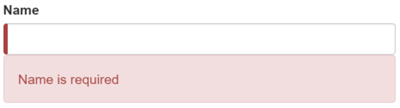
<link rel="stylesheet" href="styles.css">

## Show and Hide Validation Error messages

We can do better.

The "Name" input box is required. Clearing it turns the bar red. That says something is wrong but we don't know what is wrong or what to do about it. We can leverage the ng-invalid class to reveal a helpful message.

Here's the way it should look when the user deletes the name:



To achieve this effect we extend the <input> tag with

1. a [local template variable](https://angular.io/docs/ts/latest/guide/template-syntax.html#local-vars)
2. the "is required" message in a nearby <div> which we'll display only if the control is invalid.

Here's how we do it for the name input box:

app/hero-form.component.html (excerpt)

1. <input type="text" class="form-control" required
2. [(ngModel)]="model.name"
3. ngControl="name" #name="ngForm" >
4. <div [hidden]="name.valid" class="alert alert-danger">
5. Name is required
6. </div>

When we added the ngControl directive, we bound it to the the model's name property.

Here we initialize a template local variable (name) with the value "ngForm" (#name="ngForm"). Angular recognizes that syntax and re-sets the name local template variable to the ngControl directive instance. In other words, the name local template variable becomes a handle on the ngControl object for this input box.

Now we can control visibility of the "name" error message by binding the message <div> element's hidden property to the ngControlobject's valid property. The message is hidden while the control is valid; the message is revealed when the control becomes invalid.

### The NgForm directive

We just set a template local variable with the value of an NgForm directive. Why did that work? We didn't add the [**NgForm**](https://angular.io/docs/ts/latest/api/core/NgForm-class.html)**directive** explicitly.

Angular added it surreptiously, wrapping it around the <form> element

The NgForm directive supplements the form element with additional features. It collects Controls (elements identified by an ngControl directive) and monitors their properties including their validity. It also has its own valid property which is true only if every contained control is valid.

The Hero Alter Ego is optional so we can leave that be.

Hero Power selection is required. We can add the same kind of error handling to the <select> if we want but it's not imperative because the selection box already constrains the power to valid value.

## 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, we'll update the <form> tag with another Angular directive, NgSubmit, and bind it to the HeroFormComponent.submit() method with an event binding

<form (ngSubmit)="onSubmit()" #heroForm="ngForm">

We slipped in something extra there at the end! We defined a template local variable, **#heroForm**, and initialized it with the value, "ngForm".

The variable heroForm is now a handle to the NgForm directive that we [discussed earlier](https://angular.io/docs/ts/latest/guide/forms.html#ngForm) This time heroForm remains a reference to the form as a whole.

Later in the template we bind the button's disabled property to the form's over-all validity via the heroForm variable. Here's that bit of markup:

1. <button type="submit" class="btn btn-default"
2. [disabled]="!heroForm.form.valid">Submit</button>

Re-run the application. The form opens in a valid state and the button is enabled.

Now delete the Name. We violate the "name required" rule which is duely noted in our error message as before. And now the Submit button is also disabled.

Not impressed? Think about it for a moment. What would we have to do to wire the button's enable/disabled state to the form's validity without Angular's help?

For us, it was as simple as

1. Define a template local variable on the (enhanced) form element
2. Reference that variable in a button some 50 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 us anything new about forms. But this is an opportunity to exercise some of our newly won binding skills. If you're not interested, you can skip to the chapter's conclusion and not miss a thing.

Let's do something more strikingly visual. Let's hide the data entry area and display something else.

Start by wrapping the form in a <div> and bind its hidden property to the HeroFormComponent.submitted property.

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 the submitted property is false until we submit the form, as this fragment from theHeroFormComponent reminds us:

submitted = false;

onSubmit() { this.submitted = true; }

When we click the Submit button, the submitted flag becomes true and the form disappears as planned.

Now we need to show something else while the form is in the submitted state. Add the following block of HTML below the <div> wrapper we just wrote:

app/hero-form.component.html (excerpt)

1. <div [hidden]="!submitted">
2. <h2>You submitted the following:</h2>
3. <div class="row">
4. <div class="col-xs-3">Name</div>
5. <div class="col-xs-9 pull-left">{{ model.name }}</div>
6. </div>
7. <div class="row">
8. <div class="col-xs-3">Alter Ego</div>
9. <div class="col-xs-9 pull-left">{{ model.alterEgo }}</div>
10. </div>
11. <div class="row">
12. <div class="col-xs-3">Power</div>
13. <div class="col-xs-9 pull-left">{{ model.power }}</div>
14. </div>
15. <br>
16. <button class="btn btn-default" (click)="submitted=false">Edit</button>
17. </div>

There's our hero again, displayed read-only with interpolation bindings. This slug of HTML only appears while the component is in the submitted state.

We added an Edit button whose click event is bound to an expression that clears the submitted flag.

When we click it, this block disappears and the editable form reappears.

That's as much drama as we can muster for now.

## Conclusion

The Angular 2 form discussed in this chapter 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.
* The ngSubmit directive for handling the form submission.
* Template local variables such as #heroForm, #name, #alter-ego and #power.
* The ngModel directive for two-way data binding.
* The ngControl for validation and form element change tracking.
* The local 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.

Our final project folder structure should look like this:

angular2-forms

├── node\_modules

├── app

| ├── app.component.ts

| ├── boot.ts

| ├── hero.ts

| ├── hero-form.component.html

| └── hero-form.component.ts

├── index.html

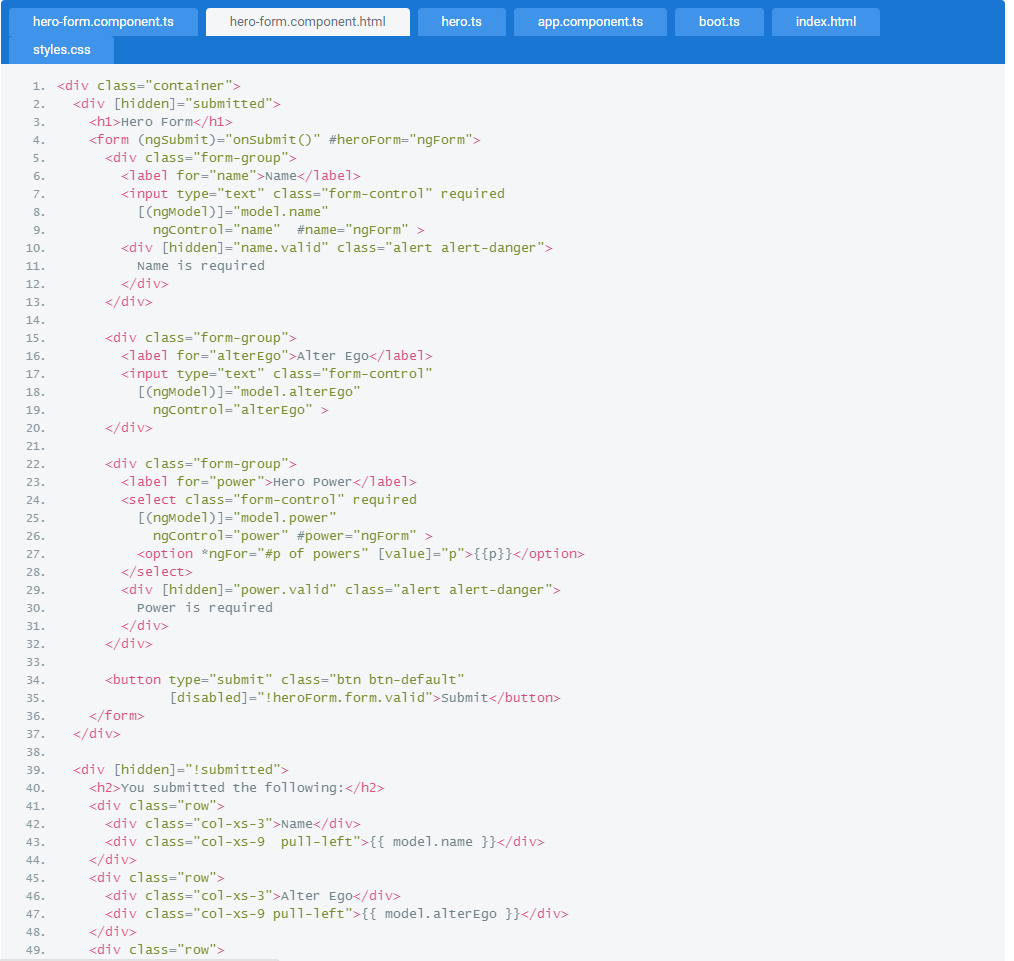
├── styles.css

├── tsconfig.json

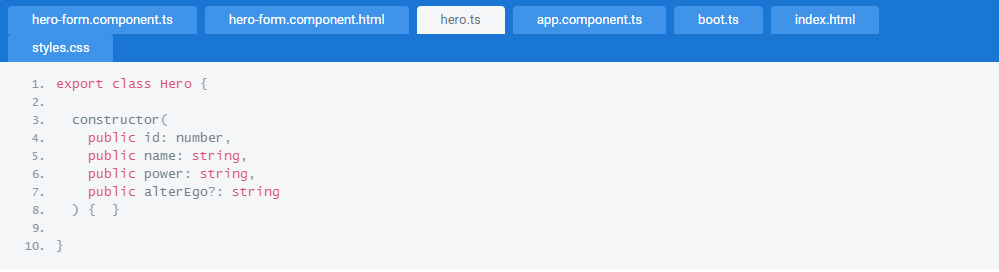
└── package.json

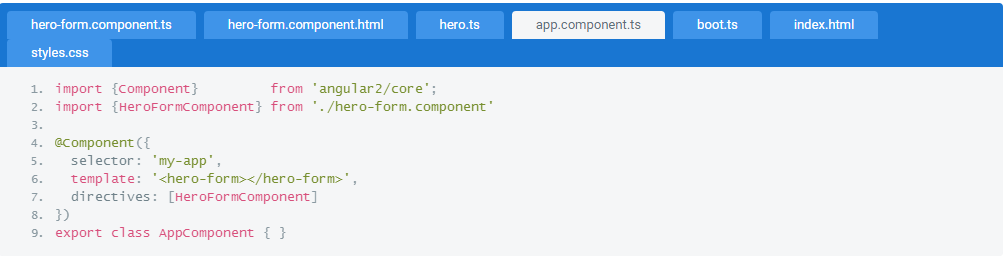
Here’s the final version of the source:

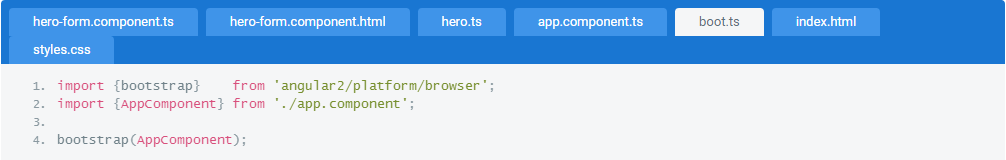


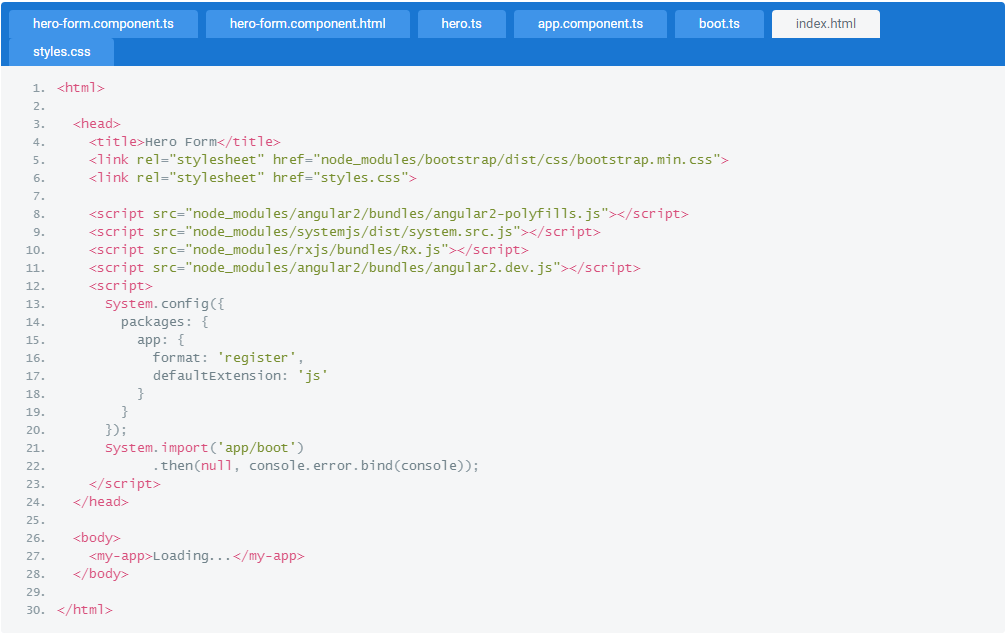


## 









1. DEPENDENCY INJECTION

Angular's dependency injection system creates and delivers dependent services "just-in-time".

Dependency Injection is an important application design pattern. Angular has its own Dependency Injection framework and we really can't build an Angular application without it.

In this chapter we'll learn what Dependency Injection is, why we want it, and how to use it.

## Why Dependency Injection?

Let's start with the following code.

1. class Engine {}
2. class Tires {}
3. class Car {
4. private engine: Engine;
5. private tires: Tires;
6. constructor() {
7. this.engine = new Engine();
8. this.tires = new Tires();
9. }
10. // Method using the engine and tires
11. drive() {}
12. }

Our Car creates everything it needs inside its constructor. What's the problem?

The problem is that our Car class is brittle, inflexible, and hard to test.

Our Car needs an engine and tires. Instead of asking for them, the Car constructor creates its own copies by "new-ing" them from the very specific classes, Engine and Tires.

What if the Engine class evolves and its constructor requires a parameter? Our Car is broken and stays broken until we rewrite it along the lines of this.engine = new Engine(theNewParameter). We didn't care about Engine constructor parameters when we first wroteCar. We don't really care about them now. But we'll have to start caring because when the definion of Engine changes, our Car class must change. That makes Car brittle.

What if we want to put a different brand of tires on our Car. Too bad. We're locked into whatever brand the Tires class creates. That makes our Car 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, we can think of other dependencies that should be shared ... like the onboard wireless connection to the manufacturer's service center. Our Carlacks the flexibility to share services that have been created previously for other consumers.

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

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

We have no control over the car's hidden dependencies. When we can't control the dependencies, a class become difficult to test.

How can we make Car more robust, more flexible, and more testable?

That's super easy. We probably already know what to do. We change our Car constructor to this:

1. constructor(engine: Engine, tires: Tires) {
2. this.engine = engine;
3. this.tires = tires;
4. }

See what happened? We moved the definition of the dependencies to the constructor. Our Car class no longer creates an engine or tires. It just consumes them.

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

1. var 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 itself. We can pass in any kind of engine or tires we like, as long as they conform to the general API requirements of an engine or tires.

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:

1. var car = new Car(new Engine(theNewParameter), new Tires());

The critical point is this: Car itself did not have to change. We'll take care of the consumer's problem soon enough.

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

1. var car = new Car(new MockEngine(), new MockLowPressureTires());

**We 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. TheCar class shed its problems at the consumer's expense. We need something that takes care of assembling these parts for us.

We could write a giant class to do that:

1. class SuperFactory {
2. createEngine = () => new Engine();
3. createTires = () => new Tires();
4. createCar = () => new Car(this.createEngine(), this.createTires());
5. }

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

Wouldn't it be nice if we could simply list the things we 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. We register some classes with this Injector and it figures out how to create them.

When we need a Car, we simply ask the Injector to get it for us and we're good to go.

1. function main() {
2. var injector = new Injector([Car, Engine, Tires, Logger]);
3. var car = injector.get(Car);
4. car.drive();
5. }

Everyone wins. The Car knows nothing about creating an Engine or Tires. The consumer knows nothing about creating a Car. We 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 InjectionFramework** is all about.

Now that we know what Dependency Injection is and appreciate its benefits, let's 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.

That sounds nice. What does it do for us when building components in Angular? Let's see, one step at a time.

We'll begin with a simplified version of the HeroesComponent that we built in the [The Tour of Heroes](https://angular.io/docs/ts/latest/tutorial/).

1. import {Component} from 'angular2/angular2';
2. import {Hero} from './hero';
3. import {HEROES} from './mock-heroes';
4. @Component({
5. selector: 'my-heroes'
6. templateUrl: 'app/heroes.component.html'
7. })
8. export class HeroesComponent {
9. heroes: Hero[] = HEROES;
10. }

It assigns a list of mocked heroes to its heroes property for binding within the template. Pretty straight forward.

Those heroes are currently a fixed, in-memory collection, defined in another file and imported by the component. That works in the early stages of development but it's far from ideal. As soon as we try to test this component or want to get our heroes data from a remote server, we'll have to change this component's implementation of heroes and fix every other use of the HEROES mock data.

Let's make a service that hides how we get Hero data.

Write this service in its own file. See [this note](https://angular.io/docs/ts/latest/guide/dependency-injection.html#forward-ref) to understand why.

1. import {Hero} from './hero';
2. import {HEROES} from './mock-heroes';
3. class HeroService {
4. heroes: Hero[];
5. constructor() {
6. this.heroes = HEROES;
7. }
8. getHeroes() {
9. return this.heroes;
10. }
11. }

Our HeroService exposes a getHeroes() method that returns the same mock data as before but none of its consumers need to know that.

A service is nothing more than a class in Angular 2. It remains nothing more than a class until we register it with the Angular injector.

### Configuring the Injector

We don't have to create the injector. Angular creates an application-wide injector for us during the bootstrap process.

1. bootstrap(HeroesComponent);

Let’s configure the injector at the same time that we bootstrap by adding our HeroService to an array in the second argument. We'll explain that array when we talk about [providers](https://angular.io/docs/ts/latest/guide/dependency-injection.html#providers) later in this chapter.

1. bootstrap(AppComponent, [HeroService]);

That’s it! The injector now knows about the HeroService which is available for injection across our entire application.

### Preparing the HeroesComponent for injection

The HeroesComponent should get its heroes from this service. Per the dependency injection pattern, the component must "ask for" the service in its constructor [as we explained earlier](https://angular.io/docs/ts/latest/guide/dependency-injection.html#ctor-injection)".

1. constructor(heroService: HeroService) {
2. this.heroes = heroService.getHeroes();
3. }

Adding a parameter to the constructor isn't all that's happening here.

We are writing the app in TypeScript and have followed the parameter name with a type notation, :HeroService. The class is also decorated with the@Component decorator (scroll up to confirm that fact).

When the TypeScript compiler evaluates this class, it sees the decorator and adds class metadata into the generated JavaScript code. Within that metadata lurks the information that associates the heroService parameter with the HeroService class.

That's how the Angular injector will know to inject an instance of the HeroService when it creates a new HeroesComponent.

### Creating the HeroesComponent with the injector (implicitly)

When we introduced the idea of an injector above, we showed how to create a new Car with that injector.

1. var car = injector.get(Car);

Search the entire Tour of Heroes source. We won't find a single line like

1. var hc = injector.get(HeroesComponent);

We could write code like that if we wanted to. We just don't have to. Angular does that for us when it renders a HeroesComponent whether we ask for it in an HTML template ...

1. <my-heroes></heroes>

... or navigate to a HeroesComponent view with the [router](https://angular.io/docs/ts/latest/guide/router.html).

### Singleton services

We might wonder what happens when we inject the HeroService into other components. Do we get the same instance every time?

Yes we do. Dependencies are singletons. We’ll discuss that later in our chapter about [Hierarchical Injectors](https://angular.io/docs/ts/latest/guide/hierarchical-dependency-injection.html).

### Testing the component

We emphasized earlier that designing a class for dependency injection makes it easier to test.

Mission accomplished! We don't even need the Angular Dependency Injection system to test the HeroesComponent. We simply create a newHeroesComponent with a mock service and poke at it:

1. it("should have heroes when created", () => {
2. let hc = new HeroesComponent(mockService);
3. expect(hc.heroes.length).toEqual(mockService.getHeroes().length);
4. })

### When the service needs a service

Our 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? We'd apply the same "constructor injection" pattern.

Here's a rewrite of HeroService with a new constructor that takes a logger parameter.

1. import {Injectable} from 'angular2/angular2';
2. import {Hero} from './hero';
3. import {HEROES} from './mock-heroes';
4. import {Logger} from './logger';
5. @Injectable()
6. class HeroService {
7. heroes: Hero[];
8. constructor(private logger: Logger) {
9. this.heroes = HEROES;
10. }
11. getHeroes() {
12. this.logger.log('Getting heroes ...')
13. return this.heroes;
14. }
15. }

The constructor now asks for an injected instance of a Logger and stores it in a private property called logger. We call that property within our getHeroes() method when anyone asks for heroes.

**The @Injectable() decoration catches our eye!**

**Always include the parentheses!** Always call @Injectable(). It's easy to forget the parentheses. Our application will fail mysteriously if we do. It bears repeating: **always include the parentheses.**

We haven't seen @Injectable() before. As it happens, we could have added it to HeroService. We didn't bother because we didn't need it then.

We need it now ... now that our service has an injected dependency. We need it because Angular requires constructor parameter metadata in order to inject a Logger. As [we mentioned earlier](https://angular.io/docs/ts/latest/guide/dependency-injection.html#di-metadata), TypeScript only generates metadata for classes that have a decorator. .

The HeroesComponent has an injected dependency too. Why don't we add @Injectable() to the HeroesComponent? We can add it if we really want to. It isn't necessary because the HeroesComponent is already decorated with @Component. TypeScript generates metadata forany class with a decorator and any decorator will do.

## Injector Providers

Remember when we added the HeroService to an array in the [bootstrap](https://angular.io/docs/ts/latest/guide/dependency-injection.html#bootstrap) process?

1. bootstrap(AppComponent, [HeroService]);

That list of classes is actually a list of **providers**.

"Providers" create the instances of the things that we ask the injector to inject. There are many ways to "provide" a thing that has the necessary shape and behavior to serve as a HeroService. A class is a natural provider - it's meant to be created. But it's not the only way to produce something injectable. We could hand the injector an object to return. We could give it a factory function to call. Any of these approaches might be a good choice under the right circumstances.

What matters is that the injector knows what to do when something asks for a HeroService.

### The Provider Class

When we wrote ...

1. [HeroService];

we used a short-hand expression for provider registration. Angular expanded that short-hand into a call to the Angular provide method

1. [provide(HeroService, {useClass:HeroService})];

and the provide method in turn creates a new instance of the Angular [Provider class](https://angular.io/docs/ts/latest/api/core/Provider-class.html):

1. [new Provider(HeroService, {useClass:HeroService})]

This provider instance associates a HeroService token with code that can create an instance of a HeroService.

The first parameter is the [token](https://angular.io/docs/ts/latest/guide/dependency-injection.html#token) that serves as the key for both locating a dependency value and registering the provider.

The second parameter is a provider definition object which we think of as a "recipe" for creating the dependency value. There are many ways to create dependency values ... and many ways to write a recipe.

### Alternative Class Providers

Occasionally we'll ask a different class to provide the service.

We do that regularly when testing a component that we're creating with dependency injection. In this example, we tell the injector to return aMockHeroService when something asks for the HeroService.

1. beforeEachProviders(() => [
2. provide(HeroService, {useClass: MockHeroService});
3. ]);

### Value Providers

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

We do that a lot when we write tests. We might write the following test setup for tests that explore how the HeroComponent behaves when the HeroService returns an empty hero list.

1. beforeEachProviders(() => {
2. let emptyHeroService = { getHeroes: () => [] };
3. return [ provide(HeroService, {useValue: emptyHeroService}) ];
4. });

Notice we defined the recipe with useValue instead of useClass.

### Factory Providers

Sometimes the best choice for a provider is neither a class nor a value.

Suppose our HeroService has some cool new feature that we're only offering to "special" users. The HeroService shouldn't know about users and we won't know if the current user is special until runtime anyway. We decide to extend our HeroService constructor to accept auseCoolFeature flag that toggles the feature on or off. We rewrite the HeroService again as follows.

1. @Injectable()
2. class HeroService {
3. heroes: Hero[];
4. constructor(private logger: Logger, private useCoolFeature: boolean) {
5. this.heroes = HEROES;
6. }
7. getHeroes() {
8. let msg = this.useCoolFeature ? 'the cool new way' : 'the old way';
9. this.logger.log('Getting heroes ...' + msg)
10. return this.heroes;
11. }
12. }

The feature flag is a simple boolean value. We'd like to inject the flag but it seems silly to write an entire class for a simple flag.

We can replace the HeroService provider with a factory function that creates a properly configured HeroService for the current user. We'll' build up to that result, beginning with our definition of the factory function:

1. let heroServiceFactory = (logger: Logger, userService: UserService) => {
2. return new HeroService(logger, userService.user.isSpecial);
3. }

The factory takes two parameters: the logger service and a user service. The logger we pass straight to the constructor as we did before.

We'll know to use the cool new feature if the userService.user.isSpecial flag is true, a fact we can't know until runtime.

We use dependency injection everywhere so of course the factory function depends on two injected services: Logger and UserService. We declare those requirements in our provider definition object:

1. let heroServiceDefinition = {
2. useFactory: heroServiceFactory,
3. deps: [Logger, UserService]
4. };

The useFactory field tells Angular that the provider is a factory function and that its implementation is the heroServiceFactory.

The deps property is an array of [provider tokens](https://angular.io/docs/ts/latest/guide/dependency-injection.html#token). The Logger and UserService classes serve as tokens for their own class providers.

Finally, we create the provider and adjust the bootstrapping to include that provider among its provider registrations.

1. let heroServiceProvider = provide(HeroService, heroServiceDefinition);
2. bootstrap(AppComponent, [heroServiceProvider, Logger, UserService]);

### String tokens

Sometimes we have an object dependency rather than a class dependency.

Applications often define configuration objects with lots of small facts like the title of the application or the address of a web api endpoint. These configuration objects aren't always instances of a class. They're just objects ... like this one:

1. let config = {
2. apiEndpoint: 'api.heroes.com',
3. title: 'The Hero Employment Agency'
4. };

We'd like to make this config object available for injection. We know we can register an object with a "Value Provider". But what do we use for the token?

Until now, we've always asked the class to play the token role whether we wrote a provider with a class, value, or factory recipe. This time we don't have a class to serve as a token. There is no Config class.

Fortunately, the token can be a string, a class type, or an [OpaqueToken](https://angular.io/docs/ts/latest/api/core/OpaqueToken-class.html). Internally, the Provider turns the string and class parameter into anOpaqueToken; the injector locates dependency values and providers by this token.

We'll register our configuration object with a string-based token!

1. bootstrap(AppComponent, [
2. // other providers //
3. provide('App.config', {useValue:config})
4. ]);

Let's apply what we've learned and update the HeroesComponent constructor so it can display the configured title. Right now the constructor signature is

1. constructor(heroService: HeroService)

We might think we can add the config dependency by writing:

1. // FAIL!
2. constructor(heroService: HeroService, config: config)

That's not going to work. There is no type called config and we didn't register the config object under that name anyway. We'll need a little help from another Angular decorator called @Inject.

1. import {Inject} from 'angular2/angular2'
2. constructor(heroService: HeroService, @Inject('app.config') config)

# Next Steps

We learned the basics of Angular Dependency Injection in this chapter.

The Angular Dependency Injection is more capable than we've described. We can learn more about its advanced features, beginning with its support for nested injectors, in the [Hierarchical Dependency Injection](https://angular.io/docs/ts/latest/guide/hierarchical-dependency-injection.html) chapter.

### Appendix: Why we recommend one class per file

Developers expect one class per file. Multiple classes per file is confusing and is best avoided. If we define every class in its own file, there is nothing in this note to worry about. Move along!

If we scorn this advice and we add our HeroService class to the HeroesComponent file anyway, **define the HeroesComponent last!** If we put it define component before the service, we'll get a runtime null reference error.

To understand why, paste the following incorrect, ultra-simplified rendition of these two classes into the [TypeScript playground](http://www.typescriptlang.org/Playground).

1. class HeroesComponent {
2. static $providers=[HeroService]
3. }
4. class HeroService { }
5. alert(HeroesComponent.$providers)

The HeroService is incorrectly defined below the HeroComponent.

The $providers static property represents the metadata about the injected HeroService that TypeScript compiler would add to the component class.

The alert simulates the action of the Dependency Injector at runtime when it attempts to create a HeroesComponent.

Run it. The alert appears but displays nothing. This is the equivalent of the null reference error thrown at runtime.

We understand why when we review the generated JavaScript which looks like this:

1. var HeroesComponent = (function () {
2. function HeroesComponent() {
3. }
4. HeroesComponent.$providers = [HeroService];
5. return HeroesComponent;
6. })();
7. var HeroService = (function () {
8. function HeroService() {
9. }
10. return HeroService;
11. })();
12. alert(HeroesComponent.$providers);

Notice that the TypeScript compiler turns classes into function expressions assigned to variables. The value of the captured HeroServicevariable is undefined when the $providers array is assigned. The HeroService variable gets its value too late to be captured.

Reverse the order of class definition so that the HeroService appears before the HeroesComponent that requires it. Run again. This time the alert displays the HeroService function definition.

If we insist on defining the HeroService in the same file and insist on defining the component first, Angular offers a way to make that work. The forwardRef() method let's us reference a class before it has been defined. Learn more about this problem and the forwardRef() in this [blog post](http://blog.thoughtram.io/angular/2015/09/03/forward-references-in-angular-2.html).

1. TEMPLATE SYNTAX

How to write templates that display data and consume user events with the help of data binding.

# Template Syntax

Our Angular application manages what the user sees and does through the interaction of a Component class instance and its user-facing template.

Many of us are familiar with the Component/Template duality from our experience with Model-View-Controller or Model-View-ViewModel. In Angular, the Component plays the part of the Controller/ViewModel and the Template represents the view.

Let’s find out what it takes to write a Template for our view.

We’ll cover these basic elements of Template Syntax

[HTML](https://angular.io/docs/ts/latest/guide/template-syntax.html#html)

[Interpolation](https://angular.io/docs/ts/latest/guide/template-syntax.html#interpolation)

[Template expressions](https://angular.io/docs/ts/latest/guide/template-syntax.html#template-expressions)

[Binding syntax](https://angular.io/docs/ts/latest/guide/template-syntax.html#binding-syntax)

[Property Binding](https://angular.io/docs/ts/latest/guide/template-syntax.html#property-binding)

[Attribute, Class and Style Bindings](https://angular.io/docs/ts/latest/guide/template-syntax.html#other-bindings)

[Event Binding](https://angular.io/docs/ts/latest/guide/template-syntax.html#event-binding)

[Two-way data binding with NgModel](https://angular.io/docs/ts/latest/guide/template-syntax.html#ngModel)

[Built-in Directives](https://angular.io/docs/ts/latest/guide/template-syntax.html#directives)

[\* and <template>](https://angular.io/docs/ts/latest/guide/template-syntax.html#star-template)

[Local template variables](https://angular.io/docs/ts/latest/guide/template-syntax.html#local-vars)

[Input and Output Properties](https://angular.io/docs/ts/latest/guide/template-syntax.html#inputs-outputs)

[Template Expression Operators](https://angular.io/docs/ts/latest/guide/template-syntax.html#expression-operators)

[Live Example](https://angular.io/resources/live-examples/template-syntax/ts/plnkr.html).

## HTML

HTML is the language of the Angular template. Our “[QuickStart](https://angular.io/docs/ts/latest/guide/quickstart.html)” application had a template that was pure HTML

<h3>My First Angular Application</h3>

Almost all HTML syntax is valid template syntax. The <script> element is a notable exception; it is forbidden in order to eliminate any possibility of JavaScript injection attacks (in practice it is simply ignored).

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

We can extend the HTML vocabulary of our templates with Components and Directives that appear as new elements and attributes. And we are about to learn how to get and set DOM values dynamically through data binding.

Let’s turn to the first form of data binding - interpolation - to see how much richer Template HTML can be.

## Interpolation

We met the double-curly braces of interpolation, {{ and }}, early in our Angular education.

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

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

<h3>

{{title}}

<img src="{{heroImageUrl}}" style="height:30px">

</h3>

The material 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 this example, Angular evaluates the title and heroImageUrl properties and "fills in the blanks", displaying first a bold application title and then a heroic image.

More generally, the material 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 within braces:

<!-- "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 as we do here with getVal():

<!-- "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 concatenates them with neighboring literal strings. Finally, it assigns this composite interpolated result to an **element or directive property**.

We appear to be inserting the result between element tags and assigning to attributes. It's convenient to think so and we rarely suffer for this mistake. But it is not literally true. Interpolation is actually a special syntax that Angular converts into a [Property Binding](https://angular.io/docs/ts/latest/guide/template-syntax.html#property-binding) as we explain below. The implications and consequences can be profound.

But before we explore that assertion, we’ll take a closer look at template expressions.

## Template Expressions

We saw a template expression within the interpolation braces. We’ll see template expressions again in [Property Bindings](https://angular.io/docs/ts/latest/guide/template-syntax.html#property-binding)([property]="expression") and [Event Bindings](https://angular.io/docs/ts/latest/guide/template-syntax.html#event-binding) ((event)="expression").

A template expression is a JavaScript-like expression. Many JavaScript expressions are legal template expressions but not all and there are a few language extensions. Notable differences include:

* Assignment is prohibited except in [Event Bindings](https://angular.io/docs/ts/latest/guide/template-syntax.html#event-binding).
* The new operator is prohibited.
* The bit-wise operators, | and &, are not supported.
* Increment and decrement operators, ++ and --, aren’t supported.
* [Template expression operators](https://angular.io/docs/ts/latest/guide/template-syntax.html#expression-operators), such as | and ?., add new meaning.

Perhaps more surprising, we cannot refer to anything in the global namespace. We can’t refer to window or document. We can’t callconsole.log. We are restricted to referencing members of the expression context.

The **expression context** is typically the **component instance** supporting a particular template instance.

We speak of component and template **instances**. Angular creates multiple concrete instances from a component class and its template.

For example, we may define a component and template to display a list item and tell Angular to create new instances of that component/template pair for each item in a list. There’s a separate, independent expression context for each item in that list as well.

When we see title wrapped in double-curly braces, {{ }}., we know that it is a property of a parent component. When we see[disabled]="isUnchanged" or (click)="onCancel()”, we know we are referring to that component's isUnchanged property andonCancel method respectively.

The component itself is usually the expression context in which case the template expression usually references that component. The expression context may include an object other than the component.

A [local template variable](https://angular.io/docs/ts/latest/guide/template-syntax.html#local-vars) is one such supplemental context object; we’ll discuss that option below.

Another is the **$event** variable that contains information about an event raised on an element; we’ll talk about that when we consider [Event Bindings](https://angular.io/docs/ts/latest/guide/template-syntax.html#event-binding).

Although we can write quite complex template expressions, we strongly discourage that practice. Most readers frown on JavaScript in the HTML. 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.

Now that we have a feel for template expressions, we’re ready to learn about the varieties of data binding syntax beyond Interpolation.

## Binding syntax overview

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

Angular provides many kinds of data binding and we’ll discuss each of them in this chapter. First we'll take a high level view of Angular data binding and its syntax.

We can group all bindings into three categories by the direction in which data flows. Each category has its distinctive syntax:

|  |  |  |
| --- | --- | --- |
| Data Direction | Syntax | Binding 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) = "expression"  on-target = "expression" | Event |
| Two way | [(target)] = "expression"  bindon-target = "expression" | Two-way |

**Template expressions must be surrounded in quotes** except for interpolation expressions which must not be quoted.

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

What is that target? Before we can answer that question, we must challenge ourselves to look at Template HTML in a new way.

### A New Mental Model

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

But it’s also significantly different than the HTML we’re used to. We really need a new mental model.

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

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

<div><b>{{currentHero.fullName}}</b></div>

<img src="images/hero.png">

<button disabled>Save</button>

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

Then we learn to create new elements with Components that encapsulate HTML and drop them into our templates as if they were native HTML elements

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

<hero-detail></hero-detail>

That’s “HTML Plus”.

Now we start to learn about data binding. The first binding we meet might look like this:

<!-- Bind button disabled state to `isUnchanged` property -->

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

We’ll get to that peculiar bracket notation in a moment. Looking beyond it, our intuition tells us that we’re binding to the button's disabledattribute and setting it to the current value of the component’s isUnchanged property.

Our intuition is wrong! Our everyday HTML mental model is misleading us. In fact, once we start data binding, we are no longer working with HTML attributes. We aren't setting attributes. We 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 DOM (the 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 the way we think!

That last category can be especially confusing ... until we understand this general rule:

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

For example, when the browser renders <input type="text" value="Bob">, it creates a corresponding DOM node with a value property initializedto "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 we discover if we 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 we add thedisabled attribute, it's 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 we cannot enable a button by writing <button disabled="false">Still Disabled</button>.

Setting the button's disabled property (e.g. 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 is so important, we’ll say it again.

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

A WORLD WITHOUT ATTRIBUTES

In the world of Angular 2, the only role of attributes is to initialize element and directive state. When we data bind, we're dealing exclusively with element and directive properties and events. Attributes effectively disappear.

With this model firmly in mind, we are ready to discuss the variety of target properties to which we may bind.

### 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:

|  |  |  |
| --- | --- | --- |
| Binding Type | Target | Examples |
| Property | Element Property Component Property Directive property | <img [src] = "heroImageUrl">  <hero-detail [hero]="currentHero"></hero-detail>  <div [ngClass] = "{selected: isSelected}"></div> |
| Event | Element Event Component Event Directive Event | <button (click) = "onSave()">Save</button>  <hero-detail (deleted)="onHeroDeleted()"></hero-detail>  <div myClick (myClick)="clicked=$event">click me</div> |
| Two-way | Directive Event Property | <input [(ngModel)]="heroName"> |
| Attribute | Attribute (the exception) | <button [attr.aria-label]="help">help</button> |
| Class | class Property | <div [class.special]="isSpecial">Special</div> |
| Style | style Property | <button [style.color] = "isSpecial ? 'red' : 'green'"> |

Let’s descend from the architectural clouds and look at each of these binding types in concrete detail.

## Property Binding

We write a template **Property Binding** when we want to set a property of a view element to the value of a template expression.

The most common Property Binding sets an element property to a component property value as when we bind the source property of an image element to the component’s heroImageUrl property.

<img [src]="heroImageUrl">

… or disable a button when the component says that it isUnchanged.

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

… or set a property of a directive

<div [ngClass]="'special'">NgClass is special</div>

… or set the model property of a custom component (a great way for parent and child components to communicate)

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

People often describe Property Binding as “one way data binding” because it can flow a value in one direction, from a component’s data property to an element property.

### Binding Target

A name between enclosing square brackets identifies the target property. The target property in this example is the image element’s srcproperty.

<img [src]="heroImageUrl">

Some developers prefer the bind- prefix alternative, known as the “canonical form”:

<img bind-src="heroImageUrl">

The target name is always the name of a property, even when it appears to be the name of something else. We 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:

<div [ngClass]="'special'">NgClass is special</div>

Technically, Angular is matching the name to a directive [input](https://angular.io/docs/ts/latest/guide/template-syntax.html#inputs-outputs), one of the property names listed in the directive’s inputs array or a property decorated with @Input(). Such inputs map to the directive’s own properties.

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

### Property Binding template expressions

Evaluation of the expression should have no visible side-effects. The expression language itself does its part to keep us safe. We can’t assign a value to anything in a Property Binding expression nor use the increment and decorator operators.

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

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

The template expression should evaluate to a value of the type expected by the target property. Most native element properties do expect a string. The image src should be set to a string, an URL for the resource providing the image. On the other hand, the disabled property of a button expects a Boolean value so our expression should evaluate to true or false.

The hero property of the HeroDetail component expects a Hero object which is exactly what we’re sending in the Property Binding:

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

This is good news for the Angular developer. If hero were an attribute we could not set it to a Hero object.

<!-- Doesn't work! HeroDetailComponent expects a Hero, not a string -->

<hero-detail hero="…what do we do here??? …"></hero-detail>

We can't set an attribute to an object. We can only set it to a string. Internally, the attribute may be able to convert that string to an object before setting the like-named element property. That’s good to know but not helpful to us when we're trying to pass a significant data object from one component element to another. The power of Property Binding is its ability to bypass the attribute and set the element property directly with a value of the appropriate type.

### Property Binding or Interpolation?

We often have a choice between Interpolation and Property Binding. The following binding pairs do the same thing

<img src="{{heroImageUrl}}">

<img [src]="'' + heroImageUrl">

<div>The title is {{title}}</div>

<div [textContent]="'The title is '+title"></div>

Interpolation is actually a convenient alternative for Property Binding in many cases. In fact, Angular translates those Interpolations into the corresponding Property Bindings before rendering the view.

There is no technical reason to prefer one form to the other. We lean toward readability which tends to favor interpolation. We suggest establishing coding style rules for the organization and choosing the form that both conforms to the rules and feels most natural for the task at hand.

## Attribute, Class, and Style Bindings

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

### Attribute Binding

We 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.

We have stressed throughout this chapter 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?

**We must use Attribute Binding when there is no element property to bind.**

Consider the [aria](https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA), [svg](https://developer.mozilla.org/en-US/docs/Web/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.

We become painfull aware of this fact when we try to write something like this:

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

… and get the 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 only set properties, not attributes.

We need an Attribute Binding to create and bind to such attributes.

Attribute Binding syntax resembles Property Binding. Instead of an element property between brackets, we start with the keyword **attr**followed by the name of the attribute and then set it with an expression that resolves to a string.

Here we bind [attr.colspan] to a calulated value:

<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 the Attribute Binding is to set aria attributes as we see in this example

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

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

### Class Binding

We can add and remove CSS class names from an element’s class attribute with the **Class Binding**.

Class Binding syntax resembles Property Binding. Instead of an element property between brackets, we start with the keyword classfollowed by the name of a CSS class: [class.class-name].

In the following examples we see how to add and remove the application's "special" class with class bindings. We start by setting the attribute without binding:

<!-- standard class attribute setting -->

<div class="bad curly special">Bad curly special</div>

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

<!-- reset all class names with a string binding -->

<div class="bad curly special"

[class]="'bad curly'">Bad curly</div>

Finally, we bind to a specific class name. Angular adds the class when the template expression evaluates to something truthy and removes the class name when the expression is falsey.

<!-- 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, we generally prefer the [NgClass directive](https://angular.io/docs/ts/latest/guide/template-syntax.html#ngClass) for managing multiple class names at the same time.

### Style Binding

We can set inline styles with a **Style Binding**.

Style Binding syntax resembles Property Binding. Instead of an element property between brackets, we start with the key word stylefollowed by the name of a CSS style property: [style.style-property].

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

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

Some Style Binding styles have unit extension. Here we conditionally set the fontSize in “em” and “%” units .

<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, we generally prefer the [NgStyle directive](https://angular.io/docs/ts/latest/guide/template-syntax.html#ngStyle) when setting several inline styles at the same time.

## Event Binding

The bindings we’ve met so far flow data in one direction from the component to an element.

Users don’t just stare at the screen. They enter text into a input box. They pick items from a list. They click buttons. Such user actions may result in a flow of data in the opposite direction, **from an element to the component**.

The only way to know about a user action is to listen for certain events such as keystrokes, mouse movements, clicks and touches. We declare our interest in user actions through Angular Event Binding.

The following Event Binding listens for the button’s click event and calls the component's onSave() method:

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

Event Binding syntax consists of a target event on the left of an equal sign and a template expression on the right: (click)="onSave()"

### Binding target

A **name between enclosing parentheses** identifies the target event. In the following example, the target is the button’s click event.

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

Some developers prefer the on- prefix alternative, known as the “canonical form”:

<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:

<!-- `myClick` is an event on the custom `MyClickDirective` -->

<div myClick (myClick)="clickity=$event">click with myClick</div>

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 expressions

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

When the event is raised, the handler executes the template expression. The template expression typically involves a receiver that wants to do something in response to the event such as take a value from the HTML control and store it in 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 itself. If the target event is a native DOM element event, the $event is a[DOM event object](https://developer.mozilla.org/en-US/docs/Web/Events) with properties such as target and target.value.

Consider this example:

<input [value]="currentHero.firstName"

(input)="currentHero.firstName=$event.target.value" >

We’re binding the input box value to a firstName property and we’re listening for changes by binding to the input box’s input event. When the user makes changes, the input event is raised, and the binding executes the expression within a context that includes the DOM event object, $event.

We must follow the $event.target.value path to get the changed text so we can update the firstName

If the event belongs to a directive (remember: components are directives), $event has whatever shape the directive chose to produce. Consider a HeroDetailComponent that produces deleted events with an EventEmitter.

HeroDetailComponent.ts (excerpt)

deleted = new EventEmitter<Hero>();

onDelete() {

this.deleted.emit(this.hero);

}

When something invokes the onDeleted() method, we "emit" a Hero object.

Now imagine a parent component that listens for that event with an Event Binding.

<hero-detail (deleted)="onHeroDeleted($event)" [hero]="currentHero">

</hero-detail>

The event binding surfaces the hero-to-delete emitted by HeroDetail to the expression via the $hero variable. We pass it along to the parent onHeroDeleted() method. That method presumably knows how to delete the hero.

Evaluation of an Event Binding template expression may have side-effects as this one clearly does. They are not just OK (unlike in property bindings); they are expected.

The expression could update the model thereby triggering other changes that percolate through the system, changes that are ultimately displayed in this view and other views. The event processing may result in queries and saves to a remote server. It's all good.

### Event bubbling and propagation

Angular invokes the event-handling expression if the event is raised by the current element or one of its child elements.

<div class="parent-div" (click)="onClickMe($event)">Click me

<div class="child-div">Click me too!</div>

</div>

Many DOM events, both [native](https://developer.mozilla.org/en-US/docs/Web/Guide/Events/Overview_of_Events_and_Handlers) and [custom](https://developer.mozilla.org/en-US/docs/Web/Guide/Events/Creating_and_triggering_events), “bubble” events up their ancestor tree of DOM elements until an event handler along the way prevents further propagation.

EventEmitter events don’t bubble.

The result of an Event Binding expression determines if [event propagation](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Examples#Example_5:_Event_Propagation) continues or stops with the current element.

Event propagation stops if the binding expression returns a falsey value (as does a method with no return value). Clicking the button in this next example triggers a save; the click doesn't make it to the outer <div> so it's save is not called:

<!-- Will save only once -->

<div (click)="onSave()">

<button (click)="onSave()">Save, no propagation</button>

</div>

Propagation continues if the expression returns a truthy value. The click is heard both by the button and the outer <div>, causing a double save:

<!-- Will save twice -->

<div (click)="onSave()">

<button (click)="onSave() || true">Save w/ propagation</button>

</div>

## Two-Way Binding with NgModel

When developing data entry forms we often want to both display a data property and update that property when the user makes changes.

The NgModel directive serves that purpose as seen in this example:

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

If we prefer the “canonical” prefix form to the punctuation form, we can write

<input bindon-ngModel="currentHero.firstName">

There’s a story behind this construction, a story that builds on the Property and Event Binding techniques we learned previously.

We could have achieved the same result as NgModel with separate bindings to the the <input> element's value property and inputevent.

<input [value]="currentHero.firstName"

(input)="currentHero.firstName=$event.target.value" >

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

That ngModel directive hides these onerous details. It wraps the element’s value property, listens to the input event, and raises its own event with the changed text ready for us to consume.

<input

[ngModel]="currentHero.firstName"

(ngModelChange)="currentHero.firstName=$event">

That’s an improvement. It should be better.

We shouldn't have to mention the data property twice. Angular should be able to read the component’s data property and set it with a single declaration — which it can with the [( )] syntax:

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

Internally, Angular maps the term, ngModel, to an ngModel input property and an ngModelChange output property. That’s a specific example of a more general pattern in which it matches [(x)] to an x input property for Property Binding and an x-change output property for Event Binding.

We can write our own two-way binding directive that follows this pattern if we're ever in the mood to do so.

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

Well NgModel can only set the target property. What if we need to do something more or something different when the user changes the value?

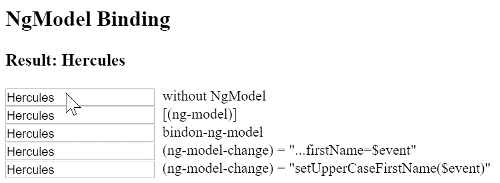
Let's try something silly like forcing the input value to uppercase.

<input

[ngModel]="currentHero.firstName"

(ngModelChange)="setUpperCaseFirstName($event)">

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



## Built-in Directives

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

We don’t need many of those directives in Angular 2. Quite often we can achieve the same results with the more capable and expressive Angular 2 binding system. Why create a directive to handle a click when we can write a simple binding such as this?

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

We still benefit from directives that simplify complex tasks. Angular still ships with built-in directives; just not as many. We will write our own directives; just not as many.

In this segment we review some of the most frequently used built-in directivest.

### NgClass

We typically control how elements appear by adding and removing CSS classes dynamically. We can bind to NgClass to add or remove several classes simultaneously.

We prefer to use a [Class Binding](https://angular.io/docs/ts/latest/guide/template-syntax.html#class-binding) to add or remove a single class.

<!-- toggle the "special" class on/off with a property -->

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

The NgClass directive may be the better choice when we want to add or remove many classes at the same time.

Our favorite way to apply NgClass is by binding it to a key:value control object. Each key of the object is a class name and its value is trueif the class should be added, false if it should be removed.

Consider a component method such as setClasses that returns its approval of two class names:

setClasses() {

return {

saveable: this.canSave, // true

modified: !this.isUnchanged, // false

special: this.isSpecial, // true

}

}

Now add an NgClass property binding to call it like this

<div [ngClass]="setClasses()">This div is saveable and special</div>

### NgStyle

We may set inline styles dynamically based on the state of the component. We bind to NgStyle to set many inline styles simultaneously.

We prefer to use a [Style Binding](https://angular.io/docs/ts/latest/guide/template-syntax.html#style-binding) to set a single style value.

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

This div is x-large

</div>

The NgStyle directive may be the better choice when we want to set many inline styles at the same time.

We apply NgStyle by binding it to a key:value control object. Each key of the object is a style name and its value is whatever is appropriate for that style.

Consider a component method such as setStyles that returns an object defining three styles:

setStyles() {

return {

// CSS property names

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

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

'font-size': this.isSpecial ? 'x-large': 'smaller', // larger

}

}

Now add an NgStyle property binding to call it like this

<div [ngStyle]="setStyles()">

This div is italic, normal weight, and x-large

</div>

Alternatively, we can return an object with camelCase style property names with the same effects:

setStyles2() {

return {

// camelCase style properties work too

fontStyle: this.canSave ? 'italic' : 'normal', // italic

fontWeight: !this.isUnchanged ? 'bold' : 'normal', // normal

fontSize: this.isSpecial ? 'x-large': 'smaller', // larger

}

}

### NgIf

We can add an element sub-tree (an element and its children) to the DOM by binding an NgIf directive to a truthy expression.

<div \*ngIf="currentHero">Hello, {{currentHero.firstName}}</div>

The leading asterisk (\*) in front of ngIf is a critical part of this syntax. See the section below on [\* and <template>](https://angular.io/docs/ts/latest/guide/template-syntax.html#star-template).

Binding to a falsey expression removes the element sub-tree from the DOM.

<!-- not displayed because nullHero is falsey.

`nullHero.firstName` never has a chance to fail -->

<div \*ngIf="nullHero">Hello, {{nullHero.firstName}}</div>

<!-- Hero Detail is not in the DOM because isActive is false-->

<hero-detail \*ngIf="isActive"></hero-detail>

#### Visibility and NgIf are not the same

We can show and hide an element sub-tree (the element and its children) with a [Class](https://angular.io/docs/ts/latest/guide/template-syntax.html#class-binding) or a [Style](https://angular.io/docs/ts/latest/guide/template-syntax.html#style-binding) binding:

<!-- 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 a sub-tree is quite different from excluding a sub-tree with NgIf.

When we hide the element sub-tree, it remains in the DOM. Components in the sub-tree are preserved along with their state. Angular may continue to check for changes even to invisible properties. The sub-tree may tie up substantial memory and computing resources.

When NgIf is false, Angular physically removes the element sub-tree from the DOM. It destroys components in the sub-tree along with their state which may free up substantial resources resulting in better performance for the user.

The show/hide technique is probably fine for small element trees. We should be wary when hiding large trees; NgIf may be the safer choice. Always measure before leaping to conclusions.

### NgSwitch

We bind to NgSwitch when we want to display one element tree (an element and its children) from a set of possible elment trees based on some condition. Angular only puts the selected element tree into the DOM.

Here’s an example:

<span [ngSwitch]="toeChoice">

<template [ngSwitchWhen]="'Eenie'">Eenie</template>

<template [ngSwitchWhen]="'Meanie'">Meanie</template>

<template [ngSwitchWhen]="'Miney'">Miney</template>

<template [ngSwitchWhen]="'Moe'">Moe</template>

<template ngSwitchDefault>Other</template>

</span>

We bind the parent NgSwitch directive to an expression returning a “switch value”. The value is a string in this example but it can be a value of any type.

The parent NgSwitch directive controls a set of child<template> elements. Each <template> wraps a candidate subtree. A template is either pegged to a “match value” expression or marked as the default template.

**At any particular moment, only one of these templates will be rendered**

If the template’s “match value” equals the “switch value”, Angular adds the template’s sub-tree to the DOM. If no template is a match and there is a default template, Angular adds the default template’s sub-tree to the DOM. Angular removes and destroys the sub-tree’s of all other templates.

There are three collaborating directives at work here.

1. ngSwitch - bound to an expression that returns the switch value.
2. ngSwitchWhen - bound to an expression returning a match value.
3. ngSwitchDefault - a marker attribute on the default template.

### NgFor

NgFor is a “repeater” directive. It will be familiar if we’ve written repeaters for other view engines.

Our goal is to present a list of items. We define a block of HTML that defines how a single item should be displayed. We 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>.

<div \*ngFor="#hero of heroes">{{hero.fullName}}</div>

We can apply an NgFor to a component element as well as we do in this example:

<hero-detail \*ngFor="#hero of heroes" [hero]="hero"></hero-detail>

The leading asterisk (\*) in front of ngFor is a critical part of this syntax. See the section below on [\* and <template>](https://angular.io/docs/ts/latest/guide/template-syntax.html#star-template).

The text assigned to \*ngFor is the instruction that guides the repeater process.

#### NgFor Micro-syntax

The string assigned to \*ngFor is not a [template expression](https://angular.io/docs/ts/latest/guide/template-syntax.html#template-expressions). It’s a little language of its own called a “micro-syntax” that Angular interprets. In this example it means:

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

Angular translates this instruction into a new set of elements and bindings. We’ll talk about this in the next section about templates.

In our two examples, the ngFor directive iterates over the heroes array returned by the parent component’s heroes property and stamps out instances of the element to which it is applied. Angular creates a fresh instance of the template for each hero in the array.

The (#) character before "hero" identifies a [local template variable](https://angular.io/docs/ts/latest/guide/template-syntax.html#local-vars) called hero.

We reference this variable within the template to access a hero’s properties as we’re doing in the interpolation or we can pass it in a binding to a component element as we're doing with hero-detail.

#### NgFor with index

The ngFor directive supports an optional index that increases from 0 to the length of the array for each iteration. We can capture that in another local template variable (i) and use it in our template too.

This next example stamps out rows that display like "1 - Hercules Son of Zeus":

<div \*ngFor="#hero of heroes, #i=index">{{i + 1}} - {{hero.fullName}}</div>

## \* and <template>

When we reviewed the ngFor and ngIf built-in directives we called out an oddity of the syntax, the asterisk (\*) that appears before the directive name.

This is a bit of “**syntactic sugar**” that makes it easier to read and write directives that modify HTML layout with the help of templates. NgFor,NgIf, and NgSwitch all add and remove element subtrees that are wrapped in <template> tags.

With the [NgSwitch](https://angular.io/docs/ts/latest/guide/template-syntax.html#ngSwitch) directive we always write the <template> tags explicitly as we saw [above](https://angular.io/docs/ts/latest/guide/template-syntax.html#ngSwitch). There isn’t much choice; we define a different template for each switch choice and let the directive render the template that matches the switch value.

[NgFor](https://angular.io/docs/ts/latest/guide/template-syntax.html#ngFor) and [NgIf](https://angular.io/docs/ts/latest/guide/template-syntax.html#ngIf) only need one template, the template-to-repeat and the template-to-include respectively.

The (\*) prefix syntax is a convenient way for developers to skip the <template> wrapper tags and focus directly on the HTML element to repeat or include. Angular sees the (\*) and expands the HTML into the <template> tags for us.

### Expanding \*ngIf

We can do that ourselves if we wish. Instead of writing ...

<hero-detail \*ngIf="currentHero" [hero]="currentHero"></hero-detail>

… we can write:

<template [ngIf]="currentHero">

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

</template>

Notice the (\*) is gone and we have a [Property Binding](https://angular.io/docs/ts/latest/guide/template-syntax.html#property-binding) to the ngIf directive, applied in this case to the <template> rather than the application’s hero-detail component.

The [hero]="currentHero" binding remains on the child <hero-detail> element inside the template. Angular does that too.

REMEMBER THE BRACKETS!

Don’t make the mistake of writing ngIf="currentHero"! That syntax assigns the string value, "currentHero", to ngIf. In JavaScript a non-empty string is a truthy value so ngIf would always be true and Angular will always display the hero-detail … even when there is no currentHero!

### Expanding \*ngFor

A similar transformation applies to \*ngFor. We can "de-sugar" the syntax ourselves and go from ...

<hero-detail \*ngFor="#hero of heroes" [hero]="hero"></hero-detail>

... to

<hero-detail template="ngFor #hero of heroes" [hero]="hero"></hero-detail>

... and from there to

<template ngFor #hero [ngForOf]="heroes">

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

</template>

This is a bit more complex than NgIf because a repeater has more moving parts to configure. In this case, we have to remember theNgForOf directive that identifies the list. It should be clear why we prefer the \*ngFor syntax to writing out this expanded HTML ourselves.

## Local template variables

A **local template variable** is a vehicle for moving data across element lines.

We've seen the #hero local template variable several times in this chapter, most prominently when writing [NgFor](https://angular.io/docs/ts/latest/guide/template-syntax.html#ngFor) repeaters.

In the [\* and <templates>](https://angular.io/docs/ts/latest/guide/template-syntax.html#star-template) segment we learned how Angular expands an \*ngFor on a component tag into a <template> that wraps the component.

<template ngFor #hero [ngForOf]="heroes">

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

</template>

The (#) prefix character in front of "hero" means that we're defining a hero variable.

Some folks don't like the (#) character. The var- prefix is the “cannonical” alternative to "#". We could have declared our variable as var-hero.

We defined hero on the outer <template> element where it becomes the current hero item as Angular iterates through the list of heroes.

The hero variable appears again in the binding on the inner <hero-detail> component element. That's how each instance of the <hero-detail> gets its hero.

### Referencing a local template variable

We can reference a local template variable on the same element, on a sibling element, or on any of its child elements.

Here are two other examples of creating and consuming a local template variable:

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

<input #phone placeholder="phone number">

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

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

<input var-fax placeholder="phone number">

<button (click)="callFax(fax.value)">Fax</button>

### How it gets its value

The value assigned to the variable depends upon the context.

When a directive is present on the element, as it is in the earlier NgFor <hero-detail> component example, the directive sets the value. Accordingly, the NgFor directive set the hero variable to a hero item from the heroes array.

When there is no directive present, as in phone and fax examples, Angular sets the variable to the element on which it was defined. We defined these variables on the input elements. We’re passing those input element objects across to the button elements where they become arguments to the call() methods in the event bindings.

### NgForm and local template variables

Let's look at one final example, a Form, the poster child for local template variables.

The HTML for a form can be quite involved as we saw in the [Forms](https://angular.io/docs/ts/latest/guide/forms.html) chapter. The following is a simplified example — and it's not simple at all.

1. <form (ngSubmit)="onSubmit(theForm)" #theForm="ngForm">
2. <div class="form-group">
3. <label for="name">Name</label>
4. <input class="form-control" required ngControl="firstName"
5. [(ngModel)]="currentHero.firstName">
6. </div>
7. <button type="submit" [disabled]="!theForm.form.valid">Submit</button>
8. </form>

A local template variable, theForm, appears three times in this example, separated by a large amount of HTML.

1. <form (ngSubmit)="onSubmit(theForm)" #theForm="ngForm">
2. <!-- . . . -->
3. <button type="submit" [disabled]="!theForm.form.valid">Submit</button>
4. </form>

What is the value of theForm?

It would be the [HTMLFormElement](https://developer.mozilla.org/en-US/docs/Web/API/HTMLFormElement) if Angular hadn't taken it over. It's actually ngForm, a reference to the Angular built-in NgForm directive that wraps the native HTMLFormElement and endows it with additional super powers such as the ability to track the validity of user input.

This explains how we can disable the submit button by checking theForm.form.valid and pass an object with rich information to the parent component's onSubmit method.

## Input and Output Properties

We can only bind to **target directive** properties that are either **inputs** or **outputs**.

We're drawing a sharp distinction between a data binding **target** and a data binding **source**.

The target is to the left of the (=) in a binding expression. The source is on the right of the (=).

Every member of a **source** directive (typically a component) is automatically available for binding. We don't have to do anything special to access a component member in the quoted template expression to the right of the (=).

We have limited access to members of a **target** directive (typically a component). We can only bind to input and output properties of target components to the left of the (=).

Also remember that a component is a directive. In this section, we use the terms "directive" and "component" interchangeably.

In this chapter we’ve focused mainly on binding to component members within template expressions on the right side of the binding declaration. A member in that position is a binding “data source”. It's not a target for binding.

In the following example, iconUrl and onSave are members of the AppComponent referenced within template expressions to the right of the (=).

<img [src]="iconUrl"/>

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

They are neither inputs nor outputs of AppComponent. They are data sources for their bindings.

Now look at the HeroDetailComponent when it is the **target of a binding**.

<hero-detail [hero]="currentHero" (deleted)="onHeroDeleted($event)">

</hero-detail>

Both HeroDetail.hero and HeroDetail.deleted are on the **left side** of binding expressions. HeroDetail.hero is the target of a Property Binding. HeroDetail.deleted is the target of an Event Binding.

**Data flow into the HeroDetail.hero target property** from the template expression. Therefore HeroDetail.hero is an **input** property from the perspective of HeroDetail.

**Events stream out of the HeroDetail.deleted target property** and toward the receiver within the template expression. ThereforeHeroDetail.deleted is an **output** property from the perspective of HeroDetail.

When we peek inside HeroDetailComponent we see that these properties are marked with decorators as input and output properties.

@Input() hero: Hero;

@Output() deleted = new EventEmitter<Hero>();

Alternatively, we can identify them in the inputs and outputs arrays of the directive metadata as seen in this example:

@Component({

inputs: ['hero'],

outputs: ['deleted'],

})

We can specify an input/output property with a decorator or in one the metadata arrays. Don't do both!

### Aliasing input/output properties

Sometimes we want the public name of the property to be different from the internal name.

This is frequently the case with [Attribute Directives](https://angular.io/docs/ts/latest/guide/attribute-directives.html). Directive consumers expect to bind to the name of the directive. For example, we expect to bind to the myClick event property of the MyClickDirective.

The directive name is often a poor choice for the the internal property name because it rarely describes what the property does. The corresponding MyClickDirective internal property is called clicks.

Fortunately, we can alias the internal name to meet the conventional needs of the directive's consumer. We alias in decorator syntax like this:

@Output('myClick') clicks = new EventEmitter<string>();

The equivalent aliasing with the outputs array requires a colon-delimited string with the internal property name on the left and the public name on the right:

@Directive({

outputs:['clicks:myClick']

})

## Template Expression Operators

The template expression language employs a subset of JavaScript syntax supplemented with some special operators for specific scenarios. We'll cover two of them, "pipe" and "Elvis".

### The Pipe Operator ( | )

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

Angular [Pipes](https://angular.io/docs/ts/latest/guide/pipes.html) 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 are easy to apply within template expressions using the **pipe operator ( | )**:

<!-- Force title to uppercase -->

<div>{{ title | uppercase }}</div>

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

We can chain expressions through multiple pipes

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

<div>{{ title | uppercase | lowercase }}</div>

And we can configure them too:

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

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

The json pipe is particular helpful for debugging our bindings:

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

<!-- Output:

{ "firstName": "Hercules", "lastName": "Son of Zeus",

"birthdate": "1970-02-25T08:00:00.000Z",

"url": "http://www.imdb.com/title/tt0065832/",

"rate": 325, "id": 1 }

-->

### The Elvis Operator ( ?. ) and null property paths

The Angular **“Elvis” 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.

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

Let’s elaborate on the problem and this particular solution.

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

The title is {{ title }}

The view still renders but the displayed value is blank; we 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 where we’re displaying the firstName of a null hero.

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

JavaScript throws a null reference error and so does Angular:

TypeError: Cannot read property 'firstName' of null in [null]

Worse, the entire view disappears.

We could claim that this is reasonable behavior if we believed that the hero property must never be null. If it must never be null and yet it is null, we've made 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 we know the data will arrive eventually.

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

Unfortunately, our app crashes when the currentHero is null.

We could code around that problem with [NgIf](https://angular.io/docs/ts/latest/guide/template-syntax.html#ngIf)

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

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

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

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

These approaches have merit but they 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 **“Elvis” 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 and there are no errors.

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

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

It works perfectly with long property paths too:

1. a?.b?.c?.d
2. PIPES

Pipes transform displayed values within a template.

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, we could push their raw toString values directly to screen. That rarely makes for a good user experience. Almost everyone prefers a simple birthday date (April 15, 1988) to the original raw string format ( Fri Apr 15 1988 00:00:00 GMT-0700 (Pacific Daylight Time) ).

Clearly some values benefit from a bit of massage. We soon discover that we desire many of the same transformations repeatedly, both within and across many applications. We almost think of them as styles. In fact, we'd like to apply them in our HTML templates as we do styles.

Welcome, Angular pipes, the simple display-value transformations that we can declare in our HTML!

[Live Example](https://angular.io/resources/live-examples/pipes/ts/plnkr.html).

## Using Pipes

A pipe takes in data as input and transforms it to a desired output. We'll illustrate by transforming a component's birthday property into a human-friendly date:

app/hero-birthday1.component.ts

1. import {Component} from 'angular2/core'
2. @Component({
3. selector: 'hero-birthday',
4. template: `<p>The hero's birthday is {{ birthday | date }}</p>`
5. })
6. export class HeroBirthday {
7. birthday = new Date(1988,3,15); // April 15, 1988
8. }

Focus on the component's template.

<p>The hero's birthday is {{ birthday | date }}</p>

Inside the interpolation expression we flow the component's birthday value through the [pipe operator](https://angular.io/docs/ts/latest/guide/template-syntax.html#pipe) ( | ) to the [Date pipe](https://angular.io/docs/ts/latest/api/common/DatePipe-class.html) function on the right. All pipes work this way.

## Built-in pipes

Angular comes with a stock set of pipes such as DatePipe, UpperCasePipe, LowerCasePipe, CurrencyPipe, and PercentPipe. They are all immediately available for use in any template.

Learn more about these and many other built-in pipes in the the [API Reference](https://angular.io/docs/ts/latest/api/); filter for entries that include the word "pipe".

## Parameterizing a Pipe

A pipe may accept any number of optional parameters to fine-tune its output.

We add parameters to a pipe by following the pipe name with a colon ( : ) and then the parameter value (e.g., currency:'EUR'). If our pipe accepts multiple parameters, we separate the values with colons (e.g. slice:1:5)

We'll modify our birthday template to give the date pipe a format parameter. After formatting the hero's April 15th birthday should display as**04/15/88**.

<p>The hero's birthday is {{ birthday | date:"MM/dd/yy" }} </p>

The parameter value can be any valid [template expression](https://angular.io/docs/ts/latest/guide/template-expression.html#template-expressions) such as a string literal or a component property. In other words, we can control the format through a binding the same way we control the birthday value through a binding.

Let's write a second component that binds the pipe's format parameter to the component's format property. Here's the template for that component:

app/hero-birthday2.component.ts (template)

template: `

<p>The hero's birthday is {{ birthday | date:format }}</p>

<button (click)="toggleFormat()">Toggle Format</button>

`

We 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').

app/hero-birthday2.component.ts (class)

1. export class HeroBirthday {
2. birthday = new Date(1988,3,15); // April 15, 1988
3. toggle = true; // start with true == shortDate
4. get format() { return this.toggle ? 'shortDate' : 'fullDate'}
5. toggleFormat() { this.toggle = !this.toggle; }
6. }

As we click the button, the displayed date alternates between "**04/15/1988**" and "**Friday, April 15, 1988**".

Date Format Toggle

Learn more about the DatePipes format options in the [API Docs](https://angular.io/docs/ts/latest/api/core/DatePipe-class.html).

## Chaining pipes

We can chain pipes together in potentially useful combinations. In the following example, we chain the birthday to the DatePipe and on to the UpperCasePipe so we can display the birthday in uppercase. The following birthday displays as **APR 15, 1988**

<p>

The chained hero's birthday is

{{ birthday | date | uppercase}}

</p>

If we pass a parameter to a filter, we have to add parentheses to help the template compiler with the evaluation order. The following example displays **FRIDAY, APRIL 15, 1988**

<p>

The chained hero's birthday is

{{ birthday | date:'fullDate' | uppercase}}

</p>

We can add parentheses to alter the evaluation order or to provide extra clarity:

<p>

The chained hero's birthday is

{{ ( birthday | date:'fullDate' ) | uppercase}}

</p>

## Custom Pipes

We can write our own custom pipes.

Here's a custom pipe named ExponentialStrengthPipe that can boost a hero's powers:

app/exponential-strength.pipe.ts

1. import {Pipe} from 'angular2/core';
2. /\*
3. \* Raise the value exponentially
4. \* Takes an exponent argument that defaults to 1.
5. \* Usage:
6. \* value | exponentialStrength:exponent
7. \* Example:
8. \* {{ 2 | exponentialStrength:10}}
9. \* formats to: 1024
10. \*/
11. @Pipe({name: 'exponentialStrength'})
12. export class ExponentialStrengthPipe {
13. transform(value:number, args:string[]) : any {
14. return Math.pow(value, parseInt(args[0] || '1', 10));
15. }
16. }

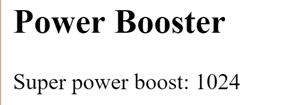
This pipe definition reveals several key points

* A pipe is a class decorated with pipe metadata.
* The pipe class implements a transform method that takes an input value and an optional array of parameter strings and returns the transformed value.
* There will be one item in the parameter array for each parameter passed to the pipe
* We tell Angular that this is a pipe by applying the @Pipe decorator which we import from the core Angular library.
* The @Pipe decorator takes an object with a name property whose value is the pipe name that we'll use within a template expression. It must be a valid JavaScript identifier. Our pipe's name is exponenentialStrength.

Now we need a component to demonstrate our pipe.

app/power-booster.component.ts

1. import {Component} from 'angular2/core';
2. import {ExponentialStrengthPipe} from './exponential-strength.pipe';
3. @Component({
4. selector: 'power-booster',
5. template: `
6. <h2>Power Booster</h2>
7. <p>
8. Super power boost: {{2 | exponentialStrength: 10}}
9. </p>
10. `,
11. pipes: [ExponentialStrengthPipe]
12. })
13. export class PowerBooster { }



Two things to note:

1. We use our custom pipe the same way we use the built-in pipes.
2. We must list our pipe in the @Component decorator's pipes array.

REMEMBER THE PIPES ARRAY!

Angular reports an error if we neglect to list our custom pipe. We didn't list the DatePipe in our previous example because all Angular built-in pipes are pre-registered. Custom pipes must be registered manually.

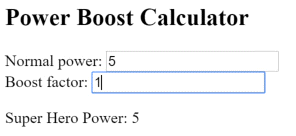
If we try the [live code](https://angular.io/resources/live-examples/pipes/ts/plnkr.html) example, we can probe its behavior by changing the value and the optional exponent in the template.

## Power Boost Calculator (extra-credit)

It's not much fun updating the template to test our custom pipe. We could upgrade the example to a "Power Boost Calculator" that combines our pipe and two-way data binding with ngModel.

/app/power-boost-calculator.component.ts

1. import {Component} from 'angular2/core';
2. import {ExponentialStrengthPipe} from './exponential-strength.pipe';
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. pipes: [ExponentialStrengthPipe]
14. })
15. export class PowerBoostCalculator {
16. power = 5;
17. factor = 1;
18. }



## Stateful Pipes

There are two categories of pipes, stateless and stateful.

Stateless pipes are pure functions that flow input data through without remembering anything or causing detectable side-effects.

Most pipes are stateless. The DatePipe in our first example is a stateless pipe. So is our custom ExponentialStrengthPipe.

Stateful pipes are conceptually similar to classes in object-oriented programming. They can manage the data they transform. A pipe that creates an HTTP request, stores the response and displays the output, is a stateful pipe. Pipes that retrieve or request data should be used cautiously, since working with network data tends to introduce error conditions that are better handled in JavaScript than in a template. We can mitigate this risk by creating a custom pipe for a particular backend and bake-in the essential error-handling.

## The stateful AsyncPipe

The Angular Async pipe is a remarkable example of a stateful pipe. The Async pipe can receive a Promise or Observable as input and subscribe to the input automatically, eventually returning the emitted value(s).

It is stateful because the pipe maintains a subscription to the input and its returned values depend on that subscription.

In the next example, we bind a simple promise to a view with the async pipe.

app/hero-async-message.component.ts

1. import {Component} from 'angular2/core';
2. // Initial view: "Message: "
3. // After 500ms: Message: You are my Hero!"
4. @Component({
5. selector: 'hero-message',
6. template: 'Message: {{delayedMessage | async}}',
7. })
8. export class HeroAsyncMessageComponent {
9. delayedMessage:Promise<string> = new Promise((resolve, reject) => {
10. setTimeout(() => resolve('You are my Hero!'), 500);
11. });
12. }

The Async pipe saves boilerplate in the component code. The component doesn't have to subscribe to the async data source, it doesn't extract the resolved values and expose them for binding, and (in the case of Obsevable stream sources like EventEmitter) the component doesn't have to unsubscribe when it is destroyed (a potent source of memory leaks).

### Implementing a Stateful Pipe

Pipes are stateless by default. We must declare a pipe to be stateful by setting the pure property of the @Pipe decorator to false. This setting tells Angular’s change detection system to check the output of this pipe each cycle, whether its input has changed or not.

Here's how we'll decorate our new stateful FetchJsonPipe that makes an HTTP fetch request and (eventually) displays the data in the server's response:

app/fetch-json.pipe.ts (metadata)

1. @Pipe({
2. name: 'fetch',
3. pure: false
4. })

Immediately below we have the finished pipe. Its input value is an url to an endpoint that returns a JSON file. The pipe makes a one-time async request to the server and eventually receives the JSON response.

app/fetch-json.pipe.ts

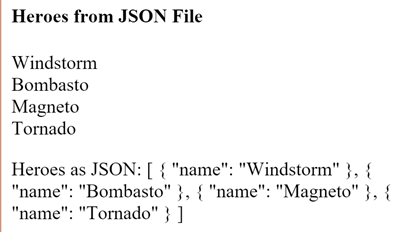
1. import {Pipe} from 'angular2/core';
2. @Pipe({
3. name: 'fetch',
4. pure: false
5. })
6. export class FetchJsonPipe {
7. private fetchedValue:any;
8. private fetchPromise:Promise<any>;
9. transform(value:string, args:string[]):any {
10. if (!this.fetchPromise) {
11. this.fetchPromise = window.fetch(value)
12. .then((result:any) => result.json())
13. .then((json:any) => this.fetchedValue = json);
14. }
15. return this.fetchedValue;
16. }
17. }

Next we demonstrate this pipe in a test component whose template defines two bindings

app/hero-list.component.ts (template)

1. template: `
2. <h4>Heroes from JSON File</h4>
3. <div \*ngFor="#hero of ('heroes.json' | fetch) ">
4. {{hero.name}}
5. </div>
6. <p>Heroes as JSON:
7. {{'heroes.json' | fetch | json}}
8. </p>
9. `,

The component renders like this:



The first binding is straight forward. An ngFor repeater displays the hero names fetched from a json source file. We're piping the literal file name, "heroes.json", through to the custom fetch pipe.

### JsonPipe

The second binding uses more pipe chaining. We take the same fetched results and display the raw hero data in JSON format by piping to the built-in JsonPipe.

DEBUGGING WITH THE JSON PIPE

The [JsonPipe](https://angular.io/docs/ts/latest/api/common/JsonPipe-class.html) is an easy way to diagnosis a mysteriously failing data binding.

Here's the complete component implementation:

app/hero-list.component.ts

1. import {Component} from 'angular2/core';
2. import {FetchJsonPipe} from './fetch-json.pipe';
3. @Component({
4. selector: 'hero-list',
5. template: `
6. <h4>Heroes from JSON File</h4>
7. <div \*ngFor="#hero of ('heroes.json' | fetch) ">
8. {{hero.name}}
9. </div>
10. <p>Heroes as JSON:
11. {{'heroes.json' | fetch | json}}
12. </p>
13. `,
14. pipes: [FetchJsonPipe]
15. })
16. export class HeroListComponent {
17. /\* I've got nothing to do ;-) \*/
18. }
19. ROUTING & NAVIGATION

In most applications, users navigate from one [view](https://angular.io/docs/ts/latest/guide/glossary.html#view) to the next as they perform application tasks.

The browser is a familiar model of application navigation. We enter a URL in the address bar and the browser navigates to a corresponding page. We click links on the page and the browser navigates to a new page. We click the browser's back and forward buttons and the browser navigates backward and forward through the history of pages we've seen.

The Angular "**Component Router**" (AKA "the router") borrows from this model. It can interpret a browser URL as an instruction to navigate to a client-generated view and pass optional parameters along to the supporting view component to help it decide what specific content to present. We can bind the router to links on a page and it will navigate to the appropriate application view when the user clicks a link. We 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.

[Live Example](https://angular.io/resources/live-examples/router/ts/plnkr.html).

## The Basics

Let's begin with a few core concepts of the Component Router. Then we can explore the details through a sequence of examples.

The **Router** is a service that presents a particular Component view for a given URL. When the browser's URL changes, the router looks for a corresponding **RouteDefinition** from which it can determine the Component to display.

A new router has no route definitions. We have to configure it. The preferred way to configure the router is with a **@RouteConfig** [decorator](https://angular.io/docs/ts/latest/guide/glossary.html#decorator)applied to a host component.

In this example, we configure the top-level AppComponent with three route definitions

app.component.ts (excerpt)

@Component({ ... })

@RouteConfig([

{path:'/crisis-center', name: 'CrisisCenter', component: CrisisListComponent},

{path:'/heroes', name: 'Heroes', component: HeroListComponent},

{path:'/hero/:id', name: 'HeroDetail', component: HeroDetailComponent}

])

export class AppComponent { }

There are several flavors of RouteDefinition. The most common by far is the named **Route** which maps a URL path to a Component

The name field is the name of the Route. The name **must** be spelled in **PascalCase**.

The :id in the third route is a token for a route parameter. In a URL such as /hero/42, "42" is the value of the id parameter. The correspondingHeroDetailComponent will use that value to find and present the hero whose id is 42. We'll learn more about route parameters later in this chapter.

Now we know how the router gets its configuration. When the browser URL for this application becomes /heroes, the router matches that URL to the RouteDefintion named Heroes and displays the HeroListComponent in a **RouterOutlet** that we've placed in the host view's HTML.

<!-- Routed views go here -->

<router-outlet></router-outlet>

Now we have routes configured and a place to render them, but how do we navigate? The URL could arrive directly from the browser address bar. But most of the time we navigate as a result of some user action such as the click of an anchor tag.

We add a **RouterLink** directive to the anchor tag and bind it to a template expression that returns an array of route link parameters (the **link parameters array**). The router ultimately resolves that array into a URL and a component view.

We see such bindings in the following AppComponent template:

template: `

<h1>Component Router</h1>

<a [routerLink]="['CrisisCenter']">Crisis Center</a>

<a [routerLink]="['Heroes']">Heroes</a>

<router-outlet></router-outlet>

`,

We're adding two anchor tags with RouterLink directives. We bind each RouterLink to an array containing the string name of a route definition. 'CrisisCenter' and 'Heroes' are the names of the Routes we configured above.

We'll learn to write more complex link expressions — and why they are arrays — [later](https://angular.io/docs/ts/latest/guide/router.html#link-parameter-array) in the chapter.

### Let's summarize

The @RouterConfig configuration tied the AppComponent to a router configured with routes. The component has a RouterOutlet where it can display views produced by the router. It has RouterLinks that users can click to navigate via the router.

The AppComponent has become a **Routing Component**, a component that can route.

Here are the key Component Router terms and their meanings:

|  |  |
| --- | --- |
| Router Part | Meaning |
| Router | Displays the application component for the active URL. Manages navigation from one component to the next. |
| @RouteConfig | Configures a router with RouteDefinitions, each mapping a URL path to a Component. |
| RouteDefinition | Defines how the router should navigate to a Component based on a URL pattern. |
| Route | The most common form of RouteDefinition consisting of a path, a route name, and a component type. |
| RouterOutlet | The directive (<router-outlet>) that marks where the router should display a view. |
| RouterLink | The directive for binding a clickable HTML element to a route. Clicking an anchor tag with a routerLink directive that is bound to a *Link Parameters Array* triggers a navigation. |
| *Link Parameters Array* | An array that the router inteprets into a routing instruction. We can bind a RouterLink to that array or pass the array as an argument to the Router.navigate method. |
| *Routing Component* | An Angular component with an attached router. |

We'll learn many more details in this chapter which covers

* [configuring a router](https://angular.io/docs/ts/latest/guide/router.html#route-config)
* the [link parameter arrays](https://angular.io/docs/ts/latest/guide/router.html#link-parameters-array) that propel router navigation
* navigating when the user clicks a data-bound [RouterLink](https://angular.io/docs/ts/latest/guide/router.html#router-link)
* navigating under [program control](https://angular.io/docs/ts/latest/guide/router.html#navigate)
* passing information in [route parameters](https://angular.io/docs/ts/latest/guide/router.html#route-parameter)
* creating a [child router](https://angular.io/docs/ts/latest/guide/router.html#child-router) with its own routes
* setting a [default route](https://angular.io/docs/ts/latest/guide/router.html#default)
* pausing, confirming and/or canceling a navigation with the the routerCanDeactivate [router lifecycle hook](https://angular.io/docs/ts/latest/guide/router.html#lifecycle-hooks)

We proceed in phases marked by milestones. Our first milestone is the ability to navigate between two placeholder views. At our last milestone, we'll have a modular, multi-view design with child routes.

We assume that you're already comfortable with the basic Angular 2 tools and concepts we introduced in the [QuickStart](https://angular.io/docs/ts/latest/quickstart.html) and the [Tour of Heroes](https://angular.io/docs/ts/latest/tutorial/) tutorial.

While we make incremental progress on a sample application, this chapter is not a tutorial. We discuss code and design decisions pertinent to routing and application design. We gloss over everything in between.

The full source is available in the [live example](https://angular.io/resources/live-examples/router/ts/plnkr.html).

## The Sample Application

We have an application in mind as we move from milestone to milestone. Our client is the Hero Employment Agency. Heroes need work and The Agency finds Crises for them to solve.

The application has two main feature areas:

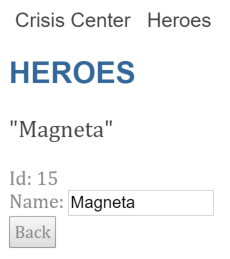
1. A Crisis Center where we maintain the list of crises for assignment to heroes.
2. A Heroes area where we maintain the list of heroes employed by The Agency.

Run the [live example](https://angular.io/resources/live-examples/router/ts/plnkr.html). It opens in the Crisis Center. We'll come back to that.

Click the Heroes link. We're presented with a list of Heroes.



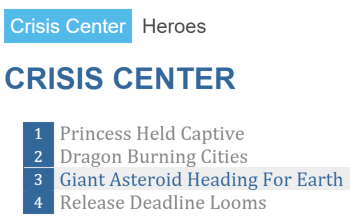
We select one and the applications takes us to a hero editing screen.



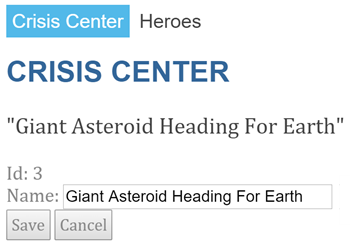
Our changes take affect immediately. We click the "Back" button and the app returns us to the Heroes list.

We could have clicked the browser's back button instead. That would have returned us to the Heroes List as well. Angular app navigation updates the browser history as normal web navigation does.

Now click the Crisis Center link. We go to the Crisis Center and its list of ongoing crises.



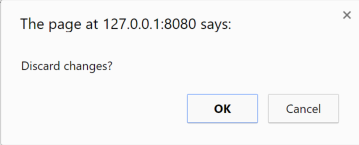
We select one and the applications takes us to a crisis editing screen.



This is a bit different from the Hero Detail. Hero Detail saves the changes as we type. In Crisis Detail our changes are temporary until we either save or discard them with by pressing the "Save" or "Cancel" buttons. Both buttons navigate back to the Crisis Center and its list of crises.

Suppose we click a crisis, make a change, but **do not click either button**. Maye we click the browser back button instead. Maybe we click the "Heroes" link.

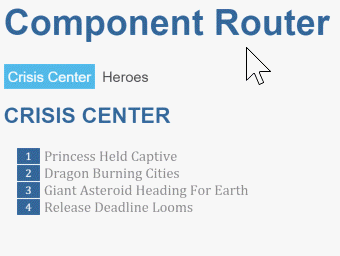
Do either. Up pops a dialog box.



We can say "OK" and lose our changes or click "Cancel" and continue editing.

The router supports a routerCanDeactivate lifecycle hook that gives us a chance to clean-up or ask the user's permission before navigating away from the current view.

Here we see an entire user session that touches all of these features.



Here's a diagram of all application routing options:

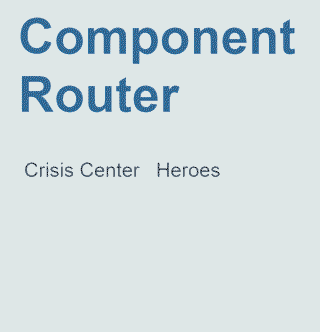


This app illustrates the router features we'll cover in this chapter

* 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 routerCanDeactivate lifecycle hook (ask permission to discard unsaved changes)

## Milestone #1: Getting Started with the Router

Let's begin with a simple version of the app that navigates between two empty views.



### Load the Component Router library

The Component Router is not part of the Angular 2 core. It is in its own library. The router is an optional service and you might prefer a different router someday.

The Component Router library is part of the Angular npm bundle. We make it available by loading its script in our index.html, right after the Angular core script.

<script src="node\_modules/angular2/bundles/router.dev.js"></script>

### Set the <base href>

The Component Router uses the browser's [history.pushState](https://developer.mozilla.org/en-US/docs/Web/API/History_API#Adding_and_modifying_history_entries) for navigation. Thanks to pushState, we can make our in-app URL paths look the way we want them to look, e.g. localhost:3000/crisis-center. Our in-app URLs can be indistinguishable from server URLs.

Modern HTML 5 browsers were the first to support pushState which is why many people refer to these URLs as "HTML 5 style" URLs.

We must **add a**[**<base href> element**](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/base)**tag** to the index.html to make this work. The href value specifies the base URL to use for all relativeURLs within a document including links to 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 our application, set the href valueexactly as shown here.

<base href="/">

HTML 5 style navigation is the Component Router default. Learn why "HTML 5" style is preferred, how to adjust its behavior, and how to switch to the older hash (#) style if necessary in the [Browser URL Styles](https://angular.io/docs/ts/latest/guide/router.html#browser-url-styles) appendix below.

#### Live example note

We have to be get tricky when we run the live example because the host service sets the application base address dynamically. That's why we replace the<base href...> with a script that writes a <base> tag on the fly to match.

<script>document.write('<base href="' + document.location + '" />');</script>

We should only need this trick for the live example, not production code.

### Booting with the router service providers

Our app launches from the boot.ts file in the ~/app folder so let's start there. It's short and all of it is relevant to routing.

boot.ts

import {AppComponent} from './app.component';

import {bootstrap} from 'angular2/platform/browser';

import {ROUTER\_PROVIDERS} from 'angular2/router';

bootstrap(AppComponent, [

ROUTER\_PROVIDERS,

]);

We import our root AppComponent and Angular's bootstrap function as expected.

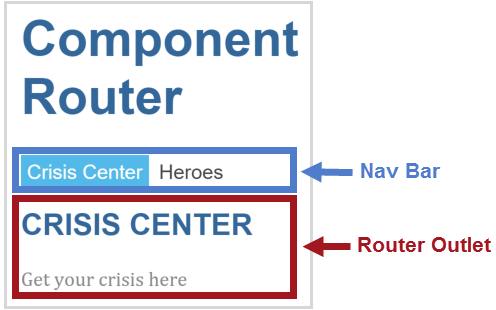
We also import ROUTER\_PROVIDERS from the router library. The router is a service implemented by a collection of Dependency Injectionproviders, most of which are identified in the ROUTER\_PROVIDERS array.

We're booting Angular with AppComponent as our app's root component and registering providers, as we often do, in the providers array in the second parameter of the bootstrap function. Providing the router providers at the root makes the Component Router available everywhere in our application.

Learn about providers, the provide function, and injected services in the [Dependency Injection chapter](https://angular.io/docs/ts/latest/guide/dependency-injection.html).

### The AppComponent shell

The root AppComponent is the application shell. It has title at the top, a navigation bar with two links, and a Router Outlet at the bottom where the router swaps views on and off the page. Here's what we mean:



The corresponding component template looks like this:

template: `

<h1>Component Router</h1>

<a [routerLink]="['CrisisCenter']">Crisis Center</a>

<a [routerLink]="['Heroes']">Heroes</a>

<router-outlet></router-outlet>

`,

### RouterOutlet

RouterOutlet is a component from the router library. The router displays views within the bounds of the <router-outlet> tags.

A template may hold exactly one **unnamed** <router-outlet>.

### RouterLink binding

Above the outlet, within the anchor tags, we see [Property Bindings](https://angular.io/docs/ts/latest/guide/template-syntax.html#property-binding) to the RouterLink directive that look like [routerLink]="[...]". We imported RouterLink from the router library.

The template expression to the right of the equals (=) returns a link parameters array.

A link parameters array holds the ingredients for router navigation:

* the name of the route that prescribes the destination component and a path for the URL
* the optional route and query parameters that go into the route URL

The arrays in this example each have a single string parameter, the name of a Route that we'll configure for this application with@RouteConfig(). We don't need to set route parameters yet.

Learn more about the link parameters array in the [appendix below](https://angular.io/docs/ts/latest/guide/router.html#link-parameters-array).

### @RouteConfig()

A router holds a list of route definitions. The list is empty for a new router. We must configure it.

A router also needs a **Host Component**, a point of origin for its navigations.

It's natural to combine the creation of a new router, its configuration, and its assignment to a host component in a single step. That's the purpose of the @RouteConfig decorator which we put to good use here:

@Component({ ... })

@RouteConfig([

{path:'/crisis-center', name: 'CrisisCenter', component: CrisisListComponent},

{path:'/heroes', name: 'Heroes', component: HeroListComponent}

])

export class AppComponent { }

The @RouteConfig decorator creates a new router. We applied the decorator to AppComponent which makes that the router's host component. The argument to @RouteConfig() is an array of **Route Definitions**.

We're supplying two definitions:

{path:'/crisis-center', name: 'CrisisCenter', component: CrisisListComponent},

{path:'/heroes', name: 'Heroes', component: HeroListComponent}

Each definition translates to a [Route](https://angular.io/docs/ts/latest/api/router/Route-class.html) which has a

* path - the URL path segment for this route
* name - the name of the route
* component - the Component associated with this route.

The router draws upon its registry of route definition when

1. the browser URL changes
2. we tell the router to go to a named route

In plain English, we might say of the first route:

1. When the browser's location URL changes to***match the path***segment*/crisis-center*, create or retrieve an instance of the*CrisisCenterComponent*and display its view.
2. When the application requests navigation to a route***named****CrisisCenter*, compose a browser URL with the path segment*/crisis-center*, update the browser's address location and history, create or retrieve an instance of the*CrisisCenterComponent*, and display that component's view.

### "Getting Started" wrap-up

We've got a very basic, navigating app, one that can switch between two views when the user clicks a link.

We've learned how to

* load the router library
* add a nav bar to the shell template with anchor tags and routerLink directives
* added a router-outlet to the shell template where views will be displayed
* configure the router with @RouterConfig
* set the router to compose "HTML 5" browser URLs.

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.

Our starter app's structure looks like this:

router-sample

node\_modules

app

app.component.ts

boot.ts

crisis-list.component.ts

hero-list.component.ts

index.html

styles.css

tsconfig.json

package.json

Here are the application-specific files

app.component.tsboot.tshero-list.component.tscrisis-list.component.ts

1. import {Component} from 'angular2/core';
2. import {RouteConfig, ROUTER\_DIRECTIVES} from 'angular2/router';
3. import {CrisisListComponent} from './crisis-list.component';
4. import {HeroListComponent} from './hero-list.component';
5. @Component({
6. selector: 'my-app',
7. template: `
8. <h1>Component Router</h1>
9. <a [routerLink]="['CrisisCenter']">Crisis Center</a>
10. <a [routerLink]="['Heroes']">Heroes</a>
11. <router-outlet></router-outlet>
12. `,
13. directives: [ROUTER\_DIRECTIVES]
14. })
15. @RouteConfig([
16. {path:'/crisis-center', name: 'CrisisCenter', component: CrisisListComponent},
17. {path:'/heroes', name: 'Heroes', component: HeroListComponent}
18. ])
19. export class AppComponent { }

## Milestone #2: The Heroes Feature

We've seen how to navigate using the RouterLink directive.

Now we'll learn some new tricks such as how to

* organize our app into feature areas
* navigate imperatively from one component to another
* pass information along in route parameters (RouteParams)

To demonstrate, we'll build out the Heroes feature.

### The Heroes "feature area"

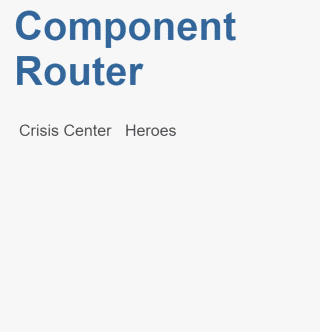
A typical application has multiple feature areas, each an island of functionality with its own workflow(s), dedicated to a particular business purpose.

We could continue to add files to the app/ folder. That's unrealistic and ultimately not maintainable. We think it's better to put each feature area in its own folder.

Our first step is to **create a separate app/heroes/ folder**. and add Hero Management feature files there.

We won't be creative about it. Our example is pretty much a copy of the code and capabilities in the "[Tutorial: Tour of Heroes](https://angular.io/docs/ts/latest/tutorial/index.html)".

Here's how the user will experience this version of the app



### Add Heroes functionality

We delete the placeholder hero-list.component.ts that's in the app/ folder.

We create a new hero-list.component.ts in the app/heroes/ folder and copy over the contents of the final heroes.component.tsfrom the tutorial. We also copy the hero-detail.component.ts and the hero.service.ts files into the heroes/ folder.

When were done organizing, we have three Hero Management files:

app/heroes

hero-detail.component.ts

hero-list.component.ts

hero.service.ts

We'll provide the HeroService during bootstrapping so that is available everywhere in the app (see boot.ts) .

Now it's time for some surgery to bring these files and the rest of the app into alignment with our application router.

### New route definition with route parameter

The new Heroes feature has two interacting components, the list and the detail. The list view is self-sufficient; we navigate to it, it gets a list of heroes and displays them. It doesn't need any outside information.

The detail view is different. It displays a particular hero. It can't know which hero on its own. That information must come from outside.

In our example, when the user selects a hero from the list, we navigate to the detail view to show that hero. We'll tell the detail view which hero to display by including the selected hero's id in the route URL.

With that plan in mind, we return to the app.component.ts to make changes to the router's configuration

First, we import the two components from their new locations in the app/heroes/ folder:

import {HeroListComponent} from './heroes/hero-list.component';

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

Then we update the @RouteConfig route definitions :

@Component({ ... })

@RouteConfig([

{path:'/crisis-center', name: 'CrisisCenter', component: CrisisListComponent},

{path:'/heroes', name: 'Heroes', component: HeroListComponent},

{path:'/hero/:id', name: 'HeroDetail', component: HeroDetailComponent}

])

export class AppComponent { }

The CrisisCenter and Heroes definitions didn't change. While we moved hero-list.component.ts to a new location in theapp/heroes/ folder, that only affects the import statement; it doesn't affect its route definition.

We added a new route definition for the HeroDetailComponent — and this definition has a twist.

{path:'/hero/:id', name: 'HeroDetail', component: HeroDetailComponent}

Notice the :id token in the the path. That creates a slot in the path for a **Route Parameter**. In this case, we're expecting the router to insert the id of a hero into that slot.

If we tell the router to navigate to the detail component and display "Magenta", we expect hero id (15) 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 "Magenta" detail view.

### Navigate to the detail imperatively

We don't navigate to the detail component by clicking a link. We won't be adding a new anchor tag to the shell navigation bar.

Instead, we'll detect when the user selects a hero from the list and command the router to present the hero detail view of the selected hero.

We'll adjust the HeroListComponent to implement these tasks, beginning with its constructor which acquires the router service and theHeroService by dependency injection:

constructor(

private \_router: Router,

private \_service: HeroService) { }

We make a few changes to the template:

template: `

<h2>HEROES</h2>

<ul>

<li \*ngFor="#hero of heroes"

(click)="onSelect(hero)">

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

</li>

</ul>

`

The template defines an \*ngFor repeater such as [we've seen before](https://angular.io/docs/ts/latest/guide/displaying-data.html#ngFor). There's a (click) [EventBinding](https://angular.io/docs/ts/latest/guide/template-syntax.html#event-binding) to the component's onSelectmethod which we implement as follows:

onSelect(hero: Hero) {

this.\_router.navigate( ['HeroDetail', { id: hero.id }] );

}

It calls the router's **navigate** method with a **Link Parameters Array**. This array is similar to the link parameters array we met [earlier](https://angular.io/docs/ts/latest/guide/router.html#shell-template) in an anchor tag while binding to the RouterLink directive. This time we see it in code rather than in HTML.

### Setting the route parameter

We're navigating to the HeroDetailComponent where we expect to see the details of the selected hero. We'll need two pieces of information: the destination and the hero's id.

Accordingly, the link parameters array has two items: the **name** of the destination route and a **route parameters object** that specifies the idof the selected hero.

['HeroDetail', { id: hero.id }] // {id: 15}

The router composes the appropriate two-part destination URL from this array:

localhost:3000/hero/15

### Getting the route parameter

How does the target HeroDetailComponent learn about that id? Certainly not by analyzing the URL! That's the router's job.

The router extracts the route parameter (id:15) from the URL and supplies it to the HeroDetailComponent via the **RouteParams** service.

As usual, we write a constructor that asks Angular to inject that service among the other services that the component require and reference them as private variables.

constructor(

private \_router:Router,

private \_routeParams:RouteParams,

private \_service:HeroService){}

Later, in the ngOnInit method, we ask the RouteParams service for the id parameter by name and tell the HeroService to fetch the hero with that id.

ngOnInit() {

let id = this.\_routeParams.get('id');

this.\_service.getHero(id).then(hero => this.hero = hero);

}

Angular calls the ngOnInit method shortly after creating an instance of the HeroDetailComponent.

We put the data access logic in the ngOnInit method rather than inside the constructor to improve the component's testability. We explore this point in greater detail in the [OnInit appendix](https://angular.io/docs/ts/latest/guide/router.html#onInit) below.

### Navigating back to the list component

The HeroDetailComponent has a "Back" button wired to its gotoHeroes method that navigates imperatively back to theHeroListComponent.

The router navigate method takes the same one-item link parameters array that we wrote for the [routerLink] directive binding. It holds the **name of the HeroListComponent route**:

gotoHeroes() {

// <a [routerLink]="['Heroes']">Heroes</a>

this.\_router.navigate(['Heroes']);

}

### Heroes App Wrap-up

We've reached the second milestone in our router education.

We've learned how to

* organize our app into feature areas
* navigate imperatively from one component to another
* pass information along in route parameters (RouteParams)

After these changes, the folder structure looks like this:

router-sample

node\_modules

app

heroes

hero-detail.component.ts

hero-list.component.ts

hero.service.ts

app.component.ts

boot.ts

crisis-list.component.ts

index.html

styles.css

tsconfig.json

package.json

### The Heroes App code

Here are the relevant files for this version of the sample application.

app.component.tsboot.tshero-list.component.tshero-detail.component.tshero.service.ts

1. import {Component} from 'angular2/core';
2. import {RouteConfig, ROUTER\_DIRECTIVES} from 'angular2/router';
3. import {CrisisListComponent} from './crisis-list.component';
4. import {HeroListComponent} from './heroes/hero-list.component';
5. import {HeroDetailComponent} from './heroes/hero-detail.component';
6. @Component({
7. selector: 'my-app',
8. template: `
9. <h1>Component Router</h1>
10. <a [routerLink]="['CrisisCenter']">Crisis Center</a>
11. <a [routerLink]="['Heroes']">Heroes</a>
12. <router-outlet></router-outlet>
13. `,
14. directives: [ROUTER\_DIRECTIVES]
15. })
16. @RouteConfig([
17. {path:'/crisis-center', name: 'CrisisCenter', component: CrisisListComponent},
18. {path:'/heroes', name: 'Heroes', component: HeroListComponent},
19. {path:'/hero/:id', name: 'HeroDetail', component: HeroDetailComponent}
20. ])
21. export class AppComponent { }

## Milestone #3: The Crisis Center

The Crisis Center is a fake view at the moment. Time to make it useful.

The new Crisis Center begins as a virtual copy of the Heroes feature. We create a new app/crisis-center folder, copy the Hero files, and change every mention of "hero" to "crisis".

A Crisis has an id and name, just like a Hero The new CrisisListComponent displays lists of crises. When the user selects a crisis, the app navigates to the CrisisDetailComponent for display and editing of the crisis name.

Voilà, instant feature module!

There's no point to this exercise unless we can learn something. We do have new ideas and techniques in mind:

* The application should navigate to the Crisis Center by default.
* The user should be able to cancel unwanted changes.
* The router should prevent navigation away from the detail view while there are pending changes.

There are also a few lingering annoyances in the Heroes implementation that we can cure in the Crisis Center.

* We currently register every route of every view at the highest level of the application. If we expand the Crisis Center with a 100 new views, we'll make 100 changes to the AppComponent route configuration. If we rename a Crisis Center component or change a route definition, we'll be changing the AppComponent too.
* If we followed Heroes lead, we'd be adding the CrisisService to the providers in boot.ts. Now both HeroService andCrisisService would be available everywhere although they're only needed in their respective feature modules. That stinks.

Changes to a sub-module such as Crisis Center shouldn't provoke changes to the AppComponent or boot.ts. We need to [separate our concerns](https://blog.8thlight.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html).

We'll fix all of these problems and add the new routing features to Crisis Center.

The most significant fix is the introduction of a **child Routing Component** and its **child router**

We'll leave Heroes in its less-than-perfect state to serve as a contrast with what we hope is a superior Crisis Center.

### A free-standing Crisis Center Feature Module

The Crisis Center is one of two application workflows. Users navigate between them depending on whether they are managing crises or heroes.

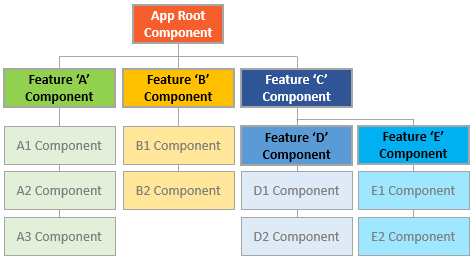
The CrisisCenter and Heroes components are children of the root AppComponent.

Unfortunately, they and their related files are physically commingled in the same folder with the AppComponent. We'd prefer to separate them in their own feature areas so they can operate and evolve independently. Someday we might re-use one or the other in a different application. Someday we might load one of them dynamically only when the user chose to enter its workflow.

Some might call it [yagni](http://martinfowler.com/bliki/Yagni.html) to even think about such things. But we're right to be nervous about the way Heroes and Crisis Center artifacts are bubbling up to the root AppComponent and blending with each other. That's a [code smell](http://martinfowler.com/bliki/CodeSmell.html).

Isolating feature area modules from each other looks good to us.

It's looking good as a general pattern for Angular applications.



* each feature area in its own module folder
* each area with its own root component
* each area root component with its own router-outlet and child routes
* area routes rarely (if ever) cross

We'll make the Crisis Center stand on its own and leave the Heroes as it is so we can compare the effort, results, and consequences. Then each of us can decide which path to prefer (as if we didn't already know).

### Child Routing Component

We create a new app/crisis-center folder and add crisis-center-component.ts to it with the following contents:

crisis-center/crisis-center.component.ts (minus imports)

1. @Component({
2. template: `
3. <h2>CRISIS CENTER</h2>
4. <router-outlet></router-outlet>
5. `,
6. directives: [RouterOutlet],
7. providers: [CrisisService]
8. })
9. @RouteConfig([
10. {path:'/', name: 'CrisisCenter', component: CrisisListComponent, useAsDefault: true},
11. {path:'/:id', name: 'CrisisDetail', component: CrisisDetailComponent},
12. {path:'/list/:id', name: 'CrisisList', component: CrisisListComponent}
13. ])
14. export class CrisisCenterComponent { }

The CrisisCenterComponent parallels the AppComponent.

The CrisisCenterComponent is the root of the Crisis Center area just as AppComponent is the root of the entire application.

This CrisisCenterComponent is a shell for crisis management just as the AppComponent is a shell to manage the high-level workflow.

AppComponent has a @RouteConfig decorator that defines the top-level routes. CrisisCenterComponent has a @RouteConfigdecorator that defines Crisis Center child routes.

The CrisisCenterComponent template is dead simple — simpler even than the AppComponent template. It has no content, no links, just a<router-outlet> for the Crisis Center child views.

It has no selector either. It doesn't need one. We don't embed this component in a parent template. We navigate to it from the outside, via a parent router (more on that soon).

### Service isolation

We add the CrisisService to the component's providers array instead of registering it with the bootstrap function in boot.ts.

providers: [CrisisService]

This step limits the scope of that service to the Crisis Center component and its sub-component tree. No component outside of the Crisis Center needs access to the CrisisService. By restricting its scope, we feel confident that we can evolve it independently without fear of breaking unrelated application modules — modules that shouldn't have access to it anyway.

### Child Route Configuration

The CrisisCenterComponent is a Routing Component like the AppComponent.

The @RouteConfig decorator that adorns the CrisisCenterComponent class defines routes in the same way that we did earlier.

app/crisis-center/crisis-center.component.ts (routes only)

@RouteConfig([

{path:'/', name: 'CrisisCenter', component: CrisisListComponent, useAsDefault: true},

{path:'/:id', name: 'CrisisDetail', component: CrisisDetailComponent},

{path:'/list/:id', name: 'CrisisList', component: CrisisListComponent}

])

There are three Crisis Center routes, two of them with an id parameter. They refer to components we haven't talked about yet but whose purpose we can guess by their names.

We cannot tell by looking at the CrisisCenterComponent that it is a child component of an application. We can't tell that its routes are child routes.

That's intentional. The Crisis Center shouldn't know that it is the child of anything. It might be the top level component of its own application. It might be repurposed in a different application. The Crisis Center itself is indifferent to these possibilities.

We make it a child component of our application by reconfiguring the routes of the top level AppComponent.

### Parent Route Configuration

Here is is the revised route configuration for the parent AppComponent:

app/app.component.ts (routes only)

1. @RouteConfig([
2. { // Crisis Center child route
3. path: '/crisis-center/...',
4. name: 'CrisisCenter',
5. component: CrisisCenterComponent,
6. useAsDefault: true
7. },
8. {path: '/heroes', name: 'Heroes', component: HeroListComponent},
9. {path: '/hero/:id', name: 'HeroDetail', component: HeroDetailComponent},
10. ])

The second and third Hero routes haven't changed. The first Crisis Center route has changed — signficantly — and we've formatted it to draw attention to the differences:

{ // Crisis Center child route

path: '/crisis-center/...',

name: 'CrisisCenter',

component: CrisisCenterComponent,

useAsDefault: true

},

Notice that the **path ends with a slash and three trailing periods (/...)**.

That means this is an incomplete route (AKA a **non-terminal route**). The finished route will be some combination of the parent /crisis-center/ route and a route from the **child router** that belongs to the designated component.

All is well. The parent route's designated component is the CrisisCenterComponent which is a Routing Component with its own router and routes.

### Default route

The other important change is the addition of the useAsDefault property. Its value is true which makes this route the default route. When the application launches, in the absence of any routing information from the browser's URL, the router will default to the Crisis Center. That's our plan.

### Routing to the Child

We've set the top level default route to go to the CrisisCenterComponent. The final route will be a combination of /crisis-center/ and one of the child CrisisCenterComponent router's three routes. Which one?

It could be any of the three. In the absence of additional information, the router can't decide and must throw an error.

We've tried the sample application and it didn't fail. We must have done something.

Scroll to the end of the child CrisisCenterComponents first route.

app/crisis-center/crisis-center.component.ts (default route)

{path:'/', name: 'CrisisCenter', component: CrisisListComponent, useAsDefault: true},

There is useAsDefault: true again. That tells the router to compose the final URL using the path from the default child route. Concatenate the base URL with /crisis-center/ and /, remove extraneous slashes, and we get:

localhost:3000//crisis-center/

### 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, we might have to accumulate the users changes. We might have to validate across fields. We might have to validate on the server. We 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 we do about unapproved, unsaved changes when the user navigates away? We'd like to pause and let the user decide what to do. Perhaps we'll cancel the navigation, stay put, and make more changes.

We need the router's cooperation to pull this off. We need router lifecycle hooks.

### Router Lifecycle Hooks

Angular components have [lifecycle hooks](https://angular.io/docs/ts/latest/guide/lifecycle-hooks.html). For example, Angular calls the hook methods of the [OnInit](https://angular.io/docs/ts/latest/api/core/OnInit-interface.html) and [OnDestroy](https://angular.io/docs/ts/latest/guide/(../api/core/OnDestroy-interface.html) interfaces when it creates and destroys components.

The router calls similar hook methods, [routerCanActivate](https://angular.io/docs/ts/latest/api/router/CanActivate-var.html) and [routerCanDeactivate](https://angular.io/docs/ts/latest/api/router/CanDeactivate-interface.html), before it navigates to a component or away from a component.

If a *can...* method returns true, the navigation proceeds. If it returns false, the router cancels the navigation and stays on the current view.

The router lifecycle hooks supplement the component lifecycle hooks. We still need the component hooks but the router hooks do what the component hooks cannot.

For example, the component hooks can't stop component creation or destruction. Because they are synchronous, they can't pause view navigation to wait for an asynchronous process to finish.

Imagine we have unsaved changes. The user starts to navigate away. We shouldn't lose the user's changes; that would be a terrible experience. So we try to save those changes to the server.

If the save fails for any reason (perhaps the data are ruled invalid), what do we do?

If we let the user move to the next screen, we have lost the context of the error. We can't block while waiting for the server — that's not possible in a browser.

We need to stop the navigation while we wait, asynchronously, for the server to return with its answer.

The router hook methods can pause asynchronously, return promises, and cancel navigation if necessary.

### Cancel and Save

Our sample application doesn't talk to a server. Fortunately, we 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. We update the entity when the user presses the Save button. We discard the changes if the user presses he Cancelbutton.

Both buttons navigate back to the crisis list after save or cancel.

crisis-detail.component.ts (excerpt)

export class CrisisDetailComponent implements OnInit, CanDeactivate {

public crisis: Crisis;

public editName: string;

cancel() {

this.editName = this.crisis.name;

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?

We'll do neither. Instead we'll ask the user to make that choice explicitly in a confirmation dialog box that waits asynchronously for the user's answer.

We could wait for the user's answer with synchronous, blocking code. Our 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 dialog service returns a [promise](http://www.html5rocks.com/en/tutorials/es6/promises/). The promise resolves when the user eventually decides to discard changes and navigate away (true) or keep the pending changes and stay in the crisis editor (false).

We execute the dialog inside the router's routerCanDeactivate lifecycle hook method.

crisis-detail.component.ts (excerpt)

1. import {CanDeactivate, ComponentInstruction} from 'angular2/router';
2. import {DialogService} from '../dialog.service';
3. export class CrisisDetailComponent implements OnInit, CanDeactivate {
4. public crisis: Crisis;
5. public editName: string;
6. }

Notice that the routerCanDeactivate method can return synchronously; it returns true immediately if there are no pending changes. But it can also return a promise and the router will wait for that promise to resolve before navigating away or staying put.

**Two critical points**

1. The router hook is optional. We don't inherit from a base class. We simply implement the method or not.
2. We rely on the router to call the hook. We don't worry about all the ways that the user could navigate away. That's the router's job. We simply write this method and let the router take it from there.

## Wrap Up

As we end our chapter, we take a parting look at the entire application.

We can always try the [live example](https://angular.io/docs/ts/latest/resources/live-examples/router/ts/plnkr.html) and download the source code from there.

Our final project folder structure looks like this:

router-sample

node\_modules

app

crisis-center/...

heroes/...

app.component.ts

boot.ts

dialog.service.ts

index.html

package.json

styles.css

tsconfig.json

The top level application files are

app.component.tsboot.tsdialog.service.tsindex.htmlstyles.css

1. import {Component} from 'angular2/core';
2. import {RouteConfig, ROUTER\_DIRECTIVES} from 'angular2/router';
3. import {CrisisCenterComponent} from './crisis-center/crisis-center.component';
4. import {HeroListComponent} from './heroes/hero-list.component';
5. import {HeroDetailComponent} from './heroes/hero-detail.component';
6. @Component({
7. selector: 'my-app',
8. template: `
9. <h1 class="title">Component Router</h1>
10. <a [routerLink]="['CrisisCenter']">Crisis Center</a>
11. <a [routerLink]="['Heroes']">Heroes</a>
12. <router-outlet></router-outlet>
13. `,
14. directives: [ROUTER\_DIRECTIVES]
15. })
16. @RouteConfig([
17. { // Crisis Center child route
18. path: '/crisis-center/...',
19. name: 'CrisisCenter',
20. component: CrisisCenterComponent,
21. useAsDefault: true
22. },
23. {path: '/heroes', name: 'Heroes', component: HeroListComponent},
24. {path: '/hero/:id', name: 'HeroDetail', component: HeroDetailComponent},
25. {path: '/disaster', name: 'Asteroid', redirectTo: ['CrisisCenter', 'CrisisDetail', {id:3}]}
26. ])
27. export class AppComponent { }

### Crisis Center

The Crisis Center feature area within the crisis-center folder follows:

app

crisis-center

crisis-center.component.ts

crisis-detail.component.ts

crisis-list.component.ts

crisis.service.ts

crisis-center.component.tscrisis-list.component.tscrisis-detail.component.tscrisis.service.ts

1. import {Component} from 'angular2/core';
2. import {RouteConfig, RouterOutlet} from 'angular2/router';
3. import {CrisisListComponent} from './crisis-list.component';
4. import {CrisisDetailComponent} from './crisis-detail.component';
5. import {CrisisService} from './crisis.service';
6. @Component({
7. template: `
8. <h2>CRISIS CENTER</h2>
9. <router-outlet></router-outlet>
10. `,
11. directives: [RouterOutlet],
12. providers: [CrisisService]
13. })
14. @RouteConfig([
15. {path:'/', name: 'CrisisCenter', component: CrisisListComponent, useAsDefault: true},
16. {path:'/:id', name: 'CrisisDetail', component: CrisisDetailComponent},
17. {path:'/list/:id', name: 'CrisisList', component: CrisisListComponent}
18. ])
19. export class CrisisCenterComponent { }

### Heroes

The Heroes feature area within the heroes folder is next:

app

heroes

hero-detail.component.ts

hero-list.component.ts

hero.service.ts

hero-list.component.tshero-detail.component.tshero.service.ts

1. // TODO SOMEDAY: Feature Componetized like HeroCenter
2. import {Component, OnInit} from 'angular2/core';
3. import {Hero, HeroService} from './hero.service';
4. import {Router} from 'angular2/router';
5. @Component({
6. template: `
7. <h2>HEROES</h2>
8. <ul>
9. <li \*ngFor="#hero of heroes"
10. (click)="onSelect(hero)">
11. <span class="badge">{{hero.id}}</span> {{hero.name}}
12. </li>
13. </ul>
14. `
15. })
16. export class HeroListComponent implements OnInit {
17. public heroes: Hero[];
18. public selectedHero: Hero;
19. constructor(
20. private \_router: Router,
21. private \_service: HeroService) { }
22. ngOnInit() {
23. this.\_service.getHeroes().then(heroes => this.heroes = heroes)
24. }
25. onSelect(hero: Hero) {
26. this.\_router.navigate( ['HeroDetail', { id: hero.id }] );
27. }
28. }

## Appendices

The balance of this chapter is a set of appendices that elaborate some of the points we covered quickly above.

The appendix material isn't essential. Continued reading is for the curious.

## Link Parameters Array

We've mentioned the Link Parameters Array several times. We've used it several times.

We've bound the RouterLink directive to such an array like this:

<a [routerLink]="['Heroes']">Heroes</a>

We've written a two element array when specifying a route parameter like this

this.\_router.navigate( ['HeroDetail', { id: hero.id }] );

These two examples cover our needs for an app with one level routing. The moment we add a child router, such as the Crisis Center, we create new link array possibilities.

We specify a default child route for Crisis Center so this simple RouterLink is fine.

<a [routerLink]="['CrisisCenter']">Crisis Center</a>

If we hadn't specified a default route, our single item array would fail because we didn't tell the router which child route to use.

// The link now fails with a "non-terminal link" error

<a [routerLink]="['CrisisCenter']">Crisis Center</a>

We'd need to write our anchor with a link array like this:

<a [routerLink]="['CrisisCenter', 'CrisisCenter']">Crisis Center</a>

Huh? Crisis Center, Crisis Center. This looks like a routing crisis!

But it actually makes sense. Let's parse it out.

* The first item in the array identifies the parent route ('CrisisCenter').
* There are no parameters for this parent route so we're done with it.
* There is no default for the child route so we need to pick one.
* We decide to go to the CrisisListComponent whose route name just happens also to be 'CrisisCenter'
* So we add that 'CrisisCenter' as the second item in the array.
* Voila! ['CrisisCenter', 'CrisisCenter'].

Let's take it a step further. This time we'll build a link parameters array that navigates from the root of the application down to the "Princess Crisis".

* The first item in the array identifies the parent route ('CrisisCenter').
* There are no parameters for this parent route so we're done with it.
* The second item identifies the child route for details about a particular crisis ('CrisisDetail').
* The details child route requires an id route parameter
* We add the "Princess Crisis" id as the third item in the array ({id:1})

It looks like this!

<a [routerLink]="['CrisisCenter', 'CrisisDetail', {id:1}]">Princess Crisis</a>

We could redefine our AppComponent template with Crisis Center routes exclusively

template: `

<h1 class="title">Component Router</h1>

<a [routerLink]="['CrisisCenter', 'CrisisCenter']">Crisis Center</a>

<a [routerLink]="['CrisisCenter', 'CrisisDetail', {id:1}]">Princess Crisis</a>

<a [routerLink]="['CrisisCenter', 'CrisisDetail', {id:2}]">Dragon Crisis</a>

<router-outlet></router-outlet>

`,

### Link Parameters Arrays in Redirects

What if we weren't constructing anchor tags with RouterLink directives? What if we wanted to add a disaster route as part of the top-level router's configuration?

We can do that!

We compose a 3-item link parameter array following the recipe we just created. This time we set the id to the "Asteroid Crisis" ({id:3}).

We can't define a normal route because that requires setting a target component. We're not defining a route to a component. We're defining aroute to a route. A route to a route is a **redirect**. Here's the redirect route we'll add to our configuration.

{path: '/disaster', name: 'Asteroid', redirectTo: ['CrisisCenter', 'CrisisDetail', {id:3}]}

We hope the picture is clear. We can write applications with one, two or more levels of routing. The link parameter array affords the flexibility to represent any routing depth and any legal sequence of route names and (optional) route parameter objects.

## Appendix: Why use an ngOnInit method

We implemented an ngOnInit method in many of our Component classes. We did so, for example, in the [HeroDetailComponent](https://angular.io/docs/ts/latest/guide/router.html#hero-detail-ctor). We might have put the ngOnInit logic inside the constructor instead. We didn't for a reason. The reason is testability.

A constructor that has major side-effects can be difficult to test because it starts doing things as soon as we create a test instance. In this case, it might have made a request to a remote server, something it shouldn't do under test. It may even be impossible to reach the server in the test environment.

The better practice is to limit what the constructor can do. Mostly it should stash parameters in local variables and perform simple instance configuration.

Yet we want an instance of this class to get the hero data from the HeroService soon after it is created. How do we ensure that happens if not in the constructor?

Angular detects when a component has certain lifecycle methods like [ngOnInit](https://angular.io/docs/ts/latest/api/core/OnInit-interface.html) and [ngOnDestroy](https://angular.io/docs/ts/latest/api/core/OnDestroy-interface.html) and calls them at the appropriate moment.

Angular will call ngOnInit when we navigate to the HeroDetailComponent, we'll get the id from the RouteParams and ask the server for the hero with that id.

We too can call that ngOnInit method in our tests if we wish ... after taking control of the injected HeroService and (perhaps) mocking it.

## Appendix: 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 a request to the server and should not reload the page.

We're talking now about the **browser** URL **not** the route URL that we record in a RouteDefinition. The browser URL is what we paste into the browser's**address bar** and email to folks so they can deep-link into an application page.

Modern HTML 5 browsers support [history.pushState](https://developer.mozilla.org/en-US/docs/Web/API/History_API#Adding_and_modifying_history_entries), a technique that changes a browser's location and history without triggering a server page request. The router can compose a "natural" URL that is indistinguishable from one that would otherwise require a page load.

Here's the Crisis Center URL in this "HTML 5 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 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 Angular Component Router supports both styles. We set our preference by providing a LocationStrategy during the bootstrapping process.

Learn about "providers" and the bootstrap process in the [Dependency Injection chapter](https://angular.io/docs/ts/latest/guide/dependency-injection#bootstrap)

### Which Strategy is Best?

We must choose a strategy and we 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 2 projects should use the default HTML 5 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.

Thist 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.

### HTML 5 URLs and the <base href>

The router use the "[HTML 5 pushState](https://developer.mozilla.org/en-US/docs/Web/API/History_API#Adding_and_modifying_history_entries)" style by default. We don't have to provide the router's PathLocationStrategy because it's loaded automatically.

We must add a [<base href> element](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/base) tag in the <head> of the index.html.

<base href="/">

Without that tag, the browser may not be 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 link.

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 HTML 5 URLs by taking two remedial steps:

1. Provide the router with an appropriate APP\_BASE\_HREF value.
2. Use **absolute URLs** for all web resources: css, images, scripts, and template html files.

Learn about the [APP\_BASE\_HREF](https://angular.io/docs/ts/latest/api/router/APP_BASE_HREF-const.html) in the API Guide.

### HashLocationStrategy

We can go old-school with the HashLocationStrategy by providing it as the router's LocationStrategy during application bootstrapping.

That means importing provide for Dependency Injection and the Location and HashLocationStrategy symbols from the router, then providing that strategy in the call to bootstrap:

1. import {bootstrap} from 'angular2/platform/browser';
2. import {ROUTER\_PROVIDERS} from 'angular2/router';
3. import {AppComponent} from './app.component';
4. // Add these symbols to register a `LocationStrategy`
5. import {provide} from 'angular2/core';
6. import {LocationStrategy,
7. HashLocationStrategy} from 'angular2/router';
8. bootstrap(AppComponent, [
9. provide(LocationStrategy,
10. {useClass: HashLocationStrategy}), // ~/src/#/crisis-center/
11. ROUTER\_PROVIDERS
12. ]);
13. LIFECYCLE HOOKS

Angular calls lifecycle hook methods on our directives and components as it creates, changes, and destroys them.

# Component Lifecycle

A Component has a lifecycle managed by Angular itself. Angular creates it, renders it, creates and renders its children, checks it when its data-bound properties change, and destroys it before removing it from the DOM.

Angular offers **Lifecycle hooks** that give us visibility into these key moments and the ability to act when they occur.

We cover these hooks in this chapter and demonstrate how they work in code.

[Live Example](https://angular.io/resources/live-examples/lifecycle-hooks/ts/plnkr.html)

## The Lifecycle Hooks

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, all of them available in the angular2/core library.

Here is the complete lifecycle hook interface inventory:

* OnInit
* OnDestroy
* DoCheck
* OnChanges
* AfterContentInit
* AfterContentChecked
* AfterViewInit
* AfterViewChecked

No directive or component will implement all of them and some of them only make sense for components.

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 names ngOnInit.

Angular calls these hook methods in the following order:

* ngOnChanges - called when an input or output binding value changes
* ngOnInit - after the first ngOnChanges
* ngDoCheck - developer's custom change detection
* ngAfterContentInit - after component content initialized
* ngAfterContentChecked - after every check of component content
* ngAfterViewInit - after component's view(s) are initialized
* ngAfterViewChecked - after every check of a component's view(s)
* ngOnDestroy - just before the directive is destroyed.

The [live example](https://angular.io/resources/live-examples/lifecycle-hooks/ts/plnkr.html) demonstrates these hooks.

## Peek-a-boo

The PeekABooComponent demonstrates all of the hooks in the same component.

Except for DoCheck. If our component superseded regular Angular change detection with its own change detection processing we would also add angDoCheck method. We would **not** implement ngOnChanges. We write either ngOnChanges or ngDoCheck, not both.

Custom change detection and ngDoCheck are on our documentation backlog.

Peek-a-boo is a demo. We'd rarely if ever implement all interfaces like this in real life.

We look forward to explaining the Peek-a-boo example and the other lifecycle hook examples in an update to this chapter. Meanwhile, please enjoy poking around in the [code](https://angular.io/resources/live-examples/lifecycle-hooks/ts/plnkr.html).

## Interface optional?

The lifecycle interfaces are optional. We recommend adding them to benefit from TypeScript's strong typing and editor tooling.

But they disappear from the transpiled JavaScript. Angular can't see them at runtime. And they are useless to someone developing in a language without interfaces (such as pure JavaScript).

Fortunately, they aren't necessary. We don't have to add the lifecycle hook interfaces to our directives and components to benefit from the hooks themselves.

Angular instead inspects our directive and component classes and calls the hook methods if they are defined. Angular will find and call methods like ngOnInit(), with or without the interfaces.

ATTRIBUTE DIRECTIVES

Attribute directives attach behavior to elements.

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

In this chapter we will

* write an attribute directive to change the background color
* apply the attribute directive to an element in a template
* respond to user-initiated events
* pass values into the directive using data binding

[Live Example](https://angular.io/resources/live-examples/attribute-directives/ts/plnkr.html)

## Directives overview

There are three kinds of directives in Angular:

1. Components
2. Structural directives
3. Attribute directives

The Component is really a directive with a template. It's the most common of the three directives and we write lots of them as we build our application.

The [Structural directive](https://angular.io/docs/ts/latest/guide/structural-directives.html) changes the DOM layout by adding and removing DOM elements. [NgFor](https://angular.io/docs/ts/latest/guide/template-syntax.html#ng-for) and [NgIf](https://angular.io/docs/ts/latest/guide/template-syntax.html#ng-if) are two familiar examples.

The Attribute directive changes the appearance or behavior of an element. The built-in [NgStyle](https://angular.io/docs/ts/latest/guide/template-syntax.html#ng-style) directive, for example, can change several element styles at the same time.

We are going to write our own attribute directive to set an element's background color when the user hovers over that element.

We don't need any directive to simply set the background color. We can set it with the special [Style Binding](https://angular.io/docs/ts/latest/guide/template-syntax.html#style-binding) like this:

<p [style.background]="'lime'">I am green with envy!</p>

That wouldn't be nearly as much fun as creating our own directive.

Besides, we're not just setting the color; we'll be changing the color in response to a user action, a mouse hover.

## Build a simple attribute directive

An attribute directive minimally requires building a controller class annotated with a Directive decorator. The Directive decorator specifies the selector identifying the attribute associated with the directive. The controller class implements the desired directive behavior.

Let's build a small illustrative example together.

### Our first draft

Create a new project folder (attribute-directives) and follow the steps in the [QuickStart](https://angular.io/docs/ts/latest/quickstart.html).

Add a new file to the app folder called highlight.directive.ts and add the following code:

app/highlight.directive.ts

1. import {Directive, ElementRef, Renderer, Input} from 'angular2/core';
2. @Directive({
3. selector: '[myHighlight]'
4. })
5. export class HighlightDirective {
6. constructor(el: ElementRef, renderer: Renderer) {
7. //el.nativeElement.style.backgroundColor = 'yellow';
8. renderer.setElementStyle(el, 'backgroundColor', 'yellow');
9. }
10. }

We begin by importing some symbols from the Angular library. We need the Directive symbol for the @Directive decorator. We need symbols for the Element Reference and the Renderer service that we will [inject](https://angular.io/docs/ts/latest/guide/dependency-injection.html) into the directive's constructor. We don't need Input now but we will need it later in the chapter.

Then we define the directive metadata in a configuration object passed as an argument to the @Directive decorator function. A@Directive decorator for an attribute directive requires a css selector to identify the HTML in the template that is associated with our directive. The [css selector for an attribute](https://developer.mozilla.org/en-US/docs/Web/CSS/Attribute_selectors) is the attribute name in square brackets.

Our directive's selector is [myHighlight]. Angular will locate all elements in the template that have an attribute named myHighlight.

### Why not call it "highlight"?

highlight is a nicer name than myHighlight and, technically, it would work if we called it that. However, the good folks at Angular strongly prefer hyphenated directive selector names. The HTML standards body will never name one of its attributes with a hyphen and there is less risk of colliding with a third-party directive name when we give ours a prefix. The ng prefix belongs to Angular. We need a prefix of our own, preferably short, and my will do for now.

After the @Directive metadata comes the directive's controller class which we are exporting to make it accessible to other components. The directive's controller class contains the logic for the directive.

Angular creates a new instance of the directive's controller class for each matching element, injecting an Element Reference and the Rendererservice as arguments to the constructor. We'll need those services to set the element's background color.

Our code shows two ways to set the color.

We could access the nativeElement property of the element reference and set the element's background color using the browser DOM API. We don't need the Renderer for this approach.

We commented this technique out. It works. But we don't like it.

We prefer the second way that relies on the Renderer service to set the element properties.

### Why prefer the Renderer?

Manipulating the DOM directly is a practice we'd rather avoid because it chains us to the browser DOM API.

The Renderer insulates our code from the browser's API. That gives us options. The rendering phase could be offloaded to a Web Worker for faster performance. Our directive might work when we ran the application outside the browser, perhaps on the server in a pre-render phase. Server-side rendering can make our application load faster and is often friendlier to Search Engine Optimizations (SEO).

## Apply the attribute directive

The AppComponent will be the test harness for our HighlightDirective. Let's give it a new template that applies the directive as an attribute to a span element. In Angular terms, the <span> element will be the attribute **host**.

We'll put the template in its own app.component.html file that looks like this:

app/app.component.html

<h1>My First Attribute Directive</h1>

<span myHighlight>Highlight me!</span>

A separate template file is clearly overkill for a 2-line template. Hang in there; we're going to expand it later. Meanwhile, we'll revise theAppComponent to reference this template.

app/app.component.ts

1. import {Component} from 'angular2/core';
2. import {HighlightDirective} from './highlight.directive'
3. @Component({
4. selector: 'my-app',
5. templateUrl: 'app/app.component.html',
6. directives: [HighlightDirective]
7. })
8. export class AppComponent { }

We've added an import statement to fetch the 'Highlight' directive and added that class to a directives array in the component metadata so that Angular will recognize our directive when it encounters myHighlight in the template. Angular would simply ignore themyHighlight attribute without it.

We run the app and see that our directive highlights the span text.



Let's recap what happened.

Angular found the myHighlight attribute on the <span> element. It created an instance of the HighlightDirective class, injecting both a reference to the element and the Renderer service into the constructor. The constructor told the Renderer to set the <span> element's background style to yellow.

## Respond to user action

We are not satisfied to simply set an element color. Our directive should set the color in response to a user action. Specifically, we want to set the color when the user mouses over the element.

We'll need to

1. detect when the user mouses into and out of the element
2. respond to those actions by setting and clearing the highlight color.

Start with event detection. We add a host property to the directive metadata and give it a configuration object that specifies two mouse events and the directive methods to call when they are raised.

host: {

'(mouseenter)': 'onMouseEnter()',

'(mouseleave)': 'onMouseLeave()'

}

The host property refers to the DOM element that hosts our attribute directive, the <span> in our case.

We could have attached an event listener to the native element (el.nativeElement) with plain old JavaScript. There are at least three problems with that approach:

1. We have to write the listeners correctly.
2. We must detach our listener when the directive is destroyed to avoid memory leaks.
3. We'd be talking to DOM API directly which, we learned, is something to avoid.

Let's roll with the host property.

Now we implement those two mouse event handlers:

onMouseEnter() { this.\_highlight("yellow"); }

onMouseLeave() { this.\_highlight(null); }

private \_highlight(color: string) {

this.renderer.setElementStyle(this.el, 'backgroundColor', color);

}

Notice that they delegate to a helper method that calls the Renderer service as we used to do in the constructor.

We no longer need the constructor body but we still want the injected ElementRef and Renderer service. We revise the constructor signature to capture the injectables in private variables and clear the body.

constructor(private el: ElementRef, private renderer: Renderer) {

}

Here's the updated directive:

app/highlight.directive.ts

1. import {Directive, ElementRef, Renderer, Input} from 'angular2/core';
2. @Directive({
3. selector: '[myHighlight]',
4. host: {
5. '(mouseenter)': 'onMouseEnter()',
6. '(mouseleave)': 'onMouseLeave()'
7. }
8. })
9. export class HighlightDirective {
10. constructor(private el: ElementRef, private renderer: Renderer) {
11. }
12. onMouseEnter() { this.\_highlight("yellow"); }
13. onMouseLeave() { this.\_highlight(null); }
14. private \_highlight(color: string) {
15. this.renderer.setElementStyle(this.el, 'backgroundColor', color);
16. }
17. }

We run the app and confirm that the background color appears as we move the mouse over the span and disappears as we move out.



## Configure the directive with binding

Currently the highlight color is hard-coded within the directive. That's inflexible. We should set the highlight color externally with a binding like this:

<p [myHighlight]="color">Highlight me!</p>

We'll extend our directive class with a bindable **input** highlightColor property and use it when we highlight text.

Here is the final version of the class:

app/highlight.directive.ts (class only)

1. export class HighlightDirective {
2. @Input('myHighlight') highlightColor: string;
3. private \_defaultColor = 'red';
4. constructor(private el: ElementRef, private renderer: Renderer) { }
5. onMouseEnter() { this.\_highlight(this.highlightColor || this.\_defaultColor); }
6. onMouseLeave() { this.\_highlight(null); }
7. private \_highlight(color:string) {
8. this.renderer.setElementStyle(this.el, 'backgroundColor', color);
9. }
10. }

The new highlightColor property is called an "input" property because data flows from the binding expression into our directive. Notice that we call the @Input() decorator function while defining the property.

@Input('myHighlight') highlightColor: string;

This @Input decorator adds metadata to the class that makes the highlightColor property available for property binding under themyHighlight alias. We must add this input metadata. Angular will reject a binding to this property if we don't declare it as an input.

See the [appendix](https://angular.io/docs/ts/latest/guide/attribute-directives.html#why-input) below to learn why.

The developer who uses our directive expects to bind to the attribute name, myHighlight. The directive property name is highlightColor. That's a disconnect.

We can resolve the discrepancy by renaming the property to myHighlight and define it as follows:

@Input() myHighlight: string;

Maybe we don't want that property name inside the directive perhaps because it doesn't express our intention well. We can **alias** the highlightColorproperty with the attribute name by passing myHighlight into the @Input decorator:

@Input('myHighlight') highlightColor: string;

Now that we're getting the highlight color as an input, we modify the onMouseEnter() method to use it instead of the hard-coded color name. We also define a red default color as a fallback in case the user neglects to bind with a color.

onMouseEnter() { this.\_highlight(this.highlightColor || this.\_defaultColor); }

Now we'll update our AppComponent template to let users pick the highlight color and bind their choice to our directive.

Here is the updated template:

1. <h1>My First Attribute Directive</h1>
2. <h4>Pick a highlight color</h4>
3. <div>
4. <input type="radio" name="colors" (click)="color='lightgreen'">Green
5. <input type="radio" name="colors" (click)="color='yellow'">Yellow
6. <input type="radio" name="colors" (click)="color='cyan'">Cyan
7. </div>
8. <p [myHighlight]="color">Highlight me!</p>

### Where is the templated color property?

The eagle-eyed may notice that the radio button click handlers in the template set a color property and we are binding that color to the directive. We should expect to find a color on the host AppComponent.

**We never defined a color property for the host AppComponent**! And yet this code works. Where is the template color value going?

Browser debugging reveals that Angular dynamically added a color property to the runtime instance of the AppComponent.

This is convenient behavior but it is also implicit behavior that could be confusing. While it's cool that this technique works, we recommend adding thecolor property to the AppComponent.

Here is our second version of the directive in action.



## Bind to a second property

Our directive only has a single, customizable property. What if we had **two properties**?

Let's let the template developer set the default color, the color that prevails until the user picks a highlight color. We'll add a second **input**property to HighlightDirective called defaultColor:

@Input() set defaultColor(colorName:string){

this.\_defaultColor = colorName || this.\_defaultColor;

}

The defaultColor property has a setter that overrides the hard-coded default color, "red". We don't need a getter.

How do we bind to it? We already "burned" the myHighlight attribute name as a binding target.

Remember that a component is a directive too. We can add as many component property bindings as we need by stringing them along in the template as in this example that sets the a, b, c properties to the string literals 'a', 'b', and 'c'.

<my-component [a]="'a'" [b]="'b'" [c]="'c'"><my-component>

We do the same thing with an attribute directive.

<p [myHighlight]="color" [defaultColor]="'violet'">

Highlight me too!

</p>

Here we're binding the user's color choice to the myHighlight attribute as we did before. We're also binding the literal string, 'violet', to thedefaultColor.

Here is the final version of the directive in action.

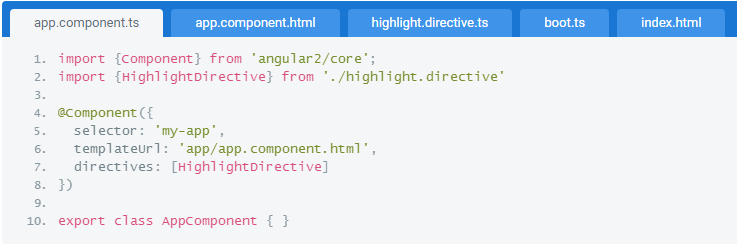


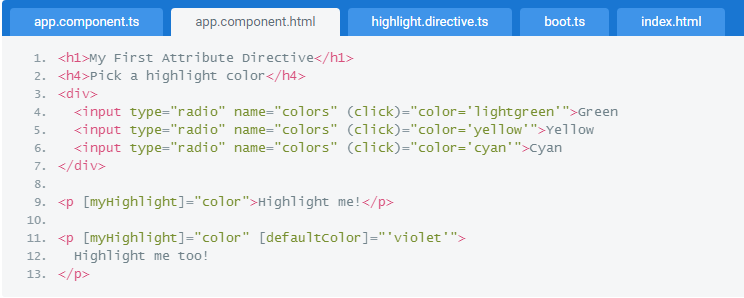
## Summary

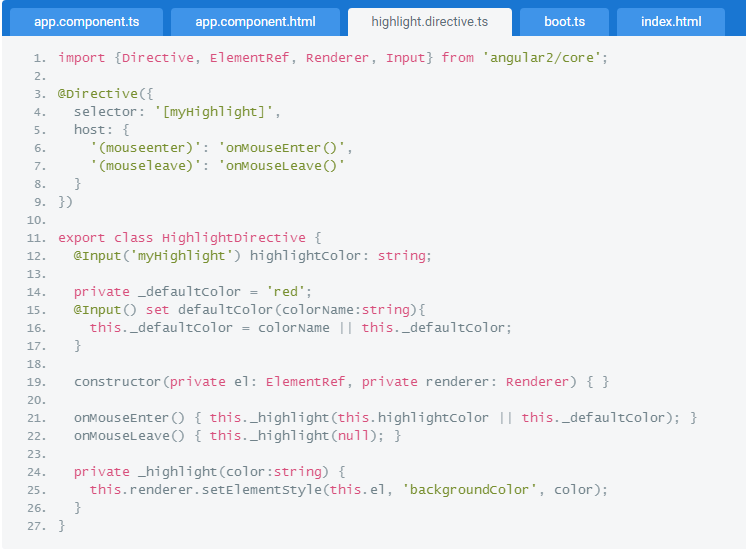
Now we know how to

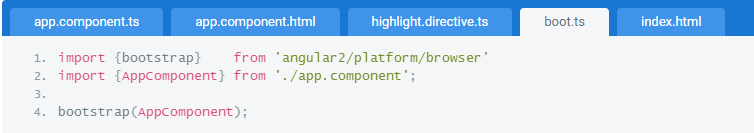
* build a simple **attribute directive** to attach behavior to an HTML element,
* use that directive in a template,
* respond to **events** to change behavior based on an event,
* and use **binding** to pass values to the attribute directive.

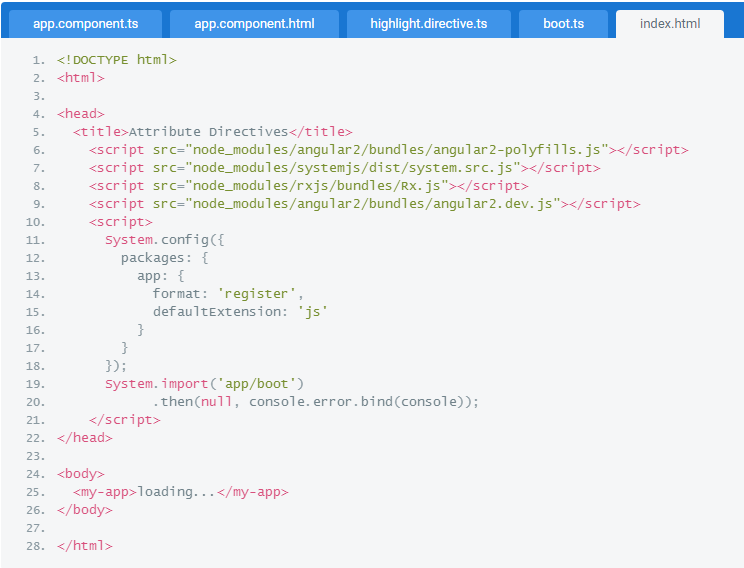
The final source:











### Appendix: Input properties

Earlier we declared the highlightColor property to be an **input** property of our HighlightDirective

We've seen properties in bindings before. We never had to declare them as anything. Why now?

Angular makes a subtle but important distinction between binding **sources** and **targets**.

In all previous bindings, the directive or component property was a binding **source**. A property is a source if it appears in the template expression to the **right** of the (=).

A property is a target when it appears to the **\*left** of the (=) ... as it is does when we bind to the myHighlight property of theHighlightDirective,

<p [myHighlight]="color">Highlight me!</p>

The 'color' in [myHighlight]="color" is a binding **source**. A source property doesn't require a declaration.

The 'myHighlight' in [myHighlight]="color" is a binding **target**. We must declare it as an input property. Angular rejects the binding with a clear error if we don't.

Angular treats a target property differently for a good reason. A component or directive in target position needs protection.

Imagine that our HighlightDirective did truly wonderous things. We graciously made a gift of it to the world.

To our surprise, some people — perhaps naively — started binding to every property of our directive. Not just the one or two properties we expected them to target. Every property. That could really mess up our directive in ways we didn't anticipate and have no desire to support.

The input declaration ensures that consumers of our directive can only bind to the properties of our public API ... nothing else.

STRUCTURAL DIRECTIVES

Angular has a powerful template engine that lets us easily manipulate the DOM structure of our elements.

One of the defining features of a single page application is its manipulation of the DOM tree. Instead of serving a whole new page every time a user navigates, whole sections of the DOM appear and disappear according to the application state. In this chapter we'll to look at how Angular manipulates the DOM and how we can do it ourselves in our own directives.

In this chapter we will

* [learn what structural directives are](https://angular.io/docs/ts/latest/guide/structural-directives.html#definition)
* [study ngIf](https://angular.io/docs/ts/latest/guide/structural-directives.html#ng-if)
* [discover the <template> element](https://angular.io/docs/ts/latest/guide/structural-directives.html#template)
* [understand the asterisk (\*) in \*ngFor](https://angular.io/docs/ts/latest/guide/structural-directives.html#asterisk)
* [write our own structural directive](https://angular.io/docs/ts/latest/guide/structural-directives.html#unless)

[Live example](https://angular.io/resources/live-examples/structural-directives/ts/plnkr.html)

## What are structural directives?

There are three kinds of Angular directives:

1. Components
2. Attribute directives
3. Structural directives

The Component is really a directive with a template. It's the most common of the three directives and we write lots of them as we build our application.

The [Attribute directive](https://angular.io/docs/ts/latest/guide/attribute-directives.html) changes the appearance or behavior of an element. The built-in [NgStyle](https://angular.io/docs/ts/latest/guide/template-syntax.html#ng-style) directive, for example, can change several element styles at the same time. We can use it to render text bold, italic, and lime green by binding to a component property that requests such a sickening result.

A Structural directive changes the DOM layout by adding and removing DOM elements. We've seen three of the built-in structural directives in other chapters: [ngIf](https://angular.io/docs/ts/latest/guide/template-syntax.html#ngIf), [ngSwitch](https://angular.io/docs/ts/latest/guide/template-syntax.html#ngSwitch) and [ngFor](https://angular.io/docs/ts/latest/guide/template-syntax.html#ngFor).

<div \*ngIf="hero">{{hero}}</div>

<div \*ngFor="#hero of heroes">{{hero}}</div>

<div [ngSwitch]="status">

<template [ngSwitchWhen]="'in-mission'">In Mission</template>

<template [ngSwitchWhen]="'ready'">Ready</template>

<template ngSwitchDefault>Unknown</template>

</div>

## NgIf Case Study

Let’s focus on ngIf. It's a great example of a structural directive: it takes a boolean and makes an entire chunk of DOM appear or disappear.

<p \*ngIf="condition">

condition is true and ngIf is true.

</p>

<p \*ngIf="!condition">

condition is false and ngIf is false.

</p>

The ngIf directive does not hide the element. Using browser developer tools we can see that, when the condition is true, the top paragraph is in the DOM and the bottom disused paragraph is completely absent from the DOM! In its place are empty <script> tags.

element not in dom

### Why remove rather than hide?

We could hide the unwanted paragraph by setting its css display style to none. The element would remain in the DOM while invisible. Instead we removed it with ngIf.

The difference matters. When we hide an element, the component's behavior continues. It remains attached to its DOM element. It continues to listen 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 descendent components — tie up resources that might be more useful elsewhere. The performance and memory burden can be substantial and the user may not benefit at all.

On the positive side, showing the element again is very quick. The component's previous state is preserved and ready to display. The component doesn't re-initialize — an operation that could be expensive.

ngIf is different. Setting ngIf to false **does** affect the component's resource consumption. Angular removes the element from DOM, stops change detection for the associated component, detaches it from DOM events (the attachments that it made) and destroys the component. The component can be garbage-collected (we hope) and free up memory.

Components often have child components which themselves have children. All of them are destroyed when ngIf destroys the common ancestor. This cleanup effort is usually a good thing.

Of course it isn't always a good thing. It might be a bad thing if we need that particular component again soon.

The component's state might be expensive to re-construct. When ngIf becomes true again, Angular recreates the component and its subtree. Angular runs every component's initialization logic again. That could be expensive ... as when a component re-fetches data that had been in memory just moments ago.

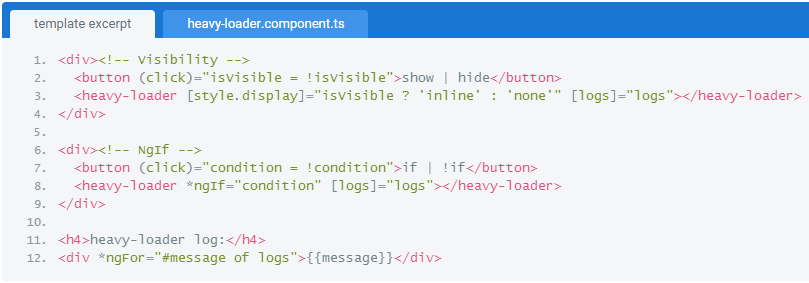
Design thought: minimize initialization effort and consider caching state in a companion service.

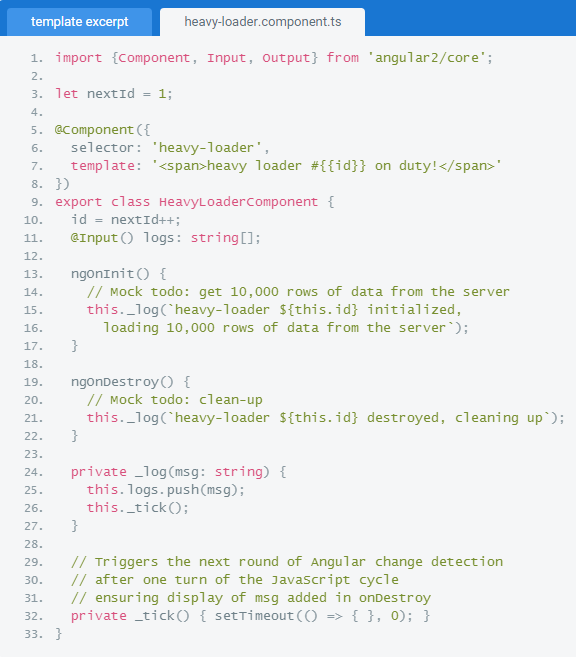
Although there are pros and cons to each approach, in general it is best to use ngIf to remove unwanted components rather than hide them.

**These same considerations apply to every structural directive, whether built-in or custom.** We should ask ourselves — and the users of our directives — to think carefully about the consequences of adding and removing elements and of creating and destroying components.

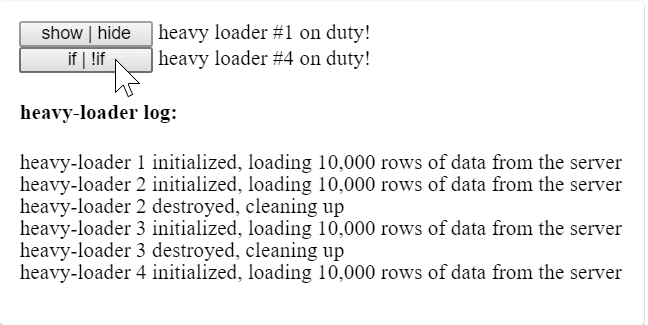
Let's see these dynamics at work. For fun, we'll stack the deck against our recommendation and consider a component called heavy-loader that **pretends** to load a ton of data when initialized.

We'll display two instances of the component. We toggle the visibility of the first one with CSS. We toggle the second into and out of the DOM with ngIf.





We also log when a component is created or destroyed using the built-in ngOnInit and ngOnDestroy [lifecycle hooks](https://angular.io/docs/ts/latest/guide/lifecycle-hooks.html). Here it is in action:



Both components are in the DOM at the start. First we toggle the component's visibility repeatedly. The component never leaves the DOM. When visible it's always the same instance and the log is quiet.

Then we toggle the second component with ngIf. We create a new instance every time and the log shows that we're paying a heavy price to create and destroy it.

If we really expected to "wink" the component like this, toggling visibility would be the better choice. In most UIs, when we "close" a component we're unlikely see it again for a long time, if ever. The ngIf would be preferred in that case.

## The <template> tag

Structural directives, like ngIf, do their magic by using the [HTML 5 template tag](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/template).

Outside of an Angular app, the <template> tag's default CSS display property is none. It's contents are **invisible** within a hidden[document fragment](https://developer.mozilla.org/en/docs/Web/API/DocumentFragment).

Inside of an app, Angular **removes** the<template> tags and their children. The contents are gone — but not forgotten as we'll see soon.

We can confirm these effects by wrapping the middle "hip" of the phrase "Hip! Hip! Hooray!" within a <template> tag.

<p>

Hip!

</p>

<template>

<p>

Hip!

</p>

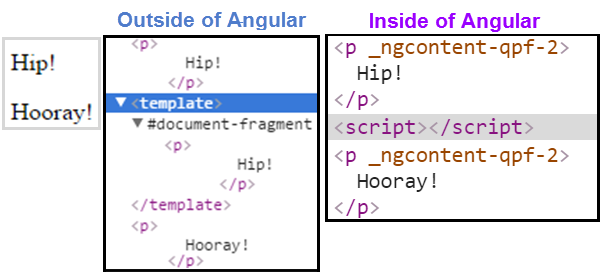
</template>

<p>

Hooray!

</p>

The display is a 'Hip!' short of perfect enthusiasm. The DOM effects are different when Angular is control.



Evidently Angular replaces the <template> tag and its contents with empty <script> tags. That's just its default behavior. It can do something different as we saw when applying a variety of ngSwitch directives to <template> tags:

<div [ngSwitch]="status">

<template [ngSwitchWhen]="'in-mission'">In Mission</template>

<template [ngSwitchWhen]="'ready'">Ready</template>

<template ngSwitchDefault>Unknown</template>

</div>

When one of those ngSwitch conditions is true, Angular inserts the template's content into the DOM.

What does this have to do with ngIf and ngFor? We didn't use a <template> tag with those directives.

## The asterisk (\*) effect

Here are those directives again. See the difference?

<div \*ngIf="hero">{{hero}}</div>

<div \*ngFor="#hero of heroes">{{hero}}</div>

We're prefixing these directive names with an asterisk (\*).

The asterisk is "syntactic sugar". It simplifies ngIf and ngFor for both the writer and the reader. Under the hood, Angular replaces the asterisk version with a more verbose <template> form.

The next two ngIf examples are effectively the same and we may write in either style:

<!-- Examples (A) and (B) are the same -->

<!-- (A) \*ngIf paragraph -->

<p \*ngIf="condition">

Our heroes are true!

</p>

<!-- (B) [ngIf] with template -->

<template [ngIf]="condition">

<p>

Our heroes are true!

</p>

</template>

Most of us would rather write in style (A).

It's worth knowing that Angular expands style (A) into style (B). It moves the paragraph and its contents inside a <template> tag. It moves the directive up to the <template> tag where it becomes a property binding, surrounded in square brackets. The boolean value of the host component's condition property determines whether the templated content is displayed or not.

Angular transforms \*ngFor in a similar manner:

<!-- Examples (A) and (B) are the same -->

<!-- (A) \*ngFor div -->

<div \*ngFor="#hero of heroes">{{ hero }}</div>

<!-- (B) ngFor with template -->

<template ngFor #hero [ngForOf]="heroes">

<div>{{ hero }}</div>

</template>

The basic pattern is the same:  create a <template>, relocate the content, and move the directive onto the <template>.

There are extra nuances stemming from Angular's [ngFor micro-syntax](https://angular.io/docs/ts/latest/guide/template-syntax#micro-syntax) which expands into an additional ngForOf property binding (the iterable) and the #hero [local template variable](https://angular.io/docs/ts/latest/guide/template-syntax#local-vars) (the current item in each iteration).

## Make a structural directive

Let's write our own structural directive, an Unless directive, the not-so-evil twin of ngIf.

Unlike ngIf which displays the template content when true, our directive displays the content when the condition is **false**.

Creating a directive is similar to creating a component.

* import the Directive decorator.
* add a CSS **attribute selector** (in brackets) that identifies our directive.
* specify the name of the public input property for binding (typically the name of the directive itself).
* apply the decorator to our implementation class.

Here is how we begin:

unless.directive.ts (excerpt)

import {Directive, Input} from 'angular2/core';

@Directive({ selector: '[myUnless]' })

export class UnlessDirective {

}

### Selector brackets [ ]

The CSS syntax for selecting an attribute is a name in square brackets. We surround our directive name in square brackets. See Directive configuration on the [cheatsheet](https://angular.io/docs/ts/latest/guide/cheatsheet.html).

### Selector name prefixes

We recommend picking a selector name with a prefix to ensure that it cannot conflict with any standard HTML attribute, now or in the future.

We do **not** prefix our unless directive name with **ng**. That prefix belongs to Angular and we don't want to confuse our directives with their directives.

Our prefix is my.

We'll need access to the template and something that can render its contents. We access the template with a TemplateRef. The renderer is a ViewContainerRef. We inject both into our constructor as private variables.

constructor(

private \_templateRef: TemplateRef,

private \_viewContainer: ViewContainerRef

) { }

The consumer of our directive will bind a true | false value to our directive's myUnless input property. The directive adds or removes the template based on that value.

Let's add the myUnless property now as a setter-only [definedProperty](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty).

@Input() set myUnless(condition: boolean) {

if (!condition) {

this.\_viewContainer.createEmbeddedView(this.\_templateRef);

} else {

this.\_viewContainer.clear();

}

}

The @Input() annotation marks this property as an input for the directive.

Nothing fancy here: if the condition is false, we render the template, otherwise we clear the element content.

The end result should look like below:

unless.directive.ts

1. import {Directive, Input} from 'angular2/core';
2. import {TemplateRef, ViewContainerRef} from 'angular2/core';
3. @Directive({ selector: '[myUnless]' })
4. export class UnlessDirective {
5. constructor(
6. private \_templateRef: TemplateRef,
7. private \_viewContainer: ViewContainerRef
8. ) { }
9. @Input() set myUnless(condition: boolean) {
10. if (!condition) {
11. this.\_viewContainer.createEmbeddedView(this.\_templateRef);
12. } else {
13. this.\_viewContainer.clear();
14. }
15. }
16. }

Now we add it to the directivesarray of the host component and try it. First we add some test HTML to the template:

<p \*myUnless="condition">

condition is false and myUnless is true.

</p>

<p \*myUnless="!condition">

condition is true and myUnless is false.

</p>

We run it and it behaves as expected, doing the opposite of ngIf. When condition is true, the top paragraph is removed (replaced by<script> tags) and the bottom paragraph appears.

myUnless is true

Our myUnless directive is dead simple. Surely we left something out. Surely ngIf is more complex?

[Look at the source code](https://github.com/angular/angular/blob/master/modules/angular2/src/common/directives/ng_if.ts). It's well documented and we shouldn't be shy about consulting the source when we want to know how something works.

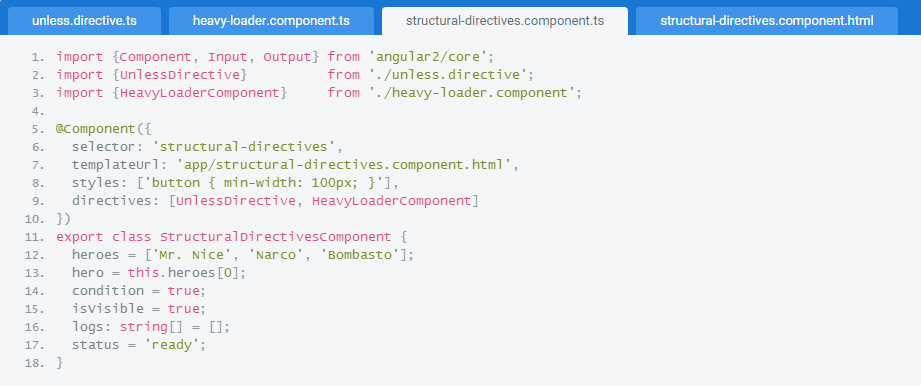
ngIf isn't much different! There are a few additional checks to improve performance (don't clear or recreate the view unless necessary) but otherwise it's much the same.

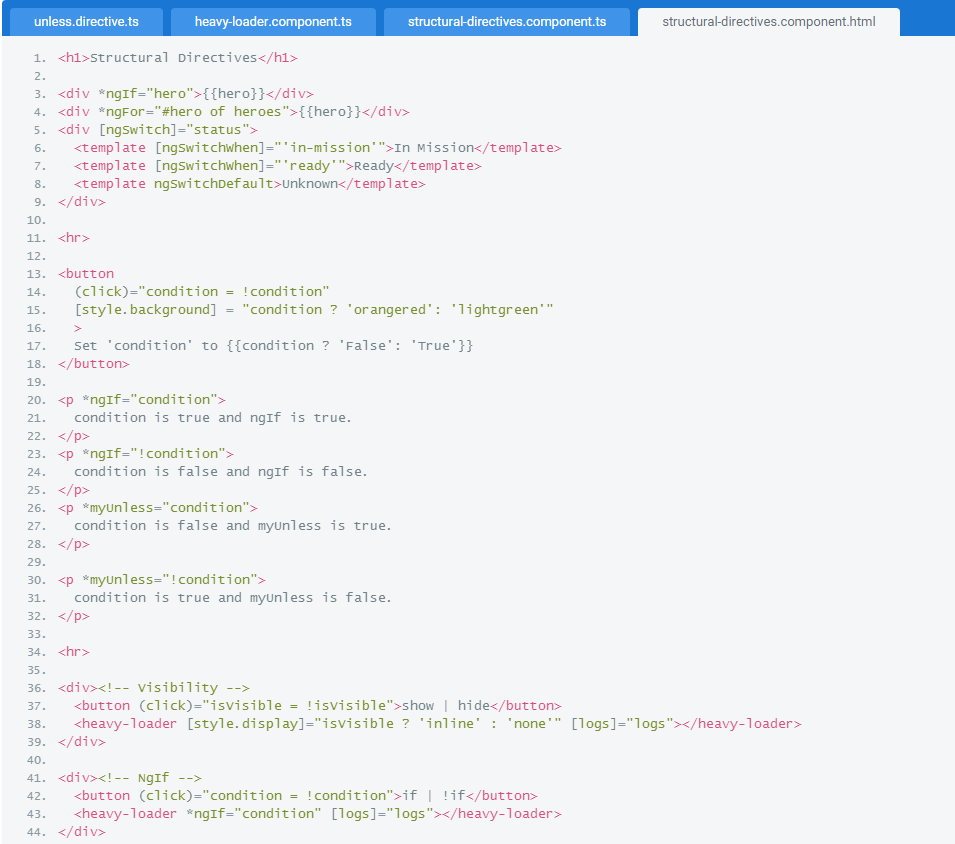
## Wrap up

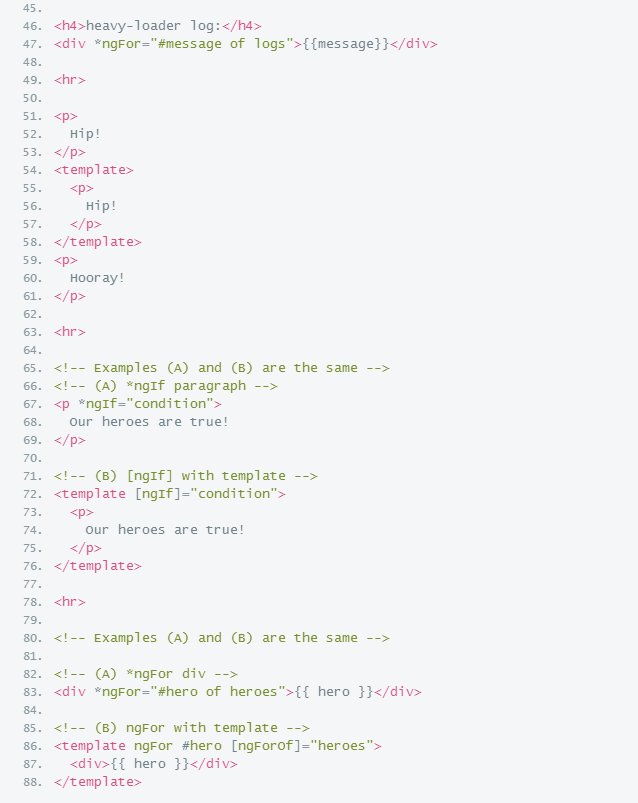
Here is the pertinent source for this chapter.











We learned that we can manipulate our HTML layout with structural directives like ngFor and ngIf and we wrote our own structural directive, myUnless, to do something similar.

Angular offers more sophisticated techniques for managing layout such as structural components that can take external content and incorporate that content within their own templates. Tab and tab pane controls are good examples.

We'll learn about structural components in a future chapter.

HIERARCHICAL INJECTORS

Angular's hierarchical dependency injection system supports nested injectors in parallel with the component tree.

We learned the basics of Angular Dependency injection in the [Dependency Injection](https://angular.io/docs/ts/latest/guide/dependency-injection.html) chapter.

Angular has an Hierarchical Dependency Injection system. There is actually a tree of injectors that parallel an application's component tree. We can re-configure the injectors at any level of that component tree with interesting and useful results.

In this chapter we explore these points and write some code.

[Live Example](https://angular.io/resources/live-examples/hierarchical-dependency-injection/ts/plnkr.html).

## The Injector Tree

In the [Dependency Injection](https://angular.io/docs/ts/latest/guide/dependency-injection.html) chapter we learned how to configure a dependency injector and how to retrieve dependencies where we need them.

We oversimplified. 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.

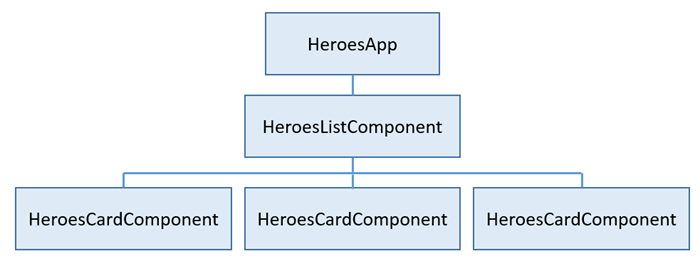
Angular doesn't literally create a separate injector for each component. Every component doesn't need its own injector and it would be horribly inefficient to create masses of injectors for no good purpose.

But it is true that every component **has an injector** (even if it shares that injector with another component) and there may be many different injector instances operating at different levels of the component tree.

It is useful to pretend that every component has its own injector.

Consider a simple variation on the Tour of Heroes application consisting of three different components: HeroesApp,HeroesListComponent and HeroesCardComponent. The HeroesApp holds a single instance of HeroesListComponent. The new twist is that the HeroesListComponent may hold and manage multiple instances of the HeroesCardComponent.

The following diagram represents the state of the component tree when there are three instances of HeroesCardComponent open simultaneously.



Each component instance gets its own injector and an injector at one level is a child injector of the injector above it in the tree.

When a component at the bottom 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 component's injector. The requests keep bubbling up until we find an injector that can handle the request or run out of component ancestors. If we run out of ancestors, Angular throws an error.

There's a third possibililty. An intermediate component can declare that it is the "host" component. The hunt for providers will climb no higher than the injector for this host component. We'll reserve discussion of this option for another day.

Such a proliferation of injectors makes little sense until we consider the possiblity that injectors at different levels can be configured with different providers. We don't have to re-configure providers at every level. But we can.

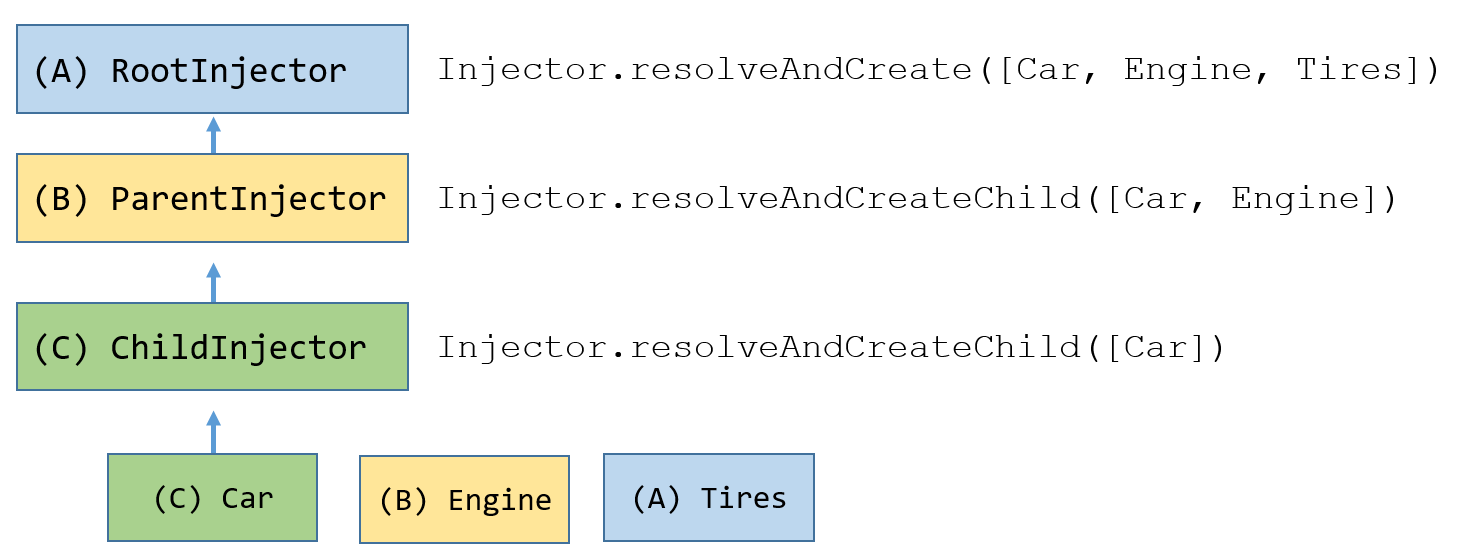
If we don't re-configure, the tree of injectors appears to be flat. All requests bubble up to the root injector that we configured with thebootstrap method.

The ability to configure one or more providers at different levels opens up interesting and useful possibilities.

Let’s return to our Car example. Suppose configured the root injector (marked as A) with providers for Car, Engine and Tires. We create a child component (B) that defines its own providers for Car and Engine This child is the parent of another component (C) that defines its own provider for Car.

Behind the scenes each component sets up its own injector with one or more providers defined for that component itself.

When we resolve an instance of Car at the deepest component (C), it's injector produces an instance of Car resolved by injector (C) with anEngine resolved by injector (B) and Tires resolved by the root injector (A).



## Component Injectors

In the previous section, we talked about injectors and how they are organized like a tree. Lookups follow the injector tree upwards until they found the requested thing to inject. But when do we actually want to provide providers on the root injector and when do we want to provide them on a child injector?

Consider you are building a component to show a list of super heroes that displays each super hero in a card with it’s name and superpower. There should also be an edit button that opens up an editor to change the name and superpower of our hero.

One important aspect of the editing functionality is that we want to allow multiple heroes to be in edit mode at the same time and that one can always either commit or cancel the proposed changes.

Let’s take a look at the HeroesListComponent which is the root component for this example.

app/heroes-list.component.ts

1. import {Component} from 'angular2/core';
2. import {EditItem} from './edit-item';
3. import {HeroesService} from './heroes.service';
4. import {HeroCardComponent} from './hero-card.component';
5. import {HeroEditorComponent} from './hero-editor.component';
6. import {Hero} from './hero';
7. @Component({
8. selector: 'heroes-list',
9. template: `
10. <div>
11. <ul>
12. <li \*ngFor="#editItem of heroes">
13. <hero-card
14. [hidden]="editItem.editing"
15. [hero]="editItem.item">
16. </hero-card>
17. <button
18. [hidden]="editItem.editing"
19. (click)="editItem.editing = true">
20. edit
21. </button>
22. <hero-editor
23. (saved)="onSaved(editItem, $event)"
24. (canceled)="onCanceled(editItem)"
25. [hidden]="!editItem.editing"
26. [hero]="editItem.item">
27. </hero-editor>
28. </li>
29. </ul>
30. </div>`,
31. directives: [HeroCardComponent, HeroEditorComponent]
32. })
33. export class HeroesListComponent {
34. heroes: Array<EditItem<Hero>>;
35. constructor(heroesService: HeroesService) {
36. this.heroes = heroesService.getHeroes()
37. .map(item => new EditItem(item));
38. }
39. onSaved (editItem: EditItem<Hero>, updatedHero: Hero) {
40. editItem.item = updatedHero;
41. editItem.editing = false;
42. }
43. onCanceled (editItem: EditItem<Hero>) {
44. editItem.editing = false;
45. }
46. }

Notice that it imports the HeroService that we’ve used before so we can skip its declaration. The only difference is that we’ve used a more formal approach for our Heromodel and defined it upfront as such.

app/hero.ts

export class Hero {

name: string;

power: string;

}

Our HeroesListComponent defines a template that creates a list of HeroCardComponents and HeroEditorComponents, each bound to an instance of hero that is returned from the HeroService. Ok, that’s not entirely true. It actually binds to an EditItem<Hero> which is a simple generic datatype that can wrap any type and indicate if the item being wrapped is currently being edited or not.

app/edit-item.ts

export class EditItem<T> {

editing: boolean

constructor (public item: T) {}

}

But how is HeroCardComponent implemented? Let’s take a look.

app/hero-card.component.ts

1. import {Component, Input} from 'angular2/core';
2. import {Hero} from './hero';
3. @Component({
4. selector: 'hero-card',
5. template: `
6. <div>
7. <span>Name:</span>
8. <span>{{hero.name}}</span>
9. </div>`
10. })
11. export class HeroCardComponent {
12. @Input() hero: Hero;
13. }

The HeroCardComponent is basically a component that defines a template to render a hero. Nothing more.

Let’s get to the interesting part and take a look at the HeroEditorComponent

app/hero-editor.component.ts

1. import {Component, Input, Output, EventEmitter} from 'angular2/core';
2. import {RestoreService} from './restore.service';
3. import {Hero} from './hero';
4. @Component({
5. selector: 'hero-editor',
6. providers: [RestoreService],
7. template: `
8. <div>
9. <span>Name:</span>
10. <input [(ngModel)]="hero.name"/>
11. <div>
12. <button (click)="onSaved()">save</button>
13. <button (click)="onCanceled()">cancel</button>
14. </div>
15. </div>`
16. })
17. export class HeroEditorComponent {
18. @Output() canceled = new EventEmitter();
19. @Output() saved = new EventEmitter();
20. constructor(private restoreService: RestoreService<Hero>) {}
21. @Input()
22. set hero (hero: Hero) {
23. this.restoreService.setItem(hero);
24. }
25. get hero () {
26. return this.restoreService.getItem();
27. }
28. onSaved () {
29. this.saved.next(this.restoreService.getItem());
30. }
31. onCanceled () {
32. this.hero = this.restoreService.restoreItem();
33. this.canceled.next(this.hero);
34. }
35. }

Now here it’s getting interesting. The HeroEditorComponentdefines a template with an input to change the name of the hero and a canceland a save button. Remember that we said we want to have the flexibility to cancel our editing and restore the old value? This means we need to maintain two copies of our Hero that we want to edit. Thinking ahead this is a perfect use case to abstract it into it’s own generic service since we have probably more cases like this in our app.

And this is where the RestoreService enters the stage.

app/estore.service.ts

1. export class RestoreService<T> {
2. originalItem: T;
3. currentItem: T;
4. setItem (item: T) {
5. this.originalItem = item;
6. this.currentItem = this.clone(item);
7. }
8. getItem () :T {
9. return this.currentItem;
10. }
11. restoreItem () :T {
12. this.currentItem = this.originalItem;
13. return this.getItem();
14. }
15. clone (item: T) :T {
16. // super poor clone implementation
17. return JSON.parse(JSON.stringify(item));
18. }
19. }

All this tiny service does is define an API to set a value of any type which can be altered, retrieved or set back to it’s initial value. That’s exactly what we need to implement the desired functionality.

Our HeroEditComponent uses this services under the hood for it’s hero property. It intercepts the get and set method to delegate the actual work to our RestoreService which in turn makes sure that we won’t work on the original item but on a copy instead.

At this point we may be scratching our heads asking what this has to do with component injectors? If closely at the metadata for ourHeroEditComponent. Notice the providers property.

providers: [RestoreService],

This adds a RestoreService provider to the injector of the HeroEditComponent. Couldn’t we simply alter our bootstrap call to this?

1. // Don't do this!
2. bootstrap(HeroesListComponent, [HeroesService, RestoreService])

Technically we could, but our component wouldn’t quite behave the way it is supposed to. Remember that each injector treats the services that it provides as singletons. However, in order to be able to have multiple instances of HeroEditComponent edit multiple heroes at the same time we need to have multiple instances of the RestoreService. More specifically each instance of HeroEditComponent needs to be bound to it’s own instance of the RestoreService.

By configuring a provider for the RestoreService on the HeroEditComponent, we get exactly one new instance of theRestoreServiceper HeroEditComponent.

Does that mean that services aren’t singletons anymore in Angular 2? Yes and no. There can be only one instance of a service type in a particular injector. But we've learned that we can have multiple injectors operating at different levels of the application's component tree. Any of those injectors could have its own instance of the service.

If we defined a RestoreService provider only on the root component, we would have exactly one instance of that service and it would be shared across the entire application.

That’s clearly not what we want in this scenario. We want each component to have its own instance of the RestoreService. Defining (or re-defining) a provider at the component level creates a new instance of the service for each new instance of that component. We've made theRestoreService a kind of "private" singleton for each HeroEditComponent, scoped to that component instance and its child components.