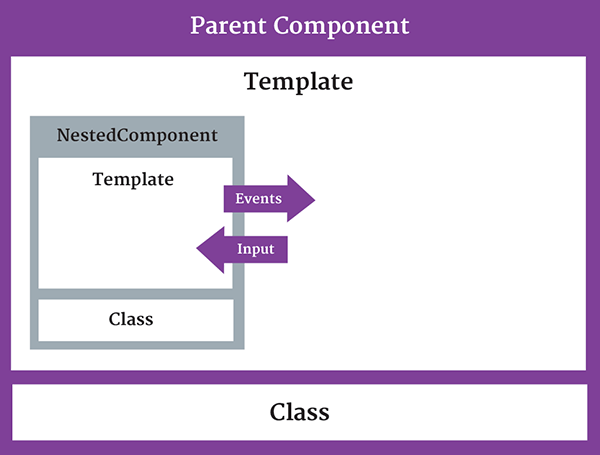
**What is Component.**

Components are a logical piece of code for Angular application. A Component consists of the following −

* **Template** − This is used to render the view for the application. This contains the HTML that needs to be rendered in the application. This part also includes the binding and directives.
* **Class** − This is like a class defined in any language such as C. This contains properties and methods. This has the code which is used to support the view. It is defined in TypeScript.
* **Metadata** − This has the extra data defined for the Angular class. It is defined with a decorator.

**Creating Component and nested component and passing data between them.**

**Building a Nested Component**

We can simply add a nested component by using a component as a directive within another component.s

Let's first create a basic component that will be nested in another component later on. This components has a title property that we use in its template:

@Component({

selector: 'child-selector',

template: 'child.component.html'

})

export class ChildComponent {

title = 'I\'m a nested component';

}

The child.component.html is just an HTML file that shows the value of the title property:

/\* child.conponent.html \*/

<h2>{{title}}</h2>

Now we want to create a container component. It looks almost identical to the nested component, except we have to specify that we want to use the nested component. We do that by adding the ChildComponent to the directives property of the Component decorator. Without doing this, the ChildComponent can not be used.

@Component({

selector: 'parent-selector',

template: 'parent.component.html',

directives: [ChildComponent]

})

export class ParentComponent { }

The container component uses the nested component by specifying its directive in the template:

/\* parent.component.html \*/

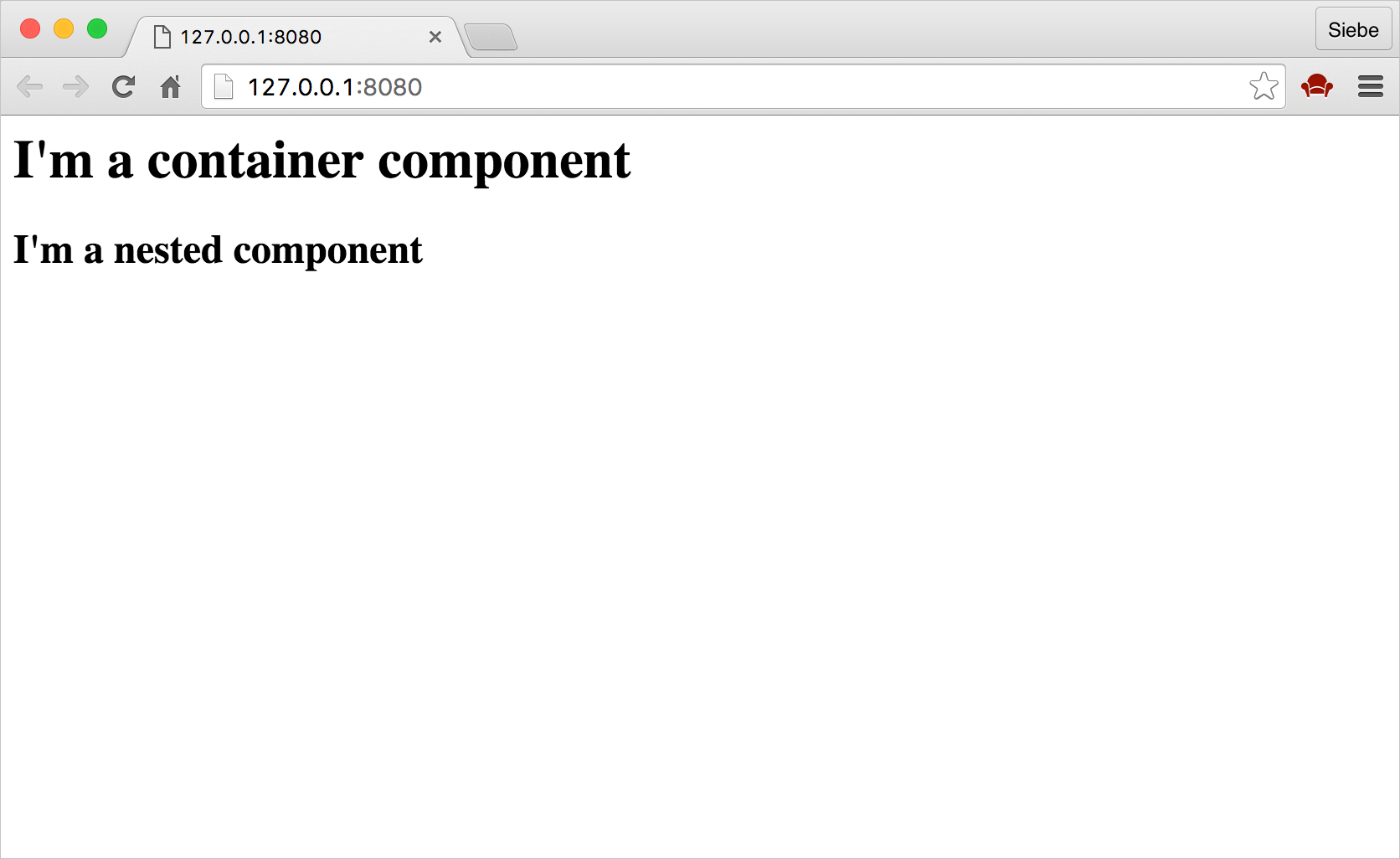
<div>

<h1>I'm a container component</h1>

<child-selector></child-selector>

</div>

Now, when we run our code we see something like this:



Nothing too fancy yet. Let's pass some data to the nested component!

**Passing data to a nested component**

If a nested component wants to receive input from its container, it must expose a property to that container. The nested component exposes a property it can use to receive input from its container using the [@Input decorator](https://angular.io/docs/ts/latest/guide/attribute-directives.html).

We use the Input decorator to decorate any property in the nested component class. This works with every property type, including objects. In our example, we'll pass a string value to the nested component's title property, so we'll mark that property with the @Input decorator:

@Component({

selector: 'child-selector',

template: 'child.component.html'

})

export class ChildComponent {

@Input() title:string;

}

Now our nested component is ready to receive input from its parent component.

In the container component, we need to define the property we want to pass to the nested component. We call it childTitle:

@Component({

selector: 'parent-selector',

template: 'parent.component.html',

directives: [ChildComponent]

})

export class ParentComponent {

childTitle:string = 'This text is passed to child';

}

Now the container component should pass the value of childTitle to the nested component by settings this property with [property binding](https://www.themarketingtechnologist.co/introduction-to-data-binding-in-angular-2-versus-angular-1/). When using property binding, we enclose the binding target in square brackets. The binding target refers to the title property of the nested component. We set the binding source to the data that the container wants to pass to the nested component, which is childTitle.

/\* parent.component.html \*/

<div>

<h1>I'm a container component</h1>

<child-selector [title]='childTitle'></child-selector>

</div>

The only time we can specify a nested component's property as a property binding target, is when that property is decorated with the @Input decorator, like we did earlier.

When we run our code, we should see the passed value in the nested component's H2 element:

https://themarketingtechnologist-ghost.s3.amazonaws.com/2016/May/2-1463923054994.png

In this example we only exposed one input property, but of course you can expose multiple input properties as needed.

**Passing data from a Nested Component**

In the previous example I showed how the container can pass data to the nested component by binding to a nested component's property, that is declared with the @Input decorator.

If the nested component wants to send information back to its container, it can raise an event. The nested component exposes an event it can use to pass output to its container using the @Output decorator.

Like with the @Input decorator, we can use the @Output decorator to decorate any property of the nested components class. However, the property type must be an event. The only way a nested component can pass data back to its container, is with an event. The data to pass is called the event payload. In Angular, an event is defined with an [EventEmitter](https://angular.io/docs/js/latest/api/core/EventEmitter-class.html) object.  
So let's start by creating a new instance of an Event Emitter and decorate the property with the @Output decorator.

@output() notify: EventEmitter<string> = new EventEmitter<string>();

If you're not familiar with generics, this syntax may look a bit odd to you. [Generics](https://www.typescriptlang.org/docs/handbook/generics.html) allow us to identify a specific type that the object instance will work with.

The generic argument, string, identifies of the event payload. So now we can only pass string values to the container. If we would want to pass a integer for example, we'd write something like this:

@output() notify: EventEmitter<number> = new EventEmitter<number>();

Although not recommended, you can let it accept any type. You can use TypeScript's any type. We'll stick with a string for now. Please note that JavaScript does not support generics by itself; it's a TypeScript feature. I've personally only played with generics in C# a bit, and I think the concept is fairly new to the Javascript community in general.

Now we've got an event emitter in place, let's use the notify event property and call its emit method to raise the notify event and pass in our payload as an argument.

this.notify.emit('payload');

We'll need a user interaction that will raise the event, so I'll add a link that will do so.

/\* child.component.html \*/

<h2>Hi, I'm a nested component</h2>

<span (click)='onClick()'>Click me please!</span>

Now we'll add the click event handler to the component, and raise the notify event:

@Component({

selector: 'child-selector',

template: 'child.component.html'

})

export class ChildComponent {

@output() notify: EventEmitter<string> = new EventEmitter<string>();

onClick() {

this.notify.emit('Click from nested component');

}

}

So every time a user clicks on the 'Click me please' link, the nested component will dispatch an event to its parent component.

The parent component receives that event and its payload. We use event binding to bind to this notify event and call a method.

/\* parent.conponent.html \*/

<div>

<h1>I'm a container component</h1>

<child-selector (notify)='onNotify($event)></child-selector>

</div>

We have to pass the $event to the handler because that variable holds the event payload.

The only time we can can specify a nested component's property as an event binding target is when that property is decorated with the @Output decorator.

Our final step is to provide the onNotify method to execute when the notify event occurs. Since the event payload is a string, the onNotify function takes in a string. We can perform any desired action in our handler, but for now let's just alert the payload.

@Component({

selector: 'parent-selector',

template: 'parent.component.html',

directives: [ChildComponent]

})

export class ParentComponent {

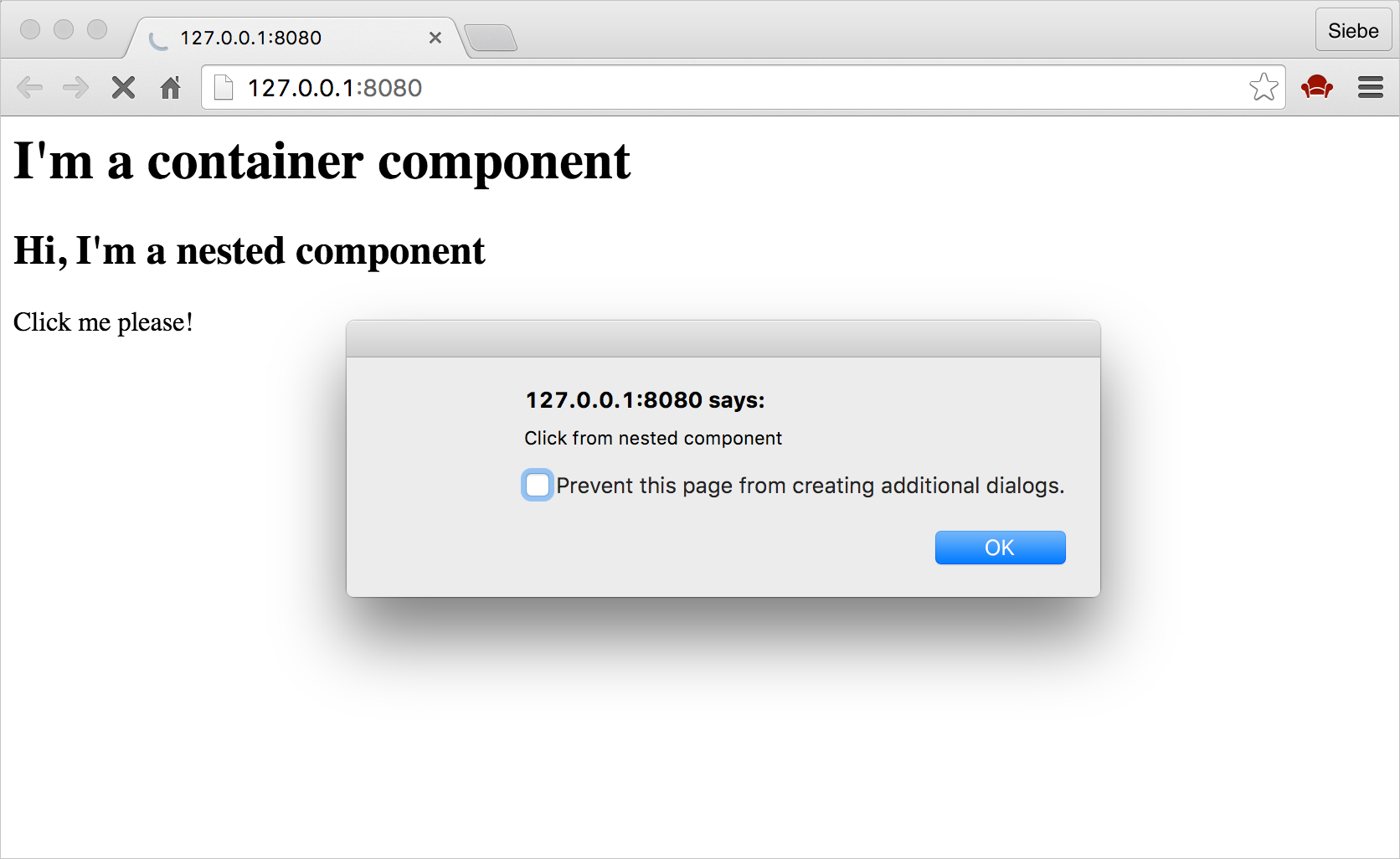
onNotify(message:string):void {

alert(message);

}

}

So now we should see an alert with the text from the nested component:



**Data Binding with Example**

Data binding is one of the most powerful features of software development technologies. Data binding is the connection bridge between View and the business logic (View Model) of the application. Data binding in Angular is the automatic synchronization between Model and the View. When the Model changes, the Views are automatically updated and vice-versa.   
  
There are many ways to bind the data in Angular. Following are the types of data binding in Angular 2.

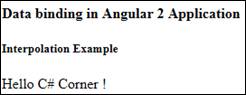
* Interpolation
* One-way binding (unidirectional)
* Two-way binding
* Event binding

**Interpolation**This is the easiest way of data binding in AngularJS. This is same as Expressions  in Angular 1.x. In interpolation, we need to supply property name in the View template, enclosed in double curly braces, e.g. {{name}}. It is used for one-way binding (Component class to View only).  
  
In the following example, I have defined one property called "name" in Component Class and it is displayed on View template by using interpolation.  
  
**app.component.ts**

1. **import** { Component } from '@angular/core';
2. @Component({
3. selector: 'test-app',
4. templateUrl: './app/databinding.html'
5. })
6. **export** **class** AppComponent {
7. name = 'C# Corner';
8. }

**databinding.html**

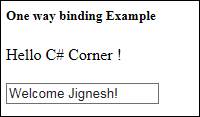
1. <h4>Data binding **in** Angular 2 Application</h4>
2. <div>
3. <h5>Interpolation Example</h5>
4. Hello {{name}} !
5. </div>

**Output**  
  
  
**One-way binding**In Angular 2, one-way data binding directive is replaced with [property]. The ng-bind directive is used for one-way binding in Angular 1.x. Angular 2.0 uses HTML DOM element property for one-way binding. The square brackets are used with property name for one-way data binding in Angular 2. For example, if we want one-way binding between Model property and template View for textbox, we need to use [value].  
  
**app.component.ts**

1. **import** { Component } from '@angular/core';
2. @Component({
3. selector: 'test-app',
4. templateUrl: './app/databinding.html'
5. })
6. **export** **class** AppComponent {
7. name = 'C# Corner';
8. welcomeText = 'Welcome Jignesh!'
9. }

**databinding.html**

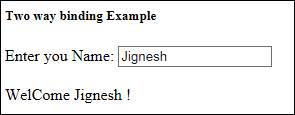
1. <div>
2. <h5>One way binding Example</h5>
3. Hello <span [innerText] = "name" ></span>!
4. <br/><br/>
5. <input type = 'text'  [value]="welcomeText" />
6. </div>

**Output**   
  
  
  
**Two-way binding**  
The ng-model directive is used for two-way data binding in Angular 1.x, but it is replaced with [(ngModel)] in Angular 2.0. The ngModel directive is part of a built-in Angular module called "FormsModule". So, we must import this module in to the template module before using the ngModel directive.  
  
**app.module.ts**

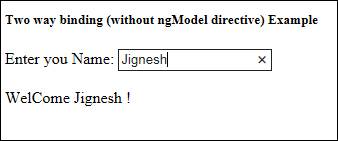
1. **import** { NgModule }      from '@angular/core';
2. **import** { BrowserModule } from '@angular/platform-browser';
3. **import** { FormsModule } from '@angular/forms';
5. **import** { AppComponent }  from './app.component';
7. @NgModule({
8. imports:      [ BrowserModule, FormsModule],
9. declarations: [ AppComponent],
10. bootstrap:    [ AppComponent ]
11. })
12. **export** **class** AppModule {
13. }

**databinding.html**

1. <div>
2. <h5>Two way binding Example</h5>
3. Enter you Name:  <input [(ngModel)]="enterName"  />
4. <br/><br/>
5. <span> WelCome {{enterName}} ! </span>
6. </div>

**Output**  
  
  
**Two-way binding without ngModel directive**Angular 2 has a feature called "template reference variables". With this feature, we are able to have direct access to an element. The template reference variable is declared by preceding an identifier with a hash/pound character (#).  
  
In the following example, we have declared a template reference variable named "txtName" on the input element. This variable is in reference to the input element. So, we can get the value property of the input element and display it with interpolation.  
  
Here, template is completely self-contained and it does not bind to the component. This solution will not work unless we bind to the element’s event. Angular only updates the bindings if something is done in response to asynchronous events, like keystrokes. This is the reason we bind keyup event to input element and we do nothing with this event. So, we are binding the number 0 with the event.   
  
**databinding.html**

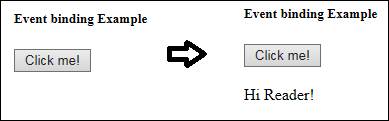
1. <div>
2. <h5>Two way binding (without ngModel directive) Example</h5>
3. Enter you Name: <input  #txtName  type = "text"  (keyup)="0" />
4. <br/><br/>
5. <span> WelCome {{txtName.value}} ! </span>
6. </div>

**Output**  
  
  
**Event Binding**Angular 2.0 directly uses the valid HTML DOM element events. For example, ng-click is now replaced with (click). The round brackets (parentheses) are used with DOM event name for event binding in Angular 2.   
  
In the following example, I have defined button click event and on button click, I have written the message on screen.  
  
**app.component.ts**

1. **import** { Component } from '@angular/core';
2. @Component({
3. selector: 'test-app',
4. templateUrl: './app/databinding.html'
5. })
6. **export** **class** AppComponent {
7. messageText = '';
8. onClickMe() {
9. **this**.messageText = "Hi Reader!";
10. }
11. }

**databinding.html**

1. <div>
2. <h5>Event binding Example</h5>
3. <button (click)="onClickMe()">Click me!</button>
4. <br/><br/>
5. <span> {{messageText}} </span>
6. </div>



There are 3 types of directives:

1. Components
2. Structural Directives
3. Attribute Directives

## Components

You cannot create an Angular application without one. A component directive requires a view along with its attached behaviour. This type of directive adds DOM elements. The naming convention for components is: name.component.ts.

**import {Component} from 'angular2/core'**

**@Component({**

**selector: 'my-app',**

**template: `<h3>The Super Button</h3><button (click)="btnClick()">Click Me</button>`**

**})**

**export class App {**

**btnClick() {**

**alert('You clicked me!');**

**}**

**}**

The above is a sample of a simple component. In this case the main component of the app, also called the root component. It contains the bare minimum needed to create a component. Once rendered, it displays a title and a button that alerts you when clicked.  
  
To tell Angular you want a class to be a component you need to import and assign the component decorator to it. Decorators in TypeScript are like annotations in Java or attributes in C#. Learn more about decorators [here](http://www.typescriptlang.org/docs/handbook/decorators.html). The component decorator allows us to declare many attributes, but we are only using the minimum here. The selector tells Angular what will be the tag name for my component. The template attribute specifies the DOM elements to add. You can also use “templateUrl” and point to a template file.

We export this class to make it available as a module and we add the btnClick method.  
  
Now in your HTML you can call this component like this:  
< my-app></my-app>

### Structural & Attribute Directives

Rather than adding new DOM elements, both these types of directives modify the existing DOM and do not have templates.

Attribute: Modifies the appearance or behavior of an element.

Structural: Modifies the DOM layout.

## 2. Attribute Directive

Examples of built-in attribute directives that ship with Angular 2 are ngStyle and ngClass. As mentioned before, these directives modify the DOM and we are going to do the same in a simple custom attribute directive of our own. The naming convention for directives is: name.directive.ts

**import {Directive, ElementRef, Renderer} from 'angular2/core';**

**@Directive({**

**selector: '[my-outline]',**

**host: {**

**'(mouseenter)': 'onMouseEnter()',**

**'(mouseleave)': 'onMouseLeave()'**

**}**

**})**

**export class MyOutline {**

**constructor(private \_element: ElementRef, private \_renderer:Renderer) { }**

**onMouseEnter() { this.\_outlineToggle(true); }**

**onMouseLeave() { this.\_outlineToggle(false); }**

**private \_outlineToggle(color: string) {**

**this.\_renderer.setElementStyle(this.\_element.nativeElement, 'border', 'solid red 1px' );**

**}**

**}**

Taking a quick look reveals that this directive looks very much like a component. But, there are differences, one is as mentioned before we do not have a template because this directive modifies the DOM.  
  
We are also introducing the host in the decorator meta data. This host refers to the DOM element that this directive is attached to. It is recommended to use the host rather than attaching listeners to the element directly. This is because we could introduce memory leaks or be confronted with naming issues. In the host object, we declare the event listeners that we are interested in listening too. These are linked to methods in the MyOutline class.  
  
MyOutline class declares a constructor that injects the ElementRef service and creates a private \_element property which gives us access to the DOM element. ElementRef  has a nativeElement property that we could use to add our border directly like so:  
  
**el.nativeElement.style.border = 'solid red 1px';**

But that is not recommended and dangerous. Instead, we use the Renderer and pass it our ElementRef’s nativeElement. The Renderer class contains everything we need to manipulate the element safely. This includes the setElementStyle method that we are using to set the border of the element.  
  
That is it, a very simple directive that adds a border to an element when the user hovers over it.

## 3. Structural Directive

Examples of built-in structural directives that ship with Angular 2 are ngIf, ngFor and ngSwitch. As mentioned before these directives change the layout of the DOM, and we will also do the same in a small simple custom example. The naming convention remains the same as before: name.directive.ts.  
  
We are now going to create a simple custom variation of the ngFor directive. Our version will take an integer input. This int represents how many times we want to repeat the element that we are attached to.

**import {Directive, Input} from 'angular2/core';**

**import {TemplateRef, ViewContainerRef} from 'angular2/core';**

**@Directive({ selector: '[repeatMe]' })**

**export class RepeatMe {**

**constructor(**

**private \_templateRef: TemplateRef,**

**private \_viewContainer: ViewContainerRef**

**) {}**

**@Input() set repeatMe(count: int) {**

**for (var i = 0; i < count; i++) {**

**this.\_viewContainer.createEmbeddedView(this.\_templateRef);**

**}**

**}**

**}**

It is worth noting at this point that the HTML 5 template tag is used in structural directives. It is your choice to use the template tag directly, other wise you can use the \* which takes care of the template tag for you. Take a look at the documentation for ngFor for more information.  
  
Much like our previous example, in our repeatMe class we declare a constructor that injects two things: [TemplateRef](https://angular.io/docs/js/latest/api/core/TemplateRef-class.html) and [ViewContainerRef](https://angular.io/docs/js/latest/api/core/ViewContainerRef-class.html#!).  TemplateRef points to our template and ViewContainerRef will handle creating the view and attaching it to the container.

Next, we use the [@Input()](https://angular.io/docs/ts/latest/api/core/Input-var.html) to specify our repeatMe input which takes in our integer and set it at the same time.  
  
With the help of the ViewContainerRef, we can use its createEmbeddedView method to do the rest. It creates a view using the template and attaches it to the element as many times as specified with the count value.

**Pipe and Custom Pipe**

pipes is a handsome way to handle transformation in templates. It makes your code more clean and structured.

Angular 2 also has the facility to create custom pipes. The general way to define a custom pipe is as follows.

import { Pipe, PipeTransform } from '@angular/core';

@Pipe({name: 'Pipename'})

export class Pipeclass implements PipeTransform {

transform(parameters): returntype { }

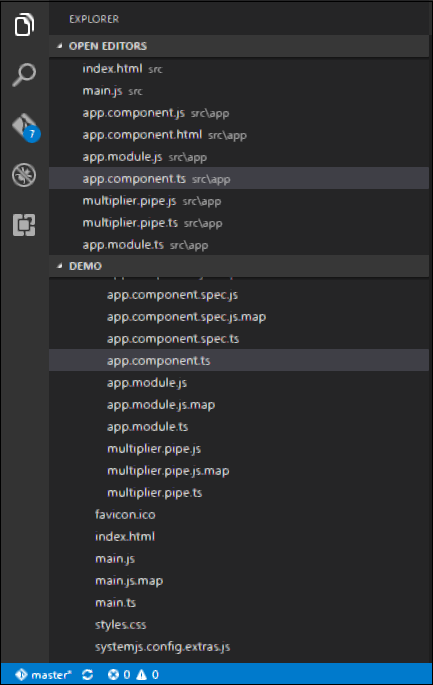
}

Where,

* **'Pipename'** − This is the name of the pipe.
* **Pipeclass** − This is name of the class assigned to the custom pipe.
* **Transform** − This is the function to work with the pipe.
* **Parameters** − This are the parameters which are passed to the pipe.
* **Returntype** − This is the return type of the pipe.

Let’s create a custom pipe that multiplies 2 numbers. We will then use that pipe in our component class.

**Step 1** − First, create a file called multiplier.pipe.ts.



**Step 2** − Place the following code in the above created file.

import {

Pipe,

PipeTransform

} from '@angular/core';

@Pipe ({

name: 'Multiplier'

})

export class MultiplierPipe implements PipeTransform {

transform(value: number, multiply: string): number {

let mul = parseFloat(multiply);

return mul \* value

}

}

Following points need to be noted about the above code.

* We are first importing the Pipe and PipeTransform modules.
* Then, we are creating a Pipe with the name 'Multiplier'.
* Creating a class called MultiplierPipe that implements the PipeTransform module.
* The transform function will then take in the value and multiple parameter and output the multiplication of both numbers.

**Step 3** − In the app.component.ts file, place the following code.

import {

Component

} from '@angular/core';

@Component ({

selector: 'my-app',

template: '<p>Multiplier: {{2 | Multiplier: 10}}</p>'

})

export class AppComponent { }

**Note** − In our template, we use our new custom pipe.

**Step 4** − Ensure the following code is placed in the app.module.ts file.

import {

NgModule

} from '@angular/core';

import {

BrowserModule

} from '@angular/platform-browser';

import {

AppComponent

} from './app.component';

import {

MultiplierPipe

} from './multiplier.pipe'

@NgModule ({

imports: [BrowserModule],

declarations: [AppComponent, MultiplierPipe],

bootstrap: [AppComponent]

})

export class AppModule {}

Following things need to be noted about the above code.

* We need to ensure to include our MultiplierPipe module.
* We also need to ensure it is included in the declarations section.

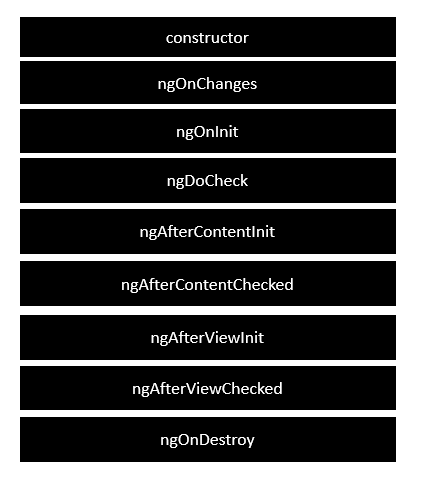
Once you save all the code changes and refresh the browser, you will get the following output.



**Angular 2 Life Cycle Hook**

Angular 2 application goes through an entire set of processes or has a lifecycle right from its initiation to the end of the application.

The following diagram shows the entire processes in the lifecycle of the Angular 2 application.



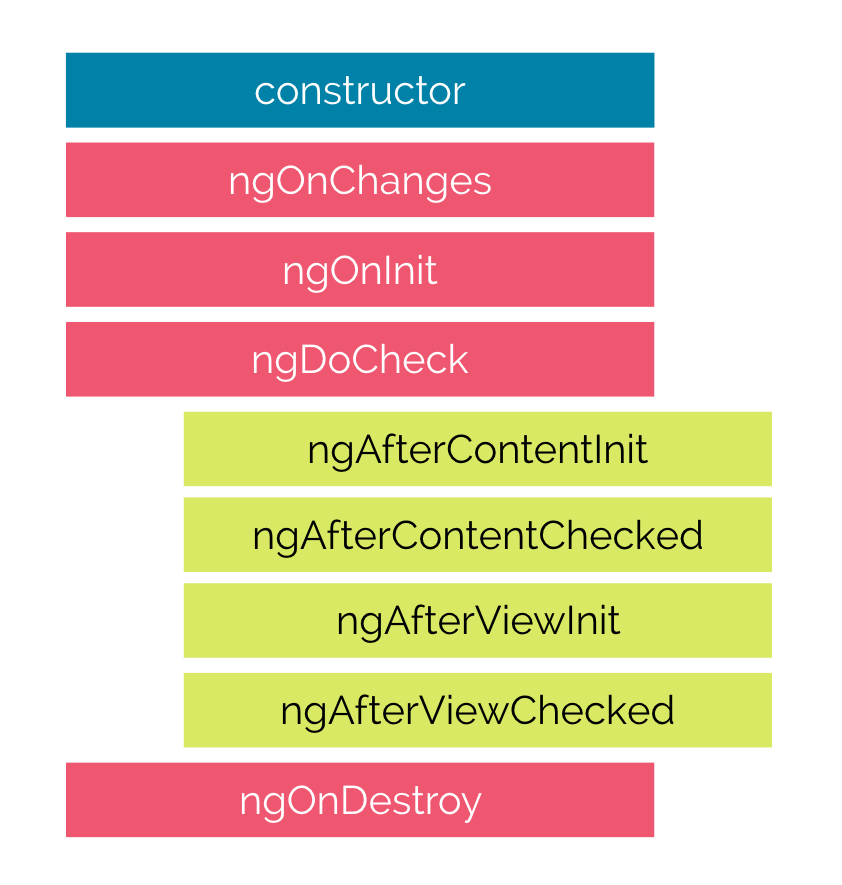
Following is a description of each lifecycle hook.

* **ngOnChanges** − When the value of a data bound property changes, then this method is called.
* **ngOnInit** − This is called whenever the initialization of the directive/component after Angular first displays the data-bound properties happens.
* **ngDoCheck** − This is for the detection and to act on changes that Angular can't or won't detect on its own.
* **ngAfterContentInit** − This is called in response after Angular projects external content into the component's view.
* **ngAfterContentChecked** − This is called in response after Angular checks the content projected into the component.
* **ngAfterViewInit** − This is called in response after Angular initializes the component's views and child views.
* **ngAfterViewChecked** − This is called in response after Angular checks the component's views and child views.
* **ngOnDestroy** − This is the cleanup phase just before Angular destroys the directive/component.

**Life Cycle Hook in Detail**

Angular is a javascript development platform for creating desktop and mobile web applications, Angular 2 will help us to create that application on a basis of components. To know about basics of component refer this [blog](http://www.techjini.com/blog/first-look-at-creating-angular-2-component/).

Let’s see about lifecycle hooks in Angular2.  
There are eight main hooks for every component to have robustness in our built applications.

[](http://www.techjini.com/wp-content/uploads/2017/04/lifecycle-hooks-A2-1.png)  
**Constructor:**  
This won’t be taken as lifecycle hooks but this will instantiate all component hooks and it is run first when the component is activated.  
> app.ts file:  
class MyComponent {  
constructor( ) {  
/\*all other modules injection and dependency injection and initialization will go here\*/  
console.log(“This is Constructor”); // printed first in console  
}  
}

**ngOnChanges:**  
This hook will be run when our component is setting or resetting the values for input properties.This will be called before ngOnInit hook whenever the changes in the input property made in a component.  
This will be used to render the DOM and updating the DOM for each and every changes made in the component attributes.  
This may run multiple times in the lifetime of the each component  
> app.ts file:  
class MyComponent {  
ngOnChanges( ) {  
/\*Called when the changes made in the input properties of the component before it binds to view\*/  
console.log(“This is ngOnChange”); // printed second in console  
}  
}  
**ngOnInit:**  
The Initialization of the component or directive of our app will be made in this hook after the angular displays the initial value of input properties.  
When changes made in input properties of the component this hook will be called after the ngOnChanges.  
This hook will run only one time after initializing all the properties of the component  
> app.ts file:  
class MyComponent {  
ngOnInit( ) {  
/\*the Initialization of every input properties and functionalities will go here\*/  
console.log(“This is ngOnInit”); // printed third in console  
}  
}

**ngDoCheck:**  
This will be called immediately after ngOnInit, And called on every change made in component properties, So the operations that have to be done on the change of any input properties can be written in this hook.  
This will be executed for every change detection in cycles even there is no properties changed  
> app.ts file:  
class MyComponent {  
ngDoCheck( ) {  
/\*The functionalities to be done for every change made in input properties will go here\*/

console.log(“This is ngDoCheck”); //printed whenever the changes made  
}  
}  
**ngAfterContentInit:**  
This will be called after the angular updates the component’s view values this will be called after ngDoCheck hook.  
This hook can be written only for component of angular application  
If the content is inserted in the component this will be called  
> app.ts file:  
class MyComponent {  
ngAfterContentInit( ) {  
/\*The functionalities to be done when initialization of the whole content in the component will go here\*/  
console.log(“This is ngAfterContentInit”); /\*printed when the whole content is initialized\*/  
}  
}

**ngAfterContentChecked:**  
This hook will be called after the angular checked the content has been projected into the component. This will be called after ngAfterContentInit and every call of ngDoCheck hook of the component  
This hook also can be written only for component of angular application  
The component this will be called when the content which is inserted has been changed

>app.ts file:  
class MyComponent {  
ngAfterContentChecked( ) {  
/\*The functionalities to be done when changes present in the component will go here\*/  
console.log(“This is ngAfterContentChecked”); /\*printed when the content is checked and changes present\*/  
}  
}

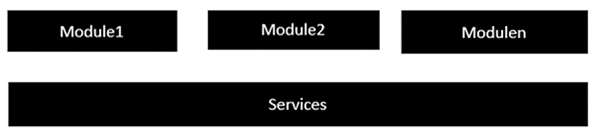
**ngAfterViewInit:**  
Called when the angular checks the view of the component and child views of the components.  
It will check all the HTML view has been initialized  
> app.ts file:  
class MyComponent {  
ngAfterViewInit( ) {  
/\*The functionalities to be done when initialization of the whole content has been projected into view will go here\*/  
console.log(“This is ngAfterViewInit”); /\*printed when the whole content is projected in view and runs only once\*/  
}  
}

**ngAfterViewChecked:**  
Initially this hook will be called after the ngAfterContentChecked hook and after every call of ngAfterViewInit and subsequent of ngAfterContentChecked.  
It will check all initialized view or child view had any changes  
> app.ts file:  
class MyComponent {  
ngAfterViewChecked( ) {  
/\*The functionalities to be done when changes present in the component and they are projected in view will go here\*/  
console.log(“This is ngAfterViewChecked”); /\*printed when the content is checked and changes present are projected in view\*/  
}  
}  
**ngDestroy:**  
This hook will be called only when the component is removed from the view of the angular component, So the deallocation of memory and removing the intervals, timeout variables will be done in this hook.

app.ts file:  
class MyComponent {  
ngDestory ( ) {  
/\*The functionalities to be done when the component is unmounted from the app will go here\*/  
console.log(“This is ngDestory”); /\*printed in after the component is unmounted\*/  
}  
}

**Services**

A service is used when a common functionality needs to be provided to various modules. For example, we could have a database functionality that could be reused among various modules. And hence you could create a service that could have the database functionality.



The following key steps need to be carried out when creating a service.

**Step 1** − Create a separate class which has the injectable decorator. The injectable decorator allows the functionality of this class to be injected and used in any Angular JS module.

@Injectable()

export class classname {

}

**Step 2** − Next in your appComponent module or the module in which you want to use the service, you need to define it as a provider in the @Component decorator.

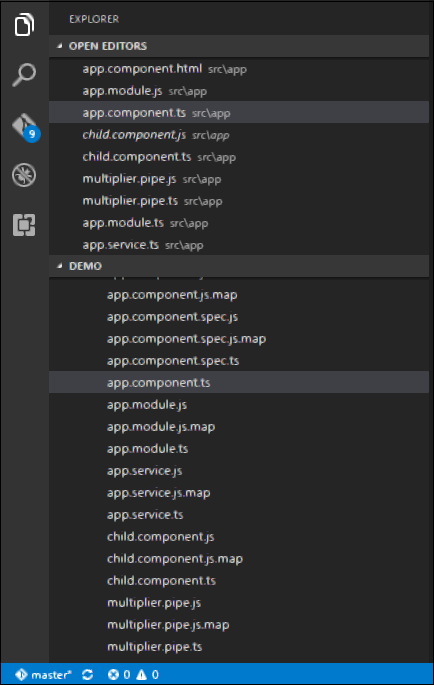
@Component ({

providers : [classname]

})

Let�s look at an example on how to achieve this. Following are the steps involved.

**Step 1** − Create a **ts** file for the service called app.service.ts.



**Step 2** − Place the following code in the file created above.

import {

Injectable

} from '@angular/core';

@Injectable()

export class appService {

getApp(): string {

return "Hello world";

}

}

Following points need to be noted about the above program.

* The Injectable decorator is imported from the angular/core module.
* We are creating a class called appService that is decorated with the Injectable decorator.
* We are creating a simple function called getApp, which returns a simple string called �Hello world�.

**Step 3** − In the app.component.ts file, place the following code.

import {

Component

} from '@angular/core';

import {

appService

} from './app.service';

@Component ({

selector: 'demo-app',

template: '<div>{{value}}</div>',

providers: [appService]

})

export class AppComponent {

value: string = "";

constructor(private \_appService: appService) { }

ngOnInit(): void {

this.value = this.\_appService.getApp();

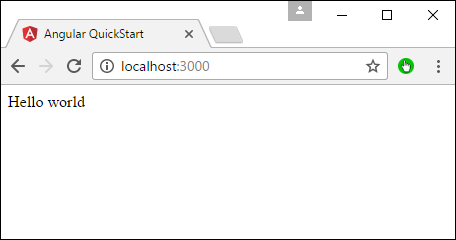
}

}

Following points need to be noted about the above program.

* First, we import our appService module in the appComponent module.
* Then, we register the service as a provider in this module.
* In the constructor, we define a variable called \_appService of the type appService so that it can be called anywhere in the appComponent module.
* As an example, in the ngOnInit lifecyclehook, we called the getApp function of the service and assign the output to the value property of the AppComponent class.

Once you save all the code changes and refresh the browser, you will get the following output.



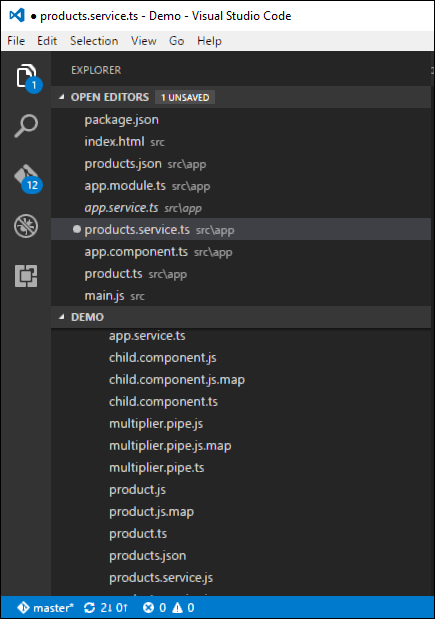
**HTTP**

The basic CRUD operation we will look into this chapter is the reading of data from a web service using Angular 2.

Example

In this example, we are going to define a data source which is a simple **json** file of products. Next, we are going to define a service which will be used to read the data from the **json** file. And then next, we will use this service in our main app.component.ts file.

**Step 1** − First let’s define our product.json file in Visual Studio code.



In the products.json file, enter the following text. This will be the data which will be taken from the Angular JS application.

[{

"ProductID": 1,

"ProductName": "ProductA"

},

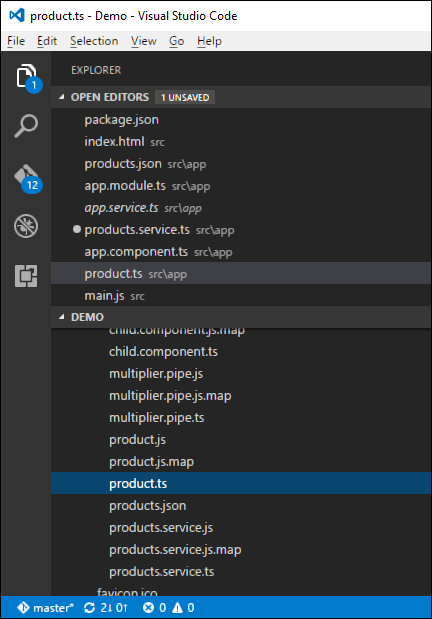
{

"ProductID": 2,

"ProductName": "ProductB"

}]

**Step 2** − Define an interface which will be the class definition to store the information from our products.json file. Create a file called products.ts.



**Step 3** − Insert the following code in the file.

export interface IProduct {

ProductID: number;

ProductName: string;

}

The above interface has the definition for the ProductID and ProductName as properties for the interface.

**Step 4** − In the app.module.ts file include the following code −

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

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

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

import { HttpModule } from '@angular/http';

@NgModule ({

imports: [ BrowserModule,HttpModule],

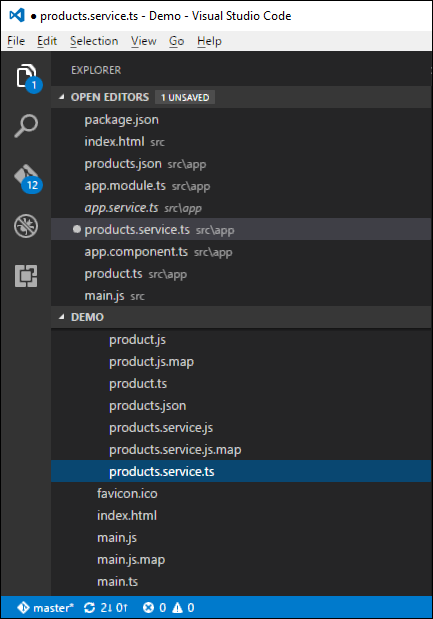
declarations: [ AppComponent],

bootstrap: [ AppComponent ]

})

export class AppModule { }

**Step 5** − Define a products.service.ts file in Visual Studio code



**Step 6** − Insert the following code in the file.

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

import { Http , Response } from '@angular/http';

import { Observable } from 'rxjs/Observable';

import 'rxjs/add/operator/map';

import 'rxjs/add/operator/do';

import { IProduct } from './product';

@Injectable()

export class ProductService {

private \_producturl='app/products.json';

constructor(private \_http: Http){}

getproducts(): Observable<IProduct[]> {

return this.\_http.get(this.\_producturl)

.map((response: Response) => <IProduct[]> response.json())

.do(data => console.log(JSON.stringify(data)));

}

}

Following points need to be noted about the above program.

* The import {Http, Response} from '@angular/http' statement is used to ensure that the http function can be used to get the data from the products.json file.
* The following statements are used to make use of the Reactive framework which can be used to create an Observable variable. The Observable framework is used to detect any changes in the http response which can then be sent back to the main application.

import { Observable } from 'rxjs/Observable';

import 'rxjs/add/operator/map';

import 'rxjs/add/operator/do';

* The statement private \_producturl = 'app/products.json' in the class is used to specify the location of our data source. It can also specify the location of web service if required.
* Next, we define a variable of the type Http which will be used to get the response from the data source.
* Once we get the data from the data source, we then use the JSON.stringify(data) command to send the data to the console in the browser.

**Step 7** − Now in the app.component.ts file, place the following code.

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

import { IProduct } from './product';

import { ProductService } from './products.service';

import { appService } from './app.service';

import { Http , Response } from '@angular/http';

import { Observable } from 'rxjs/Observable';

import 'rxjs/add/operator/map';

@Component ({

selector: 'my-app',

template: '<div>Hello</div>',

providers: [ProductService]

})

export class AppComponent {

iproducts: IProduct[];

constructor(private \_product: ProductService) {

}

ngOnInit() : void {

this.\_product.getproducts()

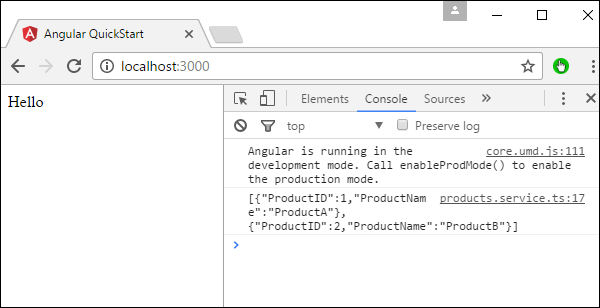
.subscribe(iproducts => this.iproducts = iproducts);

}

}

Here, the main thing in the code is the subscribe option which is used to listen to the Observable getproducts() function to listen for data from the data source.

Now save all the codes and run the application using **npm**. Go to the browser, we will see the following output.



In the Console, we will see the data being retrieved from products.json file

**Error Handeling**

Angular 2 applications have the option of error handling. This is done by including the ReactJS catch library and then using the catch function.

Let’s see the code required for error handling. This code can be added on top of the chapter for CRUD operations using http.

In the product.service.ts file, enter the following code −

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

import { Http , Response } from '@angular/http';

import { Observable } from 'rxjs/Observable';

import 'rxjs/add/operator/map';

import 'rxjs/add/operator/do';

import 'rxjs/add/operator/catch';

import { IProduct } from './product';

@Injectable()

export class ProductService {

private \_producturl = 'app/products.json';

constructor(private \_http: Http){}

getproducts(): Observable<IProduct[]> {

return this.\_http.get(this.\_producturl)

.map((response: Response) => <IProduct[]> response.json())

.do(data => console.log(JSON.stringify(data)))

.catch(this.handleError);

}

private handleError(error: Response) {

console.error(error);

return Observable.throw(error.json().error());

}

}

* The catch function contains a link to the Error Handler function.
* In the error handler function, we send the error to the console. We also throw the error back to the main program so that the execution can continue.

Now, whenever you get an error it will be redirected to the error console of the browser.

**Router**

Routing helps in directing users to different pages based on the option they choose on the main page. Hence, based on the option they choose, the required Angular Component will be rendered to the user.

Let’s see the necessary steps to see how we can implement routing in an Angular 2 application.

**Step 1** − Add the base reference tag in the index.html file.

<!DOCTYPE html>

<html>

<head>

<base href = "/">

<title>Angular QuickStart</title>

<meta charset = "UTF-8">

<meta name = "viewport" content = "width = device-width, initial-scale = 1">

<base href = "/">

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

<!-- Polyfill(s) for older browsers -->

<script src = "node\_modules/core-js/client/shim.min.js"></script>

<script src = "node\_modules/zone.js/dist/zone.js"></script>

<script src = "node\_modules/systemjs/dist/system.src.js"></script>

<script src = "systemjs.config.js"></script>

<script>

System.import('main.js').catch(function(err){ console.error(err); });

</script>

</head>

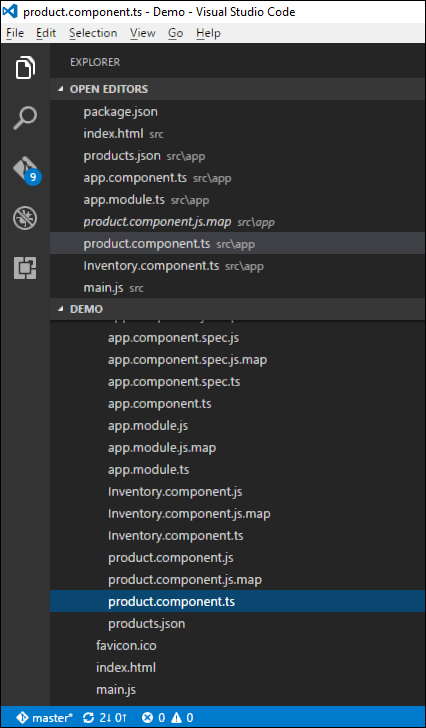
<body>

<my-app></my-app>

</body>

</html>

**Step 2** − Create two routes for the application. For this, create 2 files called **Inventory.component.ts** and **product.component.ts**



**Step 3** − Place the following code in the product.component.ts file.

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

@Component ({

selector: 'my-app',

template: 'Products',

})

export class Appproduct {

}

**Step 4** − Place the following code in the Inventory.component.ts file.

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

@Component ({

selector: 'my-app',

template: 'Inventory',

})

export class AppInventory {

}

Both of the components don’t do anything fancy, they just render the keywords based on the component. So for the Inventory component, it will display the Inventory keyword to the user. And for the products component, it will display the product keyword to the user.

**Step 5** − In the app.module.ts file, add the following code −

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

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

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

import { Appproduct } from './product.component';

import { AppInventory } from './Inventory.component';

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

const appRoutes: Routes = [

{ path: 'Product', component: Appproduct },

{ path: 'Inventory', component: AppInventory },

];

@NgModule ({

imports: [ BrowserModule,

RouterModule.forRoot(appRoutes)],

declarations: [ AppComponent,Appproduct,AppInventory],

bootstrap: [ AppComponent ]

})

export class AppModule { }

The following points need to be noted about the above program −

* The appRoutes contain 2 routes, one is the Appproduct component and the other is the AppInventory component.
* Ensure to declare both of the components.
* The RouterModule.forRoot ensures to add the routes to the application.

**Step 6** − In the app.component.ts file, add the following code.

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

@Component ({

selector: 'my-app',

template: `

<ul>

<li><a [routerLink] = "['/Product']">Product</a></li>

<li><a [routerLink] = "['/Inventory']">Inventory</a></li>

</ul>

<router-outlet></router-outlet>`

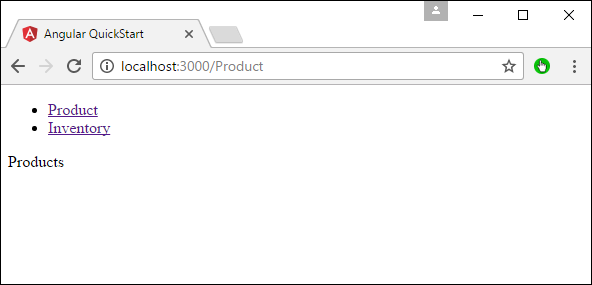
})

export class AppComponent { }

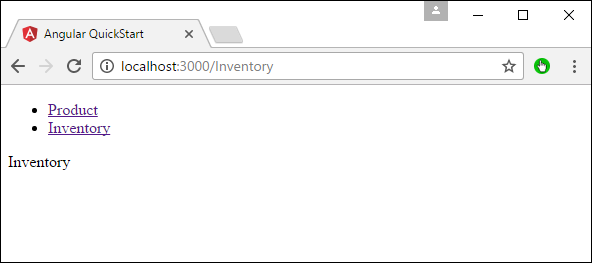
The following point needs to be noted about the above program −

* <router-outlet></router-outlet> is the placeholder to render the component based on which option the user chooses.

Now, save all the code and run the application using npm. Go to the browser, you will see the following output.



Now if you click the Inventory link, you will get the following output.

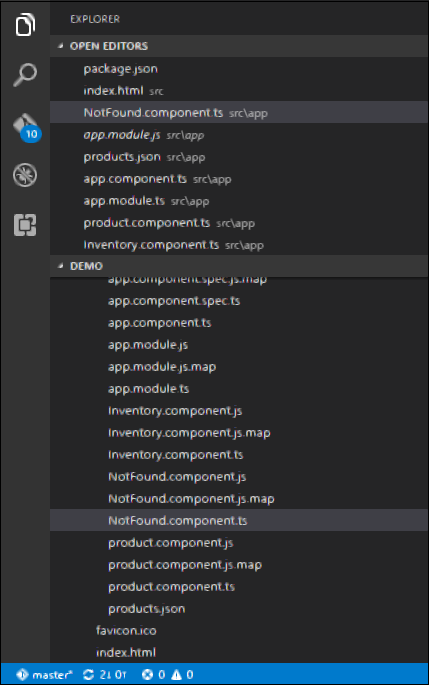


Adding an Error Route

In Routing, one can also add an error route. This can happen if the user goes to a page which does not exist in the application.

Let’s see how we can go about implementing this.

**Step 1** − Add a PageNotFound component as NotFound.component.ts as shown below −



**Step 2** − Add the following code to the new file.

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

@Component ({

selector: 'my-app',

template: 'Not Found',

})

export class PageNotFoundComponent {

}

**Step 3** − Add the following code to the app.module.ts file.

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

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

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

import { Appproduct } from './product.component'

import { AppInventory } from './Inventory.component'

import { PageNotFoundComponent } from './NotFound.component'

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

const appRoutes: Routes = [

{ path: 'Product', component: Appproduct },

{ path: 'Inventory', component: AppInventory },

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

];

@NgModule ({

imports: [ BrowserModule,

RouterModule.forRoot(appRoutes)],

declarations: [ AppComponent,Appproduct,AppInventory,PageNotFoundComponent],

bootstrap: [ AppComponent ]

})

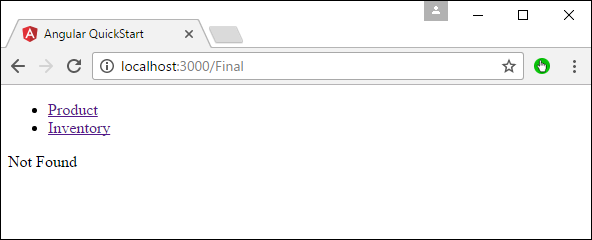
export class AppModule {

}

The following point needs to be noted about the above program −

* Now we have an extra route called path: '\*\*', component: PageNotFoundComponent. Hence, \*\* is for any route which does not fit the default route. They will be directed to the PageNotFoundComponent component.

Now, save all the code and run the application using npm. Go to your browser, and you will see the following output. Now, when you go to any wrong link you will get the following output.



Promises vs Observables

* promise:
  1. returns a single value
  2. not cancellable
  3. more readable code with try/catch and async/await
* observable
  1. works with multiple values over time
  2. cancellable
  3. supports map, filter, reduce and similar operators
  4. use Reactive Extensions (RxJS)
  5. an array whose items arrive asynchronously over time