

Angular Documentation

1.Introduction

Angular is an application design framework and development platform for creating efficient and sophisticated web-pages.

An Angular is a development platform, built on typescript. As a platform, Angular Includes:

- A component-based framework for building scalable web applications.
- A collection of well-integrated libraries that cover a wide verity of features, including routing, forms, management, client-server communication and more.
- A suite of developer tools to help you develop, build, test and update your code.

Angular provides the scalability from a signal-developer project to the enterprises level application. Angular is design to make updating as straight-forward as possible, so take advantage of the latest development with minimum effort.

1.2 Getting started:

Angular framework needs the support of Html, CSS, TypeScript and JavaScript.

1.3 Set Up:

To install angular in your system, you need to following:

- Nodejs: install the latest Nodejs version in your system.
- npm packages: Angular and Angular cli depend upon the npm packages for many features and function.
- Install Angular cli: `npm install -g @angular/cli`

1.4 Create a new Project:

Step 1: Run the cli command `ng new` and provide the project name *my-app*.

```
ng new my-app
```

Step 2: Navigate to the work space folder such as *my-app*.

Step 3: Open the Command prompt (recommended: Git bash).

```
cd my-app
```

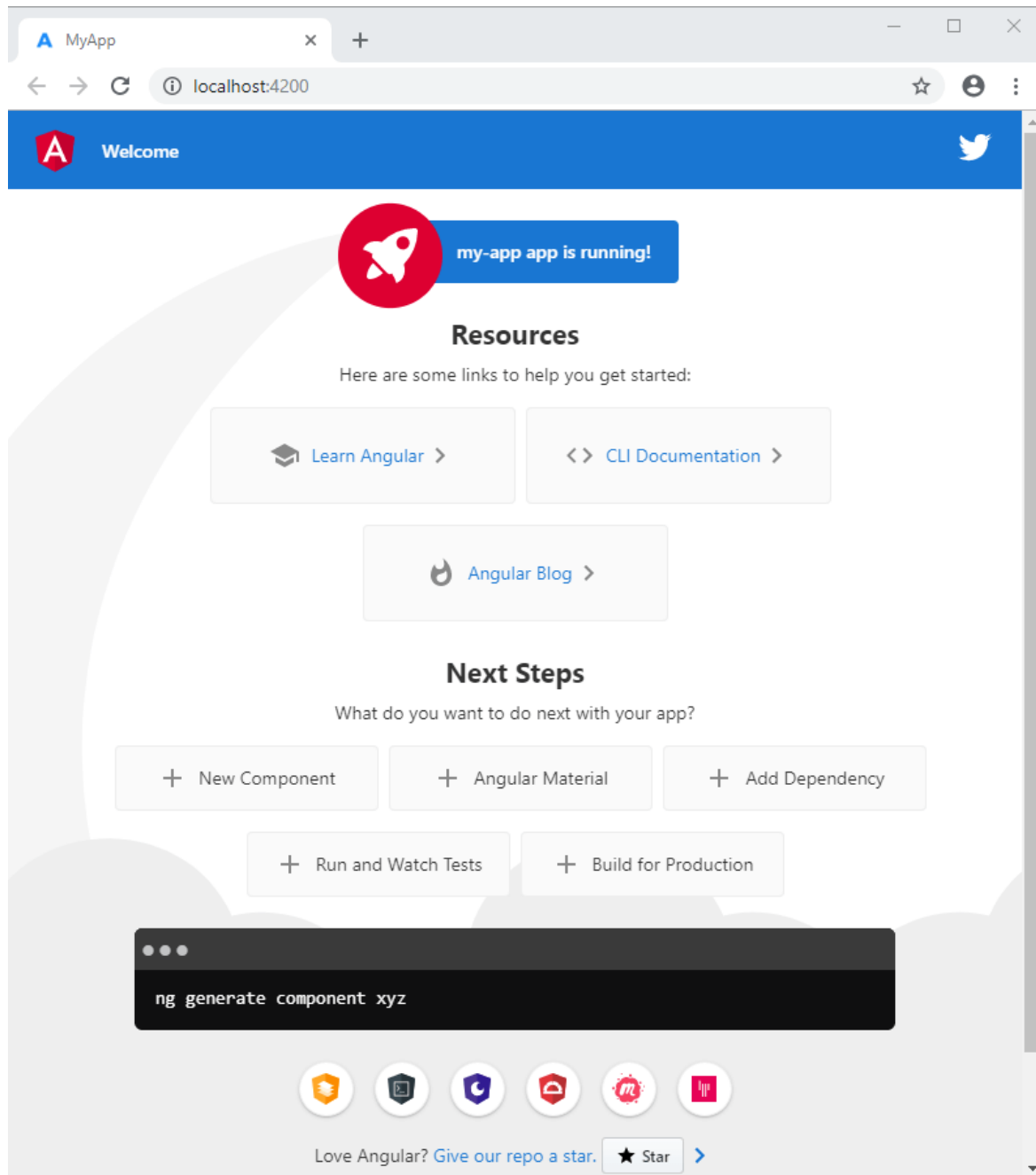
```
ng serve --open
```

The `ng serve` command launches the server, watches your files and rebuilds the app as you make changes on those files.

The `--open` automatically opens the project on:

`http://localhost:4200/`

If everything went perfectly the page look like this:



2. Angular in Detail

The *ng generate* (or simply *ng g*) command allows you to automatically generate Angular components:

The command below will generate a component in the folder you are currently at

```
ng generate component my-generated-component
```

Using the alias (same outcome as above)

```
ng g component my-generated-component
```

You can add *--flat* if you don't want to create new folder for a component

```
ng g component my-generated-component --flat
```

You can add *--spec false* if you don't want a test file to be generated (my-generated-component.spec.ts)

```
ng g component my-generated-component --spec false
```

There are several possible types of scaffolds angular-cli can generate:

Scaffold Type	Usage
Module	<i>ng g module my-module-name</i>
Component	<i>ng g component my-component-name</i>
Service	<i>ng g service my-service-name</i>
Class	<i>ng g class my-class-name</i>
Pipes	<i>ng g pipes my-pipes-name</i>
Interface	<i>ng g interface my-interface name</i>
Enum	<i>ng g enum my-new-enum</i>
Directive	<i>ng g directive my-new-directive</i>

To understand the capability of the angular framework, we need to consider the following:

1. Components.
2. Templates.
3. Directives.
4. Dependency Injection.

2.1 Component

Components are the building blocks of a UI in an angular application. These components are associated with a template and are a subset of directives.

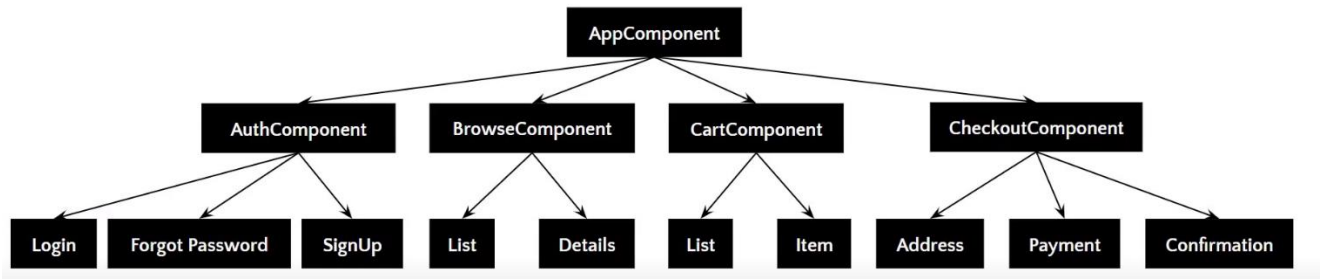


Fig: Example of component

The above image gives the tree structure of classification. There's a root component, which is the AppComponent, that then branch out to other component creating a hierarchy.

To create a new Angular component, Angular CLI is used. In the terminal type:

ng g component component-name

This will create a folder named component-name with four files.

```
newcomponent
# newcomponent.component.css
<> newcomponent.component.html
TS newcomponent.component.ts
TS newcomponent.component.spec.ts
```

Here are some of the features of angular Component:

- Component are typically custom HTML elements, and each of these elements can instantiate only one component.
- A typescript class is used to create a component. This class is then decorated with the “@Component” decorator.
- The decorator accepts a metadata object that gives information about the component.
- A component must belong to the Ng Module in order for it to be usable by another component or application.
- Component control their runtime behavior by implementing life-cycle hooks.

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'example';
}
```

The above image shows an App component, which is a pure TypeScript class decorated with the “@Component” decorator. The metadata object provides properties like selector, template-URL, and so on—the template URL points to an HTML file that defines what you see on your application.

In the index.html file, <app-root> tag corresponds to component’s selector. By doing so, Angular will inject the corresponding template of the component.

```
<body>
  <app-root></app-root>
</body>
</html>
```

2.1.1 Life Cycle Hook

When the angular application starts it creates and renders the root component. It then creates and renders its children’s & their children. It forms a tree of components. Once Angular loads the components, it starts the process of rendering the view. To do that it needs to check the input properties, evaluate the data bindings & expressions, render the projected content etc. Angular also

removes the component from the DOM, when it is no longer needs it. Angular lets us know when these events happen using lifecycle hooks. The Angular life cycle hooks are nothing but callback function, which angular invokes when a certain event occurs during the component's life cycle. Here is the complete list of life cycle hooks, which angular invokes during the component life cycle. Angular invokes them when a certain event occurs.

These are the hooks for components or directives, in call order:

1. constructor()
2. OnInit
3. DoCheck
4. OnChanges
5. OnDestroy

And these are the hooks for a component's children components:

1. AfterContentInit
2. AfterContentChecked
3. AfterViewInit
4. AfterViewChecked

Below is a summary of the Angular lifecycle hooks in a table:

OnInit	Called on initialization
OnChanges	Called when the input properties have changed
DoCheck	Developer's custom change detection
Destroy	Called before the component is destroyed
AfterContentInit	Called when the component's content <code>ngContent</code> is initialized
AfterContentChecked	Called when the component's content is updated or checked for updates
AfterViewInit	Called when the component's projected view has been initialized
AfterViewChecked	Called after the projected view has been checked

Hooks for component or directives

In Angular, components are the smallest units of building blocks for a component tree. This enables us to develop and maintain our codebase very easily. Components are classes with the `@Component()` decorator on them, like so:

```
@Component({
  selector: "app-comp",
  template: `<div>AppComponent</div>`,
  styles: ``
})
class AppComponent implements OnInit {
  //...
}
```

Directives are technically components, but they operate on the non-UI part of the DOM. Their work is to add extra functionality to the DOM. Directives are also classes, but decorated with the `@Directive()` class decorator.

Examples of in-built directives are `*ngFor`, `*ngIf`, etc. We can create our own directive by doing this:

```
@Directive({
  selector: '[highlight-text]'
})
export class HighlightDirective {
  //...
}
```

The following are lifecycle hooks for components or directives, with explanations of how they work and how to use them.

1. Constructor()

Life Cycle of a component begins, when Angular creates the component class. First method that gets invoked is class Constructor. Constructor is neither a life cycle hook nor it is specific to Angular. It is a JavaScript feature. It is a method which is invoked, when a class is created. Angular makes use of a constructor to inject dependencies.

At this point, none of the components input properties are available to use. Neither its child components are constructed. Projected contents are also not available. Hence there is not much you can do in this method. And also, it is recommended not to use it. Once Angular instantiates the class, it kick-start the first change detection cycle of the component.

2. OnInit

OnInit is a lifecycle hook that is called after Angular has initialized all data-bound properties of a directive. Define an `ngOnInit()` method to handle any additional initialization tasks. This hook is called when the first change detection is run on the component. After the first run it is switched off never to be run again. It is a best practice to add our initialization logic here.

To hook into this, we will implement the `OnInit` interface and add `ngOnInit` method in our component / directives.

```
@Component({
  //...
})
export class BComponent implements OnInit {
  ngOnInit() {
    console.log("ngOnInit called")
  }
}
```

3. OnChanges

when any data-bound property of a directive or component changes, the `OnChanges` lifecycle hook is called. Let's have a look into example –

```
@Component({
  selector: 'example',
  template:
    `<div *ngFor="let customer of customers">
      {{customer}}
    </div>
  `
})
export class ExampleComponent implements OnChanges {
  @Input() customers;
  ngOnChanges() {
    console.log("The property of customer changed");
  }
}

@Component({selector: 'example1',
  template:
    `<example [customers]="customers"></example>
  `
})
export class Example1Component {
```



```
customers = ["Customer1"];  
}
```

Here we have used interface `OnChanges` on `ExampleComponent` and called method `ngOnChanges()` which will call the `OnChanges` lifecycle hook. `Example1Component` is binding its `customers` property to `ExampleComponent`. On change of `customers` property from `Example1Component`, `ngOnChanges()` method of `ExampleComponent` will call. In short, every change of `@input` value will call `ngOnChanges()` lifecycle hook.

4. DoCheck

This lifecycle hook comes after `ngOnInit()` and basically calls to detect and act upon changes that Angular can't or won't detect on its own. `DoCheck` invokes a custom change-detection function for a directive, in addition to the check performed by the default change-detector.

```
export class ExampleComponent implements OnChanges, DoCheck  
{  
  @Input() customers;  
  
  ngOnChanges()  
  {  
    console.log("The customers property has been changed");  
  }  
  ngDoCheck()  
  {  
    console.log("ExampleComponent's ngDoCheck called.");  
  }  
}
```

5. OnDestroy()

`ngOnDestroy()` is a lifecycle hook that is called when a directive, pipe, or service is destroyed. This lifecycle hook is used for any custom cleanup that needs to occur when the instance is destroyed, by doing so you can prevent memory leakage. This lifecycle hook is mostly used to unsubscribe from observable streams and detach event handlers to avoid memory leaks.

6. AfterContentInit()

ngAfterContentInit() is called when the content of a component has initialized. In this case, content is the component that are within the <ng-content></ng-content> tags. After ngDoCheck() it is called initially.

```
@Component({
  selector: 'example2',
  template:
    `<div>
    This is a Example2 Componnet</div>
    `
})
export class Example2Component {}
@Component({
  selector: example1,
  template: `
  <ng-content></ng-content>
  `})
export class Example1Component implements AfterContentInit{
  ngAfterContentInit()
  {
  }
}}
@Component({
  template: `
  <example1><example2></example2></example1>
  `
})
export class ExampleComponent {}
```

The Example1Component will have any elements in between its tag <example1></example1> inside the ng-content tag. Now, in the Examplecomponent, the Example2Component is projected in the example1 tag. When the Example2Component is being initialized, the ngAfterContentInit hook will be called in Example1Component.

7. ngAfterContentChecked()

This lifecycle hook performs its work by knowing the change in the content of the component using Angular change detection. It gets its call after ngAftercontentInit() and also gets executed after every execution od ngDoCheck().

```

@Component({
  selector: 'example1',
  template:
    `<ng-content></ng-content>`
})
export class Example1Component implements AfterContentInit {
  ngAfterContentInit() {
  }
}
@Component({
  template:
    `<example1> {{data}} </example1>`
})
export class ExampleComponent implements AfterContentChecked {
  data: any
  ngAfterContentChecked() {
  }
}

```

The ExampleComponent component has a data property that is inside the Example1Component. When the data property changes, the ngAfterContentChecked() method will be called.

8. ngAfterViewInit()

This lifecycle hook gets called after ngAfterContentChecked(). And also gets called after a component's view and its children's views have been created and fully initialized. This lifecycle hook will be useful when we want to reference a component instance in our component using ViewChild/ViewChildren. After the initialization of view, it gets its call only once.

```

@Component({
  selector: example1,
  template:
    `
    `
  })
export class Example1Component {example1Method() {
  }}
@Component({
  template:
    `<example1 #example1></example1>`

```

```

    })
    export class ExampleComponent implements AfterViewInit{
    @ViewChild(example1) example1: Example1Component;
        constructor() {
        }
    ngAfterViewInit() {
        this.example1.example1Method();
    }
    }

```

We got the reference of example1's class Example1Component by putting # in the ExampleComponent template. Then, we used ViewChild to tell Angular to set the instance of Example1Component to example1 variable.

9. ngAfterViewChecked()

This Angular lifecycle method gets called subsequently as it checks the component's view and child view. This method gets called after ngAfterViewInit() and then for every ngAfterContentChecked() method. After checking and initialization are done, this lifecycle hook gets called.

Difference between constructor and ngOnInit here -

The Constructor is executed when the class is instantiated as a default method of the class and ensures all initialization of fields in the class. Angular, or better Dependency Injector (DI), analyses the constructor parameters and when it creates a new instance by calling a new class() it tries to find providers that match the types of the constructor parameters, resolves them, and passes them to the constructor. ngOnInit is a life cycle hook called by Angular after data bound is done for the first time and angular is done creating the component. Implementing OnInit is not mandatory but considered good practice.

```

export class ExampleComponent implements OnInit {
    constructor() {
        // Called first time before the ngOnInit()
    }
    ngOnInit() {
        // Called after the constructor and called after the first
        // ngOnChanges()
    }
}

```

Mostly we use `ngOnInit` for all the initialization or declaration and avoid doing work in the constructor. The constructor should only be used to initialize class members but shouldn't perform actual work. So you should use `constructor()` to set up Dependency Injection. `ngOnInit()` is a better place to “start” — it's where the component's bindings are resolved.

2.1.2 View Encapsulation

This defines template and style encapsulation options available for an Angular component. There are three members of the Angular view encapsulation:

- Emulated
- None
- Shadow DOM

We are going to use a demo application to understand the various members of the Angular view encapsulation.

Demo

We are going to build a simple app with buttons and styles to illustrate various members. Load up the app in your integrated development environment (I use VS Code). Open a new terminal and generate a new component with this command:

```
ng g c test
```

Angular

Now add these styles to the `styles.css` file in the root folder:

```
/* You can add global styles to this file, and also import other style files */  
  
.btn {  
  display: inline-block;  
  background: rgb(166, 190, 126);  
  color: white;  
  padding: 0.4rem 1.3rem;  
  font-size: 1rem;  
  text-align: center;  
  border: none;
```

```

    cursor: pointer;
    outline: none;
    border-radius: 2px;
    margin-right: 0.5rem;
    box-shadow: 0 1px 0 rgba(0, 0, 0, 0.45);
  }
  .btn:hover {
    opacity: 0.8;
  }

```

CSS

These are styles for the buttons we are going to use and a small effect on hover. Open the `app.component.html` file and clean it up to look like this:

```

<div style="text-align:center">
  <h1>Angular View Encapsulation</h1>
</div>
<h2>Hover over these dummy buttons</h2>
<ul>
  <li class="btn">
    <h2>
      <a target="_blank" rel="noopener" href="#">Tour of Heroes</a>
    </h2>
  </li>
  <li class="btn">
    <h2>
      <a target="_blank" rel="noopener" href="#">CLI Documentation</a>
    </h2>
  </li>
  <app-test></app-test>
</ul>
<router-outlet></router-outlet>

```

HTML

Here we have a simple list and we also brought in the test component. Go to your `test.component.html` file and replace the test works content with the code block below:

```

<li class="btn">

```

```
<h2>
  <a target="_blank" rel="noopener" href="#">Angular blog</a>
</h2>
</li>
```

HTML

Finally, go to the test.component.css file and add these rules:

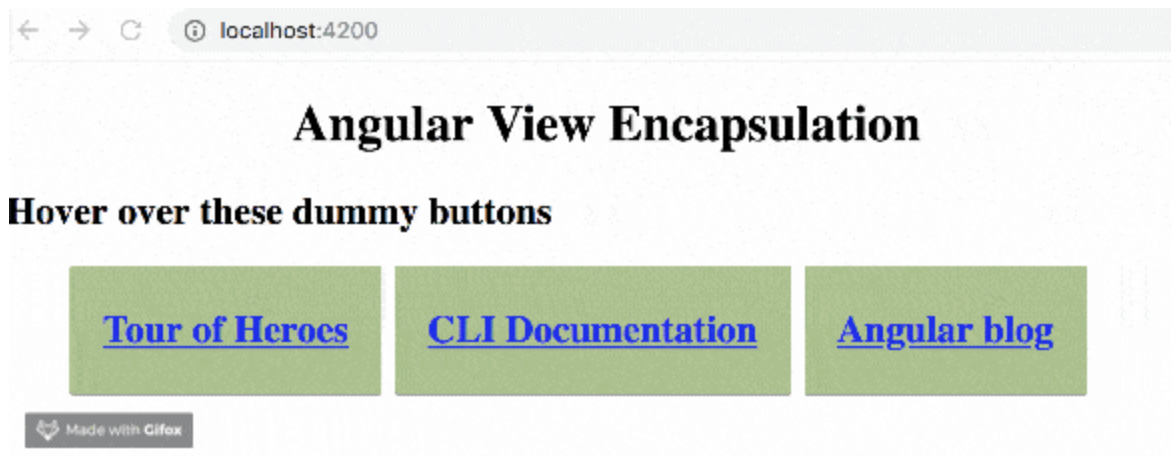
```
.btn:hover {
  opacity: 0.8;
  background: black;
}
```

CSS

This changes the color of a button on hover to black. Now everything is properly set up to test our view encapsulation members.

Run the application in the development server:

```
ng serve
```



This is how your application should look when you go to the localhost:4200 in your browser. You can notice how the scoped style in the test component does not affect the rest of the application

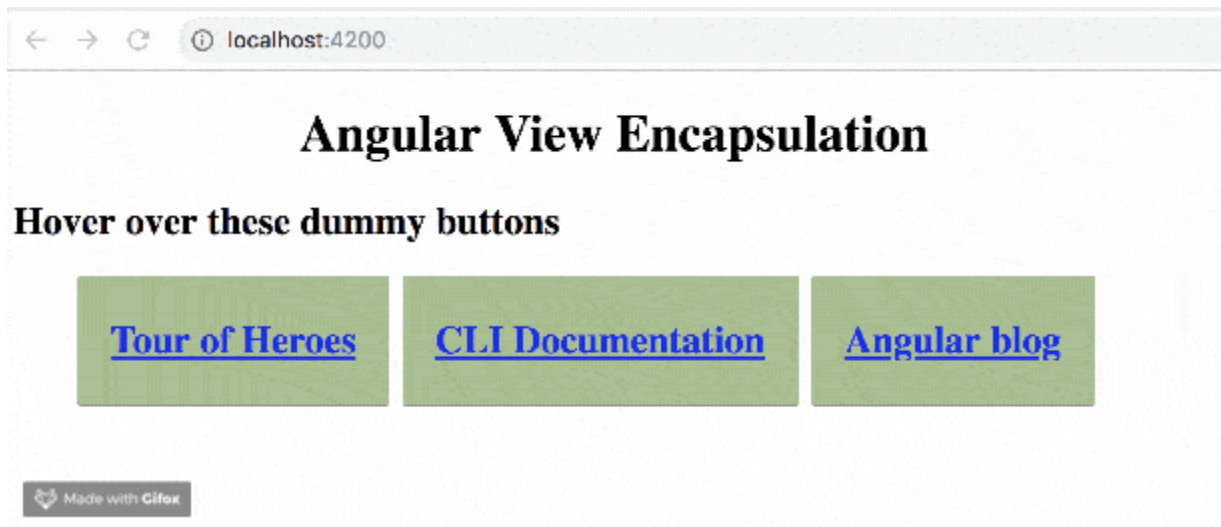
- **The None Member**

One of the options you have as an Angular developer is to specify that you do not want any kind of encapsulation of defined styles in your project. This can be very viable for projects which have a lot of contributors, like a distributed team. You might have specific style sheet or style guides that you do not want people modifying so you opt for this option. This also means that every style sheet or CSS rule you create inside the project is global no matter the location.

For our demo, we see that by default the view encapsulation is not set to none, so let's set it ourselves. In your `test.component.ts` file, go under the style section and modify the component definition section to this:

```
@component({  
  selector: "app-test",  
  templateUrl: "./test.component.html",  
  styleUrls: ["./test.component.css"],  
  encapsulation: ViewEncapsulation.None  
})
```

Now when you save your app, notice the difference in the user interface:



You can see that the scoped styles are now global, so for any component you can grant access of your style sheet to your parent component and other components in your project.

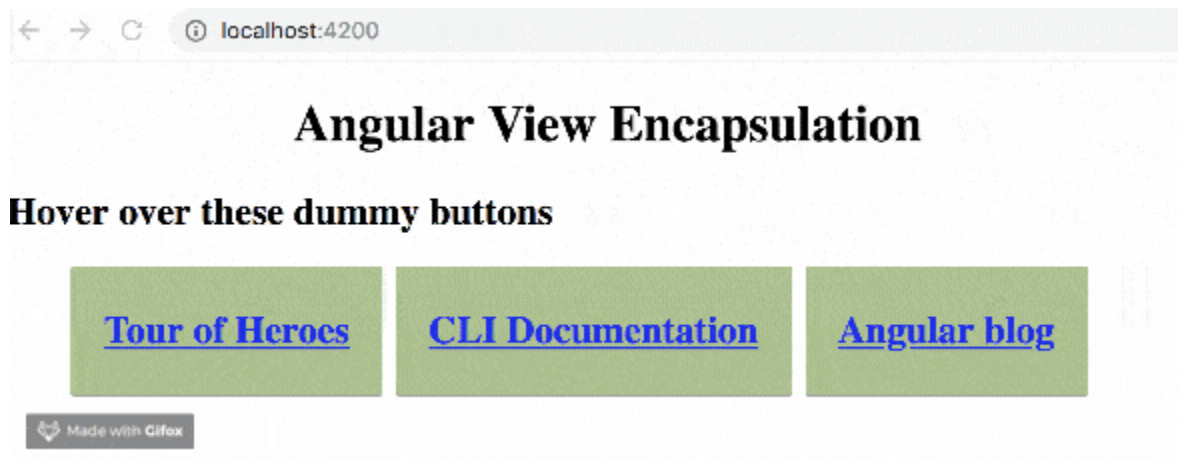
- **The Emulated Member**

This is the Angular default setup; it is simply a shadow DOM emulator. Angular achieves this by assigning custom attributes to the elements affected, especially as some browsers do not support shadow DOM. It kind of engineers a shadow DOM mechanism.

To test this out, you can remove the setup you added in the section above or explicitly define it like this:

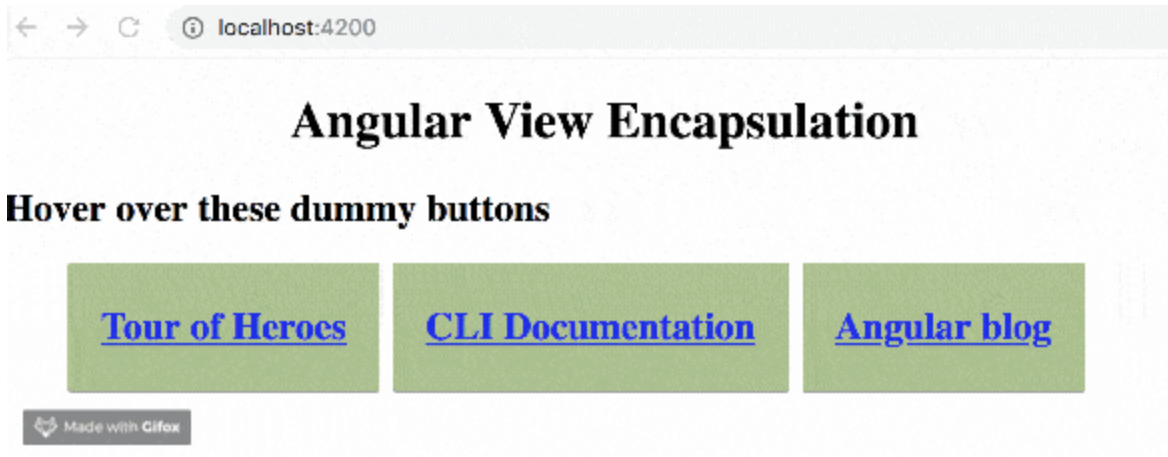
```
@component({  
  selector: "app-test",  
  templateUrl: "./test.component.html",  
  styleUrls: ["./test.component.css"],  
  encapsulation: ViewEncapsulation.Emulated  
})
```

If you run the application you see it goes back to how it was at the very start. This is the default Angular setting so without explicitly setting it up, it still does the same thing.



- **The Shadow DOM**

Here for all the browsers that support shadow DOM, you will see the same output as you saw in the emulated section. The difference is that styles are written in the document head for emulation but in the shadow DOM a shadow root is used at component level.



If you inspect, you will find that emulation used foreign ghost attributes like ng content and ng ghost but all you see with shadow DOM is the shadow root parent element.

2.1.3 Component Interaction

As we know that, Angular is a component-based approach, what good would it be if we cannot completely use this feature of interaction between components to make life easier.

But, before we can talk about this feature let us deeply focus on the structures occurring in angular application and their component-relationships. And it is a very important topic to understand how to structure our Angular application. The way to structure your angular application cannot be taught but only be caught as you evolve yourself as an angular developer. In the below image you can see one such structure in an Angular application for a general scenario.



So, visualizing the above scenario, there are four different ways of sharing/sending/interacting data to and from different components.

Parent to Child: Sharing Data via Input

This is probably the most common and straightforward method of sharing data. It works by using the `@Input()` decorator to allow data to be passed via the template. I will also show the way it is written in angular application to make it work

Parent.component.ts

```
import { Component } from '@angular/core'; @Component({
  selector: 'app-parent',
  template: `
    <app-child [childMessage]="parentMessage"></app-child>
  `,
  styleUrls: ['./parent.component.css']
})
export class ParentComponent{
  parentMessage = "message from parent"
  constructor() { }
}
```

child.component.ts

```
import { Component, Input } from '@angular/core'; @Component({
  selector: 'app-child',
  template: `
    Say {{ message }}
  `,
  styleUrls: ['./child.component.css']
})
export class ChildComponent { @Input() childMessage: string;
  constructor() { }
}
```

Child to Parent: Sharing Data Via ViewChild

ViewChild allows a one component to be injected into another, giving the parent access to its attributes and functions. One caveat, however, is that child won't be available until after the view has been initialized. This means we need to implement the `AfterViewInit` lifecycle hook to receive the data from the child.

Parent.component.ts

```
import { Component, ViewChild, AfterViewInit } from
 '@angular/core';
import { ChildComponent } from "../child/child.component";
@Component({
  selector: 'app-parent',
  template: `
    Message: {{ message }}
    <app-child></app-child>
  `,
  styleUrls: ['./parent.component.css']
})
export class ParentComponent implements AfterViewInit {
  @ViewChild(ChildComponent) child;
  constructor() {}
  message:string;
  ngAfterViewInit() {
    this.message = this.child.message
  }
}
```

child.component.ts

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-child',
  template: `
    `
  ,
  styleUrls: ['./child.component.css']
})
export class ChildComponent { message = 'Hello There!';
  constructor() {} }
```

Child to parent: sharing via Output() and Event Emitter

Another way to share data is to emit data from the child, which can be listened to by the parent. This approach is ideal when you want to share data changes that occur on things like button clicks, form entries, and other user events. In the parent, we create a function to receive the message and set it equal to the message variable. In the child, we declare a messageEvent variable with the Output decorator and set it equal to a new event emitter. Then we create a

function named `sendMessage` that calls `emit` on this event with the message we want to send. Lastly, we create a button to trigger this function. The parent can now subscribe to this `messageEvent` that's outputted by the child component, then run the `receive message` function whenever this event occurs.

Parent.component.ts

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-parent',
  template: `
    Message: {{message}}
    <app-child(messageEvent)="receiveMessage($event)"></app-child>
  `,
  styleUrls: ['./parent.component.css']
})
export class ParentComponent {
  constructor() { }
  message:string;
  receiveMessage($event) {
    this.message = $event
  }
}
```

child.component.ts

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

@Component({
  selector: 'app-child',
  template: `
    <button (click)="sendMessage()">Send Message</button>
  `,
  styleUrls: ['./child.component.css']
})
export class ChildComponent {

  message: string = "Hola Mundo!"

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

  constructor() { }

  sendMessage() {
```

```
this.messageEvent.emit(this.message)
}
}
```

Unrelated Components: Sharing Data with a Service

When passing data between components that lack a direct connection, such as siblings, grandchildren, etc, you should use a shared service. When you have data that should always be in sync, I find the [RxJS BehaviorSubject](#) very useful in this situation.

You can also use a regular RxJS Subject for sharing data via the service, but here's why I prefer a BehaviorSubject.

- It will always return the current value on subscription — there is no need to call `onnext`
- It has a `getValue()` function to extract the last value as raw data.
- It ensures that the component always receives the most recent data.

In the service, we create a private BehaviorSubject that will hold the current value of the message. We define a `currentMessage` variable handle this data stream as an observable that will be used by the components. Lastly, we create function that calls `next` on the BehaviorSubject to change its value.

The parent, child, and sibling components all receive the same treatment. We inject the `DataService` in the constructor, then subscribe to the `currentMessage` observable and set its value equal to the message variable.

Now if we create a function in any one of these components that changes the value of the message. when this function is executed the new data it's automatically broadcast to all other components.

data.service.ts

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

@Injectable()
export class DataService {

  private messageSource = new BehaviorSubject('default
```

```

message');
    currentMessage = this.messageSource.asObservable();

    constructor() { }

    changeMessage(message: string) {
        this.messageSource.next(message)
    }
}

```

parent.component.ts

```

import { Component, OnInit } from '@angular/core';
import { DataService } from "../data.service";

@Component({
    selector: 'app-parent',
    template: `
        {{message}}
    `,
    styleUrls: ['./sibling.component.css']
})
export class ParentComponent implements OnInit {

    message:string;

    constructor(private data: DataService) { }

    ngOnInit() {
        this.data.currentMessage.subscribe(message => this.message
= message)
    }

}

```

sibling.component.ts

```

import { Component, OnInit } from '@angular/core';
import { DataService } from "../data.service";
@Component({
    selector: 'app-sibling',
    template: `
        {{message}}
        <button (click)="newMessage()">New Message</button>
    `,

```

```
styleUrls: ['./sibling.component.css']
}))
export class SiblingComponent implements OnInit {
  message:string; constructor(private data: DataService) { }
  ngOnInit() {
    this.data.currentMessage.subscribe(message => this.message =
    message)
  }
  newMessage() {
    this.data.changeMessage("Hello from Sibling")
  }
}
```

2.1.4 Angular elements

Code reuse is a very important feature that all developers seek to implement. We always want to write a functionality once and use it reuse it when the need arises. By reusing code, developers can drastically cut development and maintenance time for software projects. This is most times achieved by creating custom elements.

Angular elements are Angular components packaged as custom elements, a web standard for defining new HTML elements in a framework-agnostic way.

Custom elements are a Web Platform feature currently supported by Chrome, Firefox, Opera, and Safari, and available in other browsers through polyfills. A custom element extends HTML by allowing you to define a tag whose content is created and controlled by JavaScript code. The browser maintains a CustomElementRegistry of defined custom elements (also called Web Components), which maps an instantiable JavaScript class to an HTML tag.

The `@angular/elements` package exports a `createCustomElement()` API that provides a bridge from Angular's component interface and change detection functionality to the built-in DOM API.

Transforming a component to a custom element makes all of the required Angular infrastructure available to the browser. Creating a custom element is simple and straightforward, and automatically connects your component-defined view with change detection and data binding, mapping Angular functionality to the corresponding native HTML equivalents.

Using custom elements

Custom elements bootstrap themselves - they start automatically when they are added to the DOM, and are automatically destroyed when removed from the DOM. Once a custom element is added

to the DOM for any page, it looks and behaves like any other HTML element, and does not require any special knowledge of Angular terms or usage conventions.

Easy dynamic content in an Angular app

Transforming a component to a custom element provides an easy path to creating dynamic HTML content in your Angular app. HTML content that you add directly to the DOM in an Angular app is normally displayed without Angular processing, unless you define a dynamic component, adding your own code to connect the HTML tag to your app data and participate in change detection. With a custom element, all of that wiring is taken care of automatically.

Content-rich applications

If you have a content-rich app, custom elements let you give your content providers sophisticated Angular functionality without requiring knowledge of Angular.

How it works

Use the `createCustomElement()` function to convert a component into a class that can be registered with the browser as a custom element. After you register your configured class with the browser's custom-element registry, you can use the new element just like a built-in HTML element in content that you add directly into the DOM:

TypeScript Code:

```
`<my-popup message="Use Angular!"></my-popup>`
```

Copy

When your custom element is placed on a page, the browser creates an instance of the registered class and adds it to the DOM. The content is provided by the component's template, which uses Angular template syntax, and is rendered using the component and DOM data. Input properties in the component corresponding to input attributes for the element.

Transforming components to custom elements

Angular provides the `createCustomElement()` function for converting an Angular component, together with its dependencies, to a custom element. The function collects the component's observable properties, along with the Angular functionality the browser needs to create and destroy instances and to detect and respond to changes.

The conversion process implements the **NgElementConstructor** interface and creates a constructor class that is configured to produce a self-bootstrapping instance of your component.

Use a JavaScript function, **customElements.define()**, to register the configured constructor and its associated custom-element tag with the browser's **CustomElementRegistry**. When the browser encounters the tag for the registered element, it uses the constructor to create a custom-element instance.

Mapping

A custom element *hosts* an Angular component, providing a bridge between the data and logic defined in the component and standard DOM APIs. Component properties and logic maps directly into HTML attributes and the browser's event system.

The creation API parses the component looking for input properties and defines corresponding attributes for the custom element. It transforms the property names to make them compatible with custom elements, which do not recognize case distinctions. The resulting attribute names use dash-separated lowercase. For example, for a component with `@Input('myInputProp')` `inputProp`, the corresponding custom element defines an attribute `my-input-prop`.

Component outputs are dispatched as HTML Custom Events, with the name of the custom event matching the output name. For example, for a component with `@Output()` `valueChanged = new EventEmitter()`, the corresponding custom element will dispatch events with the name "valueChanged", and the emitted data will be stored on the event's detail property. If you provide an alias, that value is used; for example, `@Output('myClick')` `clicks = new EventEmitter<string>()`; results in dispatch events with the name "myClick".

2.2 Template

In Angular, a template is a chunk of HTML. Use special syntax within a template to build on many of Angular Features.

Before learning template syntax, you should be familiar with the following:

- Angular concepts
- JavaScript
- HTML
- CSS

2.2.1 String Interpolation

In Angular, String interpolation is used to display dynamic data on HTML template (at user end). It facilitates you to make changes on component.ts file and fetch data from there to HTML template (component.html file).

For Example:

component.ts file:

```
import {Component} from '@angular/core';
@Component({
  selector: 'app-server',
  templateUrl: 'server.component.html'})
export class ServerComponent {
  serverID: number = 10;
  serverStatus: string = 'Online';
}
```

Here, we have specified serverID and serverStatus with some values. Let's use this in "component.html" file.

component.html file:

```
<p>Server with ID {{serverID}} is {{serverStatus}}. </p>
```

String Interpolation Vs Property Binding

String Interpolation and Property binding both are used for same purpose i.e., one-way databinding. But the problem is how to know which one is best suited for your application. Here, we compare both in the terms of Similarities, Difference, Security and the output you receive.

Similarities between String Interpolation and Property binding

String Interpolation and Property Binding doth are about one-way data binding. They both flow a value in one direction from our components to HTML elements.

String Interpolation

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

@Component({
  selector: 'my-app',
  template: `
    <h1>{{ fullName }}</h1>
  `,
})
export class AppComponent {
  fullName: string = 'Robert Junior';
}
```

You can see in the above example, Angular takes value of the fullName property from the component and inserts it between the opening and closing <h1> element using curly braces used to specify interpolation.

Property Binding

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

@Component({
  selector: 'my-app',
  template: `
    <h1 [innerHTML]='fullName'></h1>
  `,
})
export class AppComponent {
  fullName: string = 'Robert Junior';
}
```

In Property binding, see how Angular pulls the value from fullName property from the component and inserts it using the html property innerHtml of <h1> element.

Both examples for string interpolation and property binding will provide the same result.

Different between string interpolation and property binding

String Interpolation is a special syntax which is converted to property binding by Angular. It's a convenient alternative to property binding.

When you need to concatenate strings, you must use interpolation instead of property binding.

Example:

```
@Component({
  selector: 'my-app',
  template: `

<h1>{{citedExample}}</h1>
  </div>`
})
export class AppComponent {
  citedExample: string = 'Interpolation foe string only';
}


```

Property Binding is used when you have to set an element property to a non-string data value.

Example:

In the following example, we disable a button by binding to the Boolean property isDisabled.

```
import { Component } from '@angular/core';
@Component({
  selector: 'my-app',
  template: `

<button [disabled]='isDisabled'>Disable me</button>
  </div>`
})
export class AppComponent {
  isDisabled: boolean = true;
}


```

If you use interpolation instead of property binding, the button will always be disabled regardless isDisabled class property value is true or false.

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

@Component({
  selector: 'my-app',
  template: `<div>
    <button disabled='{{isDisabled}}'>Disable Me</button>
  </div>`
})
export class AppComponent {
  isDisabled: boolean = true/false;
}
```

2.2.2 Event Binding

Angular facilitates us to bind the events along with the methods. This process is known as event binding. Event binding is used with parenthesis ().

Example:

```
@Component({
  selector: 'app-server2',
  templateUrl: './server2.component.html',
  styleUrls: ['./server2.component.css']
})
export class Server2Component implements OnInit {
  allowNewServer = false;
  serverCreationStatus= 'No Server is created.';
  constructor() {
    setTimeout(() =>{
      this.allowNewServer = true;
    }, 5000);
  }
}
```

```

ngOnInit() {
}

}

```

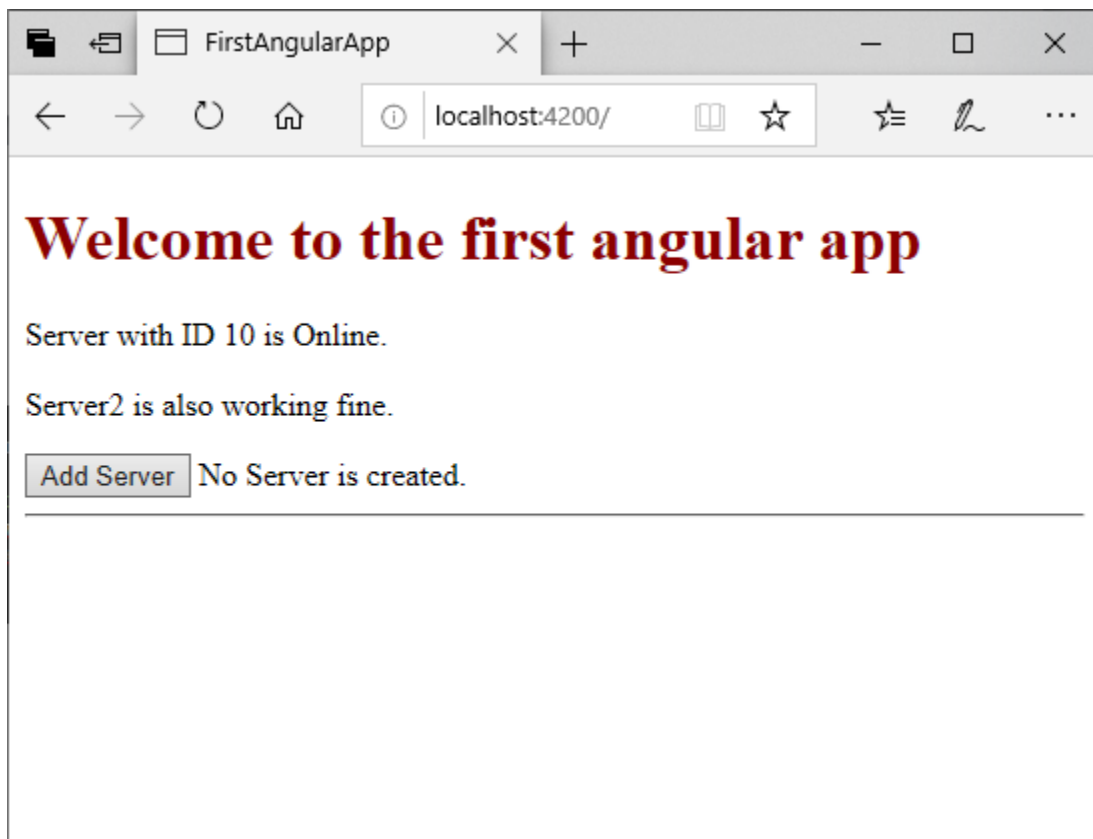
component.html file:

```

<p>
  Server2 is also working fine.
</p>
<button class="btn btn-primary"
  [disabled]="!allowNewServer" >Add Server</button>
<!--<h3 [innerText]= "allowNewServer"></h3>-->

{{serverCreationStatus}}

```



It will give an output that "No Server is created". Now, we are going to bind an event with button. Add another method onCreateServer() in component.ts file which will call the event.

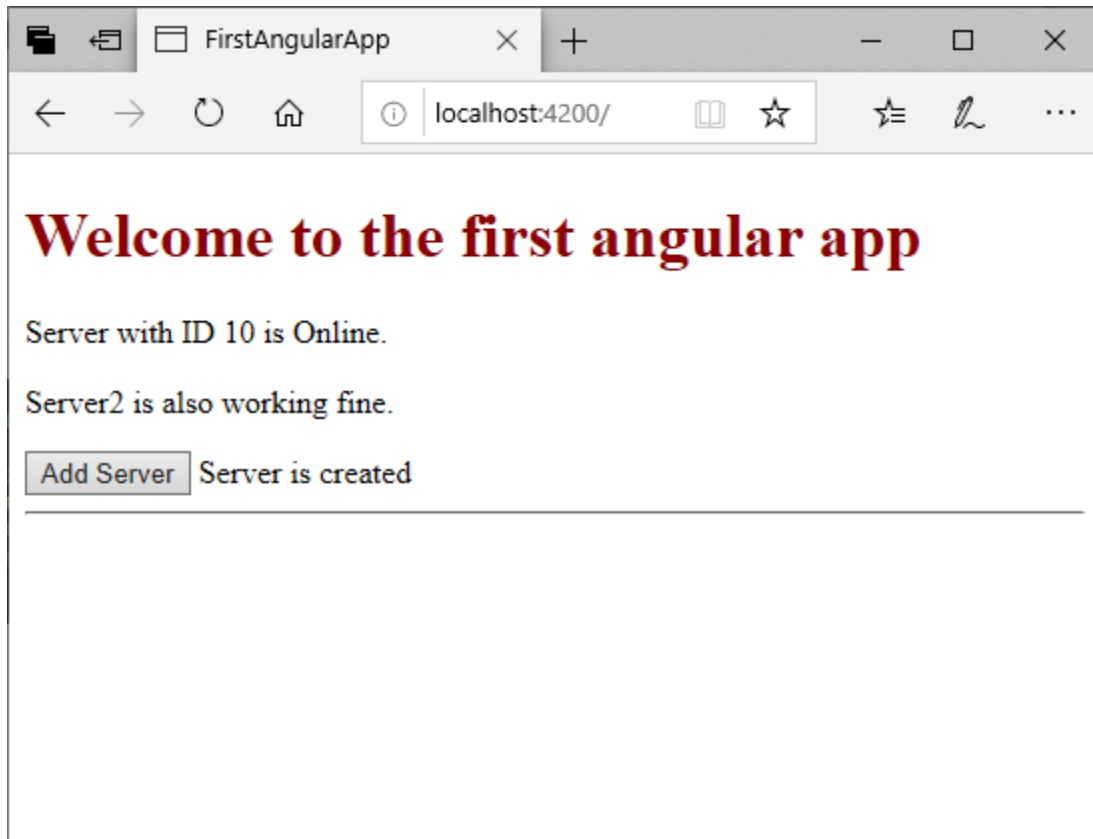
component.html file:

```
<p>
  Server2 is also working fine.
</p>
<button class="btn btn-primary"
  [disabled]="!allowNewServer"
  (click)="onCreateServer()">Add Server</button>
<!--<h3 [innerText]= "allowNewServer"></h3-->

{{serverCreationStatus}}
```

Output:

Now, after clicking on the button, you will see that it is showing server is created. This is an example of event binding.



How to use data with Event Binding?

Let's understand it by an example. Here, we will create a method named "onUpdateServerName" and add an event with it.

component.html file:

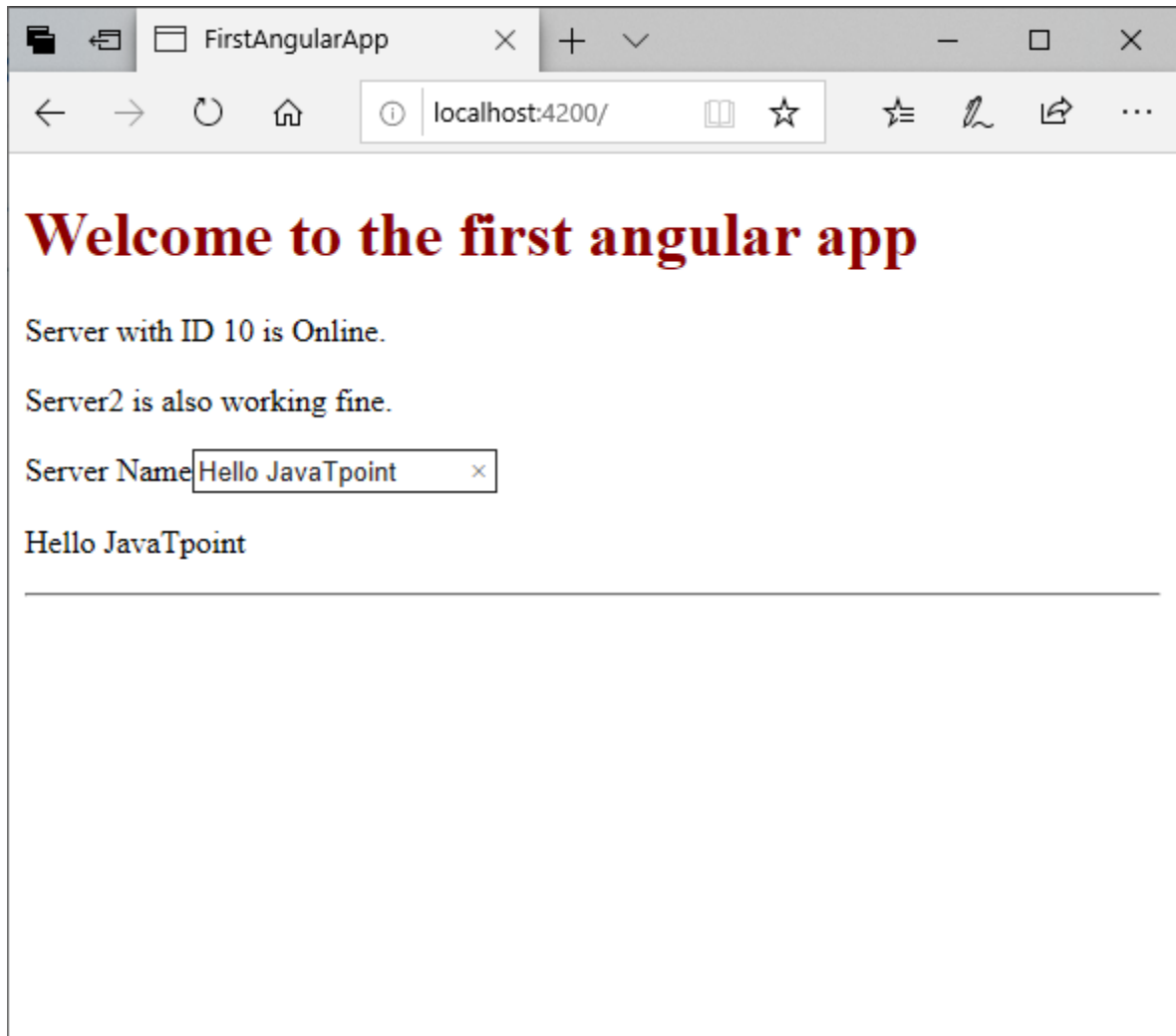
```
<label>Server Name</label>
<input type="text"
      class="form-control"
      (input)="OnUpdateServerName($event)">
<p>{{serverName}}</p>
```

component.ts file:

```
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-server2',
  templateUrl: './server2.component.html',
```

```
    styleUrls: ['./server2.component.css']
  })
  export class Server2Component implements OnInit {
    allowNewServer = false;
    serverName = '';
    constructor() {
      setTimeout(() =>{
        this.allowNewServer = true;
      }, 5000);
    }
    ngOnInit() {
    }
    onUpdateServerName(event: Event) {
      this.serverName = (<HTMLInputElement>event.target).value;
    }
  }
}
```

Output:



You can see that when you type anything in the block, it dynamically updates it below the input. This is how we can use `$event` to fetch the event's data.

2.2.3 Two-way Data Binding

We have seen that in one-way data binding any change in the template (view) were not be reflected in the component TypeScript code. To resolve this problem, Angular provides two-way data binding. The two-way binding has a feature to update data from component to view and vice-versa.

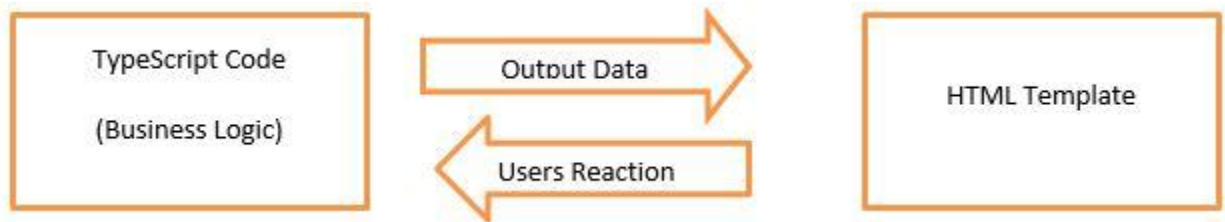
In two-way databinding, automatic synchronization of data happens between the Model and the View. Here, change is reflected in both components. Whenever you make changes in the Model, it will be reflected in the View and when you make changes in View, it will be reflected in Model.

This happens immediately and automatically, ensures that the HTML template and the TypeScript code are updated at all times.

In two-way data binding, **property binding and event binding** are combined together.

Syntax:

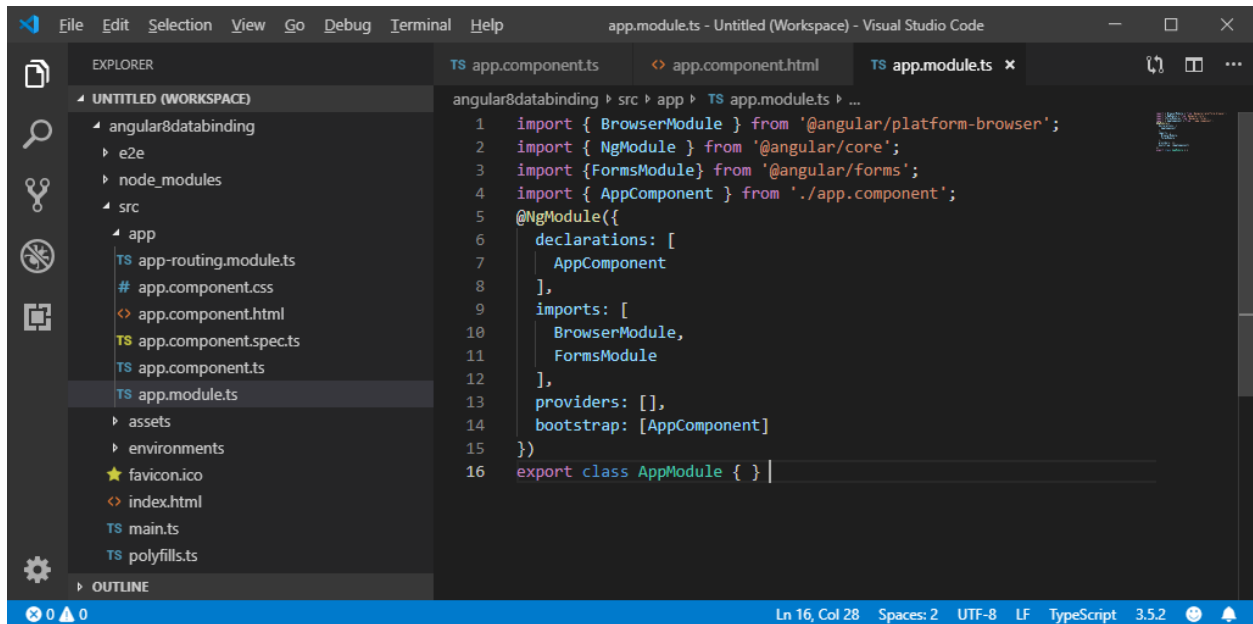
`[(ngModel)] = "[property of your component]"`



Let's take an example to understand it better.

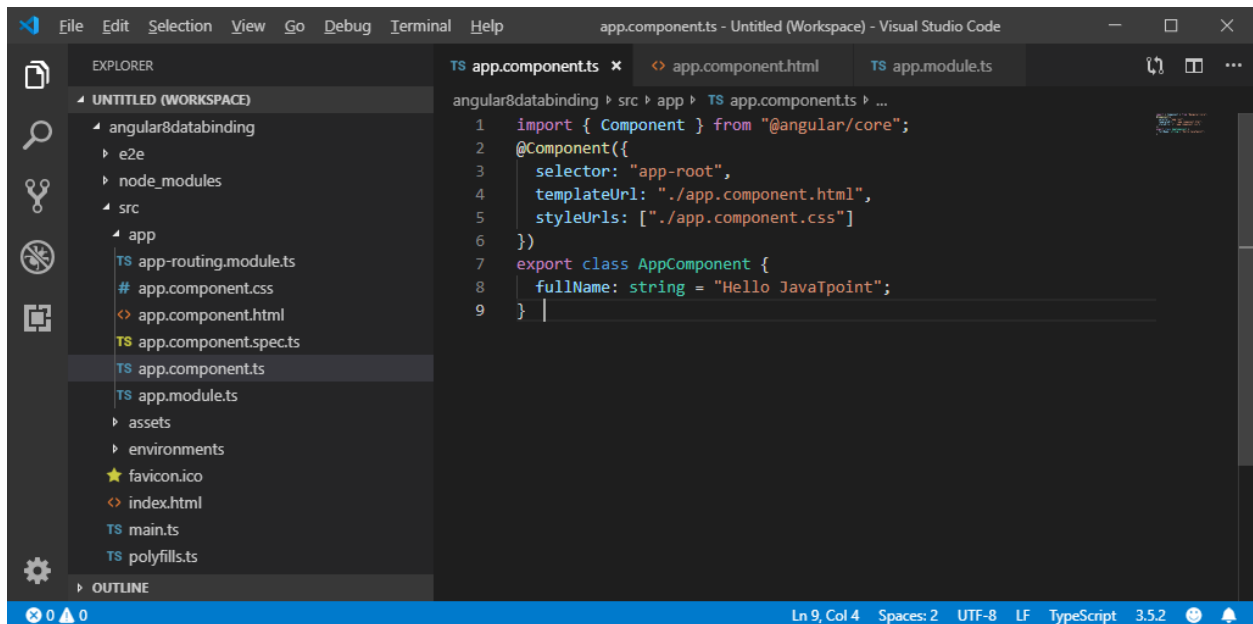
Open your project's **app.module.ts** file and use the following code:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { AppComponent } from './app.component';
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```



app.component.ts file:

```
import { Component } from "@angular/core";
@Component({
  selector: "app-root",
  templateUrl: "./app.component.html",
  styleUrls: ["./app.component.css"]
})
export class AppComponent {
  fullName: string = "Hello JavaTpoint";
}
```



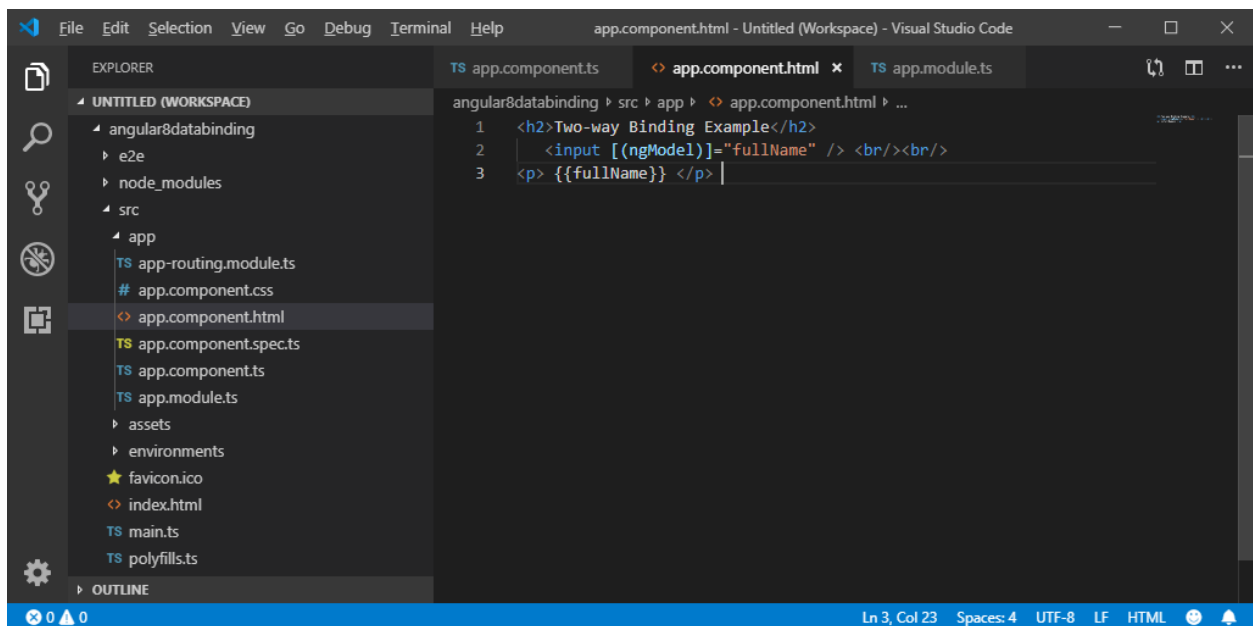
app.component.html file:

<h2>Two-way Binding Example</h2>

**<input [(ngModel)]="fullName" />

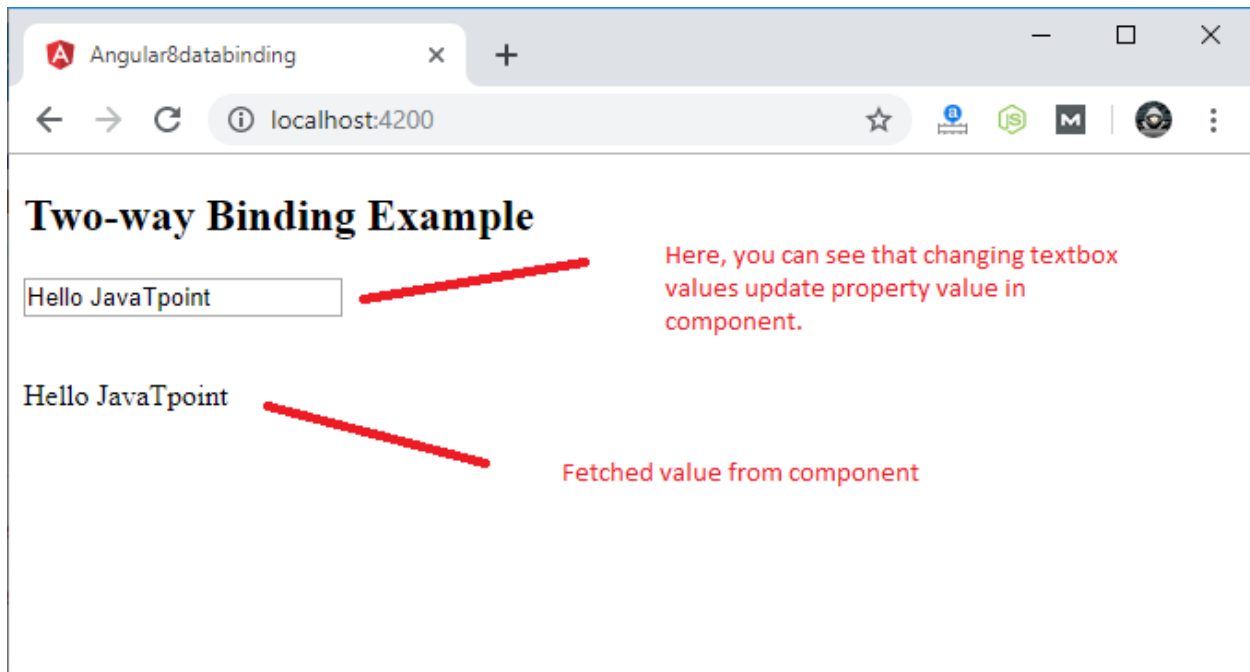
**

<p> {{fullName}} </p>



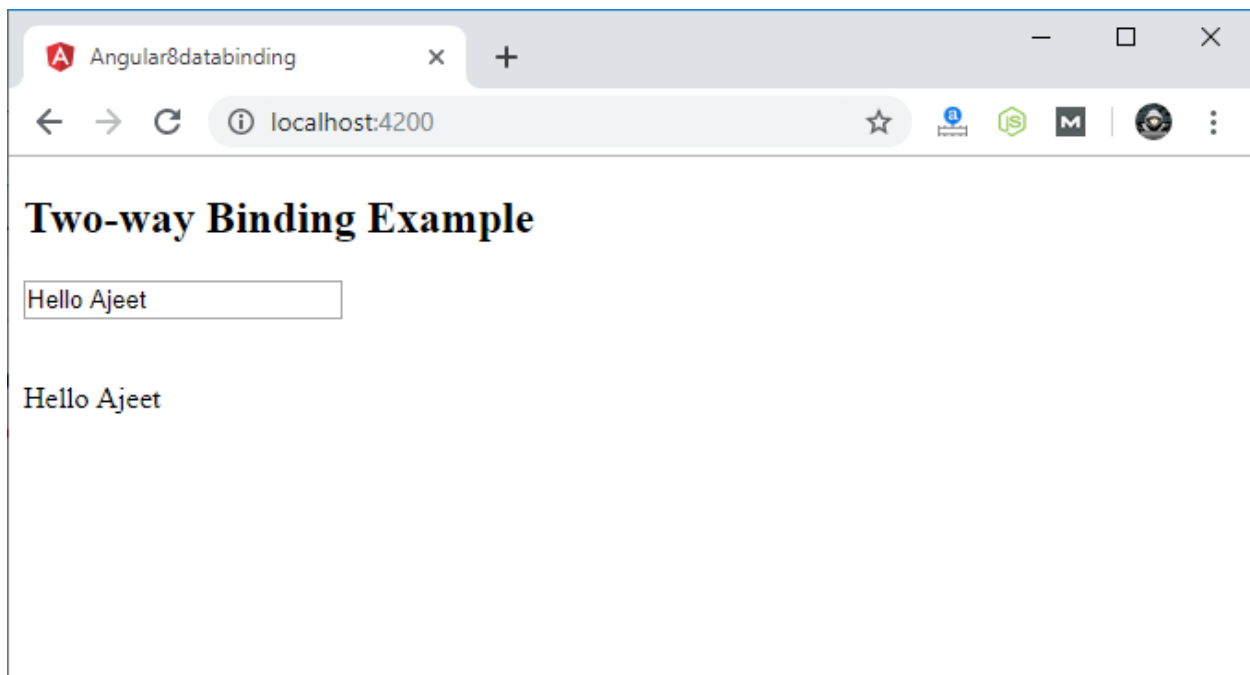
Now, start your server and open local host browser to see the result.

Output:



You can check it by changing textbox value and it will be updated in component as well.

For example:



2.2.4 Pipes

Pipes are referred as filters. It helps to transform data and manage data within interpolation, denoted by `{{ | }}`. It accepts data, arrays, integers and strings as inputs which are separated by `|` symbol. This chapter explains about pipes in detail.

1. Adding Paramter

Create a date method in your **test.component.ts** file.

```
export class TestComponent {  
  presentDate = new Date();  
}
```

Now, add the below code in your **test.component.html** file.

```
<div>  
Today's date :- {{presentDate}}  
</div>
```

Now, run the application, it will show the following output –

```
Today's date :- Mon Jun 15 2020 10:25:05 GMT+0530 (IST)
```

Here,

Date object is converted into easily readable format.

2. Date Pipe

Let's add date pipe in the above html file.

```
<div>  
Today's date :- {{presentDate | date }}  
</div>
```

You could see the below output –

```
Today's date :- Jun 15, 2020
```


3. Parameter in Date

We can add parameter in pipe using : character. We can show short, full or formatted dates using this parameter. Add the below code in **test.component.html** file.

```
<div>
short date :- {{presentDate | date:'shortDate' }} <br/>
Full date :- {{presentDate | date:'fullDate' }} <br/>
Formatted date:- {{presentDate | date:'M/dd/yyyy'}} <br/>
Hours and minutes:- {{presentDate | date:'h:mm'}}
</div>
```

You could see the below response on your screen –

```
short date :- 6/15/20
Full date :- Monday, June 15, 2020
Formatted date:- 6/15/2020
Hours and minutes:- 12:00
```

4. Chained Pipe

We can combine multiple pipes together. This will be useful when a scenario associates with more than one pipe that has to be applied for data transformation.

In the above example, if you want to show the date with uppercase letters, then we can apply both **Date** and **Uppercase** pipes together.

```
<div>
Date with uppercase :- {{presentDate | date:'fullDate' | uppercase}} <br/>
Date with lowercase :- {{presentDate | date:'medium' | lowercase}} <br/>
</div>
```

You could see the below response on your screen –

```
Date with uppercase :- MONDAY, JUNE 15, 2020 Date with lowercase :- jun 15, 2020,
12:00:00 am
```

Here,

Date, Uppercase and Lowercase are pre-defined pipes. Let's understand other types of built-in pipes in next section.

Built in Pipes

Angular supports the following built-in pipes. We will discuss one by one in brief.

5. Async Pipe

If data comes in the form of observables, then **Async pipe** subscribes to an observable and returns the transmitted values.

```
import { Observable, Observer } from 'rxjs';
export class TestComponent implements OnInit {
  timeChange = new Observable<string>((observer: Observer<string>) => {
    setInterval(() => observer.next(new
      Date().toString()), 1000);
  });
}
```

Here,

The **Async** pipe performs subscription for time changing in every one seconds and returns the result whenever gets passed to it. Main advantage is that, we don't need to call subscribe on our timeChange and don't worry about unsubscribe, if the component is removed.

Add the below code inside your test.component.html.

```
<div>
Seconds changing in Time: {{ timeChange | async }}
</div>
```

Now, run the application, you could see the seconds changing on your screen.

6. Currency Pipe

It is used to convert the given number into various countries currency format. Consider the below code in **test.component.ts** file.

```
import { Component, OnInit } from '@angular/core'; @Component({
  selector: 'app-test',
  template: `
    <div style="text-align:center">
      <h3> Currency Pipe</h3>
      <p>{{ price | currency:'EUR':true}}</p>
      <p>{{ price | currency:'INR' }}</p>
    </div>
  `
})
styleUrls: ['./test.component.scss']
})
export class TestComponent implements OnInit {
  price : number = 20000; ngOnInit() {

  }
}
```

You could see the following output on your screen –

```
Currency Pipe
€20,000.00
₹20,000.00
```

7. Slice Pipe

Slice pipe is used to return a slice of an array. It takes index as an argument. If you assign only start index, means it will print till the end of values. If you want to print specific range of values, then we can assign start and end index.

We can also use negative index to access elements. Simple example is shown below –

test.component.ts

```
import { Component, OnInit } from '@angular/core'; @Component({
  selector: 'app-test',
  template: `
    <div>
      <h3>Start index:- {{Fruits | slice:2}}</h3>
      <h4>Start and end index:- {{Fruits | slice:1:4}}</h4>
      <h5>Negative index:- {{Fruits | slice:-2}}</h5>
      <h6>Negative start and end index:- {{Fruits | slice:-4:-2}}</h6>
    </div>
  `,
  styleUrls: ['./test.component.scss']
})
export class TestComponent implements OnInit {
  Fruits = ["Apple", "Orange", "Grapes", "Mango", "Kiwi", "Pomegranate"];
  ngOnInit() {
  }
}
```

Now run your application and you could see the below output on your screen –

```
Start index:- Grapes,Mango,Kiwi,Pomegranate
Start and end index:- Orange,Grapes,Mango
Negative index:- Kiwi,Pomegranate
Negative start and end index:- Grapes,Mango
```

Here,

- `{{Fruits | slice:2}}` means it starts from second index value Grapes to till the end of value.
- `{{Fruits | slice:1:4}}` means starts from 1 to end-1 so the result is one to third index values.
- `{{Fruits | slice:-2}}` means starts from -2 to till end because no end value is specified. Hence the result is Kiwi, Pomegranate.
- `{{Fruits | slice:-4:-2}}` means starts from negative index -4 is Grapes to end-1 which is -3 so the result of `index[-4,-3]` is Grapes, Mango.

8. Decimal Pipe

It is used to format decimal values. It is also considered as CommonModule. Let's understand a simple code in `test.component.ts` file

```
import { Component, OnInit } from '@angular/core'; @Component({
  selector: 'app-test',
  template: `
    <div style="text-align:center">
      <h3>Decimal Pipe</h3>
      <p> {{decimalNum1 | number}} </p>
      <p> {{decimalNum2 | number}} </p>
    </div>
  `,
  styleUrls: ['./test.component.scss']
})
export class TestComponent implements OnInit {
  decimalNum1: number = 8.7589623;
  decimalNum2: number = 5.43;
  ngOnInit() {

  }
}
```

You could see the below output on your screen –

```
Decimal Pipe
8.759
5.43
```

9. Formatting Values

We can apply string format inside number pattern. It is based on the below format –

```
number:"{minimumIntegerDigits}.{minimumFractionDigits}
{maximumFractionDigits}"
```

-

Let's apply the above format in our code,

```
@Component({
  template: `
    <div style="text-align:center">
      <p> Apply formatting:- {{decimalNum1 | number:'3.1'}} </p>
      <p> Apply formatting:- {{decimalNum1 | number:'2.1-4'}} </p>
    </div>
  `,
})
```

Here,

{{decimalNum1 | number:'3.1'}} means three decimal place and minimum of one fraction but no constraint about maximum fraction limit. It returns the following output –

Apply formatting:- 008.759

{{decimalNum1 | number:'2.1-4'}} means two decimal places and minimum one and maximum of four fractions allowed so it returns the below output –

Apply formatting:- 08.759

10. PercentPipe

It is used to format number as percent. Formatting strings are same as DecimalPipe concept. Simple example is shown below –

```
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-test',
  template: `
    <div style="text-align:center">
      <h3>Decimal Pipe</h3>
      <p> {{decimalNum1 | percent:'2.2'}} </p>
    </div>
  `,
  styleUrls: ['./test.component.scss']
})
export class TestComponent {
  decimalNum1: number = 0.8178;
}
```

You could see the below output on your screen –

Decimal Pipe
81.78%

11. Jason Pipe

It is used to transform a JavaScript object into a JSON string. Add the below code in **test.component.ts** file as follows –

```
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-test',
  template: `
    <div style="text-align:center">
      <p ngNonBindable>{{ jsonData }}</p> (1)
      <p>{{ jsonData }}</p>
      <p ngNonBindable>{{ jsonData | json }}</p>
      <p>{{ jsonData | json }}</p>
    </div>
  `,
  styleUrls: ['./test.component.scss']
})
export class TestComponent {
  jsonData = { id: 'one', name: { username: 'user1' } }
}
```

Now, run the application, you could see the below output on your screen –

```
{{ jsonData }}
(1)
[object Object]
{{ jsonData | json }}
{"id": "one", "name": {"username": "user1" } }
```

2.2.5 SVG as Template

You can use SVG files as templates in your Angular applications. When you use an SVG as the template, you are able to use directives and bindings just like with HTML templates. Use these features to dynamically generate interactive graphics.

SVG syntax example

The following example shows the syntax for using an SVG as a template.

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

@Component({
  selector: 'app-svg',
  templateUrl: './svg.component.svg',
```

```

styleUrls: ['./svg.component.css']
})
export class SvgComponent {
  fillColor = 'rgb(255, 0, 0)';

  changeColor() {
    const r = Math.floor(Math.random() * 256);
    const g = Math.floor(Math.random() * 256);
    const b = Math.floor(Math.random() * 256);
    this.fillColor = `rgb(${r}, ${g}, ${b})`;
  }
}

```

To see property and event binding in action, add the following code to your `svg.component.svg` file:

```

<svg>
  <g>
    <rect x="0" y="0" width="100" height="100" [attr.fill]="fillColor"
(click)="changeColor()" />
    <text x="120" y="50">click the rectangle to change the fill color</text>
  </g>
</svg>

```

The example given uses a `click()` event binding and the property binding syntax (`[attr.fill]="fillColor"`).

2.3 Directives

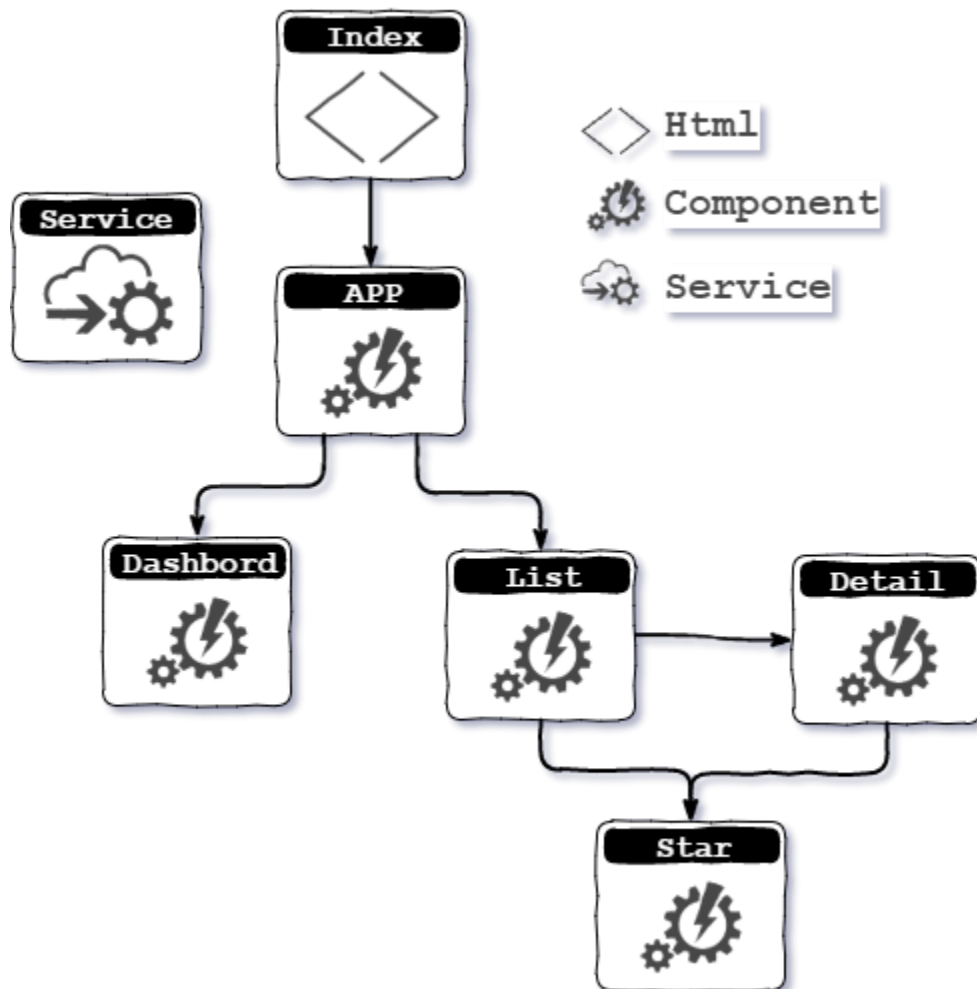
Directives are classes that add additional Behavior to elements in your angular application. Use Angular's Built-in directives to manage forms, lists, styles and what users see.

Here are three kinds of directives in Angular:

1. **Components** is a directives with a template, class and metadata.
2. **Structural directives** is a directive which change the DOM layout by adding and removing DOM elements. For Example **ngIf**, **ngFor**, **ngSwitch** are called Structural Directive
3. **Attribute directives** change the appearance or behavior of an element. **ngClass** and **ngStyle** are the example of Attribute directives.

- **Component Directive in Angular**

In Angular, "everything is a component". In fact Angular application is a hierarchy of component tree.



The above diagram is representation of a small application where **App** is the root component and **dashbord** and **list** component are it's child. **List** component inherit **Detail** component and **Star** component is child component for **List** and **Detail** Component.

- **ngIf Structural Directive**

ngIf directive is used when we want to show or hide html element based on a given condition.

Some examples are:


```

<div *ngIf="1 == 1"></div>
<!-- This will always visible since 1 == 1 is always true -->

<div *ngIf="1 == 2"></div>
<!-- This will never visible because 1 == 2 is always false -->

<div *ngIf="a > b"></div>
<!-- This will be visible if 'a' is greater then 'b' -->

```

- **ngFor Structural Directives**

ngFor directive is used to **repeat a DOM or HTML element** for a given item in array or collection.

The syntax is

```
*ngFor="let item of items"
```

Let's write some code to check how powerful is ngFor built in directive of angular.

Write Some hard value in component.ts

goto src\app\app.component.ts and enter the data below.

```

export class AppComponent {
  stringArray: string[];
  objectArray: Object[];
  complexObjectArray: Object;

  constructor() {
    this.stringArray = ['India', 'Afgan', 'New York'];
    this.objectArray = [
      { name: 'Nisar', mark: 35, city: 'India' },
      { name: 'John', mark: 12, city: 'Afgan' },
      { name: 'Amy', mark: 22, city: 'New York' }
    ];
    this.complexObjectArray = [
      {
        city: 'India',
        people: [
          { name: 'Amy', mark: 12 },
          { name: 'Lisa', mark: 22 }
        ]
      },
      {
        city: 'Afgan',
        people: [
          { name: 'John', mark: 35 },
          { name: 'Mary', mark: 36 }
        ]
      }
    ];
  }
}

```

```
};  
}
```

Repeat simple Array String

Go to src\app\app.component.html file and update the html

```
<h1>  
  Simple String  
</h1>  
<li *ngFor="let str of stringArray">  
  {{str}}  
</li>
```

This will produce following output

Repeat array of Object

In same app.component.html file update the html

```
<h1>Array of Object</h1>  
<table border="1px">  
  <thead>  
    <tr>  
      <th>Name</th>  
      <th>Mark</th>  
      <th>City</th>  
    </tr>  
  </thead>  
  <tr *ngFor="let arr of objectArray">  
    <td>{{ arr.name }}</td>  
    <td>{{ arr.mark }}</td>  
    <td>{{ arr.city }}</td>  
  </tr>  
</table>
```

This produce output

Array Of Nested Object

```
<h1>Array of Nested Object</h1>  
<div *ngFor="let item of complexObjectArray">  
  <h2>{{ item.city }}</h2>  
  <table border="1px" cellspacing="0">  
    <thead>  
      <tr>  
        <th>Name</th>  
        <th>Age</th>  
      </tr>  
    </thead>  
    <tr *ngFor="let p of item.people">  
      <td>{{ p.name }}</td>  
      <td>{{ p.mark }}</td>  
    </tr>
```

```
</table>
</div>
```

This ngFor produce the output

Track Index of array in ngFor

```
<li *ngFor="let str of stringArray; let num= index">
  {{num+1}} {{str}}
</li>
```

2.4 Dependency Injection

Dependency injection is the ability to add the functionality of components at runtime. Let's take a look at an example and the steps used to implement dependency injection.

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.

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";
  }
}
```

The 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';

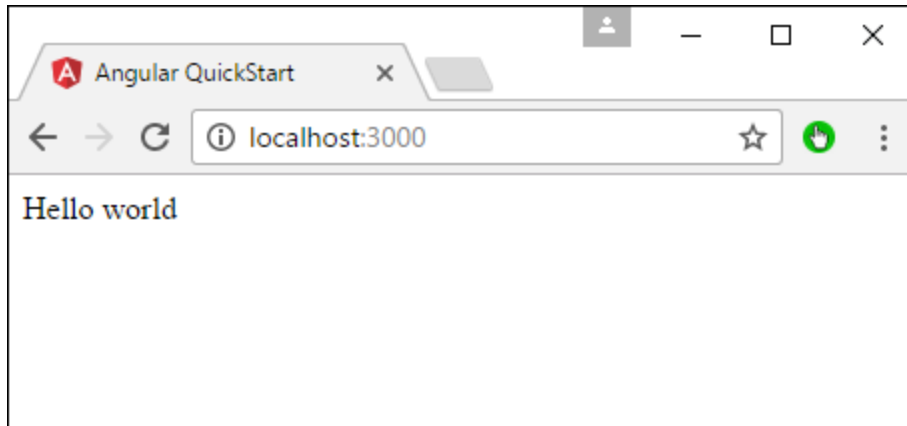
@Component({
  selector: 'my-app',
  template: '<div>{{value}}</div>',
  providers: [appService]
})

export class AppComponent {
  value: string = '';
  constructor(private _appService: appService) {}
  ngOnInit(): void {
    this.value = this._appService.getApp();
  }
}
```

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

- First, we are importing our appService module in the appComponent module.
- Then, we are registering 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 assigned the output to the value property of the AppComponent class.

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



3. Naming Convection

Naming is one of the things that we do the most as software developers, for this reason, it is important that we do it well. Good naming practices are important in software because they facilitate:

- code readability
- maintainability of code and
- on-boarding of new developers to a team.

Angular is a very opinionated framework by nature and for a good reason. It helps ensure that our coding pattern is predictable hence easy to maintain.

Golden Rules of Naming

The golden rules of naming should form the foundation of your naming pattern. They must, by all means, be applied or followed.

1. Consistency

Consistency is key when naming artifacts in your Angular code. The principle with consistency is to follow a good pattern and stick to it. Being consistent helps in finding content at a glance. In addition, it provides pointers to new developers on how to name things. Here is a quote from the official Angular documentation regarding naming consistency.

2. Giving meaningful names

All artifacts in your project; folders, files, classes, etc should be named to convey meaning. The names should provide a hint as to what an artifact does.

- Use names that convey intent
- Use names that are searchable
- Use nouns for classes, folders, and filenames
- Use verbs or verb phrases for methods or functions
- Avoid using abbreviation and notations because they can be confusing

Case Styling in Angular

Apart from the general naming guidelines discussed above, Angular mainly uses three case styles for naming artifacts. camelCase, PascalCase, and kebab-case. Knowing when and where to use each of these case styles is important. In the section below, I will go through each of the case styles and where to use them.

1. Kebab case (kebab-case)

kebab-case is a naming style where **all letters in the name are lowercase and it uses a dash to separate words** in a name. In addition, in Angular, a dot is used to separate the name, type and extension for file names.

Including the type in the file names make it easy to find a specific file type using a text editor or an IDE. In addition, they provide pattern matching for automated tasks. The kebab-case is used in naming folders, component selectors, files, and the Angular application itself. Typical files in an Angular project include component files, service files, template files, module files, etc.

Example:

A component that keeps track of a list of cars can be named ***car-list.component.ts***. A corresponding service would be ***car-list.service.ts***.

A pipe file for dates would be ***date.pipe.ts***.

A selector that shows the cars in a table could be named as highlighted in bold below.

```
@Component({ selector: 'car-list-table', templateUrl: './car-list-button.component.html' })
```

Test files are a bit special. In addition to the type and extension, they should have the suffix *spec* for unit test files and *e2e-spec* for end-to-end test files.

A date-pipe unit test file could be named *date.pipe.spec.ts*.

A car list component end-to-end test file could be named **car-list.component.e2e-spec.ts**

2. Pascal case (PascalCase)

The PascalCase is a style in which **all first letters of the words in a name are capitalized or in uppercase**. The Pascal case is mainly used for naming classes in an Angular Project.

Example:

The class name of the car-list component can be named as follows.

```
Export class CarListComponent {}
```

A pipe class that transforms dates could be named as below

```
Export class DatePipe {}
```

Class names must follow the PascalCase, regardless of whether they belong to a module, service, component, etc.

3. Camel case (camelCase)

The camelCase naming style is a bit similar to the PascalCase except that **the first letter in a name should always be lowercase. All other subsequent words in a name will have uppercase for the first letter**.

Note: The camelCase and kebab-case for single word names will be similar.

The camelCase is used for naming methods or function, properties, fields, directive selectors, and pipe selectors as highlighted below.

Examples:

Directive selectors use camelCase

```
@Directive({ selector: '[carValidate]' }) export class
CarValidateDirective {}
```

Pipe name selectors also use camel case

```
@Pipe({ name: 'initCaps' }) export class InitCapsPipe
implements PipeTransform { }
```

In Angular, method or function names also use Camel case. For example, a method to get the list of cars can be named as follows.

```
public getCars() ;
```

4. Angular Navigation and routing

The Angular Router is one of the most important libraries in an Angular application. Without it, apps would be single view/single context apps or would not be able to maintain their navigation state on browser reloads. With Angular Router, we can create rich apps that are linkable and have rich animations (when paired with Ionic of course). Let's look at the basics of the Angular Router and how we can configure it for Ionic apps.

A Simple Route

For most apps, having some sort of route is often required. The most basic configuration looks a bit like this:

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

@NgModule({
  imports: [
    ...
    RouterModule.forRoot([
      { path: '', component: LoginComponent },
      { path: 'detail', component: DetailComponent },
    ])
  ],
})
```

The simplest breakdown for what we have here is a path/component lookup. When our app loads, the router kicks things off by reading the URL the user is trying to load. In our sample, our route looks for "", which is essentially our index route. So for this, we load the LoginComponent. Fairly straight forward. This pattern of matching paths with a component continues for every entry we have in the router config. But what if we wanted to load a different path on our initial load?

Handling Redirects

For this we can use router redirects. Redirects work the same way that a typical route object does, but just includes a few different keys.

```
[
  { path: '', redirectTo: 'login', pathMatch: 'full' },
  { path: 'login', component: LoginComponent },
  { path: 'detail', component: DetailComponent },
];
```

In our redirect, we look for the index path of our app. Then if we load that, we redirect to the login route. The last key of pathMatch is required to tell the router how it should look up the path.

Since we use full, we're telling the router that we should compare the full path, even if ends up being something like /route1/route2/route3. Meaning that if we have:

```
{ path: '/route1/route2/route3', redirectTo: 'login', pathMatch: 'full' },
{ path: 'login', component: LoginComponent },
```

And load /route1/route2/route3 we'll redirect. But if we loaded /route1/route2/route4, we won't redirect, as the paths don't match fully.

Alternatively, if we used:

```
{ path: '/route1/route2', redirectTo: 'login', pathMatch: 'prefix' },
{ path: 'login', component: LoginComponent },
```

Then load both /route1/route2/route3 and /route1/route2/route4, we'll be redirected for both routes. This is because pathMatch: 'prefix' will match only part of the path.

Navigating to different routes

Talking about routes is good and all, but how does one actually navigate to said routes? For this, we can use the routerLink directive. Let's go back and take our simple router setup from earlier:

```
RouterModule.forRoot([
  { path: '', component: LoginComponent },
  { path: 'detail', component: DetailComponent },
]);
```

Now from the LoginComponent, we can use the following HTML to navigate to the detail route.

```

<ion-header>
  <ion-toolbar>
    <ion-title>Login</ion-title>
  </ion-toolbar>
</ion-header>

<ion-content class="ion-padding">
  <ion-button [routerLink]="['/detail']">Go to detail</ion-button>
</ion-content>

```

The important part here is the ion-button and routerLink directive. RouterLink works on a similar idea as typical hrefs, but instead of building out the URL as a string, it can be built as an array, which can provide more complicated paths.

We also can programmatically navigate in our app by using the router API.

```

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

@Component({
  ...
})
export class LoginComponent {

  constructor(private router: Router){}

  navigate(){
    this.router.navigate(['/detail'])
  }
}

```

Both options provide the same navigation mechanism, just fitting different use cases.

4.1 Lazy Loading Routes

Now the current way our routes are setup makes it so they are included in the same chunk as the root app.module, which is not ideal. Instead, the router has a setup that allows the components to be isolated to their own chunks.

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

@NgModule({
  imports: [
    ...
    RouterModule.forRoot([
      { path: '', redirectTo: 'login', pathMatch: 'full' },
      { path: 'login', loadChildren: () => import('./login/login.module').then(m =>
m.LoginModule) },
      { path: 'detail', loadChildren: () => import('./detail/detail.module').then(m =>
m.DetailModule) }
    ])
  ],
})
```

While similar, the loadChildren property is a way to reference a module by using native import instead of a component directly. In order to do this though, we need to create a module for each of the components.

```
...
import { RouterModule } from '@angular/router';
import { LoginComponent } from './login.component';

@NgModule({
  imports: [
    ...
    RouterModule.forChild([
      { path: '', component: LoginComponent },
    ])
  ],
})
```

Here, we have a typical Angular Module setup, along with a RouterModule import, but we're now using forChild and declaring the component in that setup. With this setup, when we run our build, we will produce separate chunks for both the app component, the login component, and the detail component.

5. Internationalization

Internationalization (i18n) is a must require feature for any modern web application. Internationalization enables the application to target any language in the world. Localization is a part of the Internationalization and it enables the application to render in a targeted local language. Angular provides complete support for internationalization and localization feature.

Let us learn how to create a simple hello world application in different language.

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

<h1 i18n="greeting|Greeting a person@@greeting">Hello</h1>
<div>
  <span i18n="time|Specify the current time@@currentTime">
    The Current time is {{ currentDate | date : 'medium' }}
  </span>
</div>
```

Here,

- **hello** is simple translation format since it contains complete text to be translated.
- **Time** is little bit complex as it contains dynamic content as well. The format of the text should follow ICU message format for translation.

We can extract the data to be translated using below command –

```
ng extract-i18n --output-path src/locale
```

Command generates **messages.xlf** file with below content –

```
<?xml version="1.0" encoding="UTF-8" ?>
<xliff version="1.2" xmlns="urn:oasis:names:tc:xliff:document:1.2">
  <file source-language="en" datatype="plaintext" original="ng2.template">
    <body>
      <trans-unit id="greeting" datatype="html">
        <source>Hello</source>
        <context-group purpose="location">
          <context context-type="sourcefile">src/app/app.component.html</context>
          <context context-type="linenumber">3</context>
        </context-group>
        <note priority="1" from="description">Greeting a person</note>
        <note priority="1" from="meaning">greeting</note>
      </trans-unit>
      <trans-unit id="currentTime" datatype="html">
        <source>
          The Current time is <x id="INTERPOLATION" equiv-text="{{ currentDate | date :
'medium' }}" />
        </source>
        <context-group purpose="location">
```

```

    <context context-type="sourcefile">src/app/app.component.html</context>
    <context context-type="linenumber">5</context>
  </context-group>
  <note priority="1" from="description">Specify the current time</note>
  <note priority="1" from="meaning">time</note>
</trans-unit>
</body>
</file>
</xliff>

```

Copy the file and rename it to **messages.hi.xlf**

Open the file with Unicode text editor. Locate **source** tag and duplicate it with **target** tag and then change the content to **hi** locale. Use google translator to find the matching text. The changed content is as follows –

```

<source>Hello</source>
|<target>हैलो</target>
<source>
  The Current time is <x id="INTERPOLATION" equiv-text="{{ currentDate | date : &apos;medium&apos; }}" />
</source>
<target>
  वर्तमान तिथि और समय <x id="INTERPOLATION" equiv-text="{{ currentDate | date : &apos;medium&apos; }}" /> है
</target>

```

Open **angular.json** and place below configuration under **build -> configuration**

```

"hi": {
  "aot": true,
  "outputPath": "dist/hi/",
  "i18nFile": "src/locale/messages.hi.xlf",
  "i18nFormat": "xlf",
  "i18nLocale": "hi",
  "i18nMissingTranslation": "error",
  "baseHref": "/hi/"
},
"en": {
  "aot": true,
  "outputPath": "dist/en/",
  "i18nFile": "src/locale/messages.xlf",
  "i18nFormat": "xlf",
  "i18nLocale": "en",
  "i18nMissingTranslation": "error",
  "baseHref": "/en/"
}

```

Here,

We have used separate setting for **hi** and **en** locale.

Set below content under **serve -> configuration**.

```
"hi": {  
  "browserTarget": "i18n-sample:build:hi"  
},  
"en": {  
  "browserTarget": "i18n-sample:build:en"  
}
```

We have added the necessary configuration. Stop the application and run below command –

```
npm run start -- --configuration=hi
```

Here,

We have specified that the hi configuration has to be used.

Navigate to <http://localhost:4200/hi> and you will see the Hindi localised content.



Finally, we have created a localized application in Angular.

6. Angular Rxjs

What is RxJS?

The **RxJS** (Reactive Extensions Library for JavaScript) is a javascript library, that allows us to work with asynchronous data streams

The Angular uses the RxJS library heavily in its framework to implement Reactive Programming. Some of the examples where reactive programming used are

- Reacting to an HTTP request in Angular
- Value changes / Status Changes in Angular Forms
- The Router and Forms modules use observables to listen for and respond to user-input events.
- You can define custom events that send observable output data from a child to a parent component.
- The HTTP module uses observables to handle AJAX requests and responses.

The RxJs has two main players

1. Observable
2. Observers (Subscribers)

What is an Observable in Angular?

Observable is a function that converts the ordinary stream of data into an observable stream of data.

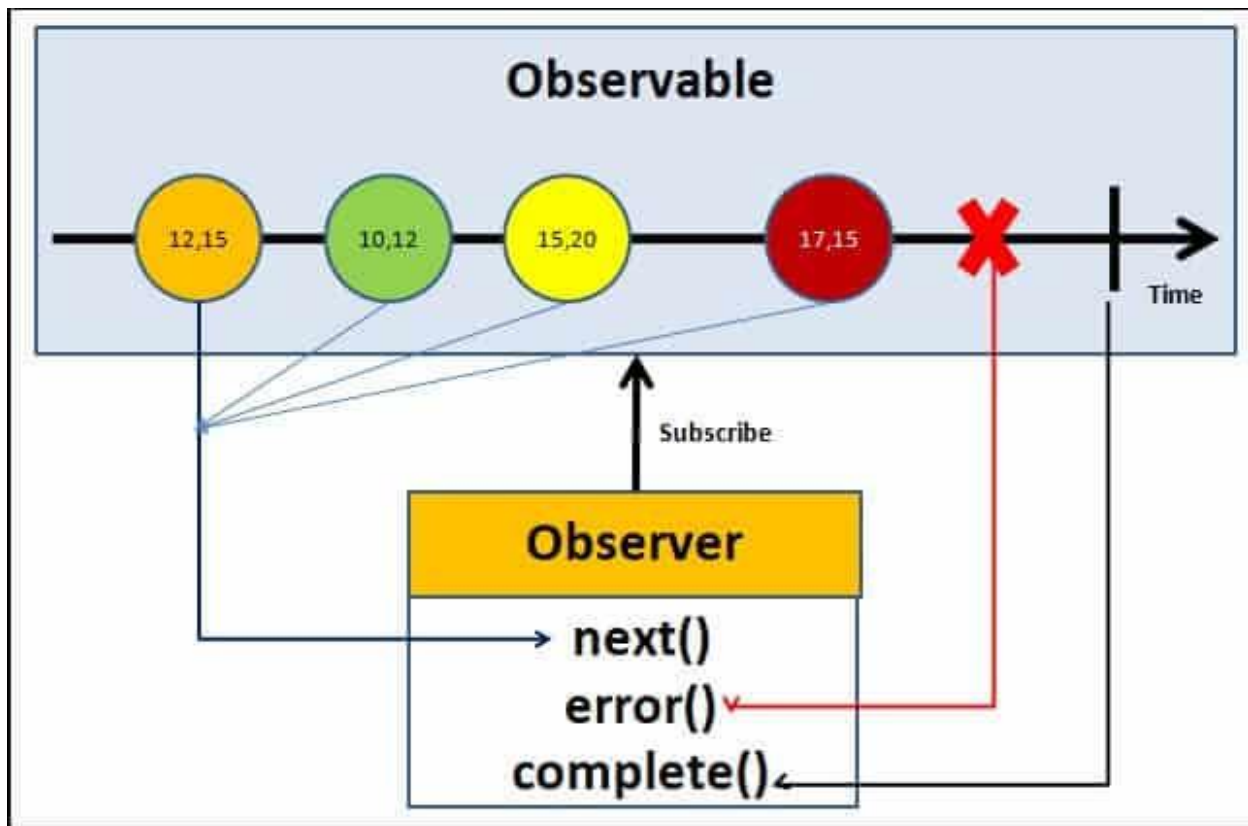
Observable stream or simple observable emits the value from the stream asynchronously. It emits the complete signals when the stream completes or an error signal if the stream errors out.

Observable are declarative. You define an observable function just like any other variable. The observable starts to emit values only when some subscribes to it.

Who are Observers (subscribers)?

The Observable on its own is useless unless someone consumes the value emitted by observable. We call them observers or subscribers. The observers communicate with the observable using callbacks.

The observable must subscribe with the observable to receive the value from the observable. While subscribing it optionally passes the three callbacks. *next()*, *error()* & *complete()*



Angular Observable Tutorial how observable and observers communicates with callbacks

The observable starts emitting the value as soon as the observer or consumer subscribes to it.

The observable invokes the `next()` callback whenever the value arrives in the stream. It passes the value as the argument to the next callback. If the error occurs, then the `error()` callback is invoked. It invokes the `complete()` callback when the stream completes.

- Observers/subscribers subscribe to Observables
- Observer registers three callbacks with the observable at the time of subscribing. i.e `next()`, `error()` & `complete()`
- All three callbacks are optional
- The observer receives the data from the observable via the `next()` callback
- They also receive the errors and completion events from the Observable via the `error()` & `complete()` callbacks

RxJs library is installed automatically when you create the angular project. Hence there is no need to install it.

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

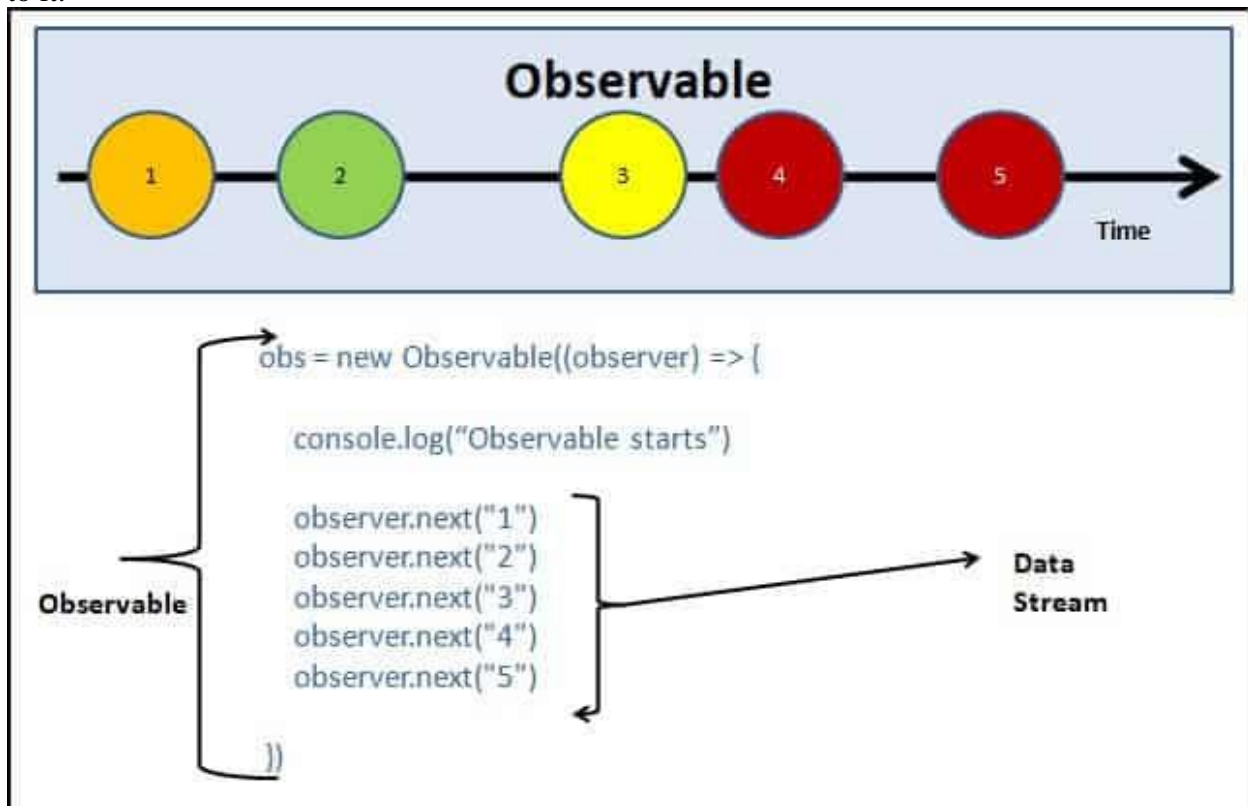

Observable Creation

There are few ways in which you can create observable in angular. Simplest is to use the Observable constructor. The observable constructor takes observer (or subscriber) as its argument. The subscriber will run when this observable's subscribe() method executes.

The following example creates an observable of a stream of numbers 1, 2, 3, 4, 5

```
1
2 obs = new Observable((observer) => {
3   console.log("Observable starts")
4   observer.next("1")
5   observer.next("2")
6   observer.next("3")
7   observer.next("4")
8   observer.next("5")
9 })
10
```

The variable `obs` is now of the type of observable. The above example declares the `obs` as the observable but does not instantiate it. To make the observable to emit values, we need to subscribe to it.



Creating observable in Angular Observable Tutorial app

In the above example, we used the Observable Constructor to create the Observable. There are many operators available with the RxJS library, which makes the task of creating the observable easy. These operators help us to create observable from an array, string, promise, any iterable, etc. Here are list some of the commonly used operators

- `create`
- `defer`
- `empty`
- `from`
- `fromEvent`
- `interval`
- `of`
- `range`
- `throwError`
- `timer`

Subscribing to observable

We subscribe to the observable, by invoking the `subscribe` method on it. We can optionally, include the three callbacks `next()`, `error()` & `complete()` as shown below

```
1
2 ngOnInit() {
3
4   this.obs.subscribe(
5     val => { console.log(val) }, //next callback
6     error => { console.log("error") }, //error callback
7     () => { console.log("Completed") } //complete callback
8   )
9 }
10
```

The complete `app.component.ts` code is as shown below.

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
  title = 'Angular Observable using RxJs - Getting Started';

  obs = new Observable((observer) => {
    console.log("Observable starts")
  })
}
```

```

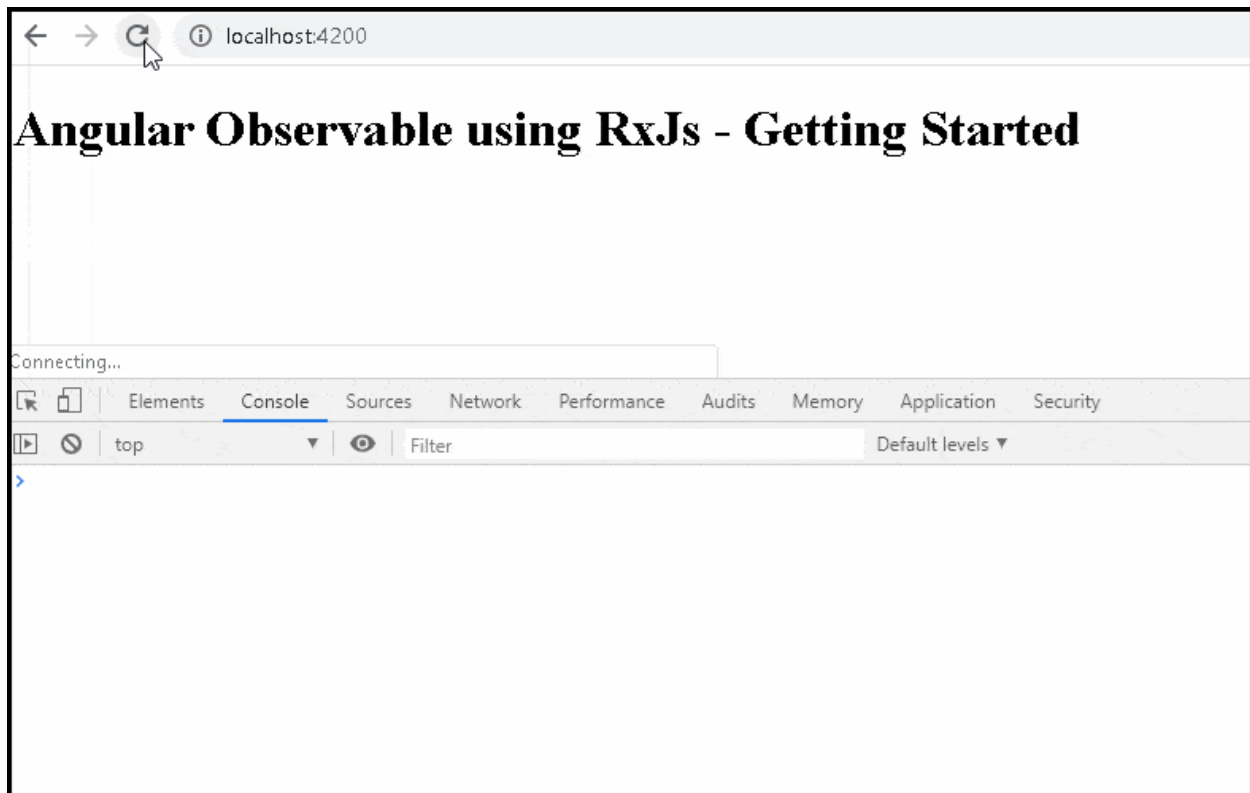
    observer.next("1")
    observer.next("2")
    observer.next("3")
    observer.next("4")
    observer.next("5")
  })

  data=[];

  ngOnInit() {

    this.obs.subscribe(
      val=> { console.log(val) },
      error => { console.log("error")},
      () => {console.log("Completed")}
    )
  }
}

```



Angular Observable tutorial example app

Error Event

As mentioned earlier, the observable can also emit an error. This is done by invoking the `error()` callback and passing the error object. The observables stop after emitting the error signal. Hence values 4 & 5 are never emitted.

```
obs = new Observable((observer) => {  
  console.log("Observable starts")  
  
  setTimeout(() => { observer.next("1") }, 1000);  
  setTimeout(() => { observer.next("2") }, 2000);  
  setTimeout(() => { observer.next("3") }, 3000);  
  setTimeout(() => { observer.error("error emitted") }, 3500); //sending error event. observable stops  
here  
  setTimeout(() => { observer.next("4") }, 4000); //this code is never called  
  setTimeout(() => { observer.next("5") }, 5000);  
  
})
```

You can send the error object as the argument to the error method.

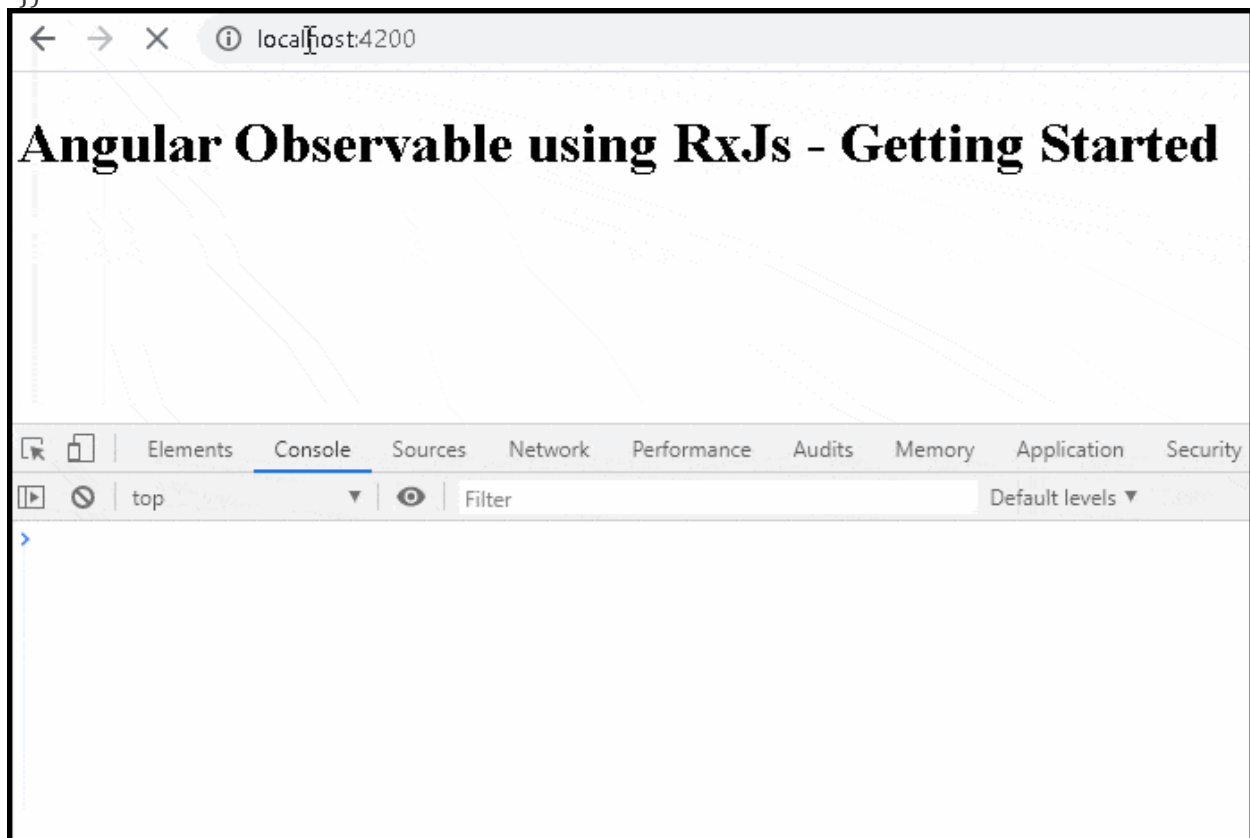


Observable with the error event

Complete Event

Similarly, the complete event. The observables stop after emitting the complete signal. Hence values 4 & 5 are never emitted.

```
obs = new Observable((observer) => {  
  console.log("Observable starts")  
  
  setTimeout(() => { observer.next("1") }, 1000);  
  setTimeout(() => { observer.next("2") }, 2000);  
  setTimeout(() => { observer.next("3") }, 3000);  
  setTimeout(() => { observer.complete() }, 3500); //sending complete event. observable stops here  
  setTimeout(() => { observer.next("4") }, 4000); //this code is never called  
  setTimeout(() => { observer.next("5") }, 5000);  
})
```



Observable with complete event

Observable Operators

The Operators are functions that operate on an Observable and return a new Observable.

The power of observable comes from the [operators](#). You can use them to manipulate the incoming observable, filter it, merge it with another observable, alter the values or subscribe to another observable.

You can also chain each operator one after the other using the [pipe](#). Each operator in the chain gets the observable from the previous operator. It modifies it and creates a new observable, which becomes the input for the next observable.

The following example shows the [filer](#) & [map](#) operators chained inside a [pipe](#). The filter operator removes all data which is less than or equal to 2 and the map operator multiplies the value by 2. The input stream is [1,2,3,4,5] , while the output is [6, 8, 10].

```
obs.pipe(  
  obs = new Observable((observer) => {  
    observer.next(1)  
    observer.next(2)  
    observer.next(3)  
    observer.next(4)  
    observer.next(5)  
    observer.complete()  
  }).pipe(  
    filter(data => data > 2), //filter Operator  
    map((val) => {return val as number * 2}), //map operator  
  )  
)
```

The following table lists some of the commonly used operators

AREA	OPERATORS
Combination	combineLatest, concat, merge, startWith , withLatestFrom, zip
Filtering	debounceTime , distinctUntilChanged, filter , take , takeUntil , takeWhile , takeLast , first , last , single , skip , skipUntil , skipWhile , skipLast ,
Transformation	bufferTime, concatMap , map , mergeMap , scan , switchMap , ExhaustMap , reduce
Utility	tap , delay , delaywhen

AREA	OPERATORS
Error Handling	throwerror , catcherror , retry , retrywhen
Multicasting	share

Unsubscribing from an Observable

We need to unsubscribe to close the observable when we no longer require it. If not it may lead to memory leak & Performance degradation.

To Unsubscribe from an observable, we need to call the `Unsubscribe()` method on the subscription. It will clean up all listeners and frees up the memory.

To do that, first, create a variable to store the subscription

```
obs: Subscription;
```

Assign the subscription to the `obs` variable

```
this.obs = this.src.subscribe(value => {
  console.log("Received " + this.id);
});
```

Call the `unsubscribe()` method in the `ngOnDestroy` method.

```
ngOnDestroy() {
  this.obs.unsubscribe();
}
```

When we destroy the component, the observable is unsubscribed and cleaned up.

But, you do not have to unsubscribe from every subscription. For Example, the observables, which emits the complete signal, close the observable.

7. State Management in Angular

State management is a key component when building applications. There are various approaches by which we can manage the state in an Angular application, each with its pros and cons. This blog post will focus on using NgRx as our state management solution. We will look at how you can use NgRx to manage your application's state by building a Recipe Admin Dashboard application. We will also learn how to secure the application using Auth0 and how it works with NgRx.

Angular Redux

Redux is a reactive state management library which is developed by Facebook and used in the React library. It is based on the **Flux pattern**. The difference between Flux and Redux is how they handle tasks; In the case of Flux, we have multiple stores and one dispatcher, whereas, in Redux, there is only one Store, which means there is no need for a dispatcher.

We can use the NgRx library to use Redux in the Angular framework. It is a reactive state management library. With NGRX, we can get all the events (data) from the Angular app and keep them all in one place.

When we want to use the stored data, we have to receive (dispatch) it from the Store using the RxJS library. RxJS is a library based on the Observable pattern used in Angular to process asynchronous operations.

We use a service to share data between components (make sure to unsubscribe the observable each time; otherwise, you risk running the observable in the background unnecessarily, which consumes resources), or we can use input/output data flow (**make sure components have parent/child relationship**).

We can use ViewChild for nested components. But in the case of a large project, these solutions increase the complexity of the project. If we have multiple components, we risk losing control over the data flow within one component.

It uses Redux in Angular: Store and unidirectional data flow to reduce the complexity of the application. The flow is clear and easy to understand for new team members.

Implementation

Once the project is set up, we start implementing our **Todo app**. The first step is to create a new module for our app (we need to do this because we will be treating the app module as the main module of the entire application).

To do this, we run the `ng g module to-do` in the terminal, and then; We import this module into the `app.module.ts` file, as can be seen below:


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

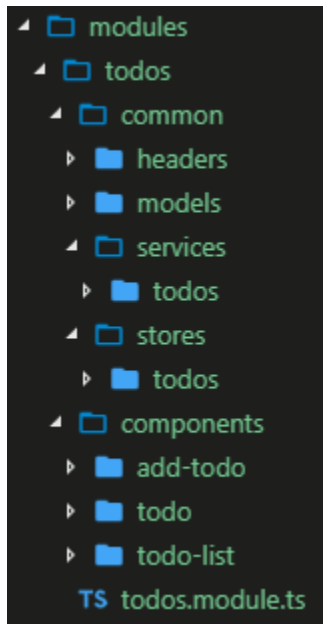
import { AppRoutingModuleModule } from './app-routing.module';
import { TodosModule } from './modules/todos/todos.module';

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

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

Figure 3. app.module.ts file.

Below is the folder structure of the **TodosModule** that we will use for the components and the common files (services, headers, models, etc.):



Project file structure.

After we done with application structure, we can start to code. The first step is to import the modules that we need to use in our **todos.module.ts**:

```
imports: [  
  CommonModule,  
  HttpClientModule,  
  NgbModule,  
  FormsModule,  
  ReactiveFormsModule,  
  NgxPaginationModule,  
  StoreModule.forRoot({})  
],  
providers: [TodosService]
```

todos.module.ts file, imports array.

By adding the StoreModule, our module has a Store now.

In NgRx, the **Store** is like an internal database that reflects the state of our application. All the StoreModule's data will be contained in the Store. Now we can write our to-do action.

An **action** is a class that implements the NgRx action interface. Action classes have two properties:

- **type:** It is a read-only string that describes what the action stands for. For example, **GET_TODO**.
- **Payload:** This property type depends on what type of data action needs to send to the reducer. In the previous example, the payload will be a string containing a to-do. Not all actions need to have a payload.

For example, to get the todo list, we want the following actions:

```
import { Action } from '@ngrx/store';
import { Todo } from '../models/todo';

export enum TodosActionType {
  GET_TODOS = 'GET_TODOS',
  GET_TODOS_SUCCESS = 'GET_TODOS_SUCCESS',
  GET_TODOS_FAILED = 'GET_TODOS_FAILED'
}

export class GetTodos implements Action {
  readonly type = TodosActionType.GET_TODOS;
}

export class GetTodosSuccess implements Action {
  readonly type = TodosActionType.GET_TODOS_SUCCESS;
  constructor(public payload: Array<Todo>) {}
}

export class GetTodosFailed implements Action {
  readonly type = TodosActionType.GET_TODOS_FAILED;
  constructor(public payload: string) {}
}

export type TodosActions = GetTodos |
  GetTodosSuccess |
  GetTodosFailed;
```

Todo actions.

To get the to-do list from the REST API, we will have an action for each type of call. These verbs will be used in the reducer.

A reducer is a function that knows what to do with the action. The reducer will take the last position of the app from the Store and return to the new position. Furthermore, a reducer is a pure function. In JavaScript, a pure function means that its return value is the same for the same number of arguments and has no side effects (the outer scope is not changed). To get the to-do list, use the reducer as shown below:

```
import { TodosActions, TodosActionType } from './todos.actions';
import { Todo } from '../models/todo';
export const initialState = {};

export function todosReducer(state = initialState, action: TodosActions) {

  switch (action.type) {

    case TodosActionType.GET_TODOS: {
      return { ...state };
    }

    case TodosActionType.GET_TODOS_SUCCESS: {
      let msgText = "";
      let bgClass = "";

      if (action.payload.length < 1) {
        msgText = 'No data found';
        bgClass = 'bg-danger';
      } else {
        msgText = 'Loading data';
        bgClass = 'bg-info';
      }

      return {
        ...state,
```

```

        todoList: action.payload,
        message: msgText,
        infoClass: bgClass
    };
}

    case TodosActionType.GET_TODOS_FAILED: {
        return { ...state };
    }
}

```

Todo reduces.

With the GET_TODOS_SUCCESS action, we can see that the reducer returns an object that contains a to-do list, a message, and a CSS class. The object will be used to display the to-do list in our application.

Also, verbs can be used in effect.

An effect uses streams to provide new sources of actions to reduce states based on external interactions Like: REST API requests or Web socket messages. In fact, the effect is a kind of middleware that we use to obtain a new state of the stored data. For example, to get a list of todos, we must have the below service:

```

import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { catchError } from 'rxjs/operators';
import { throwError } from 'rxjs';
import { Todo } from '../models/todo';
import { headers } from '../headers/headers';

@Injectable({
  providedIn: 'root'
})
export class TodosService {

  baseUrl: string;

```

```

constructor(private http: HttpClient) {
  this.baseUrl = 'http://localhost:3000';
}

getAPITodos() {
  return this.http.get(`${this.baseUrl}/todos`, { headers })
    .pipe(catchError((error: any) => throwError(error.message)));
}
}

```

Todo service.

It is a simple way to get data from the API. The **getAPITodos** method returns an observable.

And the effects of this service will be below:

```

import { Injectable } from '@angular/core';
import { Actions, Effect, ofType } from '@ngrx/effects';

import { TodosService } from '../services/todos/todos.service';

import {
  TodosActionType,
  GetTodosSuccess, GetTodosFailed,
  AddTodoSuccess, AddTodoFailed,
  UpdateTodoSuccess, UpdateTodoFailed,
  DeleteTodoSuccess,
  DeleteTodoFailed
} from './todos.actions';
import { switchMap, catchError, map } from 'rxjs/operators';
import { of } from 'rxjs';

import { Todo } from '../models/todo';

@Injectable()
export class TodosEffects {

```

```

constructor(
  private actions$: Actions,
  private todosService: TodosService
) {}

@Effect()
getTodos$ = this.actions$.pipe(
  ofType(TodosActionType.GET_TODO),
  switchMap(() =>
    this.todosService.getAPITodos().pipe(
      map((todos: Array<Todo>) => new GetTodosSuccess(todos)),
      catchError(error => of(new GetTodosFailed(error)))
    )
  )
);
}

```

Todo effects.

The effect will return GetTodosSuccess if we get the data from the API, or it will return GetTodosFailed if it fails.

To access the data, we need to send an action to the store in todo-list.component.ts :

```

import { Component, OnInit } from '@angular/core';
import { Todo } from '../common/models/todo';
import { Store } from '@ngrx/store';
import * as Todos from '../common/store/todos/todos.actions';

@Component({
  selector: 'app-todo-list',
  templateUrl: './todo-list.component.html',
  styleUrls: ['./todo-list.component.scss']
})

```

```

})
export class TodoListComponent implements OnInit {

  todos: Array<Todo>;
  message: string;
  bgClass: string;
  p = 1;

  constructor(private store: Store<any>) { }

  ngOnInit() {
    this.store.dispatch(new Todos.GetTodos());
    this.store.select('todos').subscribe(response => {

      this.todos = response.todoList;
      this.message = response.message;
      this.bgClass = response.infoClass;

      setTimeout(() => {
        this.message = "";
      }, 2000);

    }, error => {
      console.log(error);
    });
  }
}

```

todo-list.component.ts file.

The template of the **todo-list.component.ts** component is the following:


```

<div class="container-fluid" *ngIf="todos">
  <div class="row">

    <div class="col-12">

      <div class="card mt-5">
        <div class="card-header">
          <h1 class="display-6 d-inline">Todo App</h1>

          <app-add-todo></app-add-todo>

        </div>
        <div class="card-body">
          <table class="table">
            <tbody>
              <tr *ngFor="let todo of todos | paginate: { itemsPerPage: 10, currentPage: p }">
                <td>
                  <code>{{todo | json}}</code>
                </td>
              </tr>
            </tbody>
          </table>

          <pagination-controls (pageChange)="p = $event"></pagination-controls>
        </div>
      </div>

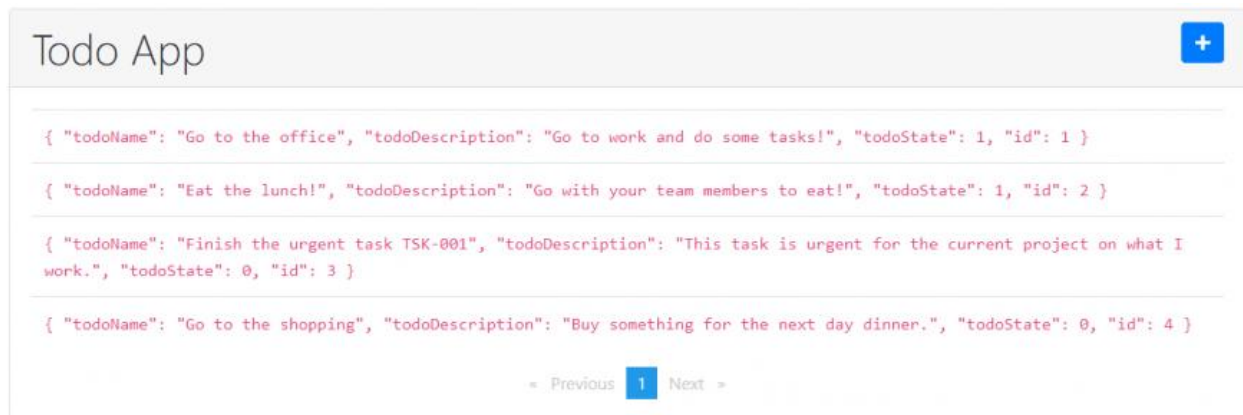
    </div>

  </div>
</div>

```

Template file for todo-list component.

The result of the GET operation will look like this:



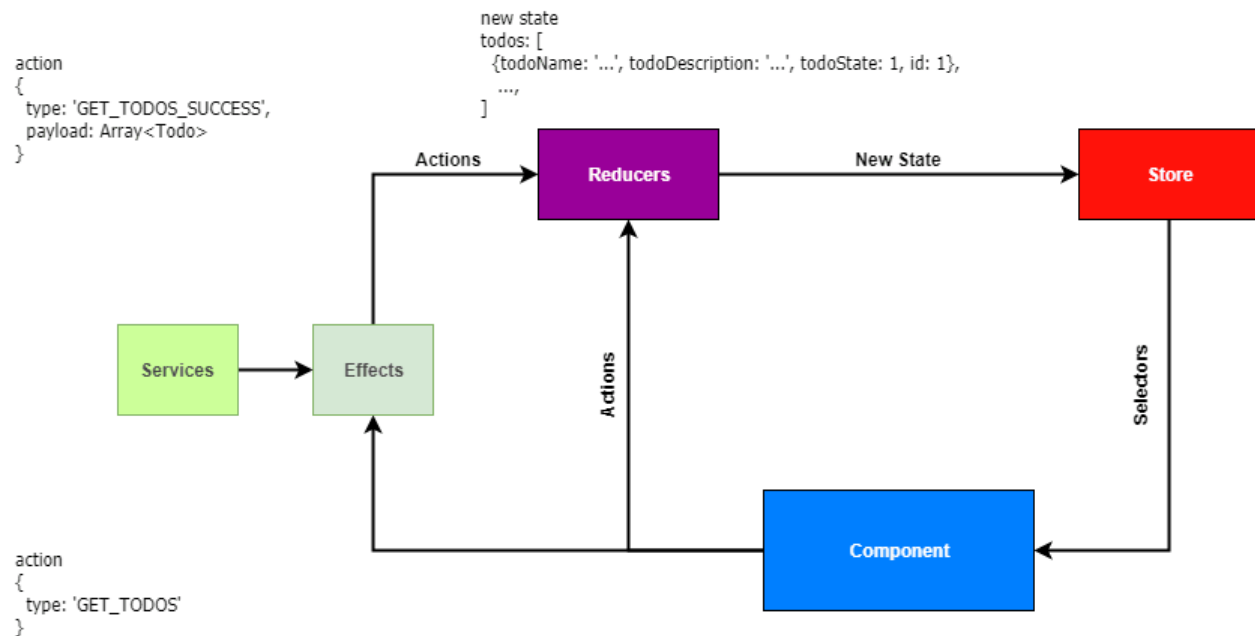
GET operation result.

To get a list of items which you select the to-do list and subscribe to it in your store. You can also add new elements to the import array from **todos.module.ts**. Then, you need to add stores for your modules (**features**), reducers and effects:









```
imports: [  
  CommonModule,  
  HttpClientModule,  
  NgbModule,  
  FormsModule,  
  ReactiveFormsModule,  
  NgxPaginationModule,  
  StoreModule.forRoot({}),  
  StoreModule.forFeature('todos', todosReducer, { metaReducers }),  
  EffectsModule.forRoot([]),  
  EffectsModule.forFeature([TodosEffects])  
],
```

todos.module.ts updated imports array.

NgRx workflow diagram for a GET to-do list. Following the steps, we can finish the application from CRUD operations to GET operations. And the following would be the diagram for GET operation using NgRx library:



Other CRUD operations (create, update, and delete) for the article can be found in the source code on Github, and after you've finished your application, it should look like this:

Todo App				
#id	#title	#description:	#done ✓	#actions
1	Go to the office	Go to work and do some tasks!	<input checked="" type="checkbox"/>	 
#id	#title	#description:	#done ✓	#actions
2	Eat the lunch!	Go with your team members to eat!	<input checked="" type="checkbox"/>	 
#id	#title	#description:	#in progress ⚙	#actions
3	Finish the urgent task TSK-001	This task is urgent for the current project on what I work.	<input type="checkbox"/>	 
#id	#title	#description:	#in progress ⚙	#actions
4	Go to the shopping	Buy something for the next day dinner.	<input type="checkbox"/>	 
< Previous 1 Next >				

Complete to-do application.

Conclusion

We can see how easy it is to add the NGRX library to an Angular project. Here are some of my conclusions before I conclude:

- The NgRx library is great if we want to use it in large applications. We need to do some configuration. If you have more than 20-30 components, this library will be helpful).
- A large application uses the **NgRx** library which is easier to understand for a new team member.
- Easy to follow data flow and debug the application.
- The NgRx library becomes more robust and flexible by using Redux in an Angular application.

8. Http Client

The `HttpClient` is a separate model in Angular and is available under the `@angular/common/http` package. The following steps show you how to use the `HttpClient` in an Angular app.

Import HttpClient Module in Root Module

We need to import it into our root module `app.module`. Also, we need to add it to the `imports` metadata array.

```
1
2import { NgModule } from '@angular/core';
3import { HttpClientModule } from '@angular/common/http';
4
5@NgModule({
6  declarations: [
7    AppComponent
8  ],
9  imports: [
10   HttpClientModule
11 ],
12 providers: [],
13 bootstrap: [AppComponent]
14})
15export class AppModule { }
16
```

Import Required Module in Component/Service

Then you should import `HttpClient` the `@angular/common/http` in the component or service.

```
1
2import { HttpClient } from '@angular/common/http';
3
```

Inject HttpClient service

Inject the HttpClient service in the constructor.

```
1
2constructor(public http: HttpClient) {
3}
4
```

Call the HttpClient.Get method

Use HttpClient.Get method to send an [HTTP Request](#). The request is sent when we Subscribe to the get() method. When the response arrives map it the desired object and display the result.

```
1
2public getData() {
3  this.httpClient.get<any[]>(`${this.baseUrl}/users/${this.userName}/repos`)
4    .subscribe(data => {
5      this.repos = data;
6    },
7    error => {
8    }
9  );
10}
11
```

What is Observable?

The Angular `HttpClient` makes use of observable. Hence it is important to understand the basics of it

Observable help us to manage async data. You can think of Observables as an array of items, which arrive asynchronously over time.

The observables implement the [observer design pattern](#), where observables maintain a list of dependents. We call these dependents as observers. The observable notifies them automatically of any state changes, usually by calling one of their methods.

Observer subscribes to an Observable. The observer reacts when the value of the Observable changes. An Observable can have multiple subscribers and all the subscribers are notified when the state of the Observable changes.

When an Observer subscribes to an observable, it needs to pass (optional) the three callbacks. `next()`, `error()` & `complete()`. The observable invokes the `next()` callback, when it receives an value. When the observable completes it invokes the `complete()` callback. And when the error occurs it invokes the `error()` callback with details of error and subscriber finishes.

The Observables are used extensively in Angular. The new `HttpClient` Module and Event system are all Observable based.

The Observables are proposed feature for the next version of Javascript. The Angular uses a Third-party library called [Reactive Extensions](#) or RxJs to implement the Observables. You can learn about RxJs from these [RxJs tutorials](#)

Observables Operators

Operators are methods that operate on an Observable and return a new observable. Each Operator modifies the value it receives. These operators are applied one after the other in a chain.

The RxJs provides several Operators, which allows you to filter, select, transform, combine and compose Observables. Examples of Operators are `map`, `filter`, `take`, `merge`, etc

How to use RxJs

The RxJs is a very large library. Hence Angular exposes a stripped-down version of Observables. You can import it using the following import statement

```
1
```

```
2import { Observable } from 'rxjs';
```

```
3
```

The above import imports only the necessary features. It does not include any of the Operators.

To use observables operators, you need to import them. The following code imports the `map` & `catchError` operators.

HTTP GET

The `HttpClient.get` sends the HTTP Get Request to the API endpoint and parses the returned result to the desired type. By default, the body of the response is parsed as JSON. If you want any other type, then you need to specify explicitly using the `observe` & `responseType` options. You can read more about [Angular HTTP Get](#)

Syntax

```
1
2 get(url: string,
3     options: {
4       headers?: HttpHeaders | { [header: string]: string | string[] };
5       params?: HttpParams | { [param: string]: string | string[] };
6       observe?: "body|events|response";
7       responseType: "arraybuffer|json|blob|text";
8       reportProgress?: boolean;
9       withCredentials?: boolean;}
10 ): Observable<>
11
```

Options

under the options, we have several configuration options, which we can use to configure the request.

headers It allows you to add HTTP headers to the outgoing requests.

observe The `HttpClient.get` method returns the body of the response parsed as JSON (or type specified by the `responseType`). Sometimes you may need to read the entire response along with the headers and status codes. To do this you can set the `observe` property to the **response**.

The allowed options are

- a `response` which returns the entire response

- body which returns only the body
- events which return the response with events.

params Allows us to Add the [URL parameters or Get Parameters](#) to the Get Request

reportProgress This is a boolean property. Set this to true, if you want to get notified of the progress of the Get Request. This is a pretty useful feature when you have a large amount of data to download (or upload) and you want the user to notify of the progress.

responseType Json is the default response type. In case you want a different type of response, then you need to use this parameter. The Allowed Options are arraybuffer, blob, JSON, and text.

withCredentials It is of boolean type. If the value is true then HttpClient.get will request data with credentials (cookies)

HTTP Post

The HttpClient.post() sends the HTTP POST request to the endpoint. Similar to the get(), we need to subscribe to the post() method to send the request. The post method parsed the body of the response as JSON and returns it. This is the default behavior. If you want any other type, then you need to specify explicitly using the observe & responseType options.

You can read [Angular HTTP Post](#)

The syntax of the HTTP Post is similar to the HTTP Get.

```

1
2 post(url: string,
3   body: any,
4   options: {
5     headers?: HttpHeaders | { [header: string]: string | string[]; };
6     observe?: "body|events|response";
7     params?: HttpParams | { [param: string]: string | string[]; };
8     reportProgress?: boolean;
9     responseType: "arraybuffer|json|blob|text";
10    withCredentials?: boolean;
11  }
12): Observable
13

```

The following is an example of HTTP Post

```

1
2addPerson(person:Person): Observable<any> {
3  const headers = { 'content-type': 'application/json'}
4  const body=JSON.stringify(person);
5  this.http.post(this.baseURL + 'people', body,{headers:headers , observe: 'response'})
6  .subscribe(
7    response=> {
8      console.log("POST completed sucessfully. The response received "+response);
9    },
10   error => {
11     console.log("Post failed with the errors");
12   },
13   () => {
14     console.log("Post Completed");
15   }
16}
17
18

```

HTTP PUT

The `HttpClient.put()` sends the `HTTP PUT` request to the endpoint. The syntax and usage are very similar to the `HTTP POST` method.

```

1
2put(url: string,
3  body: any,
4  options: {
5    headers?: HttpHeaders | { [header: string]: string | string[]; };

```

```

6  observe?: "body|events|response|";
7  params?: HttpParams | { [param: string]: string | string[]; };
8  reportProgress?: boolean;
9  responseType: "arraybuffer|json|blob|text";
10 withCredentials?: boolean;
11 }
12): Observable
13

```

HTTP PATCH

The `HttpClient.patch()` sends the `HTTP PATCH` request to the endpoint. The syntax and usage are very similar to the `HTTP POST` method.

```

1
2patch(url: string,
3  body: any,
4  options: {
5    headers?: HttpHeaders | { [header: string]: string | string[]; };
6    observe?: "body|events|response|";
7    params?: HttpParams | { [param: string]: string | string[]; };
8    reportProgress?: boolean;
9    responseType: "arraybuffer|json|blob|text";
10   withCredentials?: boolean;
11  }
12): Observable
13

```

HTTP DELETE

The `HttpClient.delete()` sends the `HTTP DELETE` request to the endpoint. The syntax and usage are very similar to the `HTTP GET` method.

```
1
2 delete(url: string,
3   options: {
4     headers?: HttpHeaders | { [header: string]: string | string[] };
5     params?: HttpParams | { [param: string]: string | string[] };
6     observe?: "body|events|response";
7     responseType: "arraybuffer|json|blob|text";
8     reportProgress?: boolean;
9     withCredentials?: boolean;
10  }): Observable<>
11
```

HttpClient Example

Now, We have a basic understanding of `HttpClient` model & observables, let us build an `HttpClient` example app.

Create a new Angular app

```
1
2ng new httpClient
3
```

import HttpClientModule

In the `app.module.ts` import the `HttpClientModule` module as shown below. We also add it to the imports array

```
1
2import { BrowserModule } from '@angular/platform-browser';
3import { NgModule } from '@angular/core';
4import { HttpClientModule } from '@angular/common/http';
5import { AppRoutingModule } from './app-routing.module';
6import { AppComponent } from './app.component';
7
8@NgModule({
9  declarations: [
10   AppComponent
11 ],
12 imports: [
13   BrowserModule,
14   AppRoutingModule,
15   HttpClientModule
16 ],
17 providers: [],
```

```
18 bootstrap: [AppComponent]
19})
20export class AppModule { }
21
22
```

Component

Now, open the `app.component.ts` and copy the following code.

```
1
2import { Component, OnInit } from '@angular/core';
3import { HttpClient } from '@angular/common/http';
4
5export class Repos {
6  id: string;
7  name: string;
8  html_url: string;
9  description: string;
10}
11
12@Component({
13  selector: 'app-root',
14  templateUrl: './app.component.html',
15})
16export class AppComponent implements OnInit {
17
18  userName: string = "tektutorialshub"
19  baseUrl: string = "https://api.github.com/";

```

```
20 repos: Repos[];
21
22
23 constructor(private http: HttpClient) {
24 }
25
26 ngOnInit() {
27   this.getRepos()
28 }
29
30
31 public getRepos() {
32
33   return this.http.get<Repos[]>(this.baseUrl + 'users/' + this.userName + '/repos')
34     .subscribe(
35       (response) => {           //Next callback
36         console.log('response received')
37         console.log(response);
38         this.repos = response;
39       },
40       (error) => {             //Error callback
41         console.error('Request failed with error')
42         alert(error);
43       },
44       () => {                   //Complete callback
45         console.log('Request completed')
46       })
47 }
```

```
48}
```

```
49
```

Import HttpClient

HttpClient is a service, which is a major component of the HTTP Module. It contains methods like GET, POST, PUT etc. We need to import it.

```
1
```

```
2import { HttpClient } from '@angular/common/http';
```

```
3
```

Repository Model

The model to handle our data.

```
1
```

```
2export class Repos {
```

```
3 id: string;
```

```
4 name: string;
```

```
5 html_url: string;
```

```
6 description: string;
```

```
7}
```

```
8
```

Inject HttpClient

Inject the HttpClient service into the component. You can learn more about [Dependency injection in Angular](#)


```

1
2 constructor(private http: HttpClient) {
3 }
4

```

Subscribe to HTTP Get

The `GetRepos` method, we invoke the `get()` method of the `HttpClient` Service. The `HttpClient.get` method allows us to cast the returned response object to a type we require. We make use of that feature and supply the type for the returned value `http.get<repos[]>`. The `get()` method returns an observable. Hence we subscribe to it.

```

1
2 public getRepos() {
3   return this.http.get<Repos[]>(this.baseUrl + 'users/' + this.userName + '/repos')
4     .subscribe(
5

```

When we subscribe to any observable, we optionally pass the three callbacks `next()`, `error()` & `complete()`. In this example we pass only two callbacks `next()` & `error()`.

Receive the Response

We receive the data in the `next()` callback. By default, the Angular reads the body of the response as JSON and casts it to an object and returns it back. Hence we can use directly in our app.

```

1
2 (response) => {           //Next callback
3   console.log('response received')
4   console.log(response);
5   this.repos = response;

```

```
6   },
```

```
7
```

Handle the errors

We handle the errors in `error` callback.

```
1
```

```
2   (error) => {                                //Error callback
```

```
3     console.error('Request failed with error')
```

```
4     alert(error);
```

```
5   },
```

```
6
```

Template

```
1
```

```
2<h1 class="heading"><strong>HTTPClient </strong> Example</h1>
```

```
3
```

```
4
```

```
5<table class='table'>
```

```
6  <thead>
```

```
7    <tr>
```

```
8      <th>ID</th>
```

```
9      <th>Name</th>
```

```
10     <th>HTML Url</th>
```

```
11     <th>description</th>
```

```
12   </tr>
```

```
13 </thead>
```

```
14 <tbody>
```

```
15   <tr *ngFor="let repo of repos;">
```

```
16     <td>{{repo.id}}</td>
```

```
17     <td>{{repo.name}}</td>
```

```
18 <td>{{repo.html_url}}</td>
19 <td>{{repo.description}}</td>
20 </tr>
21 </tbody>
22</table>
23
24<pre>{{repos | json}}</pre>
25
```

10. Role Based Authorization

In the system's security, the role-based authorization / role-based access control (RBAC) is a way of restricting and managing machine access to authorized users. RBAC (Role-based access control) confines access to the network based on the employee's role in an organization.

Understanding role-based authorization

We will make an angular app to restrict the access of sources of a page by implementing role-based authorization. This application will contain 3 pages (a login page, a home page, an admin page). In our application, the user can either be a normal user or an admin. A normal user has only access to the home page and an admin can access the home page as well as the admin page.

Start by Creating y our application and select whatever style property you want by using this command in the terminal.

```
ng new role-based-authorization --routing
```

Now, we will create folders for each module i.e., home, admin, login. And additional folders for helpers, services and models with prefix underscore (_) to differentiate them from other components.

We have made the following **3 components** in the application:

Admin Component

Admin.component.html:

This file is an admin component template that contains html code for displaying content on the admin page.

```
<xmp><div class="card mt-4">
<h4 class="card-header">Admin</h4>
<div class="card-body">
<p>Admin dashboard!!</p>
<p class="mb-1">All users from secure (admin only) api end point:</p>
<div></div>
<ul><li>
</li><li>{{user.firstName}} {{user.lastName}}</li><li>
</li></ul>
</div></div>
</xmp>
```

admin.component.ts:

The admin component calls the user service and finds all users from one secure API endpoint. It stores them in a local user's property which is accessible to the admin component template.

```
<xmp>
import { Component, OnInit } from '@angular/core';
import { first } from 'rxjs/operators';
import { User } from '../_models';
import { UserService } from '../_services';
@Component({ templateUrl: 'admin.component.html' })
export class AdminComponent implements OnInit {
  loading = false;
  users: User[] = [];
  constructor(private userService: UserService) { }
  ngOnInit() {
    this.loading = true;
    this.userService.getAll().pipe(first()).subscribe(users => {
      this.loading = false;
      this.users = users;
    });
  }
}
</xmp>
```

Home Component

home.component.html:

This file is a home component template which contains html code for displaying content on the home page.

```
<xmp>
```

```

<div class="card mt-4">

  <h4 class="card-header">Home</h4>

  <div class="card-body">

    <p>Welcome to home page!!</p>

    <p>Your role is: <strong>{{currentUser.role}}</strong>.</p>

  </div>

</div>

</xmp>

```

home.component.ts:

The home component fetches the current user from the authentication service so gets the current user from the api with the user service.

```

<xmp>

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

import { Router, ActivatedRoute } from '@angular/router';

import { FormBuilder, FormGroup, Validators } from '@angular/forms';

import { first } from 'rxjs/operators';

import { AuthenticationService } from '../_services';

@Component({ templateUrl: 'login.component.html' })

export class LoginComponent implements OnInit {

  loginForm: FormGroup;

  loading = false;

  submitted = false;

```

```
returnUrl: string;

error = '';

constructor(

private formBuilder: FormBuilder,

private route: ActivatedRoute,

private router: Router,

private authenticationService: AuthenticationService

) {

if (this.authenticationService.currentUserValue) {

this.router.navigate(['']);

}

}

ngOnInit() {

this.loginForm = this.formBuilder.group({

username: ['', Validators.required],

password: ['', Validators.required]

});

this.returnUrl = this.route.snapshot.queryParams['returnUrl'] || '';

}

get f() { return this.loginForm.controls; }

onSubmit() {

this.submitted = true;
```

```
if (this.loginForm.invalid) {  
  return;  
}  
  
this.loading = true;  
  
this.authenticationService.login(this.f.username.value, this.f.password.value)  
  .pipe(first())  
  .subscribe(  
    data => {  
      this.router.navigate([this.returnUrl]);  
    },  
    error => {  
      this.error = error;  
      this.loading = false;  
    });  
  }  
}  
  
</xmp>
```

Login component

login.component.html

The home component fetches the current user from the authentication service and then gets the current user from the api with the user service.

```
<xmp><div class="col-md-6 offset-md-3 mt-5">
```



```
</form>

</div>

</div>

</div>

</xmp>
```

login.component.ts:

This component makes use of the authentication service for logging into the application. In the case where the user is already logged in, it will automatically redirect them to the home page.

```
<xmp>

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

import { Router, ActivatedRoute } from '@angular/router';

import { FormBuilder, FormGroup, Validators } from '@angular/forms';

import { first } from 'rxjs/operators';

import { AuthenticationService } from '../_services';

@Component({ templateUrl: 'login.component.html' })

export class LoginComponent implements OnInit {

  loginForm: FormGroup;

  loading = false;

  submitted = false;

  returnUrl: string;

  error = '';

  constructor(
```

```
private FormBuilder: FormBuilder,

private route: ActivatedRoute,

private router: Router,

private authenticationService: AuthenticationService

){

if (this.authenticationService.currentUserValue) {

this.router.navigate(['&#39;/&#39;]);

}

}

ngOnInit() {

this.loginForm = this.formBuilder.group({

username: ['&#39;&#39;, Validators.required],

password: ['&#39;&#39;, Validators.required]

});

this.returnUrl = this.route.snapshot.queryParams['&#39;returnUrl&#39;'] || '&#39;/&#39;;

}

get f() { return this.loginForm.controls; }

onSubmit() {

this.submitted = true;

if (this.loginForm.invalid) {

return;

}
```

```
this.loading = true;

this.authenticationService.login(this.f.username.value, this.f.password.value)

.pipe(first())

.subscribe(

data => {

this.router.navigate([this.returnUrl]);

},

error => {

this.error = error;

this.loading = false;

});

}

}

</xmp>
```

_helpers

The followings are the files under the “_helper” folders.

Auth Guard

The auth guard is an angular route guard which is used to shut out unauthenticated or unauthorized users from accessing restricted routes.

This is done by implementing the CanActivate interface that permits the guard to determine if a route is activated with the canActivate(). If this method returns true it means that the route is activated, the user is allowed to proceed, and if this method returns false the route is blocked.

The auth guard uses the authentication service to visualize if the user is logged in, if they’re logged in, it checks if their role is permitted to access the requested route. If they’re logged in and licensed

the `canActivate()` method returns true, otherwise, it returns false and redirects the user to the login page.

Angular route guards are associated with the routes in the router config, this auth guard is used in `app.routing.ts` to protect the home page and admin page routes.

auth.guard.ts

```
<xmp>

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

import { Router, CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot } from
'@angular/router';

import { AuthenticationService } from '../_services';

@Injectable({ providedIn: 'root' })

export class AuthGuard implements CanActivate {

  constructor(

    private router: Router,

    private authenticationService: AuthenticationService

  ) {}

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {

    const currentUser = this.authenticationService.currentUserValue;

    if (currentUser) {

      if (route.data.roles &&

route.data.roles.indexOf(currentUser.role) === -1)

      {

        this.router.navigate(['</>']);

      }

    }

    return false;

  }

}
```

```

}

return true;

}

this.router.navigate(['&#39;/login&#39;],
{ queryParams: { returnUrl: state.url } });

return false;

}

}

</xmp>

```

HTTP Error Interceptor

The Error Interceptor intercepts HTTP responses from the api to check if there have been any errors. If there's a 401 Unauthorized or 403 Forbidden response the user is automatically logged out of the application, all other errors will be re-thrown up to the calling service so that an alert with the error can be displayed on the screen.

This interceptor is applied using the HttpInterceptor class contained in the HttpClientModule.

Error.interceptor.ts:

```

<xmp>

import { Injectable } from &#39;@angular/core&#39;;

import { HttpRequest, HttpHandler, HttpEvent, HttpInterceptor } from
&#39;@angular/common/http&#39;;

import { Observable, throwError } from &#39;rxjs&#39;;

import { catchError } from &#39;rxjs/operators&#39;;

import { AuthenticationService } from &#39;../_services&#39;;

```

```

@Injectable()

export class ErrorInterceptor implements HttpInterceptor {

  constructor(private authenticationService: AuthenticationService) {}

  intercept(request: HttpRequest<any>, next: HttpHandler): Observable<httpevent<any>> {

    return next.handle(request).pipe(catchError(err => {

      if ([401, 403].indexOf(err.status) !== -1) {

        this.authenticationService.logout();

        location.reload(true);

      }

      const error = err.error.message || err.statusText;

      return throwError(error);

    })))

  }

}

</httpevent<any>></any></xmp>

```

Fake Backend Provider

In this application, we used a fake backend API to intercept the HTTP requests to run and test the angular application instead of using a real backend API. We have done this by a class that implements the Angular HttpInterceptor interface.

fake-backend.ts

```

<xmp><any><httpevent<any>

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

```

```

import { HttpRequest, HttpResponse, HttpHandler, HttpEvent, HttpInterceptor,
HTTP_INTERCEPTORS } from '@angular/common/http';

import { Observable, of, throwError } from 'rxjs';

import { delay, mergeMap, materialize, dematerialize } from 'rxjs/operators';

import { User, Role } from '../_models';

const users: User[] = [

  { id: 1, username: 'admin', password: 'admin', firstName:
  'Admin', lastName: 'User', role: Role.Admin },

  { id: 2, username: 'user', password: 'user', firstName: 'Normal',
  lastName: 'User', role: Role.User }

];

@Injectable()

export class FakeBackendInterceptor implements HttpInterceptor {

  intercept(request: HttpRequest<any>, next: HttpHandler): Observable<httpevent<any>> {

    const { url, method, headers, body } = request;

    return of(null)

      .pipe(mergeMap(handleRoute))

      .pipe(materialize())

      .pipe(delay(500))

      .pipe(dematerialize());

    function handleRoute() {

      switch (true) {

        case url.endsWith('/users/authenticate') && method === 'POST':

```



```
return authenticate();

case url.endsWith('/users') && method === 'GET':
return getUsers();

case url.match(/\/users\/\d+$/) && method === 'GET':
return getUserById();

default:

return next.handle(request);
}
}

function authenticate() {

const { username, password } = body;

const user = users.find(x => x.username === username
&& x.password === password);

if (!user) return error('Username or password is incorrect');

return ok({
id: user.id,
username: user.username,
firstName: user.firstName,
lastName: user.lastName,
role: user.role,
token: `fake-jwt-token.${user.id}`
});
```

```
}

function getUsers() {

if (!isAdmin()) return unauthorized();

return ok(users);

}

function getUserById() {

if (!isLoggedIn()) return unauthorized();

if (!isAdmin() && currentUser().id !== idFromUrl())

return unauthorized();

const user = users.find(x => x.id === idFromUrl());

return ok(user);

}

function ok(body) {

return of(new HttpResponse({ status: 200, body }));

}

function unauthorized() {

return throwError({ status: 401,

error: { message: 'unauthorized' } });

}

function error(message) {

return throwError({ status: 400, error: { message } });

}
```

```
function isLoggedIn() {  
  
  const authHeader = headers.get('Authorization') || '';  
  
  return authHeader.startsWith('Bearer fake-jwt-token');  
  
}  
  
function isAdmin() {  
  
  return isLoggedIn() && currentUser().role === Role.Admin;  
  
}  
  
function currentUser() {  
  
  if (!isLoggedIn()) return;  
  
  const id = parseInt(headers.get('Authorization').split(',')[1]);  
  
  return users.find(x => x.id === id);  
  
}  
  
function idFromUrl() {  
  
  const urlParts = url.split('/');  
  
  return parseInt(urlParts[urlParts.length - 1]);  
  
}  
  
}  
  
}  
  
export const fakeBackendProvider = {  
  
  provide: HTTP_INTERCEPTORS,  
  
  useClass: FakeBackendInterceptor,  
  
  multi: true
```

```
};  
  
</httpevent<any></any></httpevent<any></any></xmp>
```

JWT Interceptor

This JWT Interceptor is used to intercept http requests from the angular application and to add a JWT auth token in the Authorization header.

jwt.interceptor.ts

```
<xmp><any><httpevent<any><any><httpevent<any>  
  
import { Injectable } from '#39;@angular/core#39;;  
  
import { HttpRequest, HttpHandler, HttpEvent, HttpInterceptor } from  
#39;@angular/common/http#39;;  
  
import { Observable } from '#39;rxjs#39;;  
  
import { environment } from '#39;@environments/environment#39;;  
  
import { AuthenticationService } from '#39;../_services#39;;  
  
@Injectable()  
  
export class JwtInterceptor implements HttpInterceptor {  
  
  constructor(private authenticationService: AuthenticationService) {}  
  
  intercept(request: HttpRequest<any>, next: HttpHandler): Observable<httpevent<any>> {  
  
    const currentUser = this.authenticationService.currentUserValue;  
  
    const isLoggedIn = currentUser && currentUser.token;  
  
    const isApiUrl = request.url.startsWith(environment.apiUrl);  
  
    if (isLoggedIn && isApiUrl) {  
  
      request = request.clone({
```

```

setHeaders: {
  Authorization: `Bearer ${currentUser.token}`
}
});
}

return next.handle(request);
}
}

</httpevent<any></any></httpevent<any></any></httpevent<any></any></xmp>

```

_models

Followings are the files under the “_models” folder:

Role Model

The role model file consists of an enum that states the roles supported by our application.

role.ts:

```

<xmp><any><httpevent<any><any><httpevent<any><any><httpevent<any>
export enum Role {
  User = &#39;User&#39;,
  Admin = &#39;Admin&#39;
}

</httpevent<any></any></httpevent<any></any></httpevent<any></any></xmp>

```

User Model

The user model file is a small class consisting of the properties of a user along with their role as well as jwt auth token.

user.ts

```
<xmp><any><httpevent<any><any><httpevent<any><any><httpevent<any>  
  
import { Role } from &quot;./role&quot;;  
  
export class User {  
  
  id: number;  
  
  username: string;  
  
  password: string;  
  
  firstName: string;  
  
  lastName: string;  
  
  role: Role;  
  
  token?: string;  
  
}  
  
</httpevent<any></any></httpevent<any></any></httpevent<any></any></xmp>
```

_services

The following files are put under the “_service” folder:

Authentication Service

This service is used to login & logout in the Angular application. When the user logs in and out, it will be notified to other components.

authentication.service.ts

```
<xmp><any><httpevent<any><any><httpevent<any><any><httpevent<any>  
  
import { Injectable } from &#39;@angular/core&#39;;  
  
import { HttpClient } from &#39;@angular/common/http&#39;;
```

```

import { BehaviorSubject, Observable } from '#39;rxjs#39;;

import { map } from '#39;rxjs/operators#39;;

import { environment } from '#39;@environments/environment#39;;

import { User } from '#39;../_models#39;;

@Injectable({ providedIn: '#39;root#39; })

export class AuthenticationService {

  private currentUserSubject: BehaviorSubject<user>;

  public currentUser: Observable<user>;

  constructor(private http: HttpClient) {

    this.currentUserSubject = new
    BehaviorSubject<user>(JSON.parse(localStorage.getItem('#39;currentUser#39;)));

    this.currentUser = this.currentUserSubject.asObservable();

  }

  public get currentUserValue(): User {

    return this.currentUserSubject.value;

  }

  login(username: string, password: string) {

    return this.http.post<any>(`${environment.apiUrl}/users/authenticate`, { username,
    password })

    .pipe(map(user => {

      if (user && user.token) {

        localStorage.setItem('#39;currentUser#39;', JSON.stringify(user));

        this.currentUserSubject.next(user);

```

```

}

return user;

});

}

logout() {

localStorage.removeItem('#39;currentUser#39;);

this.currentUserSubject.next(null);

}

}

</any></user></user></user></httpevent<any></any></httpevent<any></any></http
event<any></any></xmp>

```

User Service

The user service consists of methods to retrieve user data from the api. This acts as the interface between our Angular application and the backend api.

user.service.ts

```

<xmp><any><httpevent<any><any><httpevent<any><any><httpevent<any><user><use
r><user><any>

import { Injectable } from '#39;@angular/core#39;;

import { HttpClient } from '#39;@angular/common/http#39;;

import { environment } from '#39;@environments/environment#39;;

import { User } from '#39;../_models#39;;

@Injectable({ providedIn: '#39;root#39; })

export class UserService {

```



```

constructor(private http: HttpClient) { }

getAll() {

return this.http.get<user[]>(`${environment.apiUrl}/users`);

}

getById(id: number) {

return this.http.get<user>(`${environment.apiUrl}/users/${id}`);

}

}

</user></user[]></any></user></user></user></httpevent<any></any></httpevent<a
ny></any></httpevent<any></any></xmp>

```

Conclusion

Role-Based-Authorization is necessary for any organization to protect the resources from being used by an unauthorized person as in any organization many employees use one system.

RBA allows employees of an organization to have access rights only to the information they require to do their tasks and restricts them from accessing information that doesn't concern with their role.