@Input, @Output and EventEmitter in Angular

@Input:

Inputs provide a mechanism to allow a parent component to bind properties that a child component can have access to. The parent component pushes the properties to the child component.

In the parent component’s template, simply define property bindings in the child’s selector. The binding should point to properties available in the parent component’s class on the right side of the equal sign that you want to make available to the child component. As for the name of the binding itself (left side of the equal sign), it’s totally up to you:

Parent component template: story.component.html

<selected-story [story]="currentStory" [character]="mainCharacter">

</selected-story>

Parent component class: story.component.ts

export class StoryComponent {

currentStory: string = 'The Fox Without a Tail';

mainCharacter: string = 'Henry';

}

Now, in the child component, import Input from @angular/code and define your inputs with the @Input decorator like this:

Child component class: selected-story.component.ts

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

//...

export class SelectedStoryComponent {

@Input() story: string;

@Input('character') myCharacter: string;

}

Our child component now has access to the value of currentStory and mainCharacter from the parent component. Note how we aliased character to call the property myCharacter in the child component instead:

Child component template: selected-story.component.html

The story: {{ story }}

The character: {{ myCharacter }}

@Output:

Outputs provide a mechanism for a child component to emit events up to its parent component.

In the parent component’s template, define event bindings as part of the child component’s selector. A binding should point to a method defined in the parent component’s class that takes action on the data received from the child. $event contains the payload emitted from the child:

Parent component template: story.component.html

<selected-story (selectStory)="getStory($event)">

</selected-story>

Parent component class: story.component.ts

export class StoryComponent {

story: string = '';

getStory(story) {

this.story = story;

}

}

In the child component, import Output and EventEmitter from @angular/code and define your outputs with the @Output decorator like this:

Child component class: selected-story.component.ts

import { Component, Output, EventEmitter } from '@angular/core';

//...

export class SelectedStoryComponent {

story: string;

@Output() selectStory =new EventEmitter<string>();

onSelectStory(story: string) {

this.selectStory.emit(story);

}

}

Our EventEmitter object has an emit() method that pushes the event up to the parent component.

Now in the child component’s template, you can define event bindings that will emit back up to the parent:

Child component template: selected-story.component.html

<input #storyChoice placeholder="Your fav story">

<button (click)="onSelectStory(storyChoice.value)">

# [Angular2 - Interaction between components using a service](https://stackoverflow.com/questions/41958836/angular2-interaction-between-components-using-a-service)

Shared service:

@Injectable()

export class SharedService {

isVisibleSource: BehaviorSubject<boolean> = new BehaviorSubject(false);

constructor() { }

}

Component 1:

export class Component1 {

isVisible: boolean = false;

constructor(private sharedService: SharedService) { }

onClick(): void {

this.isVisible = !this.isVisible;

this.sharedService.isVisibleSource.next(this.isVisible);

}

}

Component 2:

export class Component2 {

constructor(private sharedService: SharedService) { }

ngOnInit() {

this.sharedService.isVisibleSource.subscribe((isVisible: boolean) => {

console.log('isVisible: ', isVisible); // => true/false

});

}

}

It is worth mentioning that BehaviorSubject upon a subscription returns the last value it holds, therefore the component from the example above will be updated with the most recent value immediately after the instantiation.

BehaviorSubject also allows to get its most recent value without even subscribing to it:

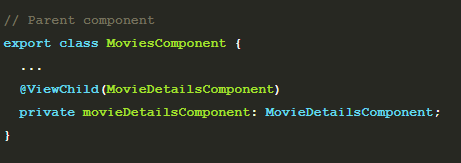
this.sharedService.isVisibleSource.getValue(); // => true/false

**ViewChild and ContentChild**

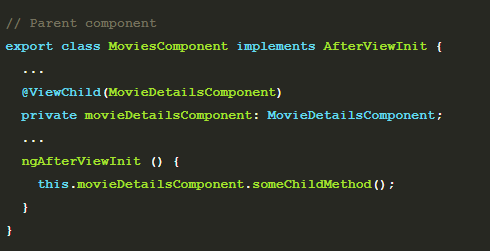
You can’t use the technique if an instance of the MoviesComponentclass itself (not its template) needs to read or write child component values, or if the parent component needs to be able to call child component methods directly. To accomplish this, you can either use a @ViewChild() or @ContentChild() decorator. Even though they are used in slightly different contexts, keep in mind that they both allow access to a child component’s public API (properties and methods) in the parent component’s class instance, just like a local variable does in the parent component’s template. Let’s take a look at how each decorator differs.

**View Child**

This decorator is used when the child component’s HTML tag is declared inside the parent component’s template, meaning the <movie-details> tag would have to be added inside the MoviesComponent template. From there, you’ll need to add a variable to reference the child component’s API inside the parent.

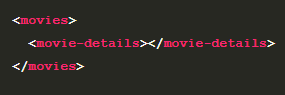


There is one additional caveat compared to the local variable technique in that the moviesDetailsComponent variable will only be set after the view has initialized. We’ll need to dive into the list of [lifecycle hooks](https://angular.io/docs/ts/latest/guide/lifecycle-hooks.html) and pull out the AfterViewInit interface. Once the interface is implemented on our parent component, we can add the ngAfterViewInit() method where we’ll finally have access to the movieDetailsComponent property.



**Content Child**

With the @ContentChild() decorator, we access the child component’s API when the child component’s HTML tag is declared in between the opening and closing tags of the parent component’s HTML tag. For our MoviesComponent and MovieDetailsComponent, it would look like



and again the child component property isn’t set at the parent component’s initialization. In this case, it will be after the content is initialized, so we’ll use the AfterContentInit lifecycle hook instead.



**JavaScript Modules vs. NgModules**

**The purpose of a NgModule is to declare each thing you create in Angular, and group them together** (like Java packages or PHP / C# namespaces).

There are two kinds of main structures:

* **“declarations” is for things you’ll use in your templates: mainly components** (~ views: the classes displaying data), but also directives and pipes,
* **“providers” is for services** (~ models: the classes getting and handling data).x

## **JavaScript modules**

In JavaScript, modules are individual files with JavaScript code in them. To make what’s in them available, you write an export statement, usually after the relevant code, like this:

export class AppComponent { ... }

Then, when you need that file’s code in another file, you import it like this:

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

## **NgModules**

NgModules are classes decorated with @[NgModule](https://angular.io/api/core/NgModule). The @[NgModule](https://angular.io/api/core/NgModule) decorator’s imports array tells Angular what other NgModules the current module needs. The modules in the imports array are different than JavaScript modules because they are NgModules rather than regular JavaScript modules. Classes with an @[NgModule](https://angular.io/api/core/NgModule) decorator are by convention kept in their own files, but what makes them an [NgModule](https://angular.io/api/core/NgModule) isn’t being in their own file, like JavaScript modules; it’s the presence of @[NgModule](https://angular.io/api/core/NgModule) and its metadata.

1. /\* These are JavaScript import statements. Angular doesn’t know anything about these. \*/
2. import { [BrowserModule](https://angular.io/api/platform-browser/BrowserModule) } from '@angular/platform-browser';
3. import { [NgModule](https://angular.io/api/core/NgModule) } from '@angular/core';
5. import { AppComponent } from './app.component';
7. /\* The @[NgModule](https://angular.io/api/core/NgModule) decorator lets Angular know that this is an NgModule. \*/
8. @[NgModule](https://angular.io/api/core/NgModule)({
9. declarations: [
10. AppComponent
11. ],
12. imports: [ /\* These are [NgModule](https://angular.io/api/core/NgModule) imports. \*/
13. [BrowserModule](https://angular.io/api/platform-browser/BrowserModule)
14. ],
15. providers: [],
16. bootstrap: [AppComponent]
17. })
18. export class AppModule { }

The NgModule classes differ from JavaScript module in the following key ways:

* An NgModule bounds [declarable classes](https://angular.io/guide/ngmodule-faq#q-declarable) only. Declarables are the only classes that matter to the [Angular compiler](https://angular.io/guide/ngmodule-faq#q-angular-compiler).
* Instead of defining all member classes in one giant file as in a JavaScript module, you list the module's classes in the @[NgModule.declarations](https://angular.io/api/core/NgModule#declarations) list.
* An NgModule can only export the [declarable classes](https://angular.io/guide/ngmodule-faq#q-declarable) it owns or imports from other modules. It doesn't declare or export any other kind of class.
* Unlike JavaScript modules, an NgModule can extend the entire application with services by adding providers to the @[NgModule.providers](https://angular.io/api/core/NgModule#providers) list.

An NgModule describes how the application parts fit together. Every application has at least one Angular module, the rootmodule that you bootstrap to launch the application. By convention, it is usually called AppModule.

the default AppModule is as follows:

1. /\* JavaScript imports \*/
2. import { [BrowserModule](https://angular.io/api/platform-browser/BrowserModule) } from '@angular/platform-browser';
3. import { [NgModule](https://angular.io/api/core/NgModule) } from '@angular/core';
4. import { [FormsModule](https://angular.io/api/forms/FormsModule) } from '@angular/forms';
5. import { [HttpClientModule](https://angular.io/api/common/http/HttpClientModule) } from '@angular/common/[http](https://angular.io/api/common/http)';
7. import { AppComponent } from './app.component';
9. /\* the AppModule class with the @[NgModule](https://angular.io/api/core/NgModule) decorator \*/
10. @[NgModule](https://angular.io/api/core/NgModule)({
11. declarations: [
12. AppComponent
13. ],
14. imports: [
15. [BrowserModule](https://angular.io/api/platform-browser/BrowserModule),
16. [FormsModule](https://angular.io/api/forms/FormsModule),
17. [HttpClientModule](https://angular.io/api/common/http/HttpClientModule)
18. ],
19. providers: [],
20. bootstrap: [AppComponent]
21. })
22. export class AppModule { }

## **The declarations array**

declarations specifies the components that are **defined in this module**. This is an important idea in Angular:

**You have to declare components in a NgModule before you can use them in your templates**.

You can think of an NgModule a bit like a “package” and declarations states **what components are “owned by” this module**

You may have noticed that when we used ng generate, the tool automatically added our components to this declarations list! The idea is that when we generated a new component, the ng tool assumed we wanted it to belong to the current NgModule.

## **The imports array**

imports describes which *dependencies* this module has. We’re creating a browser app, so we want to import the BrowserModule.

If your module depends on other modules, you list them here.

## **The providers array**

The providers array is where you list the services the app needs. When you list services here, they are available app-wide. You can scope them when using feature modules and lazy loading.

## **The bootstrap array**

The application launches by bootstrapping the root AppModule, which is also referred to as an entryComponent. Among other things, the bootstrapping process creates the component(s) listed in the bootstrap array and inserts each one into the browser DOM.

Each bootstrapped component is the base of its own tree of components. Inserting a bootstrapped component usually triggers a cascade of component creations that fill out that tree.

While you can put more than one component tree on a host web page, most applications have only one component tree and bootstrap a single root component.

This one root component is usually called AppComponent and is in the root module's bootstrap array.

**Why NgModule?**

It’s done automatically with [Angular CLI](https://github.com/angular/angular-cli), but the first thing you have to do in Angular is to load a root [NgModule](https://angular.io/guide/ngmodules) :

**The purpose of a NgModule is to declare each thing you create in Angular, and group them together** (like Java packages or PHP / C# namespaces).

There is two kind of main structures:

* **“declarations” is for things you’ll use in your templates: mainly components** (~ views: the classes displaying data), but also directives and pipes,
* **“providers” is for services** (~ models: the classes getting and handling data).

## **What classes should I add to the declarations array?**

Add [declarable](https://angular.io/guide/bootstrapping#the-declarations-array) classes—components, directives, and pipes—to a declarations list.

Declare these classes in exactly one module of the application. Declare them in a module if they belong to that particular module.

## **What is a declarable?**

Declarables are the class types—components, directives, and pipes—that you can add to a module's declarations list. They're the only classes that you can add to declarations.

## **What classes should I not add to declarations?**

Add only [declarable](https://angular.io/guide/bootstrapping#the-declarations-array) classes to an NgModule's declarations list.

Do not declare the following:

* A class that's already declared in another module, whether an app module, @NgModule, or third-party module.
* An array of directives imported from another module. For example, don't declare FORMS\_DIRECTIVES from @angular/forms because the [FormsModule](https://angular.io/api/forms/FormsModule) already declares it.
* Module classes.
* Service classes.
* Non-Angular classes and objects, such as strings, numbers, functions, entity models, configurations, business logic, and helper classes.

**The NgModule classes differ from JavaScript module in the following key ways:**

A JavaScript Module is a code written in JavaScript which means to organize it and this module comes into action by making use of an export statement like this:

export class MyComponent {}

An NgModule is the use of a module decorated by the decorator @NgModule. The imports array inside the decorator tells Angular that these modules will be needed for the current scenario

An NgModule differs from the JS Modules specifically because unlike JS Modules, NgModules do not have their own file. Also they need the presence of the decorator @NgModule and its metadata.

An NgModule has a declarations list wherein it defines all the member classes and it bounds declarable classes.

Bootstrapping

Every app has a main entry point. This application was built using Angular CLI (which is built on a tool called Webpack). We run this app by calling the command:

ng serve

ng will look at the file .angular-cli.json to find the entry point to our app. Let’s trace how ng finds the components we just built.

At a high level, it looks like this:

• .angular-cli.json specifies a "main" file, which in this case is main.ts

"main": "src/main.ts",

• main.ts is the entry-point for our app and it *bootstraps* our application

platformBrowserDynamic().bootstrapModule(AppModule)

• The bootstrap process boots **an Angular module**

• We use the AppModule to bootstrap the app. AppModule is specified in src/app/app.module.ts

bootstrap: [AppComponent]

• AppModule specifies which *component* to use as the top-level component. In this case it is AppComponent

• AppComponent has <app-user-list> tags in the template and this renders our list of users.

Feature Modules

Feature modules are NgModules for the purpose of organizing code.

As your app grows, you can organize code relevant for a specific feature. This helps apply clear boundaries for features. With feature modules, you can keep code related to a specific functionality or feature separate from other code. Delineating areas of your app helps with collaboration between developers and teams, separating directives, and managing the size of the root module.

A feature module is an organizational best practice, as opposed to a concept of the core Angular API. A feature module delivers a cohesive set of functionality focused on a specific application need such as a user workflow, routing, or forms. While you can do everything within the root module, feature modules help you partition the app into focused areas. A feature module collaborates with the root module and with other modules through the services it provides and the components, directives, and pipes that it shares.

# CoreModule vs SharedModule

# NgModule and scopes / visibility

The confusion starts with **components and services not having the same scope / visibility**:

* declarations / **components are in local scope** (private visibility),
* providers / **services are (generally) in global scope** (public visibility).

It means the **components you declared are only usable in the current module**. If you need to use them outside, in other modules, you’ll have to export them:

* **if the module is imported for components, you’ll need to import it in each module** needing them,
* **if the module is imported for services, you’ll need to import it only once**, in the first app module.

#### Modules to import each time you need them

* **CommonModule** (all the basics of Angular templating: bindings, \*ngIf, \*ngFor…), except in the first app module, because it’s already part of the BrowserModule
* FormsModule / ReactiveFormsModule
* MatXModule and other UI modules
* any other module giving you components, directives or pipes

#### Modules to import only once

* **HttpClientModule**
* BrowserAnimationsModule or NoopAnimationsModule
* any other module providing you services only.

# Core module or Core feature module

# The core module is a module that is **only imported once in the AppModule** and never again in the other modules

So why do we need to import that famous module only once ? The reason behind this is that we want everything that’s inside the core module to be a **Singleton**!!! And this is very important if you need your components/services to have only one instance.

**Shared module or Shared feature module**

The SharedModule in contradiction with the CoreModule is imported in every feature module that needs some shared components.

It’s recommended to avoid having services in the SharedModule because you will end up with a lot of instances of that service.

The SharedModule is the perfect place for importing and exporting back your UI Modules or components that are used a lot in your application. This will make your code more readable and maintainable, some good examples of the SharedModule use case are importing and exporting Angular Material modules and/or the Flex Layout Module. By doing this, you only have to import the SharedModule in your feature Module and voila !! All your imported Angular Material Modules/components are available and the import bloc on top of your file is much much smaller.

## **SharedModule**

* Create a SharedModule with the components, directives, and pipes that you use everywhere in your app. This module should consist entirely of declarations, most of them exported.
* The SharedModule may re-export other widget modules, such as CommonModule, FormsModule, and modules with the UI controls that you use most widely.
* The SharedModule should not have providers for reasons explained previously. Nor should any of its imported or re-exported modules have providers. If you deviate from this guideline, know what you're doing and why.
* Import the SharedModule in your feature modules, both those loaded when the app starts and those you lazy load later.

## **CoreModule**

* Create a CoreModule with providers for the singleton services you load when the application starts.
* Import CoreModule in the root AppModule only. Never import CoreModule in any other module.
* Consider making CoreModule a pure services module with no declarations

# Looping over object properties with ngFor in Angular

The first solution is to simply map object keys to a class property in the constructor function. Rather than iterating over properties in the object, the template for the ngFor will iterate over the keys, binding the repeated template to the current key in the array.

**export** **class** **IterateOverObject** {

public arrayOfKeys;

@Input dataObject;

**constructor**() {

**this**.arrayOfKeys = Object.keys(**this**.dataObject);

}

}

Since that exposes a string value for us to use within the repeated template, we can use it as a key mapping for binding and displaying object data. Each item in the array allows us to map to its corresponding property in the dataObject.

<**div** \*ngFor='key of arrayOfKeys'>

<**h3**>{{dataObject[key].someProperty}}</**h3**>

<**p**>{{dataObject[key].anotherPropery}}</**p**>

</**div**>

**VIEW ENCAPSULATION IN ANGULAR**

**Understanding Shadow DOM**

One common problem these brave soldiers of the Web have to face is encapsulation. You know, one of them turtles on which the Object-Oriented Programming foundation sits, upon which stands most of the modern software engineering. How do you create that boundary between the code that you wrote and the code that will consume it?

With the exception of SVG (more on that later), today’s Web platform offers only one built-in mechanism to isolate one chunk of code from another — and it ain’t pretty. Yup, I am talking about [iframes](http://www.whatwg.org/specs/web-apps/current-work/multipage/the-iframe-element.html#the-iframe-element). For most encapsulation needs, frames are too heavy and restrictive.

*What do you mean I must put each of my custom buttons in a separate iframe? What kind of insane are you?*

So we need something better. Turns out, most browsers have been sneakily employing a powerful technique to hide their gory implementation details. This technique is called the shadow DOM.

**My name is DOM, Shadow DOM**

Shadow DOM refers to the ability of the browser to include a subtree of DOM elements into the rendering of a document, but not into the main document DOM tree. Consider a simple slider:

<input id="foo" type="range">

Simple enough. There’s a slider track and there’s a thumb, which you can slide along the track.

Wait, what? There’s a separate movable element inside of the input element? How come I can’t see it from Javascript?

var slider = document.getElementsById("foo");

console.log(slider.firstChild); // returns null

Is this some sort of magic?

No magic, my fair person of the Web. Just shadow DOM in action. You see, browser developers realized that coding the appearance and behavior of HTML elements completely by hand is a) hard and b) silly. So they sort of cheated.

They created a boundary between what you, the Web developer can reach and what’s considered implementation details, thus inaccessible to you. The browser however, can traipse across this boundary at will. With this boundary in place, they were able to build all HTML elements using the same good-old Web technologies, out of the divs and spans just like you would.

Some of these are simple, like the slider above. Some get pretty complex. Check out the video element. It’s got trigger buttons, timelines, a hover-appearing volume control, you name it:

**Shadow DOM allows hidden DOM trees to be attached to elements in the regular DOM tree — this shadow DOM tree starts with a shadow root, underneath which can be attached to any elements you want, in the same way as the normal DOM.**

There are some bits of shadow DOM terminology to be aware of:

* **Shadow host**: The regular DOM node that the shadow DOM is attached to.
* **Shadow tree**: The DOM tree inside the shadow DOM.
* **Shadow boundary**: the place where the shadow DOM ends, and the regular DOM begins.
* **Shadow root**: The root node of the shadow tree.

Note that the shadow DOM is not a new thing by any means — browsers have used it for a long time to encapsulate the inner structure of an element. Think for example of a [<video>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/video)element, with the default browser controls exposed. All you see in the DOM is the <video>element, but it contains a series of buttons and other controls inside its shadow DOM. The shadow DOM spec has made it so that you are allowed to actually manipulate the shadow DOM of your own custom elements.

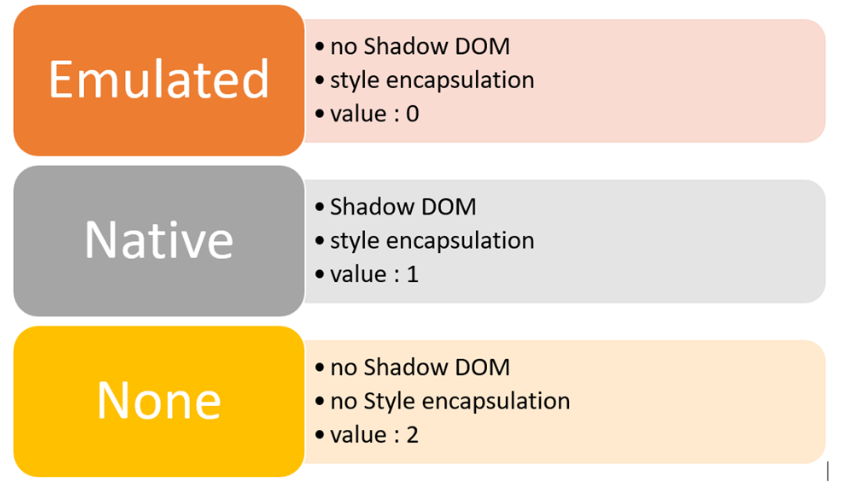
## [View Encapsulation Types](https://blog.thoughtram.io/angular/2015/06/29/shadow-dom-strategies-in-angular2.html#view-encapsulation-types)

Angular Components are made up of three things:

1. Component Class
2. Template
3. Style

The combination of these three factors makes an Angular component reusable across an application. Theoretically, when you create a component, in some way you create a web component (**theoretically, Angular Components are not web components**) to take advantage of the Shadow DOM. **You can also use Angular with browsers, which does not support the Shadow DOM because Angular has its own emulation and it can emulate(imitate) the Shadow DOM**.

To emulate the Shadow DOM and encapsulate styles, Angular provides there types of View Encapsulation. They are as follows:



Let's try to understand it using an example. I have created a component, as shown below:

#### app.component.ts

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

@Component({

selector: 'app-root',

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

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

encapsulation: ViewEncapsulation.None

})

export class AppComponent {

title = 'parent component';

}

#### app.component.html app.component.css

h1 {

background: red;

color: white;

text-transform: uppercase;

text-align: center;

}

We are setting the style of h1 in the component CSS. We have also created another component:

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

@Component({

selector: 'app-child',

template: `

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

`

})

export class AppChildComponent {

title = 'child app';

}

In AppChildComponent, we are also using the h1 tag. To understand different ViewEncapsulation options, we will change the metadata of AppComponent.

**ViewEncapsulation.None**

1. There is no shadow DOM.
2. Style is not scoped to the component.

As you run the application, you will find h1 style has applied to both components, even though we only set style the in AppComponent. This happened because in AppComponent we have set the encapsulation property to ViewEncapsulation.None.

@Component({

selector: 'app-root',

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

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

encapsulation: ViewEncapsulation.None

})

export class AppComponent {

title = 'parent component';

}

In the browser, when you examine source code, you will find the h1 style has been declared in the head section of the DOM.

Therefore, in ViewEncapsulation.None, the style gets moved to the DOM's head section and is not scoped to the component. There is no Shadow DOM for the component and the component style can affect all nodes in the DOM.

**ViewEncapsulation.Native**

1. Angular will create Shadow DOM for the component.
2. Style is scoped to the component.

As you run the application, you will find the h1 style has applied to both components, even though we only set the style only in AppComponent. This happened because in AppComponent we have set the encapsulation property to ViewEncapsulation.Native, and we are using AppChildComponnet as a child inside the template of AppComponent.

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

@Component({

selector: 'app-root',

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

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

encapsulation: ViewEncapsulation.Native

})

export class AppComponent {

title = 'parent component';

}

In the browser, when you examine source code, you will a Shadow DOM has been created for the AppComponent and the style is scoped to that.



Therefore, in ViewEncapsulation.Native, Angular creates a Shadow DOM and the style is scoped to that Shadow DOM.

**ViewEncapsulation.Emulated**

1. Angular will not create a Shadow DOM for the component.
2. Style will be scoped to the component.

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

@Component({

selector: 'app-root',

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

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

encapsulation: ViewEncapsulation.Emulated

})

export class AppComponent {

title = 'parent component';

}

As you run the application, you will find that the h1 style from AppComponent is not applied to the h1 of the AppChildComponent. This is due to emulated scoping. In this, the style is scoped only to the component. In this option, Angular only emulates the Shadow DOM and does not create a real shadow DOM. Hence, the application that runs in browsers does not support a Shadow DOM also and styles are scoped to the component as well.

Let us see how Angular achieves this. In the browser, when you examine source code, you will find the answer.



Angular has created the style in the head section of the DOM and given an arbitrary id to the component. On basis of ID, selector style is scoped to the component.

# Routing

Routes & RouterModule

First we need to setup some imports, like so:

import {Routes, RouterModule} from "@angular/router";

The mapping of URLs to *Components* we want displayed on the page is done via something called a *Route Configuration*, at it’s core it’s just an array which we can define like so:

const routes: Routes = [

{ path: '', component: HomeComponent },

{ path: 'search', component: SearchComponent }

];

* The path property describes the URL this route will handle.
* The component property is the name of the component we want to display when the URL in the browser matches this path.
* We then *install* these routes into our application by importing RouterModule.forRoot(routes) into our NgModule, like so:

NgModule({

imports: [

RouterModule.forRoot(routes, {useHash: true})]

})

class AppModule { }

# RouterOutlet Directive

We need to add a directive called router-outlet somewhere in our template HTML. This directive tells Angular *where* it should insert each of those components in the route, we’ll add ours to the AppComponent, like so:

<app-header></app-header>

<div class="m-t-1">

<router-outlet></router-outlet> (1)

</div>

# Redirects

There are a few more ways to configure our routes, for example we might like to change our routes to add some redirects like so:

const routes:Routes = [

{path: '', redirectTo: 'home', pathMatch: 'full'}, (1)

{path: 'find', redirectTo: 'search'}, (1)

{path: 'home', component: HomeComponent},

{path: 'search', component: SearchComponent}

];

|  |  |
| --- | --- |
|  | The redirectTo property describes the path we want to redirect this user to if they navigate to this URL. |

Now if the user visits the root (empty) URL they are redirected to /home instead.

For the special case of an empty URL we also need to add the pathMatch: 'full' property so Angular knows it should be matching exactly the empty string and not partially the empty string.

We’ve also added a redirect from /find to /search, since this isn’t empty we don’t need to add the pathMatch property.

# Catch all route

We can also add a *catch all* route by using the path \*\*, if the URL doesn’t match *any* of the other routes it will match this route.

const routes:Routes = [

{path: '', redirectTo: 'home', pathMatch: 'full'},

{path: 'find', redirectTo: 'search'},

{path: 'home', component: HomeComponent},

{path: 'search', component: SearchComponent},

{path: '\*\*', component: HomeComponent} (1)

];

### **Router links**

Now you have routes configured and a place to render them, but how do you navigate? The URL could arrive directly from the browser address bar. But most of the time you navigate as a result of some user action such as the click of an anchor tag.

Consider the following template:

<nav>

<[a](https://angular.io/api/router/RouterLinkWithHref) [routerLink](https://angular.io/api/router/RouterLink)="/crisis-center" [routerLinkActive](https://angular.io/api/router/RouterLinkActive)="active">Crisis Center</[a](https://angular.io/api/router/RouterLinkWithHref)>

<[a](https://angular.io/api/router/RouterLinkWithHref) [routerLink](https://angular.io/api/router/RouterLink)="/heroes" [routerLinkActive](https://angular.io/api/router/RouterLinkActive)="active">Heroes</[a](https://angular.io/api/router/RouterLinkWithHref)>

</nav>

<[router-outlet](https://angular.io/api/router/RouterOutlet)></[router-outlet](https://angular.io/api/router/RouterOutlet)>

The [RouterLink](https://angular.io/api/router/RouterLink) directives on the anchor tags give the router control over those elements. The navigation paths are fixed, so you can assign a string to the [routerLink](https://angular.io/api/router/RouterLink) (a "one-time" binding).

Had the navigation path been more dynamic, you could have bound to a template expression that returned an array of route link parameters (the link parameters array). The router resolves that array into a complete URL.

### **Active router links**

The [RouterLinkActive](https://angular.io/api/router/RouterLinkActive) directive toggles css classes for active [RouterLink](https://angular.io/api/router/RouterLink) bindings based on the current [RouterState](https://angular.io/api/router/RouterState).

On each anchor tag, you see a [property binding](https://angular.io/guide/template-syntax#property-binding) to the [RouterLinkActive](https://angular.io/api/router/RouterLinkActive) directive that look like [routerLinkActive](https://angular.io/api/router/RouterLinkActive)="...".

The template expression to the right of the equals (=) contains a space-delimited string of CSS classes that the Router will add when this link is active (and remove when the link is inactive). You set the [RouterLinkActive](https://angular.io/api/router/RouterLinkActive) directive to a string of classes such as [[routerLinkActive](https://angular.io/api/router/RouterLinkActive)]="'active fluffy'" or bind it to a component property that returns such a string.

### What are Route Guards?

Angular’s route guards are interfaces which can tell the router whether or not it should allow navigation to a requested route. They make this decision by looking for a true or false return value from a class which implements the given guard interface.

The guards are:

* CanActivate
* CanActivateChild
* CanDeactivate
* CanLoad
* Resolve

**CanActivate**

Checks to see if a user can visit a route.

**CanActivateChild**

Checks to see if a user can visit a routes children.

**CanDeactivate**

Checks to see if a user can exit a route.

**Resolve**

Performs route data retrieval before route activation.

**CanLoad**

Checks to see if a user can route to a module that lazy loaded.

## [CanActivate](https://codecraft.tv/courses/angular/routing/router-guards/#_canactivate)

Guards are implemented as services that need to be provided so we typically create them as @Injectableclasses.

Guards return either true if the user can access a route or false if they can’t.

They can also return an Observable or Promise that later on resolves to a boolean in case the guard can’t answer the question straight away, for example it might need to call an API. Angular will keep the user waiting until the guard returns true or false.

First we need to import the CanActivate interface, like so:

import {CanActivate} from "@angular/router";

Then lets create an Injectable class called AlwaysAuthGuard which implements the canActivate function, like so:

class AlwaysAuthGuard implements CanActivate {

canActivate() {

console.log("AlwaysAuthGuard");

return true;

}

}

This guard returns true all the time, so doesn’t really guard anything. It lets all users through but at the same time our guard logs "AlwaysAuthGuard" to the console so we can at least see when it’s being used.

We need to provide this guard, for this example lets configure it via our NgModule, like so:

@NgModule({

.

.

providers: [

.

.

AlwaysAuthGuard

]

})

Finally we need to add this guard to one or more of our routes, lets add it to our ArtistComponent route like so:

const routes: Routes = [

{path: '', redirectTo: 'home', pathMatch: 'full'},

{path: 'find', redirectTo: 'search'},

{path: 'home', component: HomeComponent},

{path: 'search', component: SearchComponent},

{

path: 'artist/:artistId',

component: ArtistComponent,

canActivate: [AlwaysAuthGuard], (1)

children: [

{path: '', redirectTo: 'tracks'},

{path: 'tracks', component: ArtistTrackListComponent},

{path: 'albums', component: ArtistAlbumListComponent},

]

},

{path: '\*\*', component: HomeComponent}

];

|  |  |
| --- | --- |
|  | We added our AlwaysAuthGuard to the list of canActivate guards for this route. |

#### **Note**

Since it holds an array we could have multiple guards for a single route.

#### **Note**

If this was a canActivateChild guard we would be adding it to the canActivateChild property and so on for the other guard types.

Now every-time we navigate to the ArtistComponent route we get "AlwaysAuthGuard" printed to the console so we know that the AlwaysAuthGuard is working.

The most typical use case for the CanActivate guard is some form of checking to see if the user has permissions to view a page.

Normally in an Angular application we would have a service which held whether or not the current user is logged in or what permissions they have.

We will simulate this via a mock UserService like so:

class UserService {

isLoggedIn(): boolean {

return false;

}

}

This service has one function isLoggedIn() which always returns false.

Lets create another guard called OnlyLoggedInUsersGuard which only allows logged in users to view a route.

@Injectable()

class OnlyLoggedInUsersGuard implements CanActivate { (1)

constructor(private userService: UserService) {}; (2)

canActivate() {

console.log("OnlyLoggedInUsers");

if (this.userService.isLoggedIn()) { (3)

return true;

} else {

window.alert("You don't have permission to view this page"); (4)

return false;

}

}

}

|  |  |
| --- | --- |
|  | We create a new CanActivate guard called OnlyLoggedInUsersGuard |
|  | We inject and store UserService into the constructor for our class. |
|  | If the user is logged in the guard passes and lets the user through. |
|  | If the user is *not* logged in the guard fails, we show the user an alert and the page doesn’t navigate to the new URL. |

Finally we need to add this guard to the list of guards for our search route, like so:

Copy{

path: 'artist/:artistId',

component: ArtistComponent,

canActivate: [OnlyLoggedInUsersGuard, AlwaysAuthGuard], (1)

children: [

{path: '', redirectTo: 'tracks'},

{path: 'tracks', component: ArtistTrackListComponent},

{path: 'albums', component: ArtistAlbumListComponent},

]

}

## [**CanActivateChild**](https://codecraft.tv/courses/angular/routing/router-guards/#_canactivatechild)

As we learned about guarding routes with CanActivate, we can also protect child routes with the CanActivateChild guard. The CanActivateChild guard works similarly to the CanActivate guard, but **the difference is its run before each child route is activated**. We protected our admin feature module from unauthorized access, but we could also **protect child routes within our feature module**.

### **Here's a practical example:**

1. navigating to /admin
2. canActivate is checked
3. You navigate between the children of /admin route, but canActivate isn't called because it protects /admin
4. canActivateChild is called whenever changing between children of the route its defined on.

I hope this helps you, if still unclear, you can check specific functionality by adding guards debugging them.

1. @[Injectable](https://angular.io/api/core/Injectable)()
2. class CanActivateTeam implements [CanActivateChild](https://angular.io/api/router/CanActivateChild) {
3. constructor(private permissions: Permissions, private currentUser: UserToken{}
4. [canActivateChild](https://angular.io/api/router/Route#canActivateChild)(
5. route: [ActivatedRouteSnapshot](https://angular.io/api/router/ActivatedRouteSnapshot),
6. [state](https://angular.io/api/animations/state): [RouterStateSnapshot](https://angular.io/api/router/RouterStateSnapshot)
7. ): Observable<boolean|[UrlTree](https://angular.io/api/router/UrlTree)>|Promise<boolean|[UrlTree](https://angular.io/api/router/UrlTree)>|boolean|[UrlTree](https://angular.io/api/router/UrlTree) {
8. return this.permissions.canActivate(this.currentUser, route.params.id);}}
10. @[NgModule](https://angular.io/api/core/NgModule)({
11. imports: [
12. RouterModule.forRoot([
13. {
14. path: 'root',
15. [canActivateChild](https://angular.io/api/router/Route#canActivateChild): [CanActivateTeam],
16. children: [
17. {
18. path: 'team/:id',
19. component: TeamComponent}]}])],
20. providers: [CanActivateTeam, UserToken, Permissions]})

## [**CanDeactivate**](https://codecraft.tv/courses/angular/routing/router-guards/#_candeactivate)

A third type of guard we can add to our application is a CanDeactivate guard which is usually used to warn people if they are navigating away from a page where they have some unsaved changes.

Firstly lets create a function called canDeactivate on our SearchComponent, it should be the component that decides whether or not it has unsaved changes.

canDeactivate() {

return this.itunes.results.length > 0;

}

As a proxy for unsaved changes we are just seeing if the user has performed a search, if so then the results array should be > 0.

Next lets create a CanDeactivate guard.

class UnsearchedTermGuard implements CanDeactivate<SearchComponent> { (1)

canDeactivate(component: SearchComponent, (1)

route: ActivatedRouteSnapshot,

state: RouterStateSnapshot): boolean {

console.log("UnsearchedTermGuard");

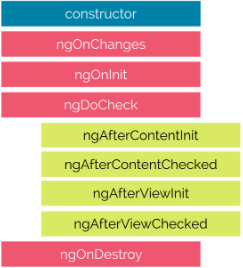
console.log(route.params);

console.log(state.url);

return component.canDeactivate() || window.confirm("Are you sure?");

}}

**Lifecycle Hooks in Angular**



**Constructor:**

The constructor of the component class is called before any other component lifecycle hook. If our component is based on any dependencies, the constructor is the best place to inject those dependencies.

**ngOnInit:**

The ngOnInit method of a component is called directly after the constructor and after the ngOnChange is triggered for the first time. It is the perfect place for initialization work.

**ngOnChanges:**

The ngOnChanges will be called first when the value of a bound property changes. It executes, every time the value of an input property changes. It will receive a changes map, containing the current and previous values of the binding, wrapped in a SimpleChange.

{"brand":{"previousValue":"","currentValue":"BMW"}}

In the case above, one change to the input property brand is reported. The value of this property has been changed from an empty string to the string “BMW”.

**ngOnDestroy:**

The ngDestroy is called in a component’s lifecycle just before the instance of the component is finally destroyed. It is the perfect place to clean the component — for example, to cancel background tasks.

Quick example:

@Directive({

selector: '[destroyDirective]'

})

export class OnDestroyDirective implements OnDestroy {

sayHello: number;

constructor() {

this.sayHiya = window.setInterval(() => console.log('hello'), 1000);

}

ngOnDestroy() {

window.clearInterval(this.sayHiya);

}

}

If we do not use the ngOnDestroy method we will have the thread logging “hello” until the end or it crashes….

More advanced phases are:

**ngDoCheck:**

ngDoCheck is triggered every time the input properties of a component or a directive are checked. We can use this lifecycle hook to extend the check with our own custom check logic. It can also be useful if we want to accelerate the change detection by checking the bare minimum and not using the default algorithm (although we usually do not use this).

**ngAfterContentInit:**

The ngAfterContentInit lifecycle hook is called after ngOnInit when the component or directive’s content has been initialised; basically when all the bindings of the component have been checked for the first time.

**ngAfterContentChecked:**

Called after every check of the component or directive’s content, effectively when all the bindings of the components have been checked; even if they haven’t changed.

**ngAfterViewInit:**

Called after ngAfterContentInit when the component’s view has been initialised. Applies to components only.

**ngAfterViewChecked:**

Called after every check of the component’s view. Applies to components only. When all the bindings of the children directives have been checked; even if they haven’t changed. It can be useful if the component is waiting for something coming from its child components.

**ngOnInit**

Let’s give you a simple example using the ngOnInit hook. The ngOnInit lifecycle hook is probably the one you’ll use most often. If you have a lot of processing to do when the component gets created, it’s good practice to do it in the ngOnInithook rather than in the constructor:

@Component({

selector: 'my-app',

templateUrl: './app.component.html'

})

export class AppComponent implements OnInit {

constructor() {}

ngOnInit() {

this.setupData();

this.doStuff();

// ...

}

setupData() {

// ...

}

doStuff() {

// ...

}

}

**ngOnChanges**

The ngOnChanges hook, with it’s SimpleChanges object, is a little different. Here’s how you would implement it. Let’s say we have a component used like this:

<my-todo [title]="title" [content]="content"></my-todo>

Now say that we want to do something when the title property changes:

import { Component, Input, SimpleChanges, OnChanges }

from '@angular/core';

@Component({

// ...

})

export class MyTodoComponent implements OnChanges {

@Input() title: string;

@Input() content: string;

constructor() { }

ngOnChanges(changes: SimpleChanges) {

for (let property in changes) {

if (property === 'title') {

console.log('Previous:', changes[property].previousValue);

console.log('Current:', changes[property].currentValue);

}

} }}

# [2 services depending on each other](https://stackoverflow.com/questions/36378751/angular2-2-services-depending-on-each-other)

This is a called circular dependency. It is not an issue with Angular2 itself. It is not allowed in any language I am aware of.

You will need to refactor your code to remove this circular dependency. Likely you will need to breakup one of these services into new service.

If you follow the single responsibility principle you will find you won't get into circular dependency trap.

Constructor injection prevents circular dependencies.

It can be broken up by injecting the Injector and requesting a dependency imperatively like:

private payrollService:PayrollService;

constructor(/\*private payrollService:PayrollService\*/ injector:Injector) {

setTimeout(() => this.payrollService = injector.get(PayrollService));}

# Lazy Loading Angular Modules

initialize an empty application:

ng new lazy-test –routing

With this command we are telling the CLI to create for us an empty project with an additional routing module already set up for us. Wonderful! This is what it looks like:



**Creating the components**

Let’s create our first component, which will be loaded normally.

cd lazy-test  
ng generate component first

This will generate a *first* folder which will contain all of our new component’s files, and the **FirstComponent**will be automatically imported in our **AppModule**. Let’s go into *app-routing.module.ts* and add this component to the main route:



*We are using pathMatch: ‘full’ because our route is an empty one, and we don’t want it to match any other route. In order to clear some stuff, go to app.component.html and delete everything but <router-outlet></router-outlet>, which is where our components will be displayed by the router.*

### Creating a lazy-loaded module

Now we can create an additional module for this app, which will be lazy-loaded. Let’s use the CLI to achieve this, with this command:

ng generate module lazy

We also want to create a component which will be the **root component** for our new lazy-loaded module: we just tell the CLI the path to our module and it will create our files for us, importing them in the correct module:

ng generate component lazy/second

We just need to declare a new set of routes for our lazy module, which will display our **SecondComponent**.



Here, we’re using an empty path because **these will be the relative routes for this module, not for the entire application**. Also, for the same reason, we use RouterModule.forChild() instead of forRoot().

### Lazy-loading the new module

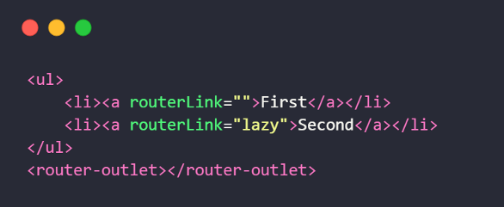
Now we need to go to app-routing.module.ts and create another route for our new LazyModule: we won’t specify any component! Instead, we’ll specify the path to the module, followed by the name of the module’s class with a hashtag. Also, we’ll use the special keyword **loadChildren**:



***Don’t****import LazyModule for any reason! If you do that, you’ll lose all the advantages of lazy-loading and you may have serious problems with debugging: specifying the module’s path is all we need to do to associate it with our route, let Angular do the rest.*

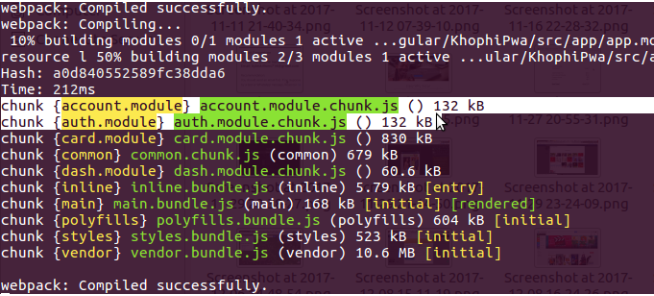
### A simple navbar

In order to see our new routing system in action, let’s go to app.component.html and modify it slightly:



For truth’s sake, we can open the browser inspector and go to the Network tab: you’ll see that once you navigate to the lazy route, a new **chunk.js** file will be loaded asynchronously: **that’s our LazyModule in action**.

When you ng serve or ng build --prod you should see the modules as chunks. Like this:



# Working with Angular 5 Template Reference Variables

A template reference variable is often a reference to a DOM element within a template. It can also be a reference to an Angular component or directive or a web component (Read more at [Angular.io](https://angular.io/guide/template-syntax#ref-vars)). That means you can easily access the variable anywhere in the template.

You declare a reference variable by using the hash symbol (#). The **#firstNameInput** declares a **firstNameInput**variable on an <input>element.

<input type="text" #firstNameInput>

<input type="text" #lastNameInput>

After that, you can access the variable anywhere inside the template. For example, I pass the variable as a parameter on an event.

<button (click)="show(lastNameInput)">Show</button>

Remember that the lastNameInput belongs to HTMLInputElement type.

show(lastName: HTMLInputElement){  
 console.log(lastName.value);  
}

Usually, the reference variable can only be accessed inside the template. However, you can use **ViewChild**decorator to reference it inside your component.

import {ViewChild, ElementRef} from '[@angular/core](http://twitter.com/angular/core)';

// Reference firstNameInput variable inside Component  
[@ViewChild](http://twitter.com/ViewChild)('firstNameInput') nameInputRef: ElementRef;

After that, you can use **this.nameInputRef** anywhere inside your Component.

show(lastName: HTMLInputElement){  
 this.fullName = this.nameInputRef.nativeElement.value + ' ' + lastName.value;  
}

**1. <ng-template>**

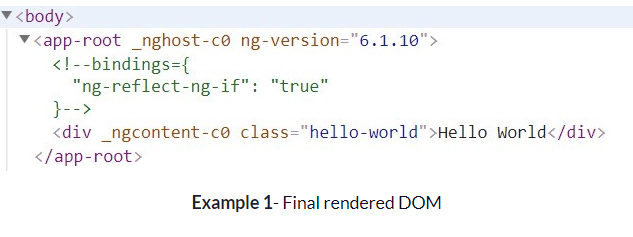
As the name suggests the <ng-template> is a template element that Angular uses with structural directives (\*ngIf, \*ngFor, [ngSwitch] and custom directives).

**These template elements only work in the presence of structural directives***.*Angular wraps the host element (to which the directive is applied) inside <ng-template> and consumes the <ng-template> in the finished DOM by replacing it with diagnostic comments.

Consider a simple example of \*ngIf:

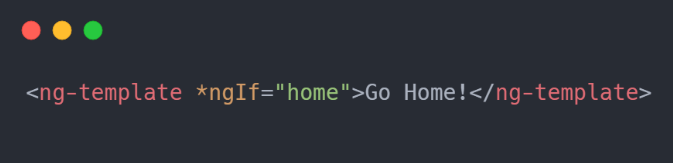


Shown above is the Angular interpretation of \*ngIf. Angular puts the host element to which the directive is applied within <ng-template> and keeps the host as it is. The final DOM is similar to what we have seen at the beginning of this article:

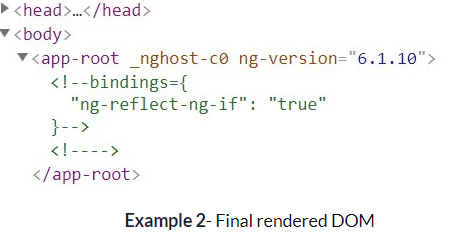


**Usage:**

We have seen how Angular uses <ng-template> but what if we want to use it? As these elements work only with a structural directive, we can write as:



Here home is a boolean property of the component set to true value. The output of the above code in DOM:

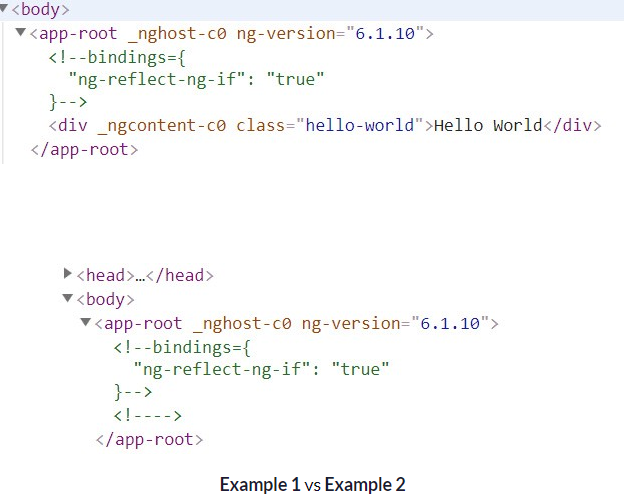


Nothing got rendered! :(

But why can’t we see our message even after using <ng-template> correctly with a structural directive?

This was the expected result. As we have already discussed, Angular replaces the <ng-template> with diagnostic comments. No doubt the above code would not generate any error, as Angular is perfectly fine with your use case. You would never get to know what exactly happened behind the scenes.

Let’s compare the above two DOMs that were rendered by Angular:

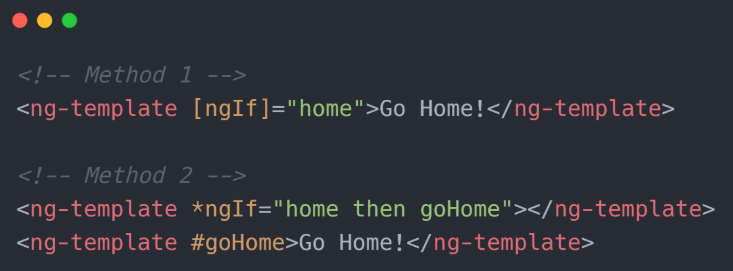


If you watch closely, there is one **extra comment tag** in the final DOM of **Example 2**. The code that Angular interpreted was:



Angular wrapped up your host <ng-template> within another <ng-template> and converted not only the outer <ng-template> to diagnostic comments but also the inner one! This is why you could not see any of your message.

To get rid of this there are two ways to get your desired result:



**Method 1:**

In this method, you are providing Angular with the de-sugared format that needs no further processing. This time Angular would only convert <ng-template> to comments but leaves the content inside it untouched (they are no longer inside any <ng-template>as they were in the previous case). Thus, it will render the content correctly.

To know more about how to use this format with other structural directives refer to this [article](https://www.concretepage.com/angular-2/angular-4-ng-template-example#ngSwitch).

**Method 2:**

This is a quite unseen format and is seldom used (using two sibling <ng-template>). Here we are giving a template reference to the \*ngIf in its then to tell it which template should be used if the condition is true.

Using multiple <ng-template> like this is not advised (you could use <ng-container>instead) as this is not what they are meant for. They are used as a container to templates that can be reused at multiple places. We will cover more on this in a later section of this article.

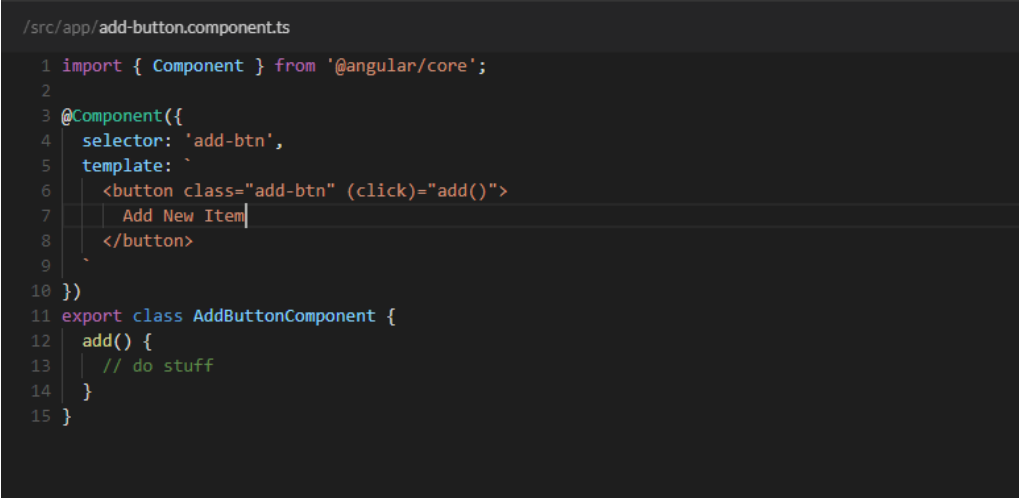
**2. <ng-content>**

One of the main goals of Angular is to help the developer create reusable and composable components. I think ng-content is one of the simplest ways to do that once you understand how it works.

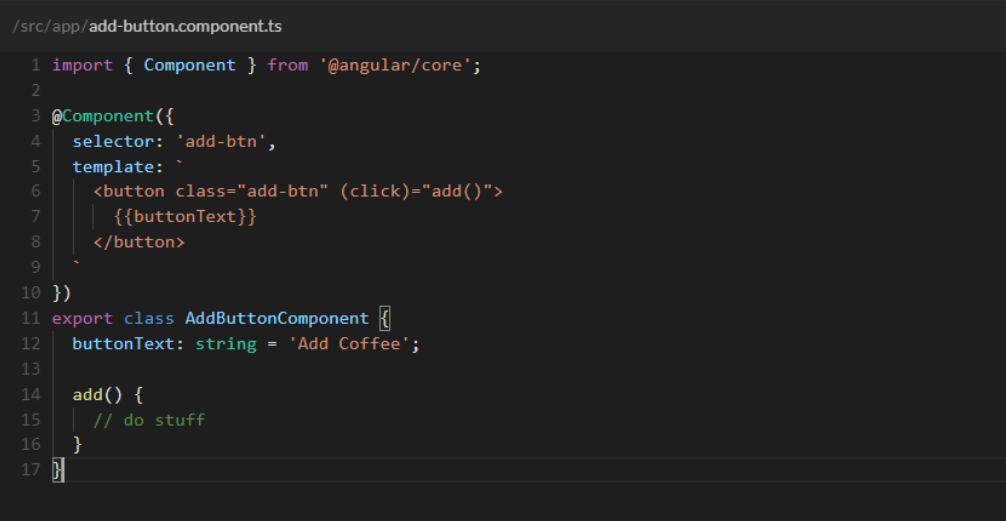
Dynamic content. That is the simplest way to explain what ng-content provides. You use the **<ng-content></ng-content>** tag as a placeholder for that dynamic content, then when the template is parsed Angular will replace that placeholder tag with your content. Think of it like curly brace interpolation, but on a bigger scale. The technical term for this is “content projection" because you are projecting content from the parent component into the designated child component.

If you understand **{{myValue}}**, then you understand the basics of what ng-content does. The difference is where that value comes from. With normal curly brace interpolation the value comes from the component. With ng-content the value comes from the component **in its execution context**.

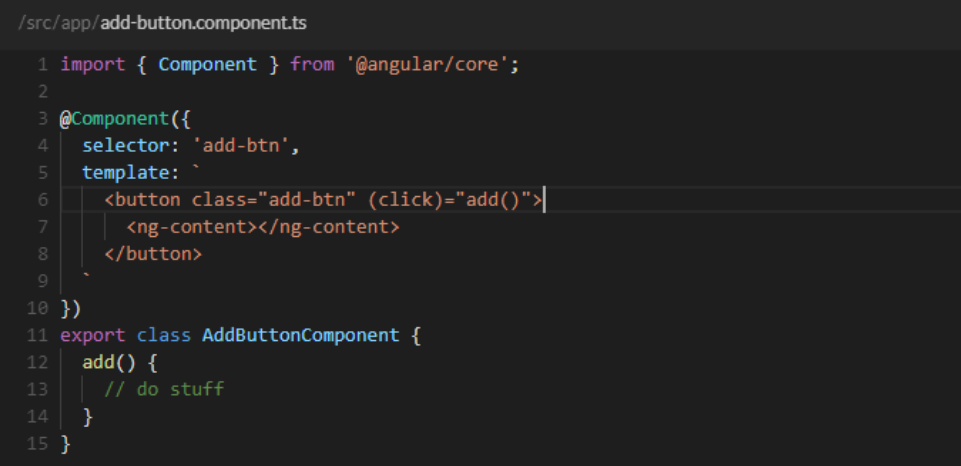
Let’s say you want to create a reusable button in your app.

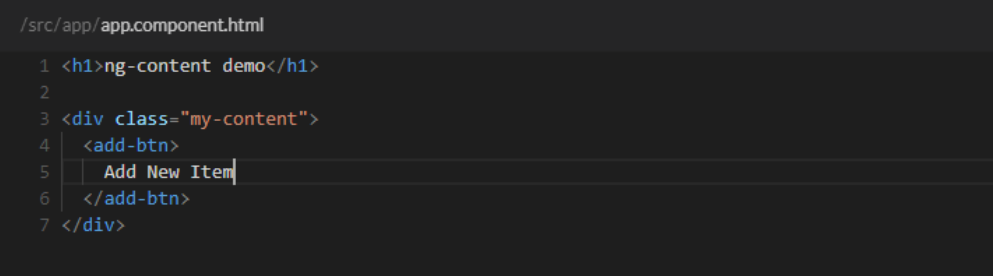


Here we can see a generic add button which triggers an event when clicked. Nothing crazy here. The main thing I want to point out is the button’s text. “Add New Item" is hard-coded in the template. But, what if we wanted to get more specific with our button text? For example, “Add Coffee". We could put that value in the component like this:



These ways work but this is where **ng-content** shines. Take a look at this:





See what’s happening here?

In the template for the reusable add button component we use the **<ng-content></ng-content>** tag as our placeholder for the button text. This is telling Angular, “Hey, I don’t know what this is supposed to be right now but, I promise to tell you later”.

Then when the button component is actually being used… BAM! We put whatever text we want inside the component and then Angular will be like “Oh Snap! Now I know what value to use for the button’s text!”.

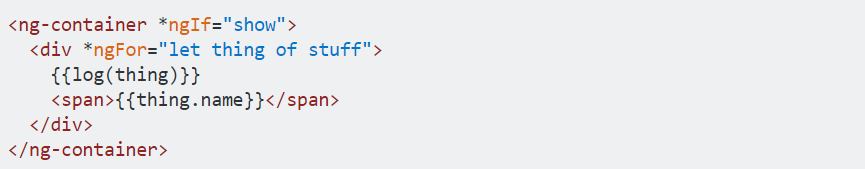
Pretty cool. It strips out so much of the extra setup we used in the earlier examples. No inputs to keep track of, no hard-coded values in the component. Just set up the original template and worry about it later when you actually need it. Dynamic text is just the tip of the metaphorical iceberg though. You can put lots of different things inside the component. Including HTML tags and even other components. Hopefully your mind is spinning with the possibilities.

###### **<ng-container>**

<ng-container> is an Angular grouping element that is similar to <ng-container> in that it doesn’t represent a DOM element. The difference is that it will always be rendered, whereas an <ng-template> will only be rendered if it is explicitly requested. <ng-container>s are useful anywhere you need an extra container for some template elements, but don’t want to (or can’t) create a container such as a div to hold them with due to syntax or style constraints.

For example, it is not allowed in Angular to put two structural directives on the same element. If you needed to loop through an array and display a <tr> for each element, but only if a different condition was met, you may want to put both an \*ngFor and \*ngIf on the <tr> element. Angular does not allow this, however, and wrapping the <tr> in a <div> to hold one of the structural directives is not valid HTML. The utility of <ng-container> shines here, where we can use the <ng-container>to hold a structural directive and contain the <tr></span> without breaking the HTML layout.

Angular v2 doesn't support more than one structural directive on the same element.  
As a workaround use the **<ng-container>** element that allows you to use separate elements for each structural directive, but it is **not stamped to the DOM**.



***Creating Custom Pipes in Angular***

Here for example we create a pipe that takes a string and reverses the order of the letters. Here's the code that would go into a reverse-str.pipe.ts file inside of your app folder:

1. import { Pipe, PipeTransform } from '@angular/core';
2. @Pipe({name: 'reverseStr'})
3. export class ReverseStr implements PipeTransform {
4. transform(value: string): string {
5. let newStr: string = "";
6. for (var i = value.length - 1; i >= 0; i--) {
7. newStr += value.charAt(i);
8. }
9. return newStr;
10. }
11. }
12. Then you’d include the custom pipe as a declaration in your app module:
13. import { BrowserModule } from '@angular/platform-browser';
14. import { NgModule } from '@angular/core';
15. import { FormsModule } from '@angular/forms';
16. import { HttpModule } from '@angular/http';
17. import { AppComponent } from './app.component';
18. import { ReverseStr } from './reverse-str.pipe.ts';
19. @NgModule({
20. declarations: [
21. AppComponent,
22. ReverseStr
23. ],
24. And finally in your templates this is how you would use this custom pipe:
25. {{ user.name | reverseStr }}

### **Creating a Custom Attribute Directive**

Create an app-highlight.directive.ts file in src/app folder and add the code snippet below.

import { Directive, ElementRef } from '@angular/core';

@Directive({

selector: '[appHighlight]'

})

export class HighlightDirective {

constructor(private eleRef: ElementRef) {

eleRef.nativeElement.style.background = 'red';

}

}

Now to get appHighlight Directive to work, we need to add our Directive to the declarations array in the app.module.ts file.

import ...;

import { ChangeThemeDirective } from './app-highlight.directive';

@NgModule({

declarations: [

AppComponent,

ChangeThemeDirective

],

Now we are going to use our newly created custom directive. I am adding the appHightlight directive in the app.component.html but you can use it anywhere in the application.

<h1 appHightlight>Highlight Me !</h1>

### Creating a Custom Structural Directive

Now create a app-not.directive.ts file in the src/app folder and add the code below.

import { Directive, Input, TemplateRef, ViewContainerRef } from '@angular/core';

@Directive({

selector: '[appNot]'

})

export class AppNotDirective {

constructor(

private templateRef: TemplateRef<any>,

private viewContainer: ViewContainerRef) { }

@Input() set appNot(condition: boolean) {

if (!condition) {

this.viewContainer.createEmbeddedView(this.templateRef);

} else {

this.viewContainer.clear(); }

}

}

## What is the difference between devDependencies and dependencies?

### **dependencies**

“dependencies” are packages required to run the application in a production-ready environment. Without these packages, your app won’t work. A couple of general examples are:

* **frameworks**: React, AngularJS, Vue.js
* **utility libraries**: lodash, Ramda, date-fns, polished

### **devDependencies**

“devDependencies” are required to develop and build your app, but are not needed to run the final version that customers will use. For example:

* **testing libraries**: Jest, Mocha, Jasmine
* **linters**: ESLint, Prettier
* **transpilers**: webpack, Babel (since production-ready code is already transpiled and minified)

When you run npm install in the root of a project with a package.json file, **all packages** in both dependencies and devDependencies are installed. This is because you’re working with the source code, so are probably developing this app and therefore need the code in every package. However, if you only want to install the packages listed under the dependencies key, then use the —-production flag, like npm install --production.

# Angular Authentication: Using Route Guards

Angular’s route guards are interfaces which can tell the router whether or not it should allow navigation to a requested route. They make this decision by looking for a true or false return value from a class which implements the given guard interface.

# Routing Decisions Based on Token Expiration

If you’re using [JSON Web Tokens](https://jwt.io/) (JWT) to secure your Angular app (and I recommend that you do), one way to make a decision about whether or not a route should be accessed is to check the token’s expiration time. It’s likely that you’re using the JWT to let your users access protected resources on your backend. If this is the case, the token won’t be useful if it is expired, so this is a good indication that the user should be considered “not authenticated”.

Create a method in your authentication service which checks whether or not the user is authenticated. Again, for the purposes of stateless authentication with JWT, that is simply a matter of whether the token is expired. The JwtHelperService class from **angular2-jwt** can be used for this.

# npm install --save @auth0/angular-jwt

# Use angular-jwt in your AuthService

// src/app/auth/auth.service.ts

import { Injectable } from '@angular/core';  
import { JwtHelperService } from '@auth0/angular-jwt';

@Injectable()  
export class AuthService

{

constructor(public jwtHelper: JwtHelperService) {}

// ...  
 public isAuthenticated(): boolean {

const token = localStorage.getItem('token'); // Check whether the token is expired and return  
 // true or false  
 return !this.jwtHelper.isTokenExpired(token);  
 }

}

# Create a new service which implements the route guard. You can call it whatever you like, but something like auth-guard.service is generally sufficient.

// src/app/auth/auth-guard.service.ts

import { Injectable } from '@angular/core';  
import { Router, CanActivate } from '@angular/router';  
import { AuthService } from './auth.service';

@Injectable()  
export class AuthGuardService implements CanActivate { constructor(public auth: AuthService, public router: Router) {} canActivate(): boolean {  
 if (!this.auth.isAuthenticated()) {  
 this.router.navigate(['login']);  
 return false;  
 }  
 return true;  
 }

}

The service injects AuthService and Router and has a single method called canActivate. This method is necessary to properly implement the CanActivate interface.

The canActivate method returns a boolean indicating whether or not navigation to a route should be allowed. If the user isn’t authenticated, they are re-routed to some other place, in this case a route called /login.

Now the guard can be applied to any routes you wish to protect.

// src/app/app.routes.ts

import { Routes, CanActivate } from '@angular/router';  
import { ProfileComponent } from './profile/profile.component';  
import { AuthGuardService as AuthGuard } from './auth/auth-guard.service';

export const ROUTES: Routes = [  
 { path: '', component: HomeComponent },  
 {   
 path: 'profile',  
 component: ProfileComponent,  
 canActivate: [AuthGuard]   
 },  
 { path: '\*\*', redirectTo: '' }  
];

# The /profile route has an extra config value now: canActivate. The AuthGuard that was created above is passed to an array for canActivatewhich means it will be run any time someone tries to access the /profileroute. If the user is authenticated, they get to the route. If not, they are redirected to the /login route.

# Angular Best Practices

## **Angular Coding Styles**

*Here’re some set of rules that you need to follow to make your project comply with the coding standard;*

* Per file, the code must not exceed from 400 lines limit
* Per function, the code must not exceed from 75 lines
* Utilize custom prefix to share feature area for all slider components
* If the values of the variables are intact, declare it with ‘const’
* If you often prefix names for field, table, and database, then you need to avoid this for Interface names like iShape, AbastractShape, etc.
* Names of properties and methods should be in lower camel case
* Always leave one empty line between imports and module such as third party and application imports and third-party module and custom module

## **Folder Structure for Angular**



## **Prevent Memory Leaks in Angular Observable**

### Using ‘take(1):

‘take(1)’ is an operator which completes the emission by taking its value and enables the ‘take(1) not to subscribe when a new value is encountered with. It will ensure that you get the data only once. Be secure with ‘take(1)’ and avoid memory leaks easily. Here’s how it happens;

data$.pipe(take(1)).subscribe(res=>console.log(res))

## **Utilizing ES6 Features**

**‘let and const’ instead of ‘var’**

### **Spread Operator**

Map, Filter, Reduce

## **Declare Variable Type Instead of Using ‘any’**