Introduction to Angular Framework

Module Overview

In this module, students will be able to learn about frontend technologies such as HTML5, CSS3, Bootstrap, JavaScript and TypeScript.

Module Objective

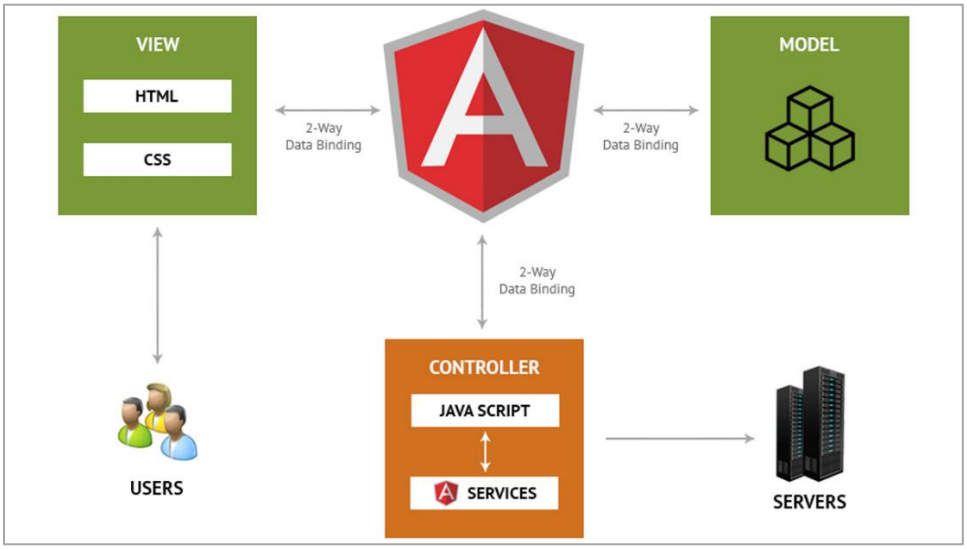
The Objectives of this module is to:

- to Set up Angular framework
- to Create the first angular app
- to Create components and setup templates
- to Use data binding, nesting, input and output properties
- to Define template, style and know angular directives
- to Know the angular pipes, services and dependency injections
- to Create template-driven and reactive forms
- to Know the component lifecycle
- to Use routing for navigation
- to Use http requests

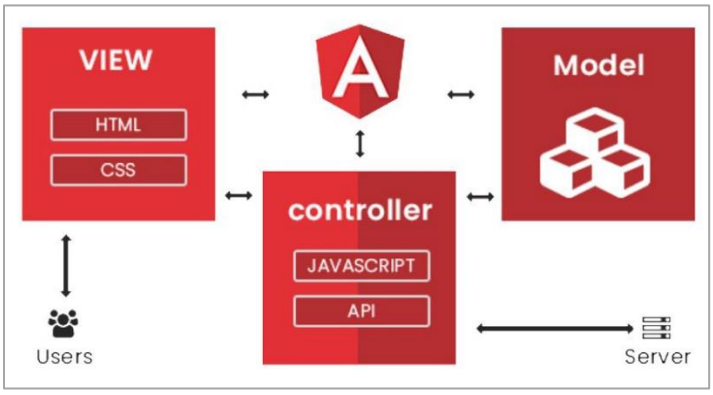
Introduction to Angular Framework, History & Overview

What is Angular?

Angular is an open-source, JavaScript framework written in TypeScript. Google maintains it, and its primary purpose is to develop single-page applications. As a framework, Angular has clear advantages while also providing a standard structure for developers to work with. It enables users to create large applications in a maintainable manner.



Angular applications are built around a design pattern called ModelView-Controller (MVC).



What is the history of Angular?

It all started in 2008 and 2009, Misko hevery (a developer at Google) working on a part time project to simplify web application development.

He developed and extended the vocabulary of HTML, not for web developers but web designers who have no or little knowledge about web development, so that if we have a static web server you can actually build a simple web application (without worrying about what is happening in the back ground).

Why do you need a Framework?

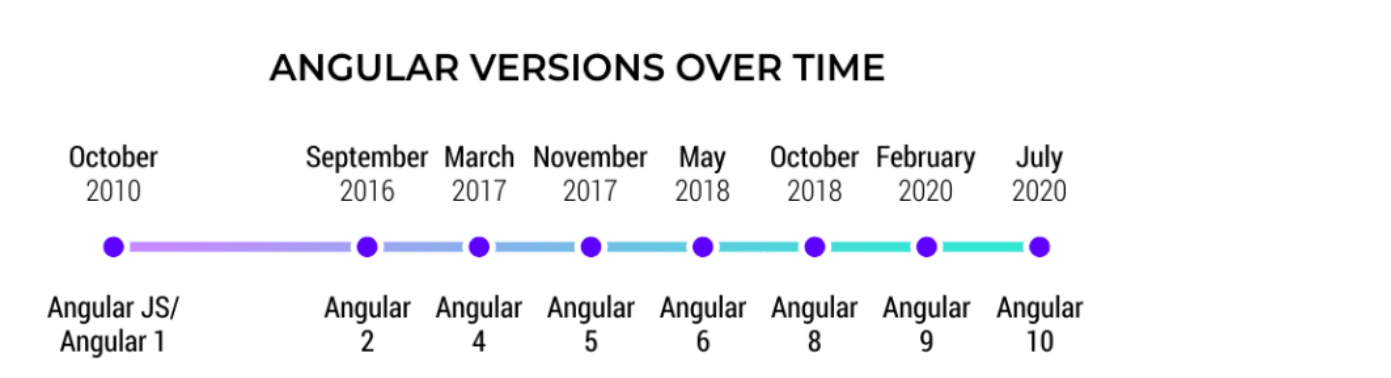
Frameworks in general boost web development efficiency and performance by providing a consistent structure so that developers don't have to keep rebuilding code from scratch. Frameworks are time-savers that offer developers a host of extra features that can be added to the software without requiring extra effort.

What are the Different Angular Versions?

"Angular" is the catch-all term for the various framework versions out there. Angular was developed in 2009, and as a result, there have been many iterations.

First, there was the original Angular, called Angular 1 and eventually known as AngularJS. Then came Angulars 2, 3, 4, 5, until finally, the current version, Angular 11, released on 11/11/2020. Each subsequent Angular version improves on its predecessor, fixing bugs, addressing issues, and accommodating increasing complexity of current platforms.

If you want to design apps better suited for mobile devices, and/or more complex apps, you had best to upgrade to its current version.



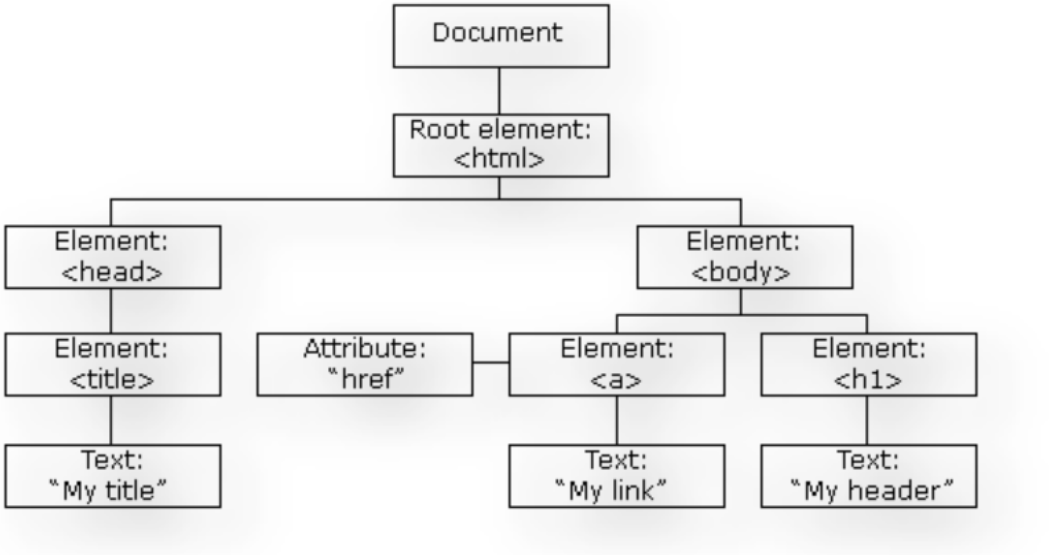
Comparison of few latest versions of Angular:

Angular 6	Angular 7	Angular 8	Angular 9
Angular Element	CLI Prompts	Ivy Engine	Default Ivy in v9
Service worker	Virtual Scrolling	Web Workers	Phantom Template Variable Menace
Internationalization (i18n)	Drag and Drop	Lazy Loading	Dependency Injection Changes in Core
Bazel Compiler	Bundle Budget	Improvement in ng-upgrade	Service Worker Updates
ng-add / ng-update	Angular Compiler	Support for Node 10	i18n Improvements
ng-update	Angular Do-Bootstrap	CLI workflow improvements	More reliable ng update
ngModelChange	Better Error Handling	Upgrading Angular Material	API Extractor Updates
TypeScript 2.7 support	TypeScript 3.1 support	TypeScript 3.4 support	Typescript 3.7 support
Improved decorator error messages	New ng-compiler	Differential Loading	Component Harness
<ng-template> updated to <template>	Native Script	Improved Web Worker Bundling	ModuleWithProviders Support

Features of Angular

1. Document Object Model

DOM (Document Object Model) treats an XML or HTML document as a tree structure in which each node represents a part of the document.



Angular uses regular DOM. Consider that ten updates are made on the same HTML page. Instead of updating the ones that were already updated, Angular will update the entire tree structure of HTML tags.

2. TypeScript

TypeScript defines a set of types to JavaScript, which helps users write JavaScript code that is easier to understand. All of the TypeScript code compiles with JavaScript and can run smoothly on any platform. TypeScript is not compulsory for developing an Angular application. However, it is highly recommended as it offers better syntactic structure—while making the codebase easier to understand and maintain.

You can install TypeScript as an NPM package with the following command:

npm install -g typescript

3. Data Binding

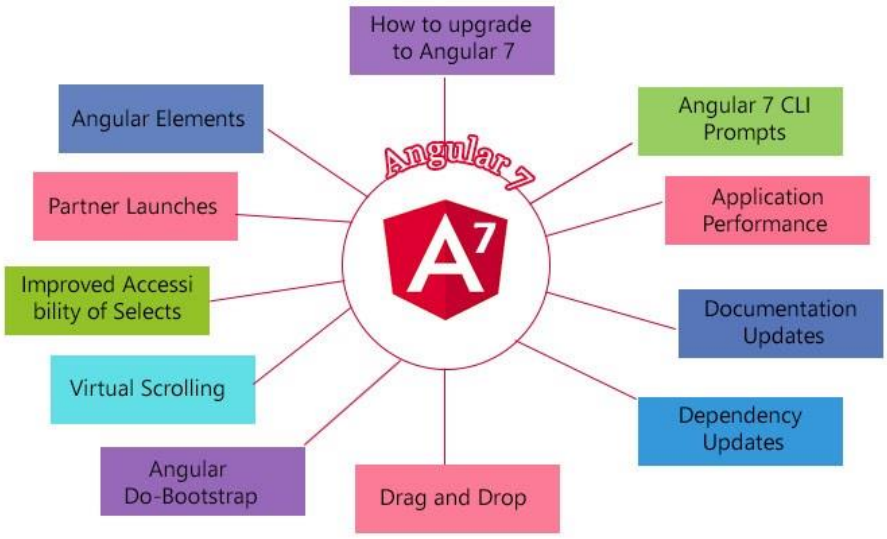
Data binding is a process that enables users to manipulate web page elements through a web browser. It employs dynamic HTML and does not require complex scripting or programming. Data binding is used in web pages that include interactive components, such as calculators, tutorials, forums, and games. It also enables a better incremental display of a web page when pages contain a large amount of data.

Angular uses the two-way binding. The model state reflects any changes made in the corresponding UI elements. Conversely, the UI state reflects any changes in the model state. This feature enables the framework to connect the DOM to the model data through the controller.

4. Testing

Angular uses the Jasmine testing framework. The Jasmine framework provides multiple functionalities to write different kinds of test cases. Karma is the task-runner for the tests that uses a configuration file to set the start-up, reporters, and testing framework.

Features of Angular 7



CLI prompts

In Angular 7, the CLI prompts have been updated to v7.0.2 with new features. For instance, it will now prompt users when typing commands like @angular/material, ng-new, and ng-add to help them discover the in-built SCSS support, routing, and more.

Angular material & component dev kit (CDK)

The Angular 7 introduced minor visual updates & improvements in Material Design that earlier received a major update this year only.

Drag & drop

The new drag-drop module basically provides a better way to easily create drag & drop interfaces, which is backed by sorting within a list, support for free dragging, animations, custom drag handles, transferring items between lists, previews, and placeholders.

Virtual scrolling

Like mentioned earlier, the new Virtual Scrolling in Angular 7 basically loads and unloads items from the DOM depending upon visible parts of lists, resulting into a much faster experiences for users having huge scrollable lists.

Application performance improvements

The development team at Google have always focused on the performance improvements, and while doing so, they recently found that most of the developers were using reflect-metadata in their production, which actually was only required in the development.

Documentation updates

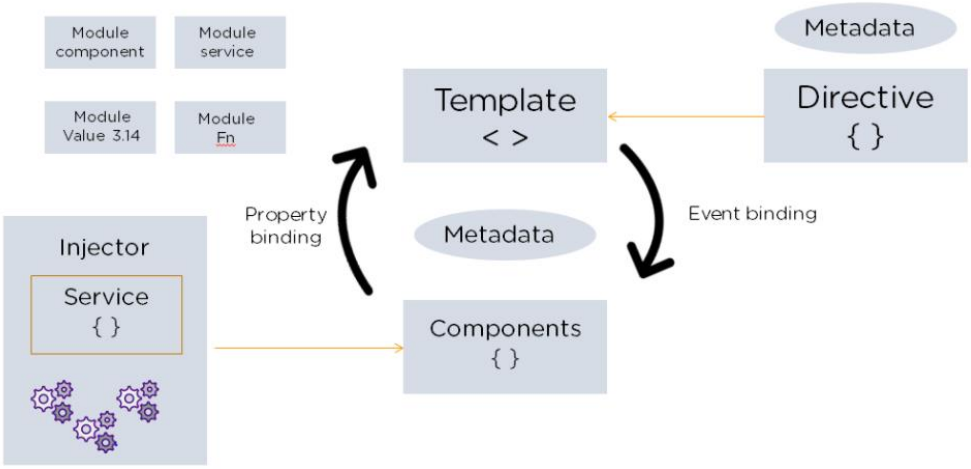
Another key improvement introduced in the Angular 7 is of the documentation update.

Dependency updates

Documentation are not the only things that have been updated. Even the dependencies have undergone upgradation for the third-party projects.

Angular Architecture

Angular is a full-fledged model-view-controller (MVC) framework. It provides clear guidance on how the application should be structured and offers bi-directional data flow while providing real DOM.



The following are the eight building blocks of an Angular application:

1. Modules

An Angular app has a root module, named AppModule, which provides the bootstrap mechanism to launch the application.

2. Components

Each component in the application defines a class that holds the application logic and data. A component generally defines a part of the user interface (UI).

3. Templates

The Angular template combines the Angular markup with HTML to modify HTML elements before they are displayed. There are two types of data binding:

Event binding: Lets your app respond to user input in the target environment by updating your application data.

Property binding: Enables users to interpolate values that are computed from your application data into the HTML.

4. Metadata

Metadata tells Angular how to process a class. It is used to decorate the class so that it can configure the expected behavior of a class.

5. Services

When you have data or logic that isn't associated with the view but has to be shared across components, a service class is created. The class is always associated with the @Injectible decorator.

6. Dependency Injection

This feature lets you keep your component classes crisp and efficient. It does not fetch data from a server, validate the user input, or log directly to the console. Instead, it delegates such tasks to the services.

Environment Setup

Let us see how you can install the prerequisites needed to run your first Angular 7 app.

1. Install Visual Studio Code IDE

You must have an IDE like Visual Studio Code IDE or JetBrains WebStorm to run your Angular 7 app. VS Code is light and easy to setup, it has a great range of built-in code editing, formatting, and refactoring features. It is free to use. It also provides a huge number of extensions that will significantly increase your productivity.

You can download VS Code from here: https://code.visualstudio.com

2. Install Node.js

You should install node.js to run your Angular 7 app. It manages npm dependencies support some browsers when loading particular pages. It provides the required libraries to run Angular project. Node.js serves your run-time environment as your localhost.

Visit the link and download from the node.js official website: https://nodejs.org/en/download/

What is Angular CLI?

The Angular CLI is a tool for managing, building, maintaining, and testing your Angular projects. CLI is an acronym for Command Line Interface. Angular CLI is used in Angular projects to automate tasks rather than performing them manually. Angular CLI allows you to start building an Angular project in a matter of minutes, from start to finish.

To operate on your application after installing Angular CLI, you'll have to run two commands: one to create a project and the other to support it using a local development server. Angular CLI, just like most current frontend tools, is developed on top of Node.js.

Some of the things you can use Angular CLI for include;

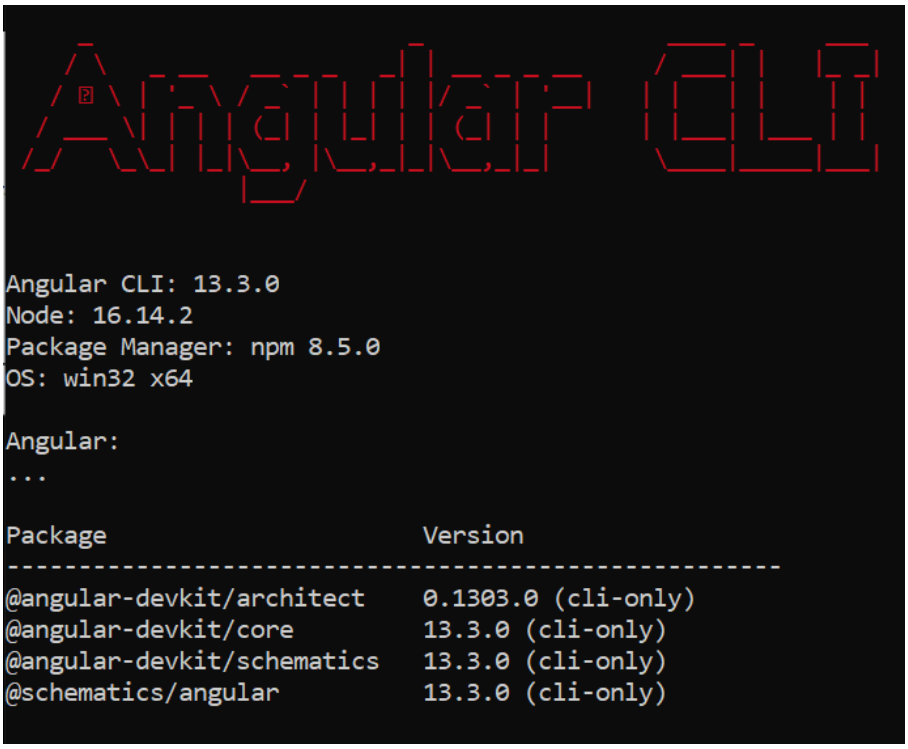
- Environment Setup and Configurations
- Developing components and building services
- Beginning, testing and launching a project
- Installation of 3rd party libraries such as Sass and Bootstrap, among others
- Angular CLI is designed to save time and effort as a developer

3. Install Angular CLI

The next step is to install Angular CLI. Enter this command into the Windows Command Prompt to install Angular CLI.

```
npm install -g @angular/cli
```

Verify the configured version after you've added all of the packages using ng –version.



What is NPM?

- npm is the world's largest Software Library (Registry)
- npm is also a software Package Manager and Installer
- The software package manager npm uses command line input to install software:

```
C:\>npm install mysoftware
```

4. Install npm

npm is installed with Node.js

This means that you have to install Node.js to get npm installed on your computer.

Download Node.js from the official Node.js web site: https://nodejs.org

NPM command and Package JSON

- The name npm (Node Package Manager) stems from when npm first was created as a package manager for Node.js.
- All npm packages are defined in files called package.json.
- The content of package.json must be written in JSON.
- At least two fields must be present in the definition file: name and version.

The Angular Framework, Angular CLI, and components used by Angular applications are packaged as npm packages and distributed using the npm registry.

You can download and install these npm packages by using the npm CLI client, which is installed with and runs as a Node.js® application. By default, the Angular CLI uses the npm client.

- package.json is a JSON file that lives in the root directory of your project. Your package.json holds important information about the project.
- It contains human-readable metadata about the project (like the project name and description) as well as functional metadata like the package version number and a list of dependencies required by the application.

```
{
  "name": "my-first-angular-app",
  "version": "0.0.0",
  "scripts": {
    "ng": "ng",
    "start": "ng serve",
    "build": "ng build",
    ....
  },
  "private": true,
  "dependencies": {
    "@angular/common": "~8.0.1",
    "@angular/core": "~8.0.1",
    "@angular/forms": "~8.0.1",
    "@angular/platform-browser": "~8.0.1",
    ....
  },
  "devDependencies": {
    "@angular-devkit/build-angular": "~0.800.0",
    "@angular/cli": "~8.0.3",
    "@angular/compiler-cli": "~8.0.1",
    ....
    "tslint": "~5.15.0",
    "typescript": "~3.4.3"
  }
}
```

Activity One

Follow the instructions:

1. Refer to the environment setup of angular given in the above modules. Download all the necessary software and install it to start with designing an angular application.
2. Make a note of downloading the recent versions of the software.

Bootstrapping Angular App

The process of loading the index.html page, app-level module, and app-level component is called bootstrapping, or loading the app. In this guide, you will learn about the internals of the bootstrapping process.

Angular takes the following steps to bootstrap the application:

- Load index.html
- Load Angular, Other Libraries, and App Code
- Execute main.ts File
- Load App-Level Module
- Load App-Level Component
- Process Template

Load index.html

The starting point of any Angular web application is the index.html page. Index.html is usually the first page to load. Let us open the file and find out what it contains. You will find it under the src folder.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>HelloWorld</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

There are no javascript files in the index.html. Neither you can see a stylesheet file. The body of the files has the following HTML tag.

```
<app-root></app-root>
```

Building Application

To run our application, we use the Angular CLI command ng serve or NPM command npm start (npm start command actually translates into ng serve.)

ng serve does build our application but does not save the compiled application to the disk. It saves it in memory and starts the development server.

We use ng build to build our app. Open the command prompt and run the command. This will build and copy the output files to the dist folder.

```
ng build
```

Now open the dist and open the index.html.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>GettingStarted</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>

  <script src="runtime-es2015.js" type="module"></script>
  <script src="runtime-es5.js" nomodule defer></script>
  <script src="polyfills-es5.js" nomodule defer></script>
  <script src="polyfills-es2015.js" type="module"></script>
  <script src="styles-es2015.js" type="module"></script>
  <script src="styles-es5.js" nomodule defer></script>
  <script src="vendor-es2015.js" type="module"></script>
  <script src="vendor-es5.js" nomodule defer></script>
  <script src="main-es2015.js" type="module"></script>
  <script src="main-es5.js" nomodule defer></script></body>
</html>
```

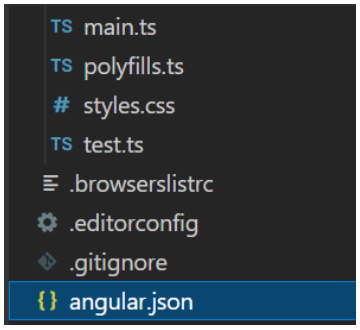
You can see that the compiler included five script files. They are runtime, polyfills, styles, vendor, & main. All these files have two versions one is es5 & the other one es2015.

Note:
Since the Angular 7, we have new feature called conditional polyfill loading. Now Angular builds two script files, one for es2015 & another for es5. The es2015 (es6) is for modern browser and es5 is older browsers, which do not support the new features of es2015.

- runtime.js: Webpack runtime file
- polyfills.js – Polyfill scripts for supporting the variety of the latest modern browsers
- styles.js – This file contains the global style rules bundled as javascript file.
- vendor.js – contains the scripts from the Angular core library and any other 3rd party library.
- main.js – code of the application.

Application Loads

When index.html is loaded, the Angular core libraries, third-party libraries are loaded. Now the angular needs to locate the entry point.

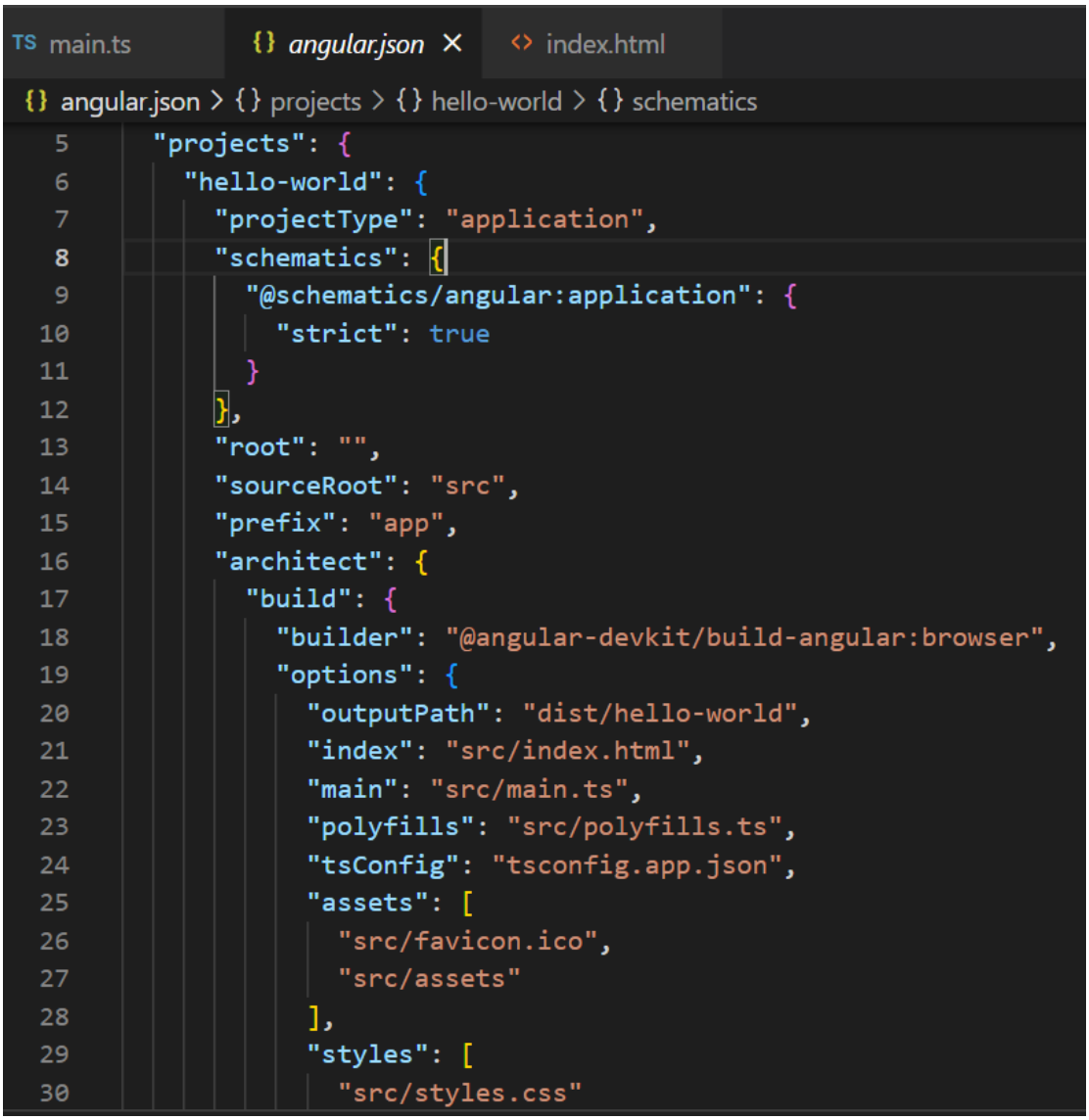


Application Entry point

The entry point of our application is main.ts. You will find it under the src folder.

angular.json

The Angular finds out the entry point from the configuration file angular.json. This file is located in the root folder of the project.



The main entry under the node projects -> GettingStarted -> architect -> build -> options points towards the src/main.ts. This file is the entry point of our application.

main.ts Application entry point

The main.ts file is as shown below.

```
src > TS main.ts > ...
1  import { enableProdMode } from '@angular/core';
2  import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
3
4  import { AppModule } from './app/app.module';
5  import { environment } from './environments/environment';
6
7  if (environment.production) {
8    enableProdMode();
9  }
10
11 platformBrowserDynamic().bootstrapModule(AppModule)
12   .catch(err => console.error(err));
13
```

Let us look at the relevant code in detail.

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

This line imports the module platformBrowserDynamic from the library@angular/platform-browser-dynamic.

What is platformBrowserDynamic

- platformBrowserDynamic is the module, which is responsible for loading the Angular application in the desktop browser.
- The Angular Applications can be bootstrapped in many ways and in many platforms. For example, we can load our application in a Desktop Browser or in a mobile device with Ionic or NativeScript.

```
import { AppModule } from './app/app.module';
```

The above line imports AppModule. The AppModule is the Root Module of the app. The Angular applications are organized as modules. Every application built in Angular must have at least one module. The module, which is loaded first when the application is loaded is called a root module.

```
platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.error(err));
```

The platformBrowserDynamic loads the root module by invoking the bootstrapModule and giving it the reference to our Root module i.e AppModule

Root Module

The angular bootstrapper loads our root module AppModule. The AppModule is located under the folder src/app. The code of our Root module is shown below.

src > app > TS app.module.ts > ...

```
1   import { NgModule } from '@angular/core';
2   import { FormsModule } from '@angular/forms';
3   import { BrowserModule } from '@angular/platform-browser';
4   import { AppRoutingModule } from './app-routing.module';
5   import { AppComponent } from './app.component';
6
7   @NgModule({
8     declarations: [
9       AppComponent
10    ],
11    imports: [
12      BrowserModule,
13      AppRoutingModule,
14      FormsModule
15    ],
16    providers: [],
17    bootstrap: [AppComponent]
18  })
19  export class AppModule { }
```

The root module must have at least one root component. The root component is loaded, when the module is loaded by the Angular.

In our example, AppComponent is our root component. Hence, we import it.

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

We use @NgModule class decorator to define an Angular Module and provide metadata about the Modules.

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

The @NgModule has several metadata properties.

imports

We need to list all the external modules required including other Angular modules, that is used by this Angular Module

Declarations

The Declarations array contains the list of components, directives, & pipes that belong to this Angular Module. We have only one component in our application AppComponent.

Providers

The Providers array, is where we register the services we create. The Angular Dependency injection framework injects these services in components, directives. pipes and other services.

Bootstrap

The component that angular should load, when this Angular Module loads. The component must be part of this module. We want AppComponent load when AppModule loads, hence we list it here.

The Angular reads the bootstrap metadata and loads the AppComponent

Component

Finally, we arrive at AppComponent, which is the root component of the AppModule. The code of our AppComponent is shown below.

src > app > TS app.component.ts > ...

```
1   import { Component } from '@angular/core';
2   import { FormsModule } from '@angular/forms';
3
4   @Component({
5     selector: 'app-root',
6     templateUrl: './app.component.html',
7     styleUrls: ['./app.component.css']
8   })
9   export class AppComponent {
10  }
```

The Class AppComponent is decorated with @Component Class Decorator.

The @Component class decorator provides the metadata about the class to the Angular. It has 3 properties in the above code. selector, templateURL & styleUrls

templateURL

This property contains an HTML template, which is going to be displayed in the browser. The template file is app.component.html

selector

This property specifies the CSS Selector, where our template will be inserted into the HTML. The CSS Selector in our code is app-root.

Template

The integral part of Angular component is Template. It is used to generate the HTML content.

The AppComponent defines the template as app.component.html and the CSS Selector is app-root.

Our index.html already have the app-root CSS selector defined.

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

The Angular locates app-root in our index.html and renders our template between those tags.

Activity Two

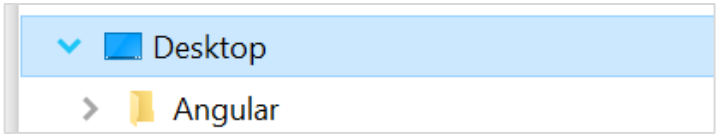
1. **Answer the following questions**
   - Explain the different metadata properties of @NgModule.
   - What do you mean by root module?
   - When does a component gets loaded?
2. **Open the Visual Studio Code and check all the files that are used to bootstrap an angular application.**

First Angular App & Directory Structure

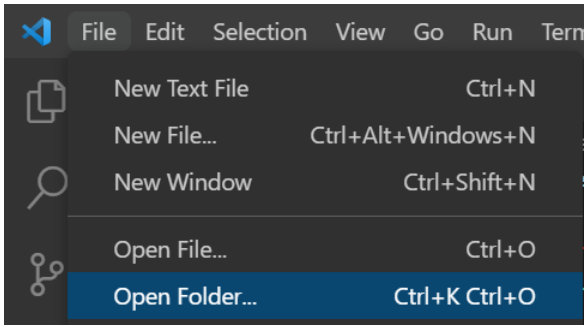
Project Setup

To create a new workspace and initial starter app:

1. Create a folder, Angular, on a desktop or your choice.



2. Open visual code, click on File, select Open Folder (ctrl+O) then click on it.



3. Start Visual Studio code app and open Terminal>New Terminal.

4. In Terminal, Type the CLI command ng new and provide the name my-app, as shown here:

5. The ng new command prompts you for information about features to include in the initial app. Accept the defaults by pressing the Enter or Return key.

```
? Would you like to add Angular routing? Yes
? Which stylesheet format would you like to use? (Use arrow keys)
> CSS
  SCSS   [ https://sass-lang.com/documentation/syntax#scss                ]
  Sass   [ https://sass-lang.com/documentation/syntax#the-indented-syntax ]
  Less   [ http://lesscss.org                                             ]
```

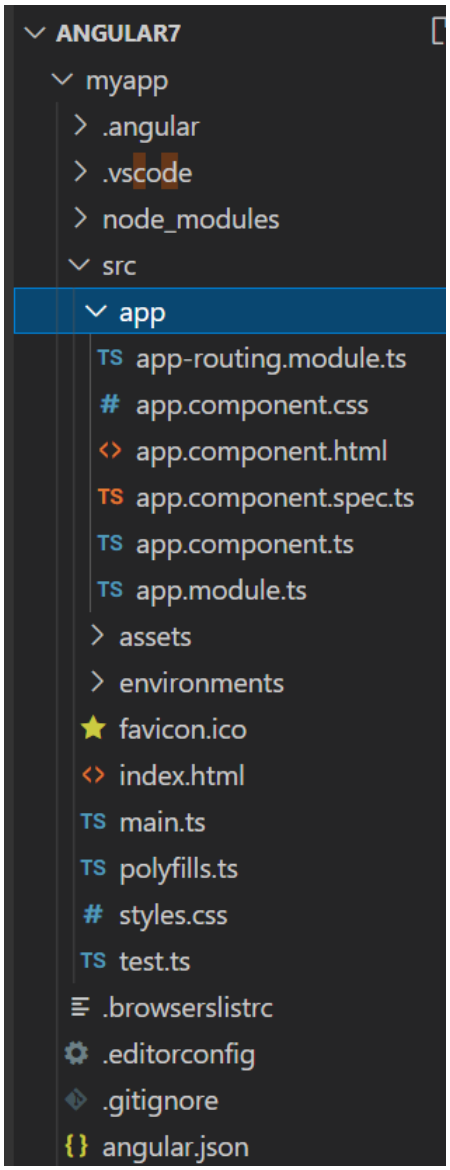
The Angular CLI installs the necessary Angular npm packages and other dependencies. This can take a few minutes.

The CLI creates a new workspace and a simple Welcome app, ready to run.

```
? Would you like to add Angular routing? Yes
? Which stylesheet format would you like to use? CSS
CREATE myapp/README.md (1059 bytes)
CREATE myapp/tsconfig.json (863 bytes)
CREATE myapp/.editorconfig (274 bytes)
CREATE myapp/.gitignore (548 bytes)
CREATE myapp/.browserslistrc (600 bytes)
CREATE myapp/karma.conf.js (1422 bytes)
CREATE myapp/tsconfig.app.json (287 bytes)
CREATE myapp/tsconfig.spec.json (333 bytes)
CREATE myapp/.vscode/extensions.json (130 bytes)
CREATE myapp/.vscode/launch.json (474 bytes)
CREATE myapp/.vscode/tasks.json (938 bytes)
CREATE myapp/src/favicon.ico (948 bytes)
CREATE myapp/src/index.html (291 bytes)
CREATE myapp/src/main.ts (372 bytes)
CREATE myapp/src/polyfills.ts (2338 bytes)
CREATE myapp/src/styles.css (80 bytes)
CREATE myapp/src/test.ts (749 bytes)
CREATE myapp/src/assets/.gitkeep (0 bytes)
CREATE myapp/src/environments/environment.prod.ts (51 bytes)
CREATE myapp/src/environments/environment.ts (658 bytes)
CREATE myapp/src/app/app-routing.module.ts (245 bytes)
CREATE myapp/src/app/app.module.ts (393 bytes)
CREATE myapp/src/app/app.component.html (23364 bytes)
CREATE myapp/src/app/app.component.spec.ts (1070 bytes)
CREATE myapp/src/app/app.component.ts (209 bytes)
CREATE myapp/src/app/app.component.css (0 bytes)
✓ Packages installed successfully.
'git' is not recognized as an internal or external command,
operable program or batch file.
```

When the command is run, Angular creates a skeleton application under the folder. It also includes a bunch of files and other important necessities for the application.

Introduction to directory structure:



- **node_modules** It saves all the dev dependencies (used only at development time) and dependencies (used for development as well as needed in production time), any new dependency when added to project it is automatically saved to this folder.
- **src** This directory contains all of our work related to project i.e. creating components, creating services, adding CSS to the respective page, etc.
- **package.json** This file stores the information about the libraries added and used in the project with their specified version installed. Whenever a new library is added to the project it's name and version is added to the dependencies in package.json

Other files: As a beginner you don't need these files at this time, don't bother about that. These all are used for editor configurations and information needed at compile time. The builtin webpack in angular CLI manages all for you.

Inside src folder:

- **index.html** This is the entry point for the application, app-root tag is the entry point of the application on this single page application, on this page angular will add or remove the content from the DOM or will add new content to the DOM. Base href="/" is important for routing purposes.
- **style.css** This file is the global stylesheet. you can add CSS classes or selectors which are common to many components, for example, you can import custom fonts, import bootstrap.css, etc.
- **assets** It contains the js images, fonts, icons and many other files for your project.

Inside app folder:

- **app.module.ts:** This is the main module file for the project. Each Angular project is divided into modules to make the project management easy.
- **app.component.html** This file is used to make changes to the page. You can edit this file as an HTML file.
- **app.component.spec.ts** These are automatically generated files which contain unit tests for source component.
- **app.component.ts** You can do the processing of the HTML structure in the .ts file. The processing will include activities such as connecting to the database, interacting with other components, routing, services, etc.
- **app.component.css** Here you can add CSS for your component.

Creating the Angular Application

Let us create an application where, you can add the name, age, email id, password of the employee to register for the company.

Step 1: In Terminal, Type the CLI command ng new and provide the name my-app, as shown here:

```
\Desktop\Angular7> ng new myapp
```

Step 2: To run the application, change the directory to the folder created:

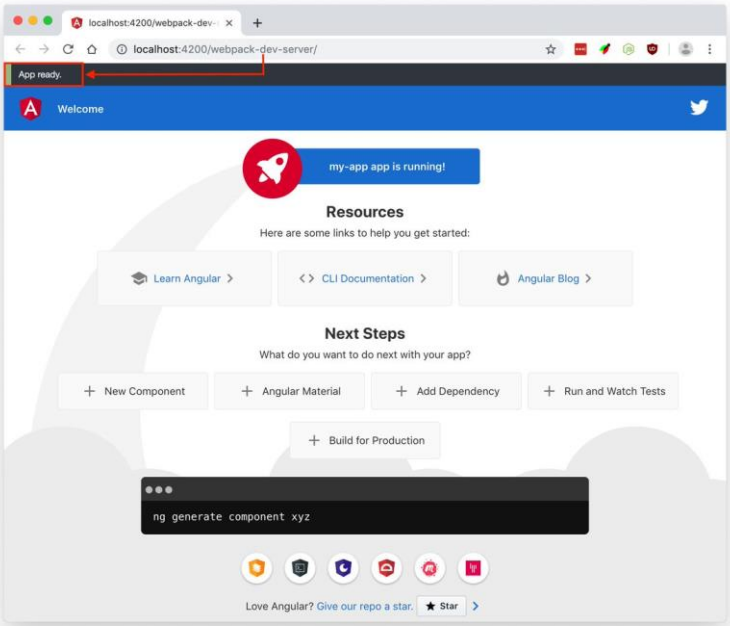
```
\Desktop\Angular7> cd myapp
\Desktop\Angular7\myapp> ng serve
```

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

In the terminal, you can see that the application is compiled successfully.

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

styles.css, styles.js | styles        | 207.34 kB |
main.js               | main          |  50.44 kB |
runtime.js            | runtime       |   6.51 kB |

                      | Initial Total |   2.58 MB

Build at: 2022-06-14T14:30:33.855Z - Hash: ac8a6beaa24e49d0 - Time: 31339ms

** Angular Live Development Server is listening on localhost:4200, open your browser on http://localhost:4
200/ **


✓ Compiled successfully.
✓ Browser application bundle generation complete.

5 unchanged chunks

Build at: 2022-06-14T15:02:54.196Z - Hash: ac8a6beaa24e49d0 - Time: 1714ms
```

Step 3: Open your browser on http://localhost:4200/. You can see the output as shown below.



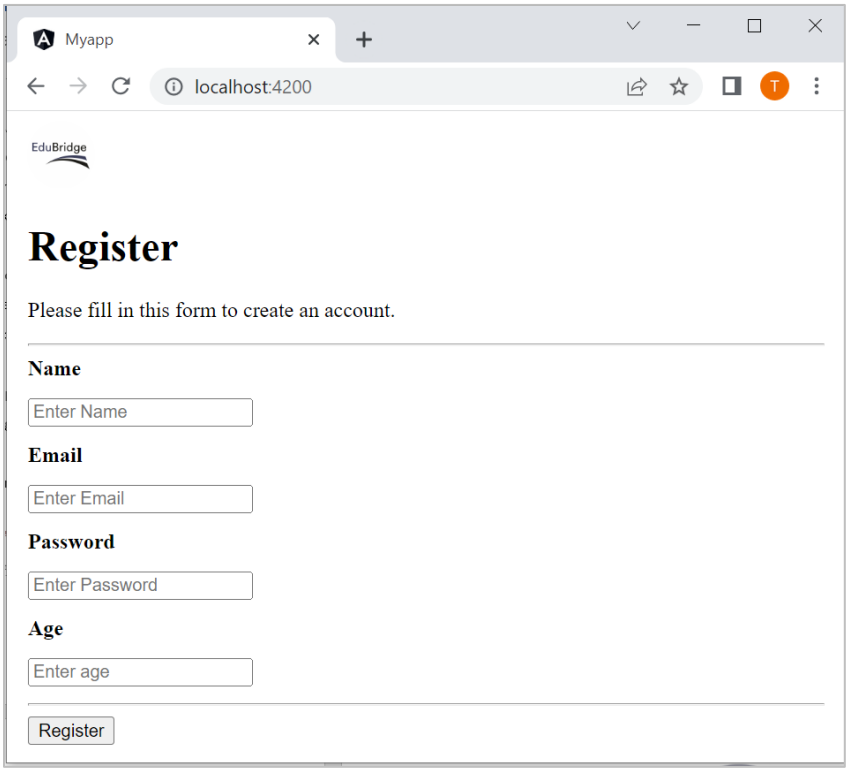
Step 4: Now let us change the coding to create our registration form that displays the text and textboxes. Open app.component.html file that is used to make changes to the page.

Code:

```
<html>
  <body>
  <form action="/action_page.php">
    <div class="container">
      <img class="img-responsive" width="50" alt="Edubridge Logo"
src="assets/Images/Logo.png">
      <h1>Register</h1>
      <p>Please fill in this form to create an account.</p>
      <hr>
      <label for="name"><b>Name</b></label><pre></pre>
      <input type="text" placeholder="Enter Name" name="name" id="name"
required><pre></pre>

      <label for="email"><b>Email</b></label><pre></pre>
      <input type="text" placeholder="Enter Email" name="email" id="email"
required><pre></pre>

      <label for="psw"><b>Password</b></label><pre></pre>
      <input type="password" placeholder="Enter Password" name="psw" id="psw"
required><pre></pre>

      <label for="age"><b>Age</b></label><pre></pre>
      <input type="text" placeholder="Enter age" name="age" id="age"
required><pre></pre>
      <hr>
      <button type="submit" class="registerbtn">Register</button>
    </div>
  </form>
```

Step 5: To save the changes in the code, click File > Save. Now, open browser to see the application.

Output:



As you can see the application looks structured but not styled. We can style Angular applications with standard CSS. That means you can apply everything you know about CSS stylesheets, selectors, rules, and media queries directly to Angular applications.

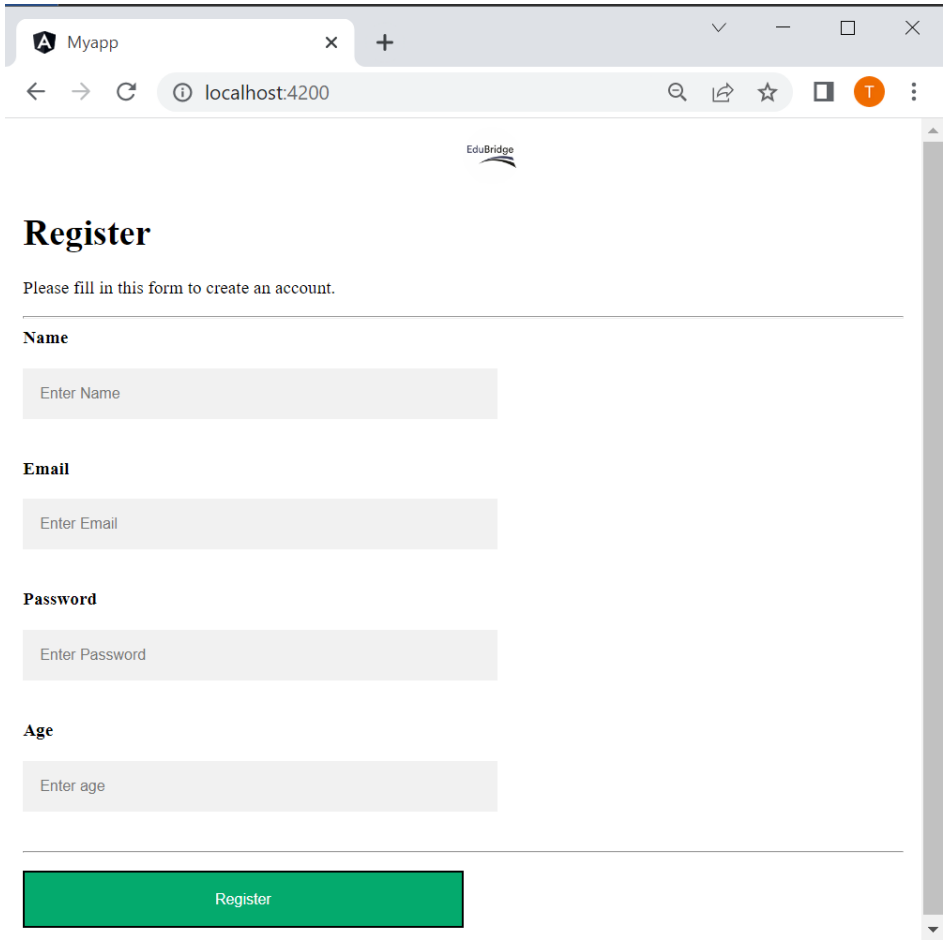
Step 6: Open app.component.css file and add different elements and properties to style the application.

Code:

```
input[type=text], input[type=password] {
  width: 50%;
  padding: 15px;
  margin: 5px 0 22px 0;
  display: inline-block;
  border: none;
  background: #f1f1f1;
}
.img-responsive{
    margin-left: 50%;
```

```
}
/* Set a style for the submit button */
.registerbtn {
  background-color: #04AA6D;
  color: white;
  padding: 16px 20px;
  margin: 8px 0;
  cursor: pointer;
  width: 50%;
}
```

Output:



Activity Three

Follow the instructions and create your first angular project

1. Open the Visual Studio code.
2. Create an angular application to display the school registration from.

Angular Fundamentals

The architecture of an Angular application relies on certain fundamental concepts. The basic building blocks of the Angular framework are Angular components that are organized into NgModules. NgModules collect related code into functional sets; an Angular application is defined by a set of NgModules. An application always has at least a root module that enables bootstrapping, and typically has many more feature modules.

- Components define views, which are sets of screen elements that Angular can choose among and modify according to your program logic and data
- Components use services, which provide specific functionality not directly related to views. Service providers can be injected into components as dependencies, making your code modular, reusable, and efficient.

Modules, components and services are classes that use decorators. These decorators mark their type and provide metadata that tells Angular how to use them.

- The metadata for a component class associates it with a template that defines a view. A template combines ordinary HTML with Angular directives and binding markup that allow Angular to modify the HTML before rendering it for display.
- The metadata for a service class provides the information Angular needs to make it available to components through dependency injection (DI)

An application's components typically define many views, arranged hierarchically. Angular provides the Router service to help you define navigation paths among views. The router provides sophisticated in-browser navigational capabilities.

Metadata

- Metadata tells Angular how to process a class. To tell Angular that Course Component is a component, metadata is attached to the class. In TypeScript, you attach metadata by using a decorator. In the below code, you can see metadata attached to the Course Component:

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

- Here is the @Component decorator, which identifies the class immediately below it as a component class. The @Component decorator takes the required configuration object which Angular needs to create and present the component and its view.

- The most important configurations of @Component decorator are:
  - selector: Selector tells Angular to create and insert an instance of this component where it finds <app-course> tag. For example, if an app's HTML contains <app-course></app-course>, then Angular inserts an instance of the CourseComponent view between those tags.
  - templateUrl: It contains the path of this component's HTML template.
  - styleUrls: It is the component-specific style sheet.
- The metadata in the @Component tells Angular where to get the major building blocks you specify for the component. The template, metadata, and component together describe a view. The architectural takeaway is that you must add metadata to your code so that Angular knows what to do.

Module 2: Essentials of Angular

Module Overview

In this module, students will be able to learn about frontend technologies such as HTML5, CSS3, Bootstrap, JavaScript and TypeScript.

Module Objective

At the end of this module, students should be able to demonstrate appropriate knowledge, and show an understanding of the following:

Component Basics

- Setting up the templates

- Creating Components using CLI

- Nesting Components

- Data Binding - Property & Event Binding, String Interpolation, Style binding

- Two-way data binding

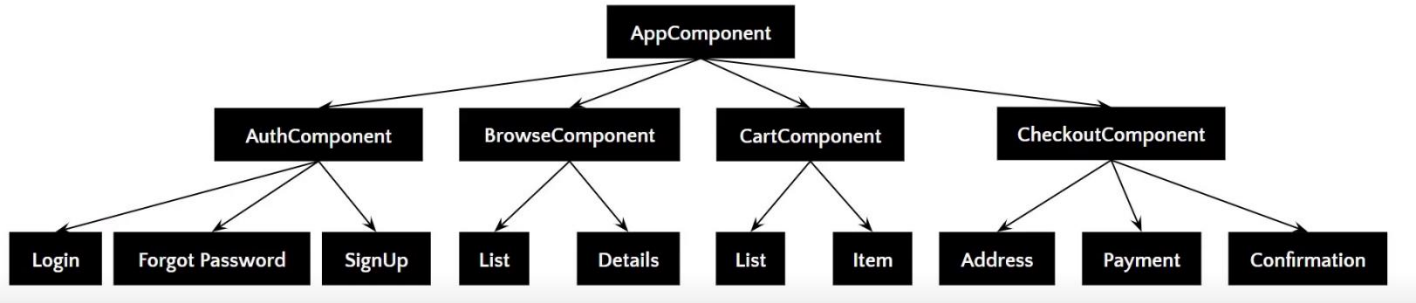
- Input Properties, Output Properties, Passing Event Data

Component Basics

Components are the main building block for Angular applications. Each component consists of:

- An HTML template that declares what renders on the page
- A TypeScript class that defines behavior
- A CSS selector that defines how the component is used in a template
- Optionally, CSS styles applied to the template

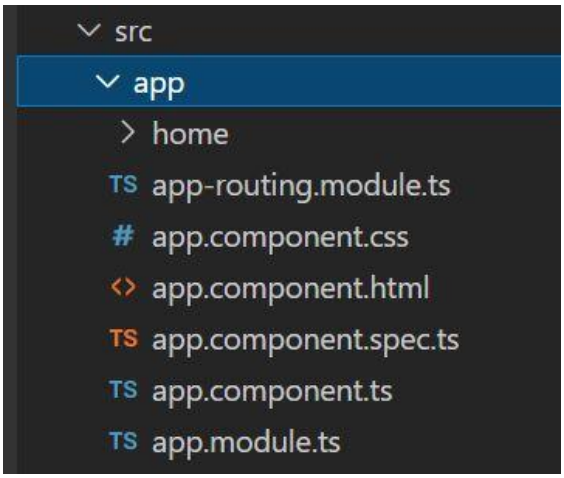
These components are associated with a template and are a subset of directives.



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

Every Angular application has at least one component, the *root component* that connects a component hierarchy with the page document object model (DOM). Each component defines a class that contains application data and logic, and is associated with an HTML *template* that defines a view to be displayed in a target environment.

Let's have a look at each of the files Inside the src/app folder and understand their role in the Angular project.



In the above screenshot, you can see that there are a total of 6 files as of now inside the src/app folder.

app.module.ts: This is the main module file for the project. Each Angular project is divided into modules to make the project management easy. By default, the whole project is a part of a single module but you can always create more modules. A module is a logical chunk of project resources that works independently or in combination with other modules. This file contains a module called AppModule.

- . If you want to add more modules at the root level, you can add.
  - **declarations** It is the reference of the array to store its components. The app component is the default component that is generated when a project is created. You have to add all your component's reference to this array to make them available in the project.
  - **imports** If you want to add any module whether angular or you have to add it to imports array to make them available in the whole project.
  - **providers** If you will create any service for your application then you will inject it into your project through this provider array. Service injected to a module is available to it and it's child module in the project hierarchy.
  - **bootstrap** This has reference to the default component created, i.e., AppComponent

app.component.html: This file is the template file for the one and only component in our project as of now. This component is called the AppComponent. The extension html indicated that this file is an HTML document and will define the view of the component.

app.component.css: This file work in combination with the HTML template and adds CSS styles to the HTML elements defined in that template. This only only contains CSS classes and styles.

app.component.ts: This is the main class file for the AppComponent component. This file contains the TypeScript class for the component and all the business logic for this component goes inside this file.

app.component.spec.ts: This file is for unit testing the AppComponent component. Let's not worry about this for now. Feel free to delete the file if it bothers you too much.

app-routing.module.ts: This is another module file for the project (just like the app.module.ts that contains the module AppModule) that handles routing or navigation for the whole project. It defines how different components will be presented to the user as the user navigates within the application.

Component Lifecycle

Components are core building blocks of Angular applications. Each component is instantiated when it is required, it does its job and then gets destroyed to free us the memory. Component instances have a lifecycle as Angular creates, updates, and destroys them. Developers can tap into key moments in that lifecycle by implementing one or more of the lifecycle hook interfaces in the Angular core library.

Lifecycle hooks allow us to execute our code when the components reach a specific stage in their lifecycle.

For example, we may want to make a call to an external API as soon as a component finished loading or we may want to notify a web-service when a component is destroyed. We can do this with the help of lifecycle hooks.

Here is some of the important lifecycle hooks that Angular provides for the components in the same sequence in which they are called after the constructor of the component class is executed.

ngOnChanges(): called when Angular (re)sets the data bound properties.

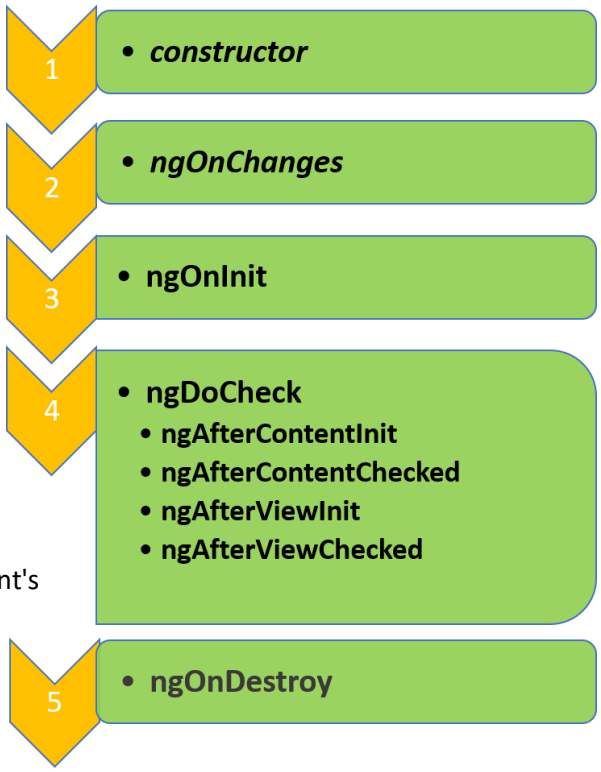
ngOnInit(): called after Angular first displays the data-bound properties and sets the directive/component's input properties. It is generally use to perform complex initializations shortly after construction and to set up the component after Angular sets the input properties.

ngAfterContentInit(): called after Angular projects external content into the component's view / the view that a directive is in.

ngAfterContentChecked(): called after Angular checks the content projected into the directive/component.

ngAfterViewInit(): called after after Angular initializes the component's views and child views.

ngOnDestroy(): called just before Angular destroys the component. This is the time to notify another part of the application that the component is going away.



Classes, Constructors, Variables and Types

Open the file src/app/app/component.ts and there is the code for the TypeScript class responsible for the AppComponent.

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

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

In the above code, you can see that we have a class called AppComponent that has a member variable defined inside it. This class does not have a constructor defined, but we can always create a constructor manually as shown below.

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

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

The code inside the constructor is executed as soon as the class is initialized.

At line#9, a variable is declared named title and is initialized with the value 'Test' which is of the type string. Optionally, we can also specify the type explicitly.

```
title: string = 'Test';
```

Setting up templates

A template is a form of HTML that tells Angular how to render the component. Templates in Angular represent a view whose role is to display data and change the data whenever an event occurs. It's default language for templates is HTML.

A template looks like regular HTML, except that it also contains Angular template syntax, which alters the HTML based on your application's logic and the state of application and DOM data. Your template can use data binding to coordinate the application and DOM data, pipes to transform data before it is displayed, and directives to apply application logic to what gets displayed.

Types of Templates

There are two ways of defining a template in an angular component.

1. **Inline Template**

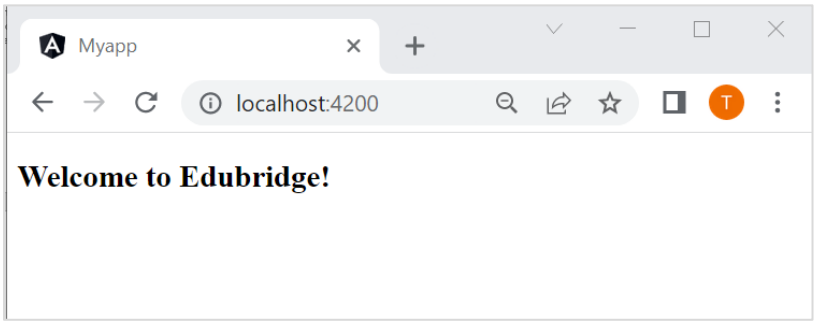
The inline template is defined by placing the HTML code in backticks ` and is linked to the component metadata using the template property of @Component decorator.

Code:

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

@Component({
  selector: 'app-root',
  template: `
  <h2>Welcome to Edubridge!</h2>
  `,
```

```
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'myapp';
  constructor(){}
}
```

Output:



2. External Template

The template is defined in a separate HTML file and is linked to the component metadata using the *@Component* decorator's templateUrl property.

expense-manager.component.html

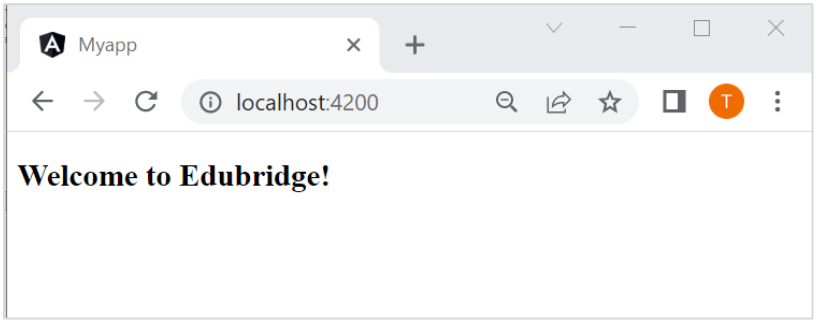
```
<html>
    <h2>Welcome to Edubridge!</h2>
</html>
```

expense-manager.component.ts

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

@Component({
  selector: 'app-expense-manager',
  templateUrl: './expense-manager.component.html',
  styleUrls: ['./expense-manager.component.css']
})
export class ExpenseManagerComponent implements OnInit {
```

```
  constructor() { }

  ngOnInit(): void {
  }
}
```

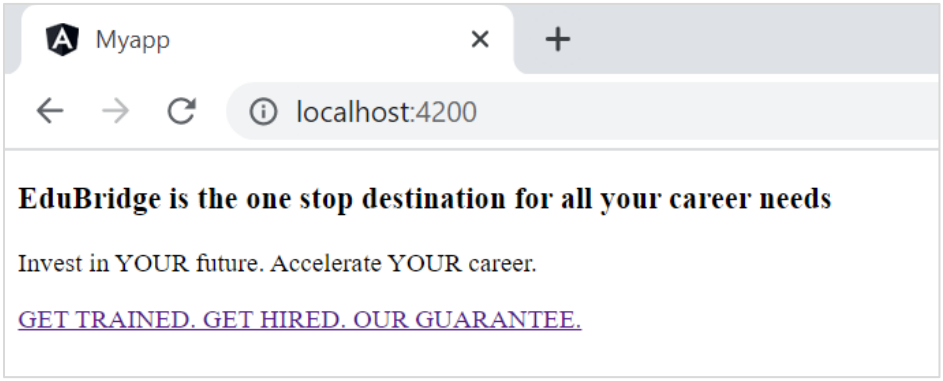
Output:



Activity Four

Follow the instructions:

1. Open the Visual Studio code.
2. Use both the ways to define template in angular and display the following output.



Creating Component using CLI

Let's assume we want to create an Angular component named home.

You can create a component by using **ng g c <component_name>**, or using **ng generate component <component_name>**. Using either of these two commands, the new component can be generated pretty easily and followed by the suitable component name of your choice.

Open a new command-line interface, navigate to the root of your Angular project and run the following command:

```
ng g c home
```

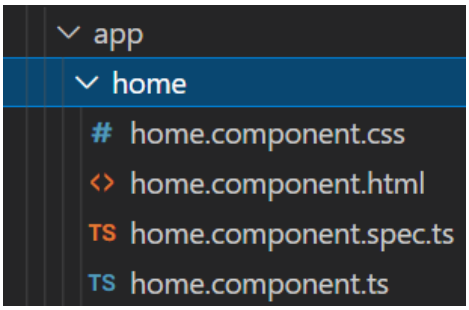
Or you can also run the following command:

```
ng generate command home
```

Angular CLI will generate 4 files for the component in the src/app folder of your project:

```
? Would you like to add Angular routing? Yes
? Which stylesheet format would you like to use? (Use arrow keys)
> CSS
  SCSS   [ https://sass-lang.com/documentation/syntax#scss                ]
  Sass   [ https://sass-lang.com/documentation/syntax#the-indented-syntax ]
  Less   [ http://lesscss.org                                             ]
```

Angular CLI will generate 4 files for the component in the src/app folder of your project:



- home.component.ts (Class with @Component decorator)
- home.component.spec.ts (For test case specification)
- home.component.html (For the template)

- home.component.css (For stylesheets)

Note:
We can also customize where the component's files are placed. Angular CLI adds the component to the declarations array of the module.

```
@NgModule({
 // [...]
 declarations: [ AppComponent, HomeComponent ]
 // [...]
})
export class AppModule { }
```

This will allow HomeComponent in any component in the AppModule.

Activity Five

Follow the instructions:

1. Open the Visual Studio code.
2. Create two components namely Home, About, Courses, Contact.
3. In the home component, create a code to display the key pointers about Edubridge with an image.
4. In the About Page, add information about Edubridge.
5. In services, add content of different courses available.
6. In contact, add a form where you can add Name, Email id and Message and a submit button.

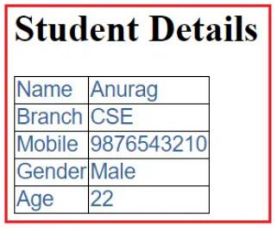
Nesting Components

What are Angular Nested Components?

The Angular framework allows us to use a component within another component and when we do so then it is called Angular Nested Components. The outside component is called the parent component and the inner component is called the child component.

Understanding Nested Components in Angular Application:

Let us discuss how to create and use a nested component in the angular application with one example. Assume that you have a component called StudentComponent and you want to use this Student Component inside the AppComponent which is our root component. Our requirement is that we need to display the student details as shown in the below image:

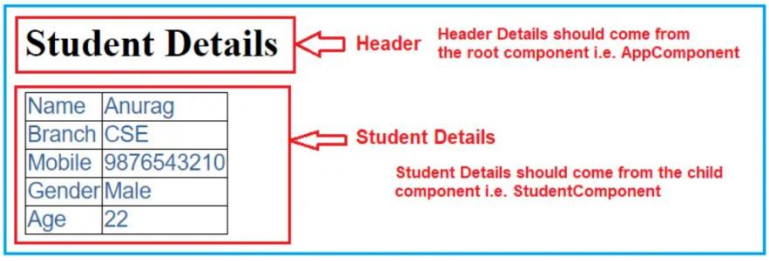


As per our requirement, we are going to create two angular components

AppComponent: The AppComponent is used to display the page header. This is our parent component.

StudentComponent: The StudentComponent is used to display the Student details. This is our child component. As this is the child component, it should be nested inside the AppComponent i.e. parent component.

The following diagram gives you a clear idea about the above two points.

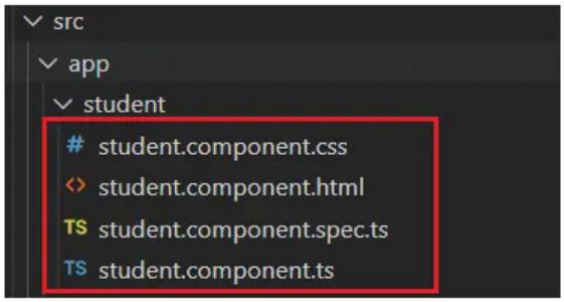


Let's see how to achieve the above requirement step by step using nested components.

Step 1: Creating student component

Open the Visual Studio Code terminal and type **ng g c student** and press the enter button as shown in the below image.

Once you type the above command and press the enter button, then it will create a folder called student within the app folder with four files as shown in the below image.



Step 2: Modifying Student.component.html file:

This file is going to contain the HTML for the student component. So, open the **student.component.html** file and then copy and paste the following HTML code in it. In the following code, we are using data binding expression (i.e. **{{}}**) to bind the data.

```
<table>
    <tr>
        <td>Name</td>
        <td>{{Name}}</td>
    </tr>
    <tr>
        <td>Branch</td>
        <td>{{Branch}}</td>
    </tr>
    <tr>
        <td>Mobile</td>
        <td>{{Mobile}}</td>
    </tr>
    <tr>
        <td>Gender</td>
        <td>{{Gender}}</td>
    </tr>
    <tr>
        <td>Age</td>
        <td>{{Age}}</td>
    </tr>
</table>
```

Step 3: Modifying the student.component.ts file:

Within the student folder, you can find a file with the name **student.component.ts** within which the **StudentComponent** is created. So, open this file and then copy and paste the following code in it. This is going to be our child component. As said earlier, this component is going to display the student details, so here, we created five variables such as Name, Branch, Mobile, Gender and Age to store the student details as well as we also assigned these variables with some default values.

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

@Component({
  selector: 'app-student',
  templateUrl: './student.component.html',
  styleUrls: ['./student.component.css']
})
export class StudentComponent {

    Name: string = 'Anurag';
    Branch: string = 'CSE';
    Mobile: number = 9876543210;
    Gender: string = 'Male';
    Age: number = 22;

}
```

Step 4: Modifying the App.Component.ts file

Now, open **app.component.ts** file and then copy and paste the following code in it. This file has the **AppComponent** and this is also the root component in the angular application and in our example, it is also going to be our parent component. As said earlier, the parent component is going to display the header details.

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
    template: `<div>
                    <h1>{{pageHeader}}</h1>
                </div>`
})
export class AppComponent {
    pageHeader: string = 'Student Details';
}
```

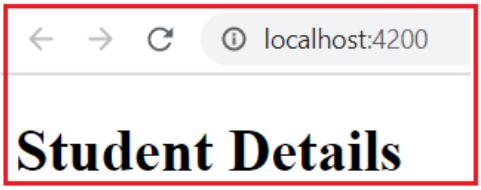
As you can see in the above code, here we are using the inline template to display the page header as it contains only three lines of code. But if you want then you can also use an external template by using the templateUrl property of @Component decorator.

Running Angular Application:

In order to compile and run the angular application with your default browser, type **ng serve -o** and press the enter key as shown below.

```
PS D:\AngularProjects\MyAngularApp> ng serve -o
```

Once you type the above command, it will compile and run your application using your default browser. If you observe the output, you are only getting the header, not the student details as shown in the below image.



This is because we have created the child component i.e. the StudentComponent but not yet used it within the parent component i.e. within the AppComponent. So, in order to display the Student details, you need to nest the StudentComponent inside the AppComponent.

How to nest a component inside another component in angular?

In order to nest a component inside another component, you need to follow the below two steps.

Step1: Open the "**app.module.ts**" file which is present inside the **app** folder, and then do the following:

First, you need to import the StudentComponent and then you need to add the StudentComponent to the declarations array as shown in the below image:

```
import { NgModule } from '@angular/core';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { StudentComponent } from './student/student.component';

@NgModule({
  declarations: [
    AppComponent,
    StudentComponent
  ],

  imports: [
    BrowserModule,
    AppRoutingModule
  ],

  providers: [],
  bootstrap: [AppComponent]
})

export class AppModule { }
```

Step 3: Open your parent component i.e. "**app.component.ts**" file and then include "**app-student**" as a directive as shown in the below image. If you remember, the "**app-student**" that we used here is nothing but a selector of **StudentComponent** i.e. our child component.

app.component.ts

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
    template: `<div>
                <h1>{{pageHeader}}</h1>
                <app-student></app-student>
              </div>`
})
export class AppComponent {
    pageHeader: string = 'Student Details';
}
```

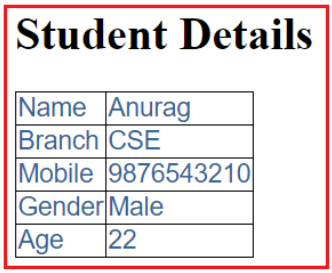
student.component.ts

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

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

In order to style the table, please add the following styles in the **styles.css** file which is present inside the src folder.

```
table {
color: #369;
font-family: Arial, Helvetica, sans-serif;
font-size:large;
border-collapse: collapse;
}
td {
border: 1px solid black;
}
```

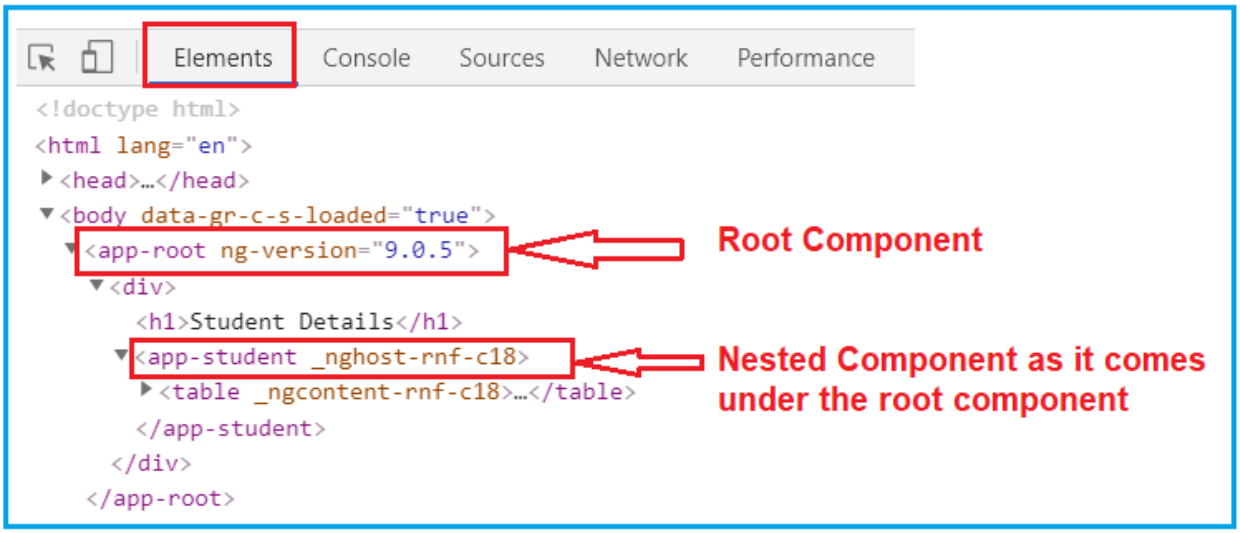
With the above changes in place, now you should see the student details along with the header as expected as shown in the below image.



Student Details

Name	Anurag
Branch	CSE
Mobile	9876543210
Gender	Male
Age	22

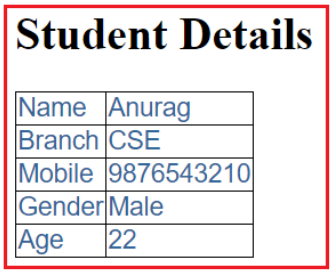
At this point, launch the browser developer's tools and click on the "**Elements**" tab and notice **<app-root>** and **<app-student>** directives are rendered as HTML tag as shown in the below image. Here, you can see the **app-student** render inside the **app-root** tag.



Activity Six

Follow the instructions:

1. Open the Visual Studio code.
2. Follow the steps given above for nested components and display the student information as shown.



Student Details

Name	Anurag
Branch	CSE
Mobile	9876543210
Gender	Male
Age	22

Data Binding

What is Angular Data Binding

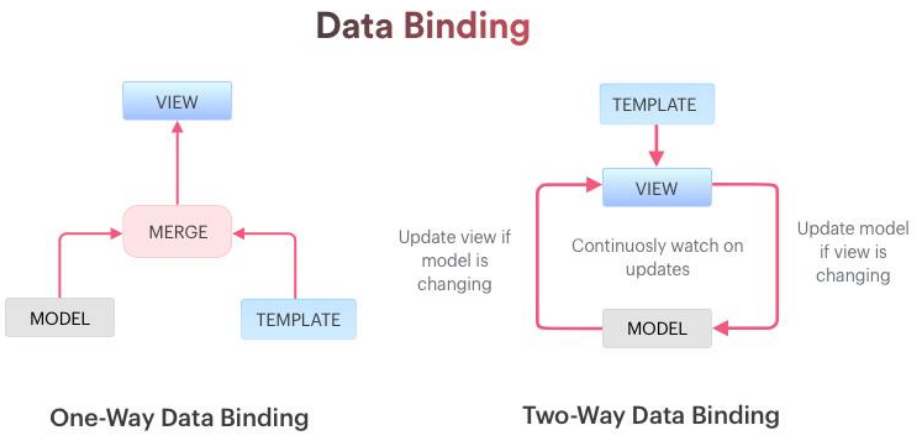
Data Binding is the mechanism that binds the applications UI or User Interface to the models. Using Data Binding, the user will be able to manipulate the elements present on the website using the browser. Therefore, whenever some variable has been changed, that particular change must be reflected in the Document Object Model or the DOM.
In Angular, Data Binding defines the interaction between the components and the DOM. Data Binding is a part of all Angular versions starting from AngularJS right through to the latest Angular 9 version.

Types of Data Binding in Angular

Angular allows both One-way and Two-way Data Binding. One-way data binding is a simple type of data binding where you are allowed to manipulate the views through the models. This implies, making changes to the Typescript code will be reflected in the corresponding HTML. In Angular, One-way data binding is achieved through:

- Interpolation or String Interpolation
- Property binding
- Event binding

Two-way data binding, on the other hand, allows synchronization of data in such a way that the views can be updated using the models and the models can be updated using views. This means that your application will be able to share information between a component class and its template.



One-way Data Binding

In one-way data binding, data flows only in one direction i.e from the models to the views. As mentioned earlier, one-way data binding in Angular can be of three types i.e Interpolation, Property binding, and Event binding.

- **Interpolation Binding**

Interpolation binding is used to return HTML output from TypeScript code i.e. from the components to the views. Here, the template expression is specified within double curly braces. Through Interpolation, strings can be added into the text that is present between HTML element tags and within attribute assignments. These strings are calculated using Template expressions.

Example:

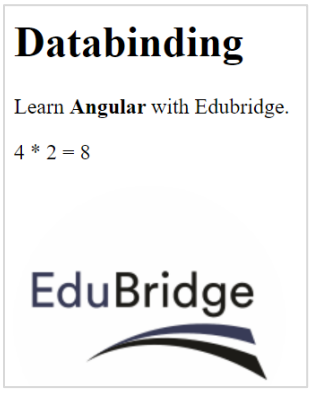
app.component.html

```
<h1>{{title}}</h1>
Learn <b> {{course}}
</b> with me.

4 * 2 = {{4 * 2}}
<div><img src="{{image}}"></div>
```

app.component.ts

```
export class AppComponent {
  title = 'Databinding';
  course ='Angular';
  image = 'paste the url here'
  constructor(){}
}
```

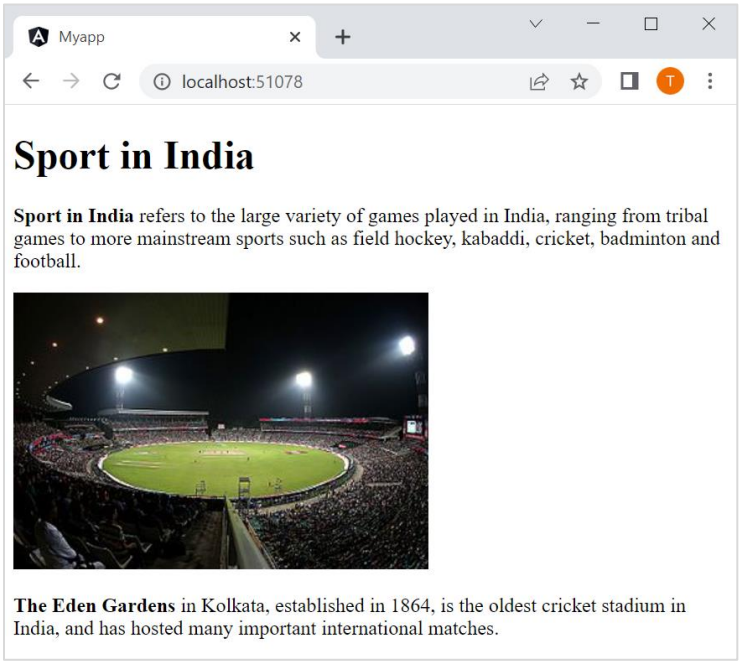
Output:



Activity Seven

Follow the instructions:

1. Open the Visual Studio code.
2. Create an angular application by using interpolation binding as shown below.



Template expressions

Template expressions are present within the two curly braces and they produce a value. Angular will execute that expression and then, it assigns that particular expression to a property of a binding target such as HTML elements, components, or directives.

NOTE: 2 * 2 present between interpolation brackets, is a template expression.

Property Binding

In Property binding, value flows from a component's property into the target elements property. Therefore, Property binding can't be used to read or pull data from the target elements or to call a method that belongs to the target

element. The events raised by the element can be acknowledged through event binding which will be covered later on in this article.

In general, one can say that the component property value will be set to the element property using Property binding.

```
<h1>Property binding</h1>

<div><img [src]="image"></div>
```

In the above example, the ***src*** property of the image element is bound to a component's *image* property.

Property binding and Interpolation

If you have noticed, you can see that interpolation and property binding can be used interchangeably. Take a look at the example pair given below:

```
<h2>Interpolation</h2>
<div><img src="{{image}}"></div>
<h2>Property binding</h2>
<div><img [src]="image"></div>
```

Please make a note that when you need to set element properties to non-string data values, you must use Property binding and not Interpolation.

Event Binding

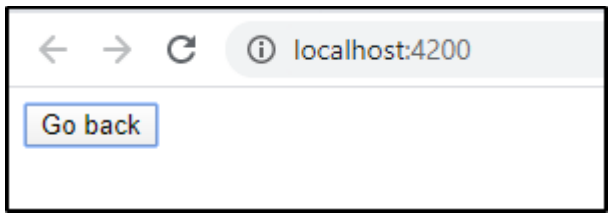
The Event binding feature lets you listen to certain events such as mouse movements, keystrokes, clicks, etc. In Angular, event binding can be achieved by specifying the target event name within regular brackets on the left of an equal to (=) sign, and the template statement on the right side within quotes (" ").

Code:

```
<div>
    <button (click)="goBack()">Go back</button>
</div>
```

The '*click*' in the above example is the target events name and '*goBack()*' is the template statement.

Output:



Whenever event binding occurs, an event handler will be set by Angular for the target event. When that particular event gets raised, the template statement is executed by the handler. Generally, receivers are involved with template statements that perform actions in response to the event. Here, binding is used to convey information about the event. These data values of the information include event string, object, etc.

Two-way Data Binding

Angular allows two-way data binding that will allow your application to share data in two directions i.e. from the components to the templates and vice versa. This makes sure that the models and the views present in your application are always synchronized. Two-way data binding will perform two things i.e. setting of the element property and listening to the element change events.

The syntax of two way binding is – [( )]. As you can see, it is a combination of the property binding syntax i.e. [ ] and the event binding syntax ( ). According to Angular, this syntax resembles “Banana in a Box”.

Code:

```
<label ><b>Name:</b>
    <input [(ngModel)]="course.name" placeholder="name"/>
</label>
```

When you execute this code, you will see that changes to either the models or the views will result in changes to the corresponding views and models. Take a look at the image below that shows the name of the course being changed from ‘Python’ to ‘Pytho’ from the views:

Example:
A single property userName in the component.

app.component.ts

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

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

```
export class AppComponent {
  title = 'myapp';
  userName = "Rita";
  constructor(){}
}
```

app.component.html

```
<div class="container">
    <div class="row">
      <div class="col-sm-6">
        <div class="form-group">
            <label for="username">User Name:</label>
            <input class="form-control" [(ngModel)]="userName">
            <p>{{ userName }}</p>
        </div>
      </div>
    </div>
  </div>
```

In the template as you can see target of two-way binding is ngModel directive and the expression is the property “userName” property of the component.

Import FormModule from @angular/forms in app.module.ts

```
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent,
  ],
  imports: [
    AppRoutingModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
```

```
export class AppModule { }
```

On running the example, you get the output as-

Activity Eight

Follow the instructions:

1. Open the Visual Studio code.
2. Follow the steps given above and create an angular application using Two-way binding.

Input Properties, Output Properties

@Input() and **@Output()** allow Angular to share data between the parent context and child directives or components. An **@Input()** property is writable while an **@Output()** property is observable.

Consider this example of a child/parent relationship:

```
<parent-component>
<child-component></child-component>
</parent-component>
```

Here, the <child-component> selector, or child directive, is embedded within a <parent-component>, which serves as the child's context.

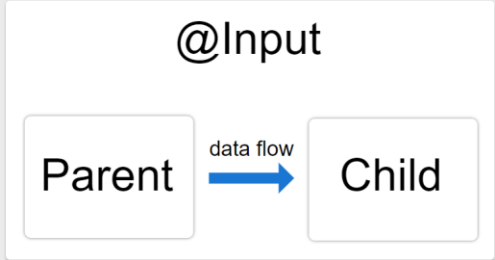
@Input() and @Output() act as the API, or application programming interface, of the child component in that they allow the child to communicate with the parent. Think of @Input() and @Output() like ports or doorways—@Input() is the

doorway into the component allowing data to flow in while @Output() is the doorway out of the component, allowing the child component to send data out.

Note:
@Input() and @Output() are independent
Though @Input() and @Output() often appear together in apps, you can use them separately. If the nested component is such that it only needs to send data to its parent, you wouldn't need an @Input(), only an @Output(). The reverse is also true in that if the child only needs to receive data from the parent, you'd only need @Input().

How to use @Input()?

Use the @Input() decorator in a child component or directive to let Angular know that a property in that component can receive its value from its parent component. It helps to remember that the data flow is from the perspective of the child component. So, an @Input() allows data to be input *into* the child component from the parent component.



To illustrate the use of @Input(), edit these parts of your app:

- The child component class and template
- The parent component class and template

In the child

To use the @Input() decorator in a child component class, first import Input and then decorate the property with @Input():

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

export class ItemDetailComponent {
  @Input() item: string; // decorate the property with @Input()
}
```

In this case, @Input() decorates the property item, which has a type of string, however, @Input() properties can have any type, such as number, string, boolean, or object. The value for item will come from the parent component.

Next, in the child component template, add the following:

```
<p>
    Today's item: {{item}}
</p>
```

In the Parent

The next step is to bind the property in the parent component's template. In this example, the parent component template is app.component.html.

First, use the child's selector, here <app-item-detail>, as a directive within the parent component template. Then, use property binding to bind the property in the child to the property of the parent.

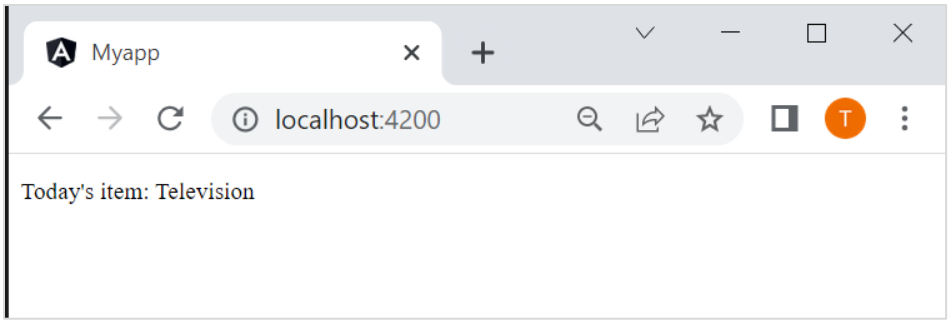
```
<app-item-detail [item]="currentItem"></app-item-detail>
```

Next, in the parent component class, app.component.ts, designate a value for currentItem:

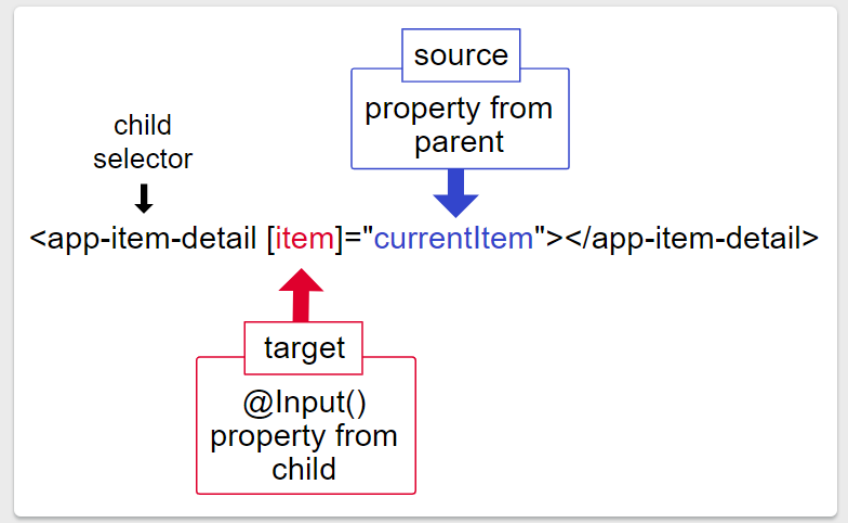
```
export class AppComponent {
  currentItem = 'Television';
}
```

With @Input(), Angular passes the value for currentItem to the child so that item renders as Television.

Output:



The following diagram shows this structure:



The target in the square brackets, [], is the property you decorate with @Input() in the child component. The binding source, the part to the right of the equal sign, is the data that the parent component passes to the nested component.

The key takeaway is that when binding to a child component's property in a parent component—that is, what's in square brackets—you must decorate the property with @Input() in the child component.

Activity Nine

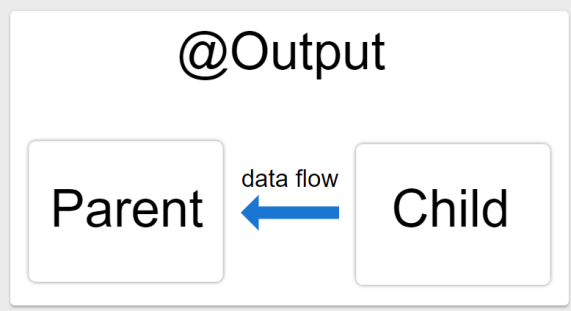
Follow the instructions:

1. Open the Visual Studio code.
2. Follow the step by step instruction given above and create an angular application using @Input property.

How to use @Output()?

Use the @Output() decorator in the child component or directive to allow data to flow from the child out to the parent.

An @Output() property should normally be initialized to an Angular EventEmitter with values flowing out of the component as events.



Just like with @Input(), you can use @Output() on a property of the child component but its type should be EventEmitter.

What is EventEmitter?

EventEmitter is responsible for raising the event. The @output property normally is of type EventEmitter. The child component will use the emit() method to emit an event along with the data.

@Output() marks a property in a child component as a doorway through which data can travel from the child to the parent. The child component then has to raise an event so the parent knows something has changed. To raise an event, @Output() works hand in hand with EventEmitter, which is a class in @angular/core that you use to emit custom events.

When you use @Output(), edit these parts of your app:

- The child component class and template
- The parent component class and template

Note:
The HTML element <input> and the Angular decorator @Input() are different. This documentation is about component communication in Angular as it pertains to @Input() and @Output().

The following example shows how to set up an @Output() in a child component that pushes data you enter in an HTML <input> to an array in the parent component.

In the child

This example features an <input> where a user can enter a value and click a <button> that raises an event. The EventEmitter then relays the data to the parent component.

First, be sure to import Output and EventEmitter in the child component class:

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

Next, still in the child, decorate a property with @Output() in the component class. The following example @Output() is called newItemEvent and its type is EventEmitter, which means it's an event.

```
@Output() newItemEvent = new EventEmitter<string>();
```

The different parts of the above declaration are as follows:

- **@Output()**—a decorator function marking the property as a way for data to go from the child to the parent
- **newItemEvent**—the name of the @Output()
- **EventEmitter<string>**—the @Output()'s type
- **new EventEmitter<string>()**—tells Angular to create a new event emitter and that the data it emits is of type string. The type could be any type, such as number, boolean, and so on. For more information on EventEmitter, see the EventEmitter API documentation.

Next, create an addNewItem() method in the same component class:

```
export class ItemOutputComponent {

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

  addNewItem(value: string) {
    this.newItemEvent.emit(value);
  }
}
```

The addNewItem() function uses the @Output(), newItemEvent, to raise an event in which it emits the value the user types into the <input>. In other words, when the user clicks the add button in the UI, the child lets the parent know about the event and gives that data to the parent.

In the child's template

- The child's template has two controls. The first is an HTML <input> with a template reference variable , #newItem, where the user types in an item name. Whatever the user types into the <input> gets stored in the #newItem variable.

```
<label>Add an item: <input #newItem></label>
<button (click)="addNewItem(newItem.value)">Add to parent's
list</button>
```

- The second element is a <button> with an event binding. You know it's an event binding because the part to the left of the equal sign is in parentheses, (click).
- The (click) event is bound to the addNewItem() method in the child component class which takes as its argument whatever the value of #newItem is.
- Now the child component has an @Output() for sending data to the parent and a method for raising an event. The next step is in the parent.

In the parent

In this example, the parent component is AppComponent, but you could use any component in which you could nest the child.

The AppComponent in this example features a list of items in an array and a method for adding more items to the array.

```
export class AppComponent {
  items = ['Television', 'Air Conditioner', 'Refrigerator', 'Dining
Table'];

  addItem(newItem: string) {
    this.items.push(newItem);
  }
}
```

The addItem() method takes an argument in the form of a string and then pushes, or adds, that string to the items array.

In the parent's template

Next, in the parent's template, bind the parent's method to the child's event. Put the child selector, here <app-item-output>, within the parent component's template, app.component.html.

```
<app-item-output (newItemEvent)="addItem($event)"></app-item-
output>
```

The event binding, (newItemEvent)='addItem($event)', tells Angular to connect the event in the child, newItemEvent, to the method in the parent, addItem(), and that the event that the child is notifying the parent about is to be the

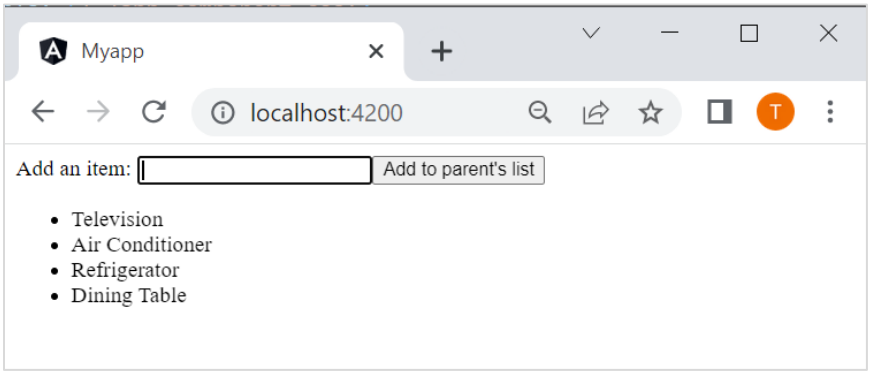
argument of addItem(). In other words, this is where the actual hand off of data takes place. The $event contains the data that the user types into the <input> in the child template UI.

Now, in order to see the @Output() working, add the following to the parent's template:

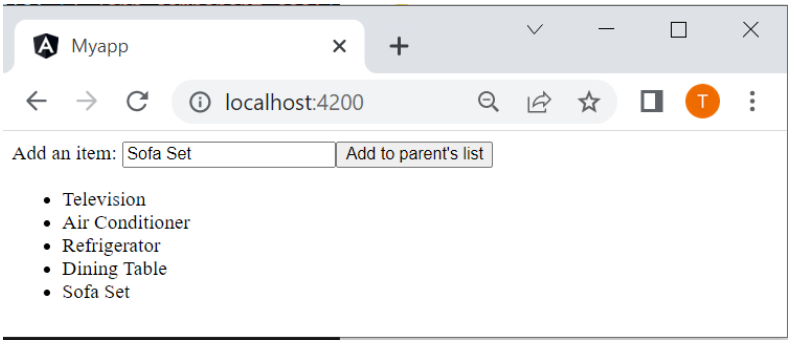
```
<ul>
  <li *ngFor="let item of items">{{item}}</li>
</ul>
```

The *ngFor iterates over the items in the items array. When you enter a value in the child's <input> and click the button, the child emits the event and the parent's addItem() method pushes the value to the items array and it renders in the list.

Output:



After adding the new item the output looks like this:



Activity Ten

Follow the instructions:

1. Open the Visual Studio code.

2. Follow the step by step instruction given above and create an angular application using @Output property.

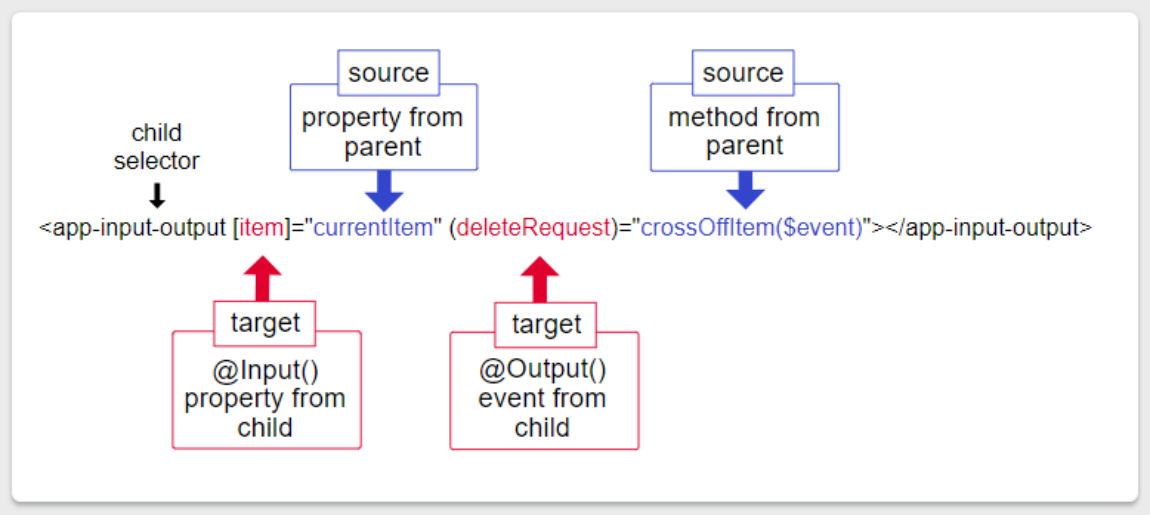
@Input() and @Output() together

You can use @Input() and @Output() on the same child component as in the following:

```
<app-input-output [item]="currentItem"
(deleteRequest)="crossOffItem($event)"></app-input-output>
```

The target, item, which is an @Input() property in the child component class, receives its value from the parent's property, currentItem. When you click delete, the child component raises an event, deleteRequest, which is the argument for the parent's crossOffItem() method.

The following diagram is of an @Input() and an @Output() on the same child component and shows the different parts of each:



As the diagram shows, use inputs and outputs together in the same manner as using them separately. Here, the child selector is <app-input-output> with item and deleteRequest being @Input() and @Output() properties in the child component class. The property currentItem and the method crossOffItem() are both in the parent component class.

To combine property and event bindings using the banana-in-a-box syntax, [()], see Two-way Binding.

@Input() and @Output() declarations

Instead of using the @Input() and @Output() decorators to declare inputs and outputs, you can identify members in the inputs and outputs arrays of the directive metadata, as in this example:

```
// tslint:disable: no-inputs-metadata-property no-outputs-metadata-
property
inputs: ['clearanceItem'],
outputs: ['buyEvent']
// tslint:enable: no-inputs-metadata-property no-outputs-metadata-
property
```

While declaring inputs and outputs in the @Directive and @Component metadata is possible, it is a better practice to use the @Input() and @Output() class decorators instead, as follows:

```
@Input() item: string;
@Output() deleteRequest = new EventEmitter<string>();
```

Note:
If you get a template parse error when trying to use inputs or outputs, but you know that the properties do indeed exist, double-check that your properties are annotated with @Input() / @Output() or that you've declared them in an inputs/outputs array:

Exercise

Answer the following questions.

1. What is data binding?
2. What are the types of databinding? Explain with examples.
3. How is @input property used?
4. How is @output property used?
5. How are nesting components used?

Module 3: Templates, Styles & Directives

Templates

We have learnt in the last module that a template is a form of HTML that tells Angular how to render the component. There are two ways of defining a template in an angular component.

- **Inline template:** The inline template is defined by placing the HTML code in backticks ` and is linked to the component metadata using the template property of @Component decorator.
- **External Template:** The template is defined in a separate HTML file and is linked to the component metadata using the @Component decorator's templateUrl property.

Angular 7 uses the <ng-template> as the tag instead of <template>which is used in Angular2. The reason it started to use <ng-template> instead of <template> from Angular 4 onwards is because there is a name conflict between the <template> tag and the html <template> standard tag. It will deprecate completely going ahead. This was one of the major changes made in Angular 4 version.

Let us now use the template along with the if else condition and see the output.

app.component.html

```
<!--The content below is only a placeholder and can be replaced.-->
<div style = "text-align:center">
  <h1>Welcome to {{title}}.</h1>
</div>

<div> Months :
  <select (change) = "changemonths($event)" name = "month">
     <option *ngFor = "let i of months">{{i}}</option>
  </select>
</div>
<br/>

<div>
```

```
  <span *ngIf = "isavailable;then condition1 else condition2">
     Condition is valid.
  </span>
  <ng-template #condition1>Condition is valid from template</ng-template>
  <ng-template #condition2>Condition is invalid from template</ng-template>
</div>
<button (click) = "myClickFunction($event)">Click Me</button>
```

For the Span tag, we have added the if statement with the else condition and will call template condition1, else condition2.

The templates are to be called as follows −

```
  <ng-template #condition1>Condition is valid from template</ng-template>
  <ng-template #condition2>Condition is invalid from template</ng-template>
```

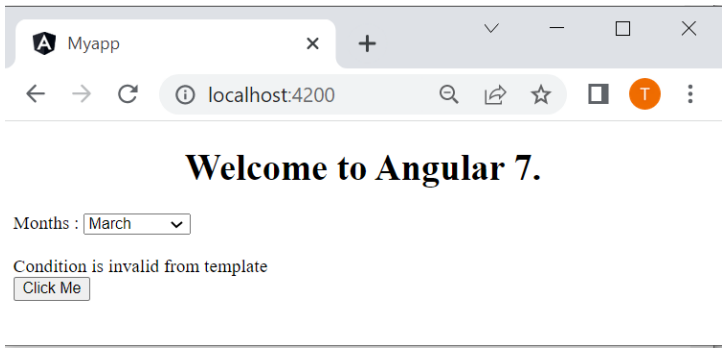
If the condition is true, then the condition1 template is called, otherwise condition2.

app.component.ts

```
import { Component } from '@angular/core';
@Component({
   selector: 'app-root',
   templateUrl: './app.component.html',
   styleUrls: ['./app.component.css']
})
export class AppComponent {
   title = 'Angular 7';

   // declared array of months.
   months = ["January", "February", "March", "April", "May", "June", "July",
      "August", "September", "October", "November", "December"];
   isavailable = false; // variable is set to true

   myClickFunction(event: any) {
      //just added console.log which will display the event details in browser on click of
the button.
      alert("Button is clicked");
```

```
      console.log(event);
   }
   changemonths(event: any) {
      alert("Changed month from the Dropdown");
   }
}
```

The output in the browser is as follows –



The variable isavailable is false so the condition2 template is printed. If you click the button, the respective template will be called.

app.component.ts

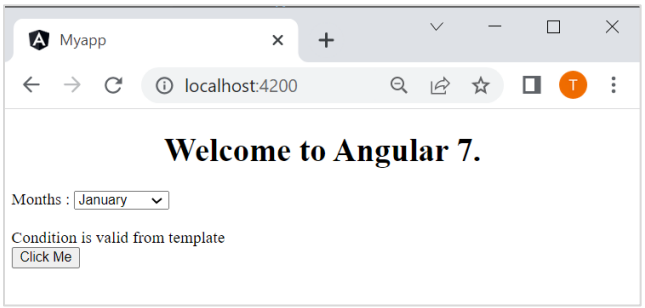
```
import { Component } from '@angular/core';
@Component({
   selector: 'app-root',
   templateUrl: './app.component.html',
   styleUrls: ['./app.component.css']
})
export class AppComponent {
   title = 'Angular 7';

   // declared array of months.
   months = ["January", "February", "March", "April", "May", "June", "July",
      "August", "September", "October", "November", "December"];

   isavailable = false; //variable is set to true
   myClickFunction(event: any) {
```

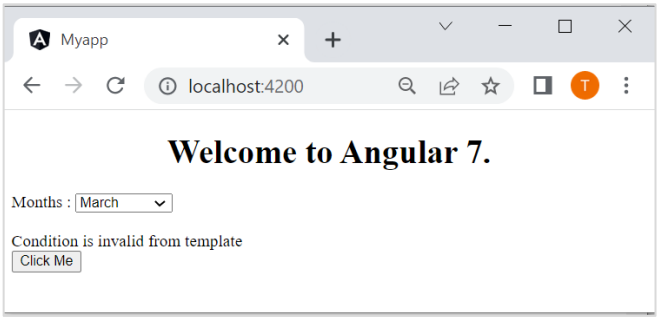
```
      this.isavailable = !this.isavailable;
      // variable is toggled onclick of the button
   }
   changemonths(event: any) {
      alert("Changed month from the Dropdown");
   }
}
```

The isavailable variable is toggled on click of the button as shown below –

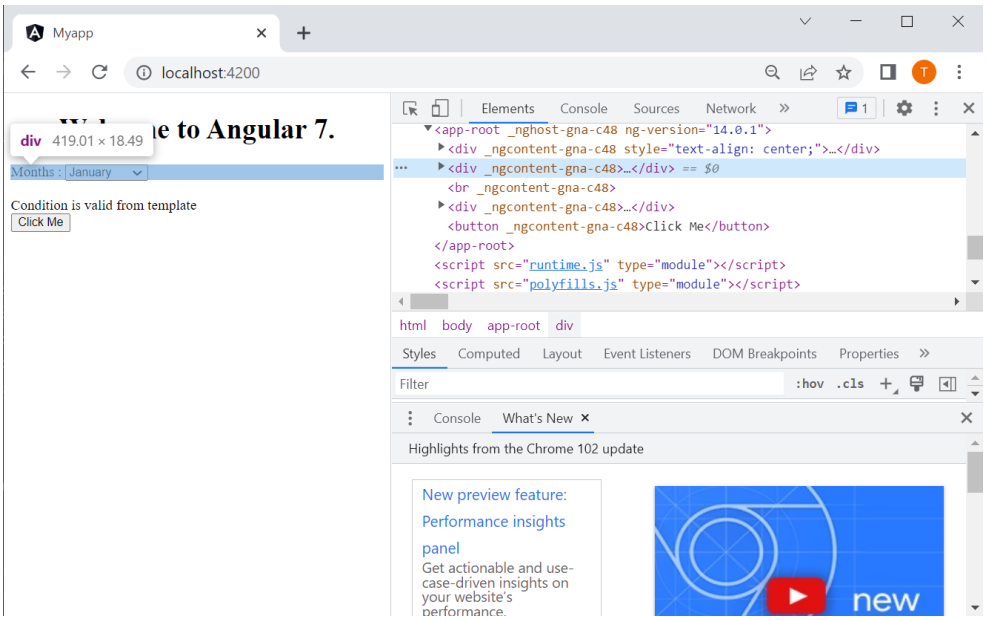
```
myClickFunction(event: any) {
      this.isavailable = !this.isavailable;
```

When you click on the button based on the value of the isavailable variable the respective template will be displayed –





If you inspect the browser, you will see that you never get the span tag in the dom. The following example will help you understand the same.



Though in app.component.html we have added span tag and the <ng-template> for the condition

We do not see the span tag and also the <ng-template> in the dom structure when we inspect the same in browser.

The following line of code in html will help us get the span tag in the dom –

```
<!--The content below is only a placeholder and can be replaced.-->
<div style = "text-align:center">
   <h1> Welcome to {{title}}. </h1>
</div>

<div> Months :
   <select (change) = "changemonths($event)" name = "month">
      <option *ngFor = "let i of months">{{i}}</option>
   </select>
</div>
<br/>

<div>
   <span *ngIf = "isavailable; else condition2">
      Condition is valid.
   </span>
   <ng-template #condition1>Condition is valid from template </ng-template>
```

```
   <ng-template #condition2>Condition is invalid from template</ng-template>
</div>
<button (click) = "myClickFunction($event)">Click Me</button>
```

If we remove the then condition, we get the “Condition is valid” message in the browser and the span tag is also available in the dom. For example, in app.component.ts, we have made the isavailable variable as true.

Styles

Angular applications are styled with standard CSS. That means you can apply everything you know about CSS stylesheets, selectors, rules, and media queries directly to Angular applications.

Additionally, Angular can bundle component styles with components, enabling a more modular design than regular stylesheets.

How Angular Handles Styling

An Angular component ideally consists of the presentation file, the style sheet, the component file itself and the test file. This means that for every component created or generated by the CLI there is a specific style sheet for it. Angular was built in such a way that the styles defined inside the component style sheet are scoped to only that component alone no matter the class name. This is a lot like local and global variable definition and how they are scoped; this scoping mechanism is known as encapsulation

Using component styles

For every Angular component you write, you can define not only an HTML template, but also the CSS styles that go with that template, specifying any selectors, rules, and media queries that you need.

One way to do this is to set the styles property in the component metadata. The styles property takes an array of strings that contain CSS code. Usually you give it one string, as in the following example:

```
@Component({
  selector: 'app-root',
  template: `
    <h1>Tour of Heroes</h1>
    <app-hero-main [hero]="hero"></app-hero-main>
  `,
  styles: ['h1 { font-weight: normal; }']
})
export class HeroAppComponent {
/* . . . */
}
```

Special selectors

Component styles have a few special selectors from the world of shadow DOM style scoping. The following sections describe these selectors.

- **:host**

Use the :host pseudo-class selector to target styles in the element that hosts the component (as opposed to targeting elements inside the component's template).

```
:host {
  display: block;
  border: 1px solid black;
}
```

The :host selector is the only way to target the host element. You can't reach the host element from inside the component with other selectors because it's not part of the component's own template. The host element is in a parent component's template.

Use the function form to apply host styles conditionally by including another selector inside parentheses after :host.

The next example targets the host element again, but only when it also has the active CSS class.

```
:host(.active) {
```

```
  border-width: 3px;
}
```

- **:host-context**

Sometimes it's useful to apply styles based on some condition outside of a component's view. For example, a CSS theme class could be applied to the document <body> element, and you want to change how your component looks based on that.

Use the :host-context() pseudo-class selector, which works just like the function form of :host(). The :host-context() selector looks for a CSS class in any ancestor of the component host element, up to the document root. The :host-context() selector is useful when combined with another selector.

The following example applies a background-color style to all <h2> elements inside the component, only if some ancestor element has the CSS class theme-light.

```
:host-context(.theme-light) h2 {
  background-color: #eef;
}
```

- **/deep/**

Component styles normally apply only to the HTML in the component's own template.

Use the /deep/ selector to force a style down through the child component tree into all the child component views. The /deep/ selector works to any depth of nested components, and it applies to both the view children and content children of the component.

The following example targets all <h3> elements, from the host element down through this component to all of its child elements in the DOM.

```
:host /deep/ h3 {
  font-style: italic;
```

```
}
```

Loading styles into components

There are several ways to add styles to a component:

- By setting styles or styleUrls metadata.
- Inline in the template HTML.
- With CSS imports.

Styles in metadata

You can add a styles array property to the @Component decorator. Each string in the array (usually just one string) defines the CSS.

```
@Component({
  selector: 'hero-app',
  template: `
    <h1>Tour of Heroes</h1>
    <hero-app-main [hero]=hero></hero-app-main>`,
  styles: ['h1 { font-weight: normal; }']
})
export class HeroAppComponent {
/* . . . */
}
```

Style URLs in metadata

You can load styles from external CSS files by adding a styleUrls attribute into a component's @Component decorator:

```
@Component({
  selector: 'hero-details',
  template: `
    <h2>{{hero.name}}</h2>
    <hero-team [hero]=hero></hero-team>
    <ng-content></ng-content>
```

```
  `,
  styleUrls: ['app/hero-details.component.css']
})
export class HeroDetailsComponent {
/* . . . */
}
```

Template inline styles

You can embed styles directly into the HTML template by putting them inside <style> tags.

```
@Component({
  selector: 'hero-controls',
  template: `
    <style>
      button {
        background-color: white;
        border: 1px solid #777;
      }
    </style>
    <h3>Controls</h3>
    <button (click)="activate()">Activate</button>
  `
})
```

Template link tags

You can also embed <link> tags into the component's HTML template.

As with styleUrls, the link tag's href URL is relative to the application root, not the component file.

```
@Component({
  selector: 'hero-team',
  template: `
    <link rel="stylesheet" href="app/hero-team.component.css">
    <h3>Team</h3>
    <ul>
      <li *ngFor="let member of hero.team">
```

```
        {{member}}
      </li>
    </ul>`
})
```

CSS @imports

You can also import CSS files into the CSS files using the standard CSS @import rule. For details, see @import on the MDN site.

In this case, the URL is relative to the CSS file into which you're importing.

```
@import 'hero-details-box.css';
```

View encapsulation

Encapsulation is a very critical aspect of the modern web components standard which supports keeping every component modular and independent. The shadow DOM is a part of the modern web component standard that ensures encapsulation is carried out through its API, providing a way to attach a separated DOM to an element. So basically, the shadow DOM allows you to hide DOM logic behind other elements without affecting any other part of the application so that you can use scoped styles in your component in isolation.

View encapsulation defines whether the template and styles defined within the component can affect the whole application or vice versa. Angular provides three encapsulation strategies:

Emulated (default) - styles from main HTML propagate to the component. Styles defined in this component's @Component decorator are scoped to this component only.

ShadowDOM - styles from main HTML do not propagate to the component. Styles defined in this

component's @Component decorator is scoped to this component only.

None - styles from the component propagate back to the main HTML and therefore are visible to all components on the page. Be careful with apps that have None and Native components in the application. All components with None encapsulation will have their styles duplicated in all components with Native encapsulation.

Example Application

app.component.ts

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

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

app.component.html

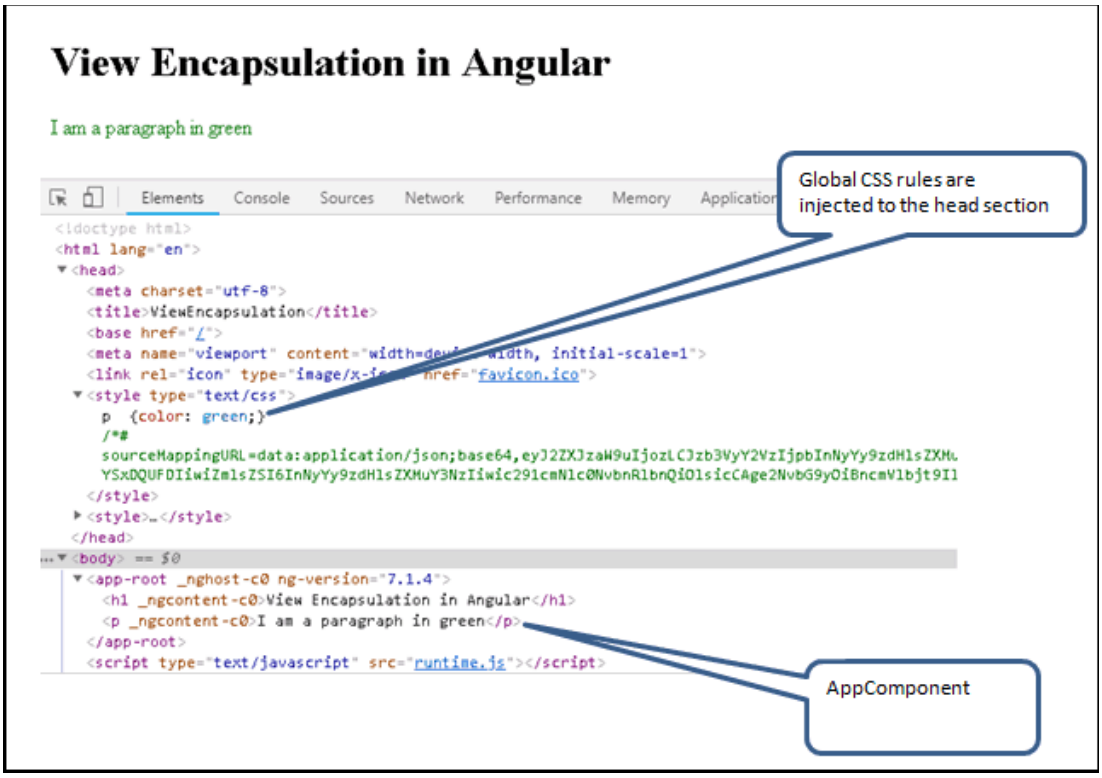
```
<h1>{{title}}</h1>
<p>I am a paragraph in green</p>
```

src/styles.css

```
p  {color: green;}
```

Run the app and you should able to see the paragraph in green.

Open the chrome developer tools and check the elements section. The CSS rules are inserted in the head section of the page.



ViewEncapsulation.None

The ViewEncapsulation.None is used, when we do not want any encapsulation. When you use this, the styles defined in one component affects the elements of the other components.

Now, let us look at ViewEncapsulation.None does.

Create a new component ViewNoneComponent and add the following code.

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

@Component({
  selector: 'app-none',
  template: `<p>I am not encapsulated and in blue
             (ViewEncapsulation.None) </p>`,
  styles: ['p { color:blue}'],
  encapsulation: ViewEncapsulation.None
})
```

```
export class ViewNoneComponent {
}
```

We have added encapsulation: ViewEncapsulation.None. We have also defined the inline style p { color:blue}

Do not forget to import & declare the component in AppModule. You also need to add the <app-none></app-none> selector in app.component.html

```
<h1>{{title}}</h1>
<p>I am a paragraph in green</p>
<app-none></app-none>
```

Run the code and as expected both the paragraphs turn blue.

That is because, the global scope of CSS styles. The style defined in the ViewNoneComponent is injected to the global style and overrides any previously defined style. The style p {color: blue;} overrides the style p {color: green;} defined in the styles.css.

The important points are

- The styles defined in the component affect the other components
- The global styles affect the element styles in the component

ViewEncapsulation.Emulated

In a HTML page, we can easily add a id or a class to the element to increase the specificity of the CSS rules so that the CSS rules do not interfere with each other..

The ViewEncapsulation.Emulated strategy in angular adds the unique HTML attributes to the component CSS styles and to the markup so as to achieve the encapsulation. This is not true encapsulation. The Angular Emulates the encapsulation, Hence the name Emulated.

Create a new component in Angular app and name it as ViewEmulatedComponent. as shown below:

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

@Component({
  selector: 'app-emulated',
  template: `<p>Using Emulator</p>`,
  styles: ['p { color:red}'],
  encapsulation: ViewEncapsulation.Emulated
})
export class ViewEmulatedComponent {
}
```

We have not added any id in the above component. Only change this component has with one ViewNoneComponent was encapsulation mode, which is set to ViewEncapsulation.Emulated

Update app.component.html

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

<p>I am a paragraph in green</p>

<app-none></app-none>

<app-emulated></app-emulated>
```

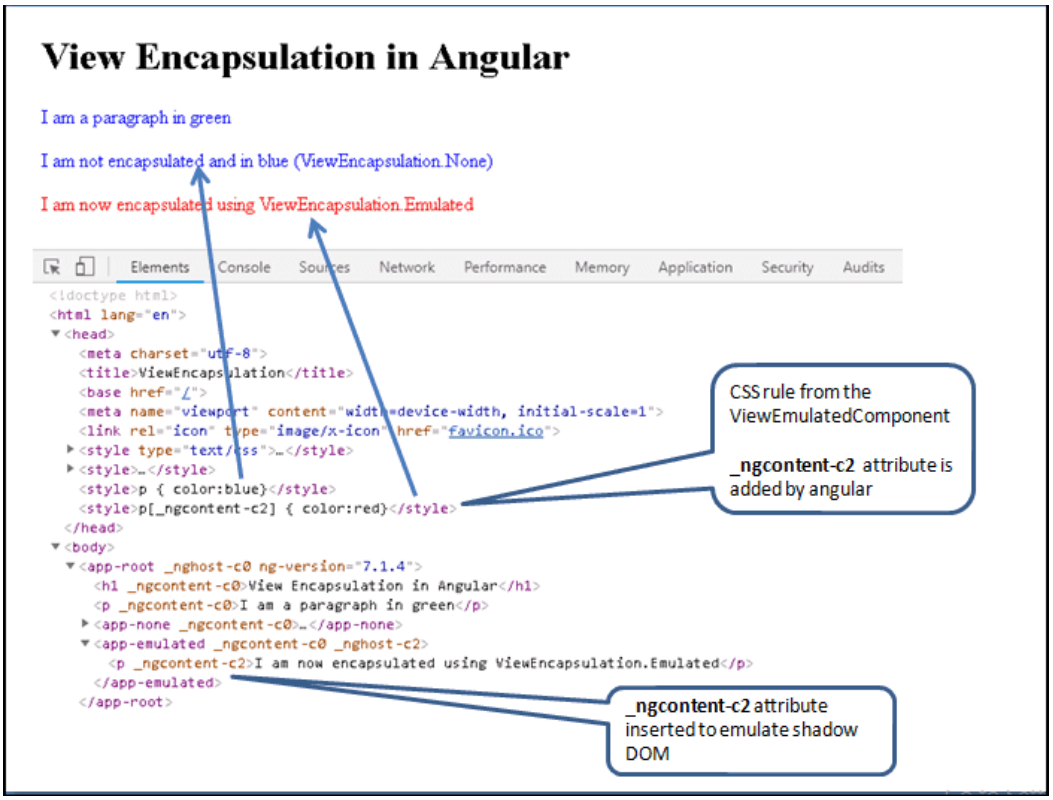
Now, you can see that the style does not spill out to other components, when you use emulated mode. i.e because angular adds _ngcontent-c# attributes to the emulated components and makes necessary changes in the generated styles

You can see this by opening the chrome developer console

_ngcontent-c2 attribute is added in the style and to the p element, making the style local to the component

```
<style>p[_ngcontent-c2] { color:red}</style>
```

```
<app-emulated _ngcontent-c0 _nghost-c2>
    <p _ngcontent-c2>Using Emulator</p>
</app-emulated>
```



The important points are

- This strategy isolates the component styles. They do not bleed out to other components.
- The global styles may affect the element styles in the component
- The Angular adds the attributes to the styles and mark up

ViewEncapsulation.ShadowDOM

The Shadow DOM is a scoped sub-tree of the DOM. It is attached to a element (called shadow host) of the DOM tree. The shadow dom do not appear as child node of the shadow host, when you traverse the main DOM.

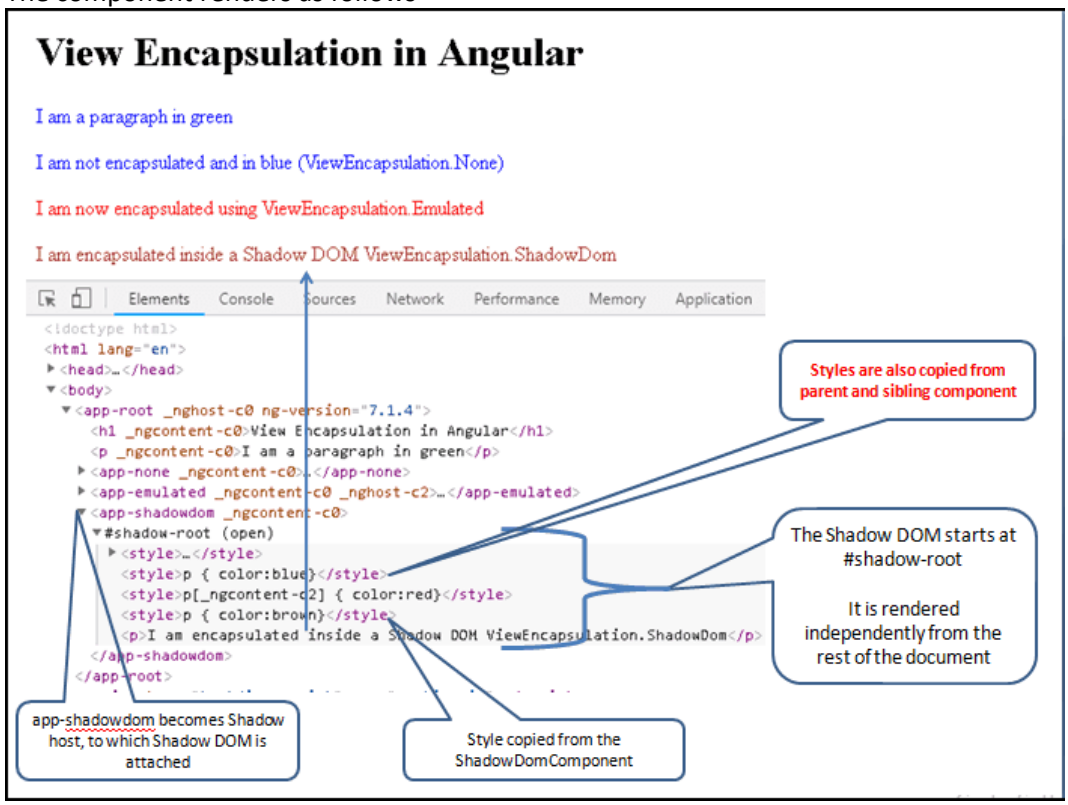
The browser keeps the shadow DOM separate from the main DOM. The rendering of the Shadow dom and the main DOM happens separately. The browser flattens them together before displaying it to the user. The feature, state & style of the Shadow DOM stays private and not affected by the main DOM. Hence it achieves the true encapsulation.

To create shadow dom in angular , all we need to do is to add the ViewEncapsulation.ShadowDom as the encapsulation strategy.

Create a new component ViewShadowdomComponent and add the following code

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

@Component({
  selector: 'app-shadowdom',
  template: `<p>I am encapsulated inside a Shadow DOM ViewEncapsulation.ShadowDom</p>`,
  styles: ['p { color:brown}'],
  encapsulation: ViewEncapsulation.ShadowDom
})
export class ViewShadowdomComponent {

}
```

The component renders as follows



The angular renders the component inside the #shadow root element. The styles from the component along with the styles from the parent and other components are also injected inside the shadow root.

Activity Eleven

Follow the instructions:

1. Open the Visual Studio code.
2. Follow the step by step instruction given above and create an angular application as shown below.
3. Use ViewEncapsulation.None to get the output as shown below:

Test Encapsulation Component1

Test Encapsulation Component2

Adding Bootstrap

What is Bootstrap?

Bootstrap is the most popular front-end open-source CSS framework and a toolkit to design your customized and mobile-first responsive websites. It comes with CSS- and JavaScript-based design templates, Sass variables and mixins, a responsive grid system, extensive prebuilt components, and powerful JavaScript plugins for typography, forms, buttons, navigations, carousels, slides, and many other interface components.

Install Bootstrap in Your Angular Project

In this step, we will add Bootstrap to your Angular project. There are many ways to install Bootstrap into your Angular project.

- Install from npm using npm install command.
- Download and add the Bootstrap files to your Angular project's src/assets folder.
- Use Bootstrap from a CDN.

We will use the first method and install Bootstrap through npm in the command line interface.

```
C:\projects\my-app>npm install bootstrap
```

This command will add the bootstrap package to package.json. Also, it will install bootstrap assets in the node_modules/bootstrap folder.

You need to install jQuery next. Let's use the following command in the command-line interface:

```
C:\projects\my-app>npm install jquery
```

Add Bootstrap to Angular app

Let us look at different methods to add Bootstrap to your Angular project:

- Using angular.json
- Using index.html
- Using styles.css
- Using ng-bootstrap and ngx-bootstrap
- Using Schematics

Add Bootstrap to Angular using angular.json

You can add Bootstrap to Angular by including the below files in your angular.json file:

- node_modules/bootstrap/dist/css/bootstrap.css in the projects>architect>build>styles array,
- node_modules/jquery/dist/jquery.js in the projects>architect>build>scripts array,
- node_modules/bootstrap/dist/js/bootstrap.js in the projects>architect>build>scripts array

As shown below: my-app/angular.json

```
{
  "$schema": "./node_modules/@angular/cli/lib/config/schema.json",
  "version": 1,
  "newProjectRoot": "projects",
  "projects": {
    "my-app": {
      "projectType": "application",
      "schematics": {
        "@schematics/angular:application": {
          "strict": true
        }
      },
      "root": "",
      "sourceRoot": "src",
      "prefix": "app",
      "architect": {
        "build": {
          "builder": "@angular-devkit/build-angular:browser",
          "options": {
            "outputPath": "dist/my-app",
            "index": "src/index.html",
            "main": "src/main.ts",
            "polyfills": "src/polyfills.ts",
            "tsConfig": "tsconfig.app.json",
            "assets": [
              "src/favicon.ico",
              "src/assets"
            ],
            "styles": [
              "src/styles.css",
              "node_modules/bootstrap/dist/css/bootstrap.css"
            ],
            "scripts": [
              "node_modules/jquery/dist/jquery.js",
              "node_modules/bootstrap/dist/js/bootstrap.js"
            ]
          },
```

Add Bootstrap to Angular using index.html

You can also add Bootstrap to Angular by adding tags in your src/index.html file.

- A <link> tag to add the bootstrap.css file in the <head> section,
- A <script> tag to add the jquery.js file before the </body> tag,
- A <script> tag to add the bootstrap.js file before the </body> tag.

After adding these tags, you can see your src/index.html file like the below example:

my-app/src/index.html

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>MyApp</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
  <link rel="stylesheet" href="../node_modules/bootstrap/dist/css/bootstrap.css">
</head>
<body>
  <app-root></app-root>
  <script src="../node_modules/jquery/dist/jquery.js"></script>
  <script src="../node_modules/bootstrap/dist/js/bootstrap.js"></script>
</body>
</html>
```

Add Bootstrap to Angular using styles.css

Also, you can add Bootstrap to your Angular project by importing the bootstrap.css file in your src/styles.css file as follows:

my-app/src/styles.css

```
@import "~bootstrap/dist/css/bootstrap.css";
```

This method replaces the previous methods, and you don't need to add this file to the styles array of the angular.json file or the index.html file. However, you can add JavaScript file(s) using the script array of the angular.json file or using the <script> tag(s) in the index.html file as you did in the previous two methods.

Add Bootstrap to Angular using ng-bootstrap ngx-bootstrap

In this method, ng-bootstrap completely replaces JavaScript implementation for components. You don't have to add bootstrap.js or bootstrap.min.js. Also, you should not include jQuery or popper.js libraries when you follow this method to add Bootstrap to your Angular project.

First, install the library as shown below:

```
C:\projects\my-app>npm install @ng-bootstrap/ng-bootstrap
```

After that, you should import the main module and add it to your app root module as shown below:

my-app/src/app/app.modules.ts

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

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import {NgbModule} from '@ng-bootstrap/ng-bootstrap';

@NgModule({
  declarations: [
    AppComponent,
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    NgbModule.forRoot()
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

This ng-bootstrap needs the Bootstrap CSS files to be present in your project. As shown below, you can add it in the styles array of your angular.json file.

my-app/angular.json

```
{
  "$schema": "./node_modules/@angular/cli/lib/config/schema.json",
  "version": 1,
  "newProjectRoot": "projects",
  "projects": {
    "my-app": {
      "projectType": "application",
      "schematics": {
        "@schematics/angular:application": {
          "strict": true
        }
      },
      "root": "",
      "sourceRoot": "src",
      "prefix": "app",
      "architect": {
        "build": {
          "builder": "@angular-devkit/build-angular:browser",
          "options": {
            "outputPath": "dist/my-app",
            "index": "src/index.html",
            "main": "src/main.ts",
            "polyfills": "src/polyfills.ts",
            "tsConfig": "tsconfig.app.json",
            "assets": [
              "src/favicon.ico",
              "src/assets"
            ],
            "styles": [
              "src/styles.css",
              "./node_modules/bootstrap/dist/css/bootstrap.css"
            ],
            "scripts": []
          },
```

Now you can use Bootstrap and add Bootstrap components to your Angular application.

Also, you can use the ngx-bootstrap library to add Bootstrap to your Angular project. First, you have to install ngx-bootstrap to your project as follows:

Before installing, make sure that you are in your Angular project folder. According to this example, it's C:\projects\my-app>.

```
C:\projects\my-app>npm install ngx-bootstrap
```

After that, you should add the Bootstrap CSS file. You can include Bootstrap from CDN within the <head> tag in the index.html file, as shown below.

my-app/src/index.html

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>MyApp</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
  <link href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css" rel="stylesheet">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

Then update your src/app/app.module.ts file as per the below example:

my-app/src/app/app.module.ts

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

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { BsDropdownModule} from 'ngx-bootstrap/dropdown';
import { AlertModule } from 'ngx-bootstrap';

@NgModule({
  declarations: [
    AppComponent,
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    BsDropdownModule.forRoot(),
    AlertModule.forRoot()

  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

This example imports components BsDropdownModule and AlertModule. You need to import the modules for each element you need to use.

Because ngx-bootstrap provides each component with its module, you need to import the required modules to your project. In this way, you can import only the components required for your project and make the application size smaller.

Add Bootstrap to Angular using Schematics

With the new ng add command in Angular 7+, it's straightforward to add support to an external library to your project. Using this command, you can add Bootstrap to Angular without extra configurations. You have just to run the below command in the terminal screen as shown in the below example:

```
C:\projects\my-app>ng add @ng-bootstrap/schematics
```

In this example, you don't have to add jQuery.js because we use ng-bootstrap. Now you have support for Bootstrap components and styles.

Directives

Directives are classes that add additional behavior to elements in your Angular applications. Use Angular's built-in directives to manage forms, lists, styles, and what users see.

The different types of Angular directives are as follows:

DIRECTIVE TYPES	DETAILS
Components	Used with a template. This type of directive is the most common directive type.
Attribute directives	Change the appearance or behavior of an element, component, or another directive.
Structural directives	Change the DOM layout by adding and removing DOM elements.

Built-in attribute directives

Attribute directives listen to and modify the behavior of other HTML elements, attributes, properties, and components.

Many NgModules such as the RouterModule and the FormsModule define their own attribute directives.

The most common attribute directives are as follows:

COMMON DIRECTIVES	DETAILS
NgClass	Adds and removes a set of CSS classes.
NgStyle	Adds and removes a set of HTML styles.
NgModel	Adds two-way data binding to an HTML form element.

Using NgClass with an expression

On the element you'd like to style, add [ngClass] and set it equal to an expression. In this case, isSpecial is a boolean set to true in app.component.ts. Because isSpecial is true, ngClass applies the class of special to the .

```
<div [ngClass]="isSpecial ? 'special' : ''">This div is special</div>
```

Using NgClass with a method

1. To use NgClass with a method, add the method to the component class. In the following example, setCurrentClasses() sets the property currentClasses with an object that adds or removes three classes based on the true or false state of three other component properties.

Each key of the object is a CSS class name. If a key is true, ngClass adds the class. If a key is false, ngClass removes the class.

```
currentClasses: Record<string, boolean> = {};
/* . . . */
setCurrentClasses() {
 // CSS classes: added/removed per current state of component properties
 this.currentClasses =  {
  saveable: this.canSave,
  modified: !this.isUnchanged,
  special:  this.isSpecial
 };
}
```

2. In the template, add the ngClass property binding to currentClasses to set the element's classes:

```
<div [ngClass]="currentClasses">This div is initially saveable, unchanged, and special.</div>
```

For this use case, Angular applies the classes on initialization and in case of changes. The full example calls setCurrentClasses() initially with ngOnInit() and when the dependent properties change through a button click. These steps are not necessary to implement ngClass. For more information, see the live example / download example app.component.ts and app.component.html.

Setting inline styles with NgStyle

Use NgStyle to set multiple inline styles simultaneously, based on the state of the component.

1. To use NgStyle, add a method to the component class.

In the following example, setCurrentStyles() sets the property currentStyles with an object that defines three styles, based on the state of three other component properties.

```
currentStyles: Record<string, string> = {};
/* . . . */
setCurrentStyles() {
  // CSS styles: set per current state of component properties
  this.currentStyles = {
```

```
    'font-style':  this.canSave      ? 'italic' : 'normal',
    'font-weight': !this.isUnchanged ? 'bold'   : 'normal',
    'font-size':   this.isSpecial    ? '24px'   : '12px'
  };
}
```

2. To set the element's styles, add an ngStyle property binding to currentStyles.

```
<div [ngStyle]="currentStyles">
  This div is initially italic, normal weight, and extra large (24px).
</div>
```

For this use case, Angular applies the styles upon initialization and in case of changes. To do this, the full example calls setCurrentStyles() initially with ngOnInit() and when the dependent properties change through a button click. However, these steps are not necessary to implement ngStyle on its own. See the live example / download example app.component.ts and app.component.html for this optional implementation.

Displaying and updating properties with ngModel

Use the NgModel directive to display a data property and update that property when the user makes changes.

1. Import FormsModule and add it to the NgModule's imports list

```
import { FormsModule } from '@angular/forms'; // <--- JavaScript import from Angular
/* . . . */
@NgModule({
  /* . . . */

  imports: [
    BrowserModule,
    FormsModule // <--- import into the NgModule
  ],
  /* . . . */
})
export class AppModule { }
```

2. Add an `[(ngModel)]` binding on an HTML `<form>` element and set it equal to the property, here `name`.

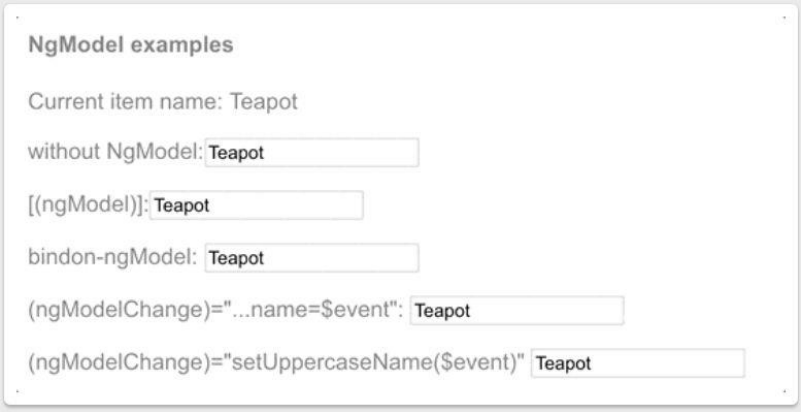
```
<label for="example-ngModel">[(ngModel)]:</label>
<input [(ngModel)]="currentItem.name" id="example-ngModel">
```

This [(ngModel)] syntax can only set a data-bound property.

To customize your configuration, write the expanded form, which separates the property and event binding. Use property binding to set the property and event binding to respond to changes. The following example changes the <input> value to uppercase:

```
<input [ngModel]="currentItem.name" (ngModelChange)="setUppercaseName($event)"
id="example-uppercase">
```

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



This section walks you through creating a highlight directive that sets the background color of the host element to yellow.

1. To create a directive, use the CLI command ng generate directive.

```
ng generate directive apply-class
```

The CLI creates src/app/apply-class.directive.ts, a corresponding test file src/app/apply-class.directive.spec.ts, and declares the directive class in the AppModule.

The CLI generates the default src/app/apply-class.directive.ts as follows:

```
import { Directive, Renderer2, ElementRef, OnInit } from '@angular/core';

@Directive({
  selector: '[applyCSSClass]'
})

export class ApplyClassDirective implements OnInit {

  constructor(
    private renderer2: Renderer2,
    private elementRef: ElementRef
  ) { }

  ngOnInit() {
    this.renderer2.addClass(this.elementRef.nativeElement, 'myClass')
  }
}
```

The @Directive() decorator's configuration property specifies the directive's CSS attribute selector, [applyCSSClass].

2. Import ElementRef from @angular/core. ElementRef grants direct access to the host DOM element through its nativeElement property.
3. Add ElementRef in the directive's constructor() to inject a reference to the host DOM element, the element to which you apply applyCSSClass.
4. Add logic to the ApplyClassDirective class that sets the background to yellow.

Applying an attribute directive

1. To use the ApplyClassDirective, add a <p> element to the HTML template with the directive as an attribute.

Src/app/app.component.html

```
<p applyCSSClass>
    Angular 9 Example Directive with Renderer2
  </p>
```

Angular creates an instance of the ApplyClassDirective class and injects a reference to the <p> element into the directive's constructor, which sets the <p> element's background style to yellow.

Handling user events

This section shows you how to detect when a user enters into or out of the element and to respond by setting or clearing the highlight color.

1. Import HostListener from '@angular/core'.

```
import { Directive, Renderer2, ElementRef, OnInit, HostListener } from '@angular/core';
```

2. Add two event handlers that respond when the mouse enters or leaves, each with the @HostListener() decorator.

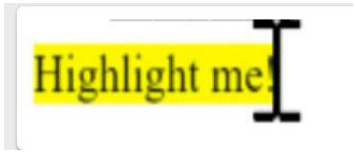
```
@HostListener('mouseenter') onMouseEnter() {
  this.highlight('yellow');
}

@HostListener('mouseleave') onMouseLeave() {
  this.highlight('');
}

private highlight(color: string) {
  this.elementRef.nativeElement.style.backgroundColor = color;
}
```

Subscribe to events of the DOM element that hosts an attribute directive, the <p> in this case, with the @HostListener() decorator.

The complete directive is as follows:

```
@Directive({
  selector: '[applyCSSClass]'
})

export class ApplyClassDirective implements OnInit {

  constructor(
    private renderer2: Renderer2,
    private elementRef: ElementRef
  ) { }

  ngOnInit() {
    this.renderer2.addClass(this.elementRef.nativeElement, 'myClass')
  }
  @HostListener('mouseenter') onMouseEnter() {
    this.highlight('yellow');
  }

  @HostListener('mouseleave') onMouseLeave() {
    this.highlight('');
  }

  private highlight(color: string) {
    this.elementRef.nativeElement.style.backgroundColor = color;
  }
}
```

The background color appears when the pointer hovers over the paragraph element and disappears as the pointer moves out.



HostBinding

Host Binding binds a Host element property to a variable in the directive or component.

As well as listening to output events from the host element a directive can also bind to input properties in the host element with @HostBinding.

This directive can change the properties of the host element, such as the list of classes that are set on the host element as well as a number of other properties.

Using the @HostBinding decorator a directive can link an internal property to an input property on the host element. So, if the internal property changed the input property on the host element would also change.

We first need something, a property on our directive which we can use as a source for binding.

We'll create a boolean called ishovering and in our onMouseOver() and onMouseOut() functions we'll set this to true and false accordingly, like so:

HostBinding Example

The following appHighLight directive, uses the HostBinding on style.border property of the parent element to the border property.

Whenever we change the value of the border, the angular will update the border property of the host element.

```
import { Directive, Renderer2, ElementRef, OnInit, HostListener} from '@angular/core';
class CardHoverDirective{
  private ishovering!: boolean;

  constructor(private el: ElementRef,
              private renderer2: Renderer2) {
  }

  @HostListener('mouseover') onMouseOver() {
    let part = this.el.nativeElement.querySelector('.card-text');
    this.renderer2.setStyle(part, 'display', 'block');
    this.ishovering = true;
  }

  @HostListener('mouseout') onMouseOut(): void {
    let part = this.el.nativeElement.querySelector('.card-text');
    this.renderer2.setStyle(part, 'display', 'none');
    this.ishovering = false;
  }
}
```

1. We've created a boolean called ishovering.
2. We set our boolean to true when we are being hovered over and false when we are not.

Now we need to link this source property to an input property on the host element, we do this by decorating our ishovering boolean with the @HostBinding decorator.

The @HostBinding decorator takes one parameter, the name of the property on the host element which we want to bind to.

If you remember we can use the alternative ngClass syntax by binding to the [class.<class-name>] property. Let's add the card-outline-primary class to our host element when the ishovering boolean is true.

```
import { Directive, Renderer2, ElementRef, OnInit, HostListener, HostBinding} from '@angular/core';
class CardHoverDirective {
  @HostBinding('class.card-outline-primary')
  private ishovering!: boolean;

  constructor(private el: ElementRef,
              private renderer: Renderer2) {
    // renderer.setElementStyle(el.nativeElement, 'backgroundColor', 'gray');
  }

  @HostListener('mouseover') onMouseOver() {
    let part = this.el.nativeElement.querySelector('.card-text');
    this.renderer.setStyle(part, 'display', 'block');
    this.ishovering = true;
  }

  @HostListener('mouseout') onMouseOut() {
    let part = this.el.nativeElement.querySelector('.card-text');
    this.renderer.setStyle(part, 'display', 'none');
    this.ishovering = false;
  }
}
```

We've added the @HostBinding decorator to the ishovering property.

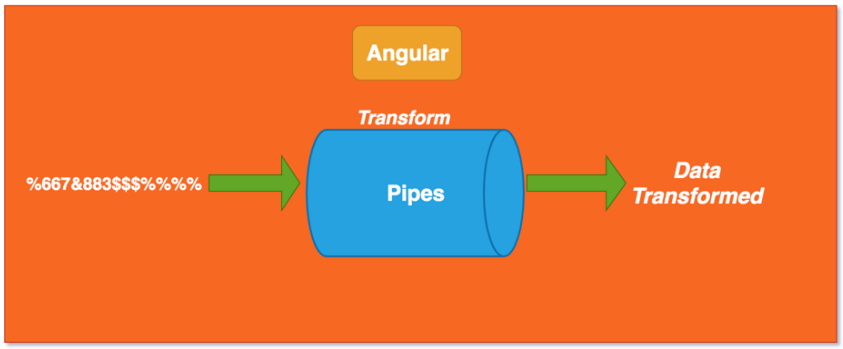
Now when we hover over a card as well as the punchline showing we adding a blue border to the card.

Module 4: Pipes, Services & Dependency Injection

What are Angular Pipes?

Angular Pipes allows its users to change the format in which data is being displayed on the screen. For instance, consider the date format. Dates can be represented in multiple ways, and the user can decide which one to use with the help of Angular Pipes.

Angular Pipes transform the output. You can think of them as makeup rooms where they beautify the data into a more desirable format. They do not alter the data but change how they appear to the user.



Technically, pipes are simple functions designed to accept an input value, process, and return a transformed value as the output. Angular supports many built-in pipes. However, you can also create custom pipes that suit your requirements. Some salient features include:

- Pipes are defined using the pipe "|" symbol.
- Pipes can be chained with other pipes.
- Pipes can be provided with arguments by using the colon (:) sign.

Some commonly used predefined Angular pipes are:

- DatePipe: Formats a date value.
- UpperCasePipe: Transforms text to uppercase.
- LowerCasePipe: Transforms text to lowercase.
- CurrencyPipe: Transforms a number to the currency string.
- PercentPipe: Transforms a number to the percentage string.
- DecimalPipe: Transforms a number into a decimal point string.

Using Built-in Angular Pipes

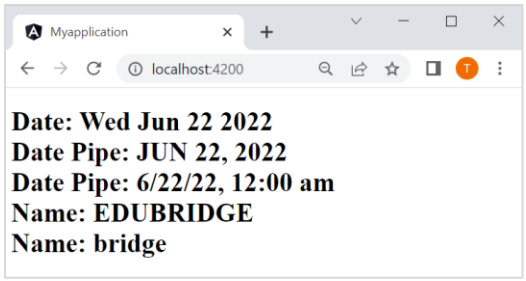
As mentioned above, Angular provides several built-in pipes to beautify the data being shown on the user interface.

In the pipes.component.ts file, we've created properties for date and name.

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

@Component({

  selector: 'app-pipes',

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

  styleUrls: ['./pipes.component.css']

})

export class PipesComponent implements OnInit {

  dateToday: string;

  name: string;

  constructor() { }

  ngOnInit(): void {

    this.dateToday = new Date().toDateString();

    this.name = "Edubridge"

  }

}
```

In Pipes.component.html, we've interpolated the properties and used pipes to format them.

```
<h1>

    Date: {{ dateToday  }} <br>

    Date Pipe: {{ dateToday | date | uppercase}}<br>

    Date Pipe: {{ dateToday | date: 'short' | lowercase}} <br>

    Name: {{ name | uppercase}} <br>

    Name: {{ name | slice:6}}

</h1>
```



In the code mentioned above, the date property initially displays in the default format. However, when used with the date pipe, the format changes. We've also used other pipes like uppercase and lowercase for better understanding.

As you can see, we have used the slice pipe for the name property. This pipe slices the text and displays it from the index position provided by the user.

Now that you know how to use pipes and their functionalities, let us show you how to create custom pipes.

Pure and Impure Pipes

Pipes in Angular are classified into Pure and Impure types. Let's have a closer look at them.

Pure Pipes

These pipes use pure functions. As a result of this, the pipe doesn't use any internal state and the output remains the same as long as the parameters passed remain the same. Angular calls the pipe only when it detects a change in the parameters being passed. A single instance of the pure pipe is used throughout all components.

Impure Pipes

An impure pipe in Angular is called for every change detection cycle regardless of the change in the input fields. Multiple pipe instances are created for these pipes and the inputs passed to these pipes are mutable.

Although by default pipes are pure, you can specify impure pipes using the pure property as shown below.

```
@Pipe({
  name: 'demopipe',
  pure : true/false
})
export class DemopipePipe implements PipeTransform {
```

Creating Custom Pipes

Angular makes provision to create custom pipes that convert the data in the format that you desire. Angular Pipes are TypeScript classes with the @Pipe decorator. The decorator has a name property in its metadata that specifies the Pipe and how and where it is used.

Attached below is a screenshot of the code that Angular has for the uppercase pipe.

```
/**
* Transforms text to uppercase.
*
* @stable
*/
@Pipe({name: 'uppercase'})
export class UpperCasePipe implements PipeTransform{
transform(value: string): string{
if(!value) return value;
if (typeof value !== 'string'){
throw invalidPipeArgumentError(UpperCasePipe, value);
}
Return value.yoUpperCase();
}
}
```

Pipe implements the PipeTransform interface. As the name suggests, it receives the value and transforms it into the desired format with the help of a transform() method.

Here are the general steps to create a custom pipe:

- Create a TypeScript Class with an export keyword.
- Decorate it with the @Pipe decorator and pass the name property to it.

- Implement the pipe transform interface in the class.
- Implement the transform method imposed due to the interface.
- Return the transformed data with the pipe.
- Add this pipe class to the declarations array of the module where you want to use it.

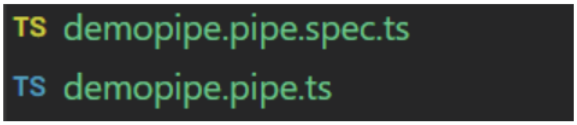
Alternatively, you can use the following command,

```
ng g pipe <nameofthepipe>
```

Example:

```
\Angular> cd myapplication
\Angular\myapplication> ng g pipe demopipe
```

Once run, two files are created.



The pipe.ts file holds the following code.

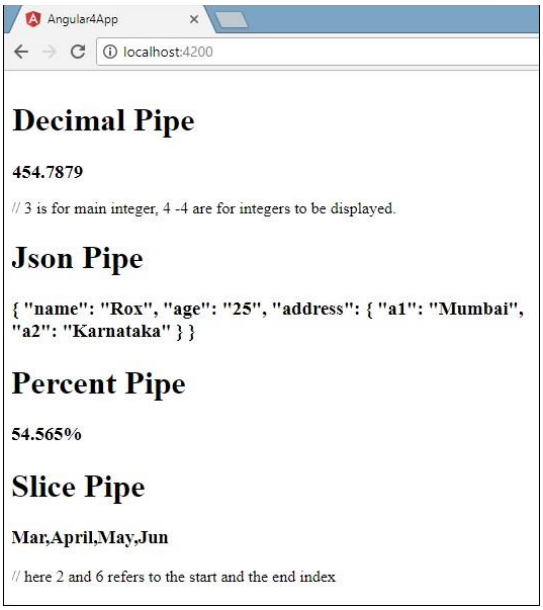
```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({

  name: 'demopipe'

})

export class DemopipePipe implements PipeTransform {

  transform(value: unknown, ...args: unknown[]): unknown {

    return null;

  }

}
```

As you can see, the PipeTransform interface and the transform method have been created.

Activity Twelve

Follow the instructions:

1. Open the Visual Studio code.
2. Understand the concept of Angular pipes and create a application as shown below:



Services & Dependency Injections

Services provides specific functionality in an Angular application. In a given Angular application, there may be one or more services can be used. Similarly, an Angular component may depend on one or more services.

Also, Angular services may depend on other services to work properly. Dependency resolution is one of the complex and time-consuming activity in developing any application. To reduce the complexity, Angular provides Dependency Injection pattern as one of the core concepts.

Types of Dependency Injection in Angular

There are three types of Dependency Injections in Angular, they are as follows:

- **Constructor injection:** Here, it provides the dependencies through a class constructor.
- **Setter injection**: The client uses a setter method into which the injector injects the dependency.
- **Interface injection:** The dependency provides an injector method that will inject the dependency into any client passed to it. On the other hand, the clients must implement an interface that exposes a setter method that accepts the dependency.

Create Angular service

An Angular service is plain Typescript class having one or more methods (functionality) along with @Injectable decorator. It enables the normal Typescript class to be used as service in Angular application.

```
import { Injectable } from '@angular/core'; @Injectable()
export class DebugService {
constructor() { }
}
```

Here, @Injectable decorator converts a plain Typescript class into Angular service.

Register Angular service

To use Dependency Injection, every service needs to be registered into the system. Angular provides multiple option to register a service. They are as follows –

- ModuleInjector @ root level

- ModuleInjector @ platform level
- ElementInjector using providers meta data
- ElementInjector using viewProviders meta data
- NullInjector

ModuleInjector @ root

ModuleInjector enforces the service to used only inside a specific module. ProvidedInmeta data available in @Injectable has to be used to specify the module in which the service can be used.

The value should refer to the one of the registered Angular Module (decorated with @NgModule). root is a special option which refers the root module of the application. The sample code is as follows −

```
import { Injectable } from '@angular/core'; @Injectable({
providedIn: 'root',
})
export class DebugService {
constructor() { }
}
```

ModuleInjector @ platform

Platform Injector is one level higher than ModuleInject and it is only in advanced and rare situation. Every Angular application starts by executing PreformBrowserDynamic().bootstrap method (see main.js), which is responsible for bootstrapping root module of Angular application.

PreformBrowserDynamic() method creates an injector configured by PlatformModule. We can configure platform level services using platformBrowser() method provided by PlatformModule.

NullInjector

NullInjector is one level higher than platform level ModuleInjector and is in the top level of the hierarchy. We could not able to register any service in the NullInjector. It resolves when the required service is not found anywhere in the hierarchy and simply throws an error.

ElementInjector using providers

ElementInjector enforces the service to be used only inside some particular components. providers and ViewProviders meta data available in @Component decorator is used to specify the list of services to be visible for the particular component. The sample code to use providers is as follows –

ExpenseEntryListComponent

```
// import statement
import { DebugService } from '../debug.service';
// component decorator
@Component({
selector: 'app-expense-entry-list',
templateUrl: './expense-entry-list.component.html',
styleUrls: ['./expense-entry-list.component.css'],
providers: [DebugService] })
```

Here, DebugService will be available only inside the ExpenseEntryListComponent and its view. To make DebugService in other component, simply use providers decorator in necessary component.

ElementInjector using viewProviders

viewProviders is similar to provider except it does not allow the service to be used inside the componentâ€™s content created using ng-content directive.

ExpenseEntryListComponent

```
// import statement
import { DebugService } from '../debug.service';
// component decorator
@Component({
selector: 'app-expense-entry-list',
templateUrl: './expense-entry-list.component.html',
styleUrls: ['./expense-entry-list.component.css'], viewProviders:
[DebugService]
})
```

Parent component can use a child component either through its view or content. Example of a parent component with child and content view is mentioned below –

Parent component view / template

```
<div>
child template in view
<child></child>
</div>
<ng-content></ng-content>
```

Child component view/template

```
<div>
child template in view
</div>
```

Parent component usage in a template (another component)

```
<parent>
<!-- child template in content -->
<child></child>
</parent>
```

Here,

- child component is used in two place. One inside the parent's view. Another inside parent content.
- Services will be available in child component, which is placed inside parent's view.
- Services will not be available in child component, which is placed inside parent's content.

Example:

we'll create a service class and inject the same into a component that displays a button for the user. The service holds employee details such as name, employee ID, and email ID. When the user clicks on the display button, the same will be displayed.

Step 1: Create a component to display the employee records. Use the command **ng g c <component name>** for the same. The component we've created is called emp_info

Step 2: Create a service using the command, ng g service <service name>. We've created a service called records. Once run, two files records.service.ts and records.service.spec.ts are created. The service consists of the employee data that needs to be displayed. We will be using three arrays for this purpose.

```
info1: string[]=["Reena Tandon",'E354', 'rt@abc.net']

info2: string[]=["Anuja Deshpande",'E673','ad@abc.net']

info3: string[]=["Shantanu Sharon",'E865','ss@abc.net']
```

With the agenda to retrieve this data in our component, we use a method, for the same. This method returns the employee data.

```
getInfo1():string[]{

    return this.info1

  }

  getInfo2():string[]{

    return this.info2

  }

  getInfo3():string[]{

    return this.info3

  }
```

Step 3: In order to retrieve this information in our emp_info.component.ts, we need three more arrays.

```
infoReceived1: string[]=[];

    infoReceived2: string[]=[];

    infoReceived3: string[]=[];

    getInfoFromService1(){

      this.infoReceived1 = this.rservice.getInfo1()

    }

    getInfoFromService2(){

      this.infoReceived2 = this.rservice.getInfo2()

    }

    getInfoFromService3(){

      this.infoReceived3 = this.rservice.getInfo3()

    }
```

As mentioned, services are implemented using dependency injection. We import the service class into the component.ts file. The key reason behind doing this is to tell Angular that when the component is created, an instance of the service class is also made to perform all the necessary tasks. We must also declare this instance in the providers' array of the component. However, to access this instance, an object is created that can access the methods and variables of the service class. We've created the object rservice for the same.

```
import { Component, OnInit } from '@angular/core';
import { RecordsService } from "../records.service"
@Component({
  selector: 'app-e-info',
  templateUrl: './e-info.component.html',
  styleUrls: ['./emp -info.component.css'],
  providers: [RecordsService]
})
export class EmpInfoComponent implements OnInit {
  infoReceived1: string[]=[];
  infoReceived2: string[]=[];
  infoReceived3: string[]=[];
  getInfoFromService1(){
    this.infoReceived1 = this.rservice.getInfo1()
  }
  getInfoFromService2(){
    this.infoReceived2 = this.rservice.getInfo2()
  }
  getInfoFromService3(){
    this.infoReceived3 = this.rservice.getInfo3()
  }
constructor(private rservice: RecordsService) { }
  ngOnInit(): void { }}
```

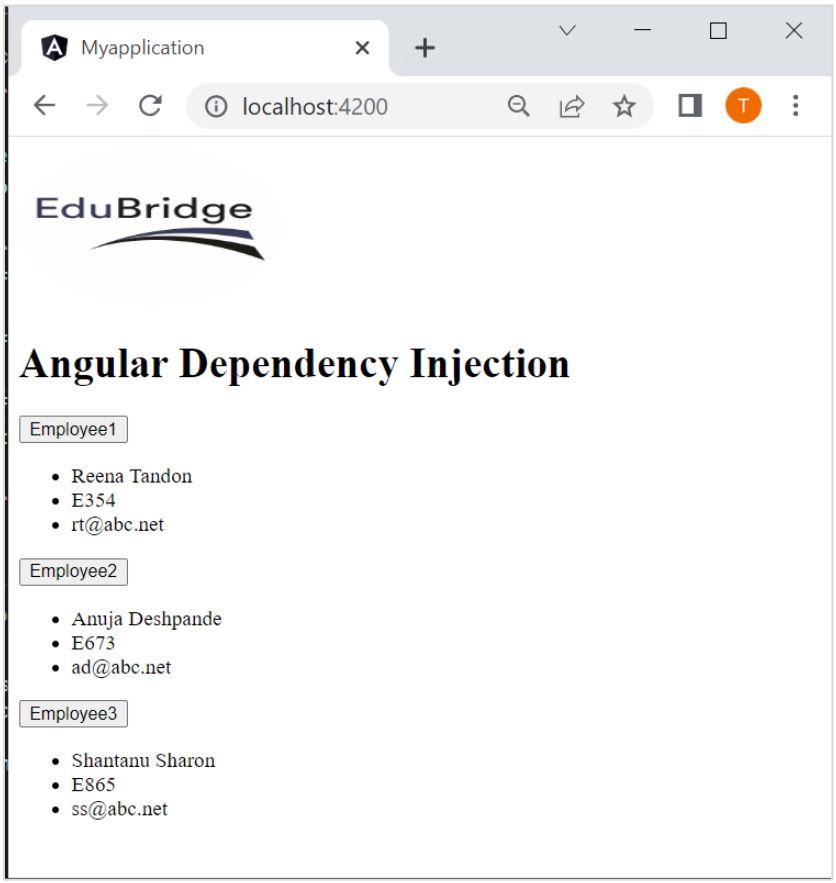
We've highlighted the code for better visibility.

Now that you've learned how to create the service and inject it into the components, let's move on to the UI of the application. We're creating a button for each employee.

Step 4: In the component.html file, we are creating an unordered list. We are also using the *ngFor directive to loop over the record fields.

```
<button type="button" name="button"
(click)='getInfoFromService1()'>Employee1</button>

<ul class="list-group">

    <li *ngFor = "let info of infoReceived1" class="list-group-
info">{{info}}</li>

</ul>

<button type="button" name="button"
(click)='getInfoFromService2()'>Employee2</button>

<ul class="list-group">

    <li *ngFor = "let info of infoReceived2" class="list-group-
info">{{info}}</li>

</ul>

<button type="button" name="button"
(click)='getInfoFromService3()'>Employee3</button>

<ul class="list-group">

    <li *ngFor = "let info of infoReceived3" class="list-group-
info">{{info}}</li>

</ul>
```

Once you run the application using the ng serve command, the output looks like this:



So, this is how you could inject dependencies to your services, classes, or modules. With this, we are concluding the demo tutorial.

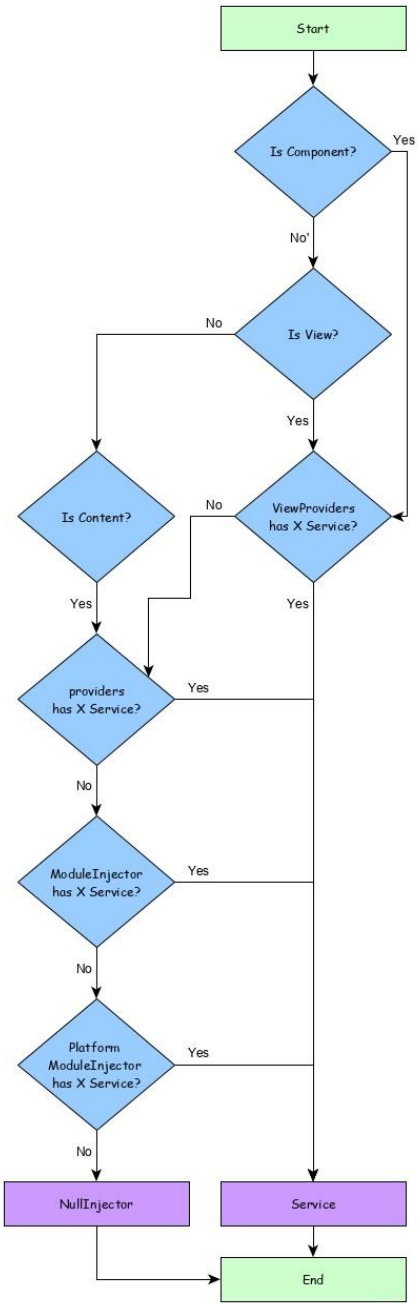
Activity Thirteen

Follow the instructions:

1. Open the Visual Studio code.
2. Understand the concept of Dependency Injections and create an application shown in the example given above.

Resolve Angular service

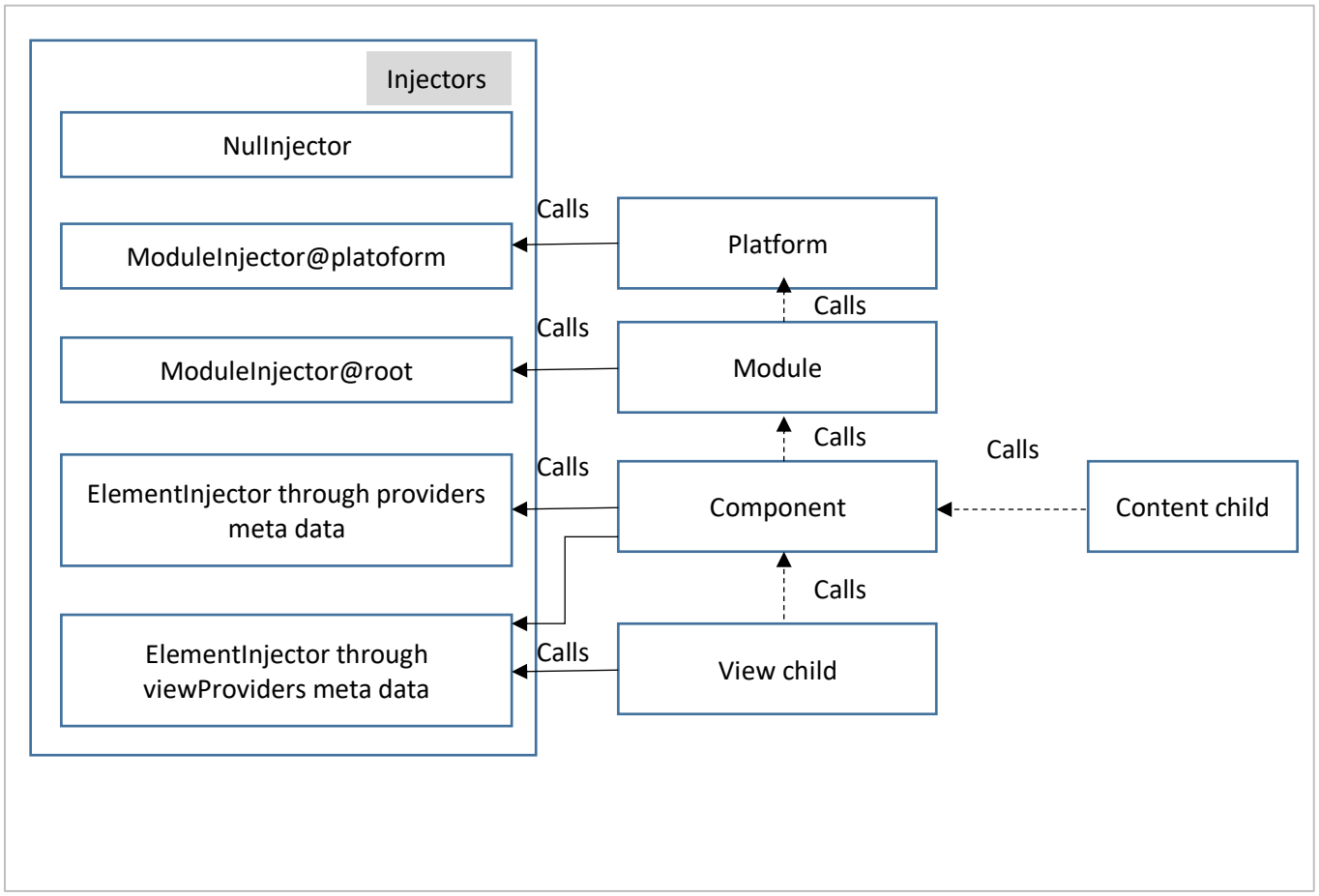
Let us see how a component can resolve a service using the below flow diagram.



Here,

- First, component tries to find the service registered using viewProviders meta data.
- If not found, component tries to find the service registered using providers meta data.
- If not found, Component tries to find the service registered using ModuleInjector
- If not found, component tries to find the service registered using PlatformInjector
- If not found, component tries to find the service registered using NullInjector, which always throws error.

The hierarchy of the Injector along with work flow of the resolving the service is as follows –



Module 5: Template-Driven and Reactive Forms

Forms are an integral part of a web application. Practically every application comes with forms to be filled in by the users. Angular forms are used to log in, update a profile, enter sensitive information, and perform many other data-entry tasks.

Types of Angular Forms

There are two types of form building supported by Angular Forms.

- Template-Driven Approach
- Reactive Approach

Template-Driven Approach

- In this method, the conventional form tag is used to create forms. Angular automatically interprets and creates a form object representation for the tag.
- Controls can be added to the form using the ngModel tag. Multiple controls can be grouped using the ngControlGroup module.
- A form value can be generated using the “form.value” object. Form data is exported as JSON values when the submit method is called.
- Basic HTML validations can be used to validate the form fields. In the case of custom validations, directives can be used.
- Arguably, this method is the simplest way to create an Angular App.

Reactive Form Approach

- This approach is the programming paradigm oriented around data flows and propagation of change.
- With Reactive forms, the component directly manages the data flows between the form controls and the data models.
- Reactive forms are code-driven, unlike the template-driven approach.

- Reactive forms break from the traditional declarative approach.
- Reactive forms eliminate the anti-pattern of updating the data model via two-way data binding.
- Typically, Reactive form control creation is synchronous and can be unit tested with synchronous programming techniques.

Key differences

The following table summarizes the key differences between reactive and template-driven forms.

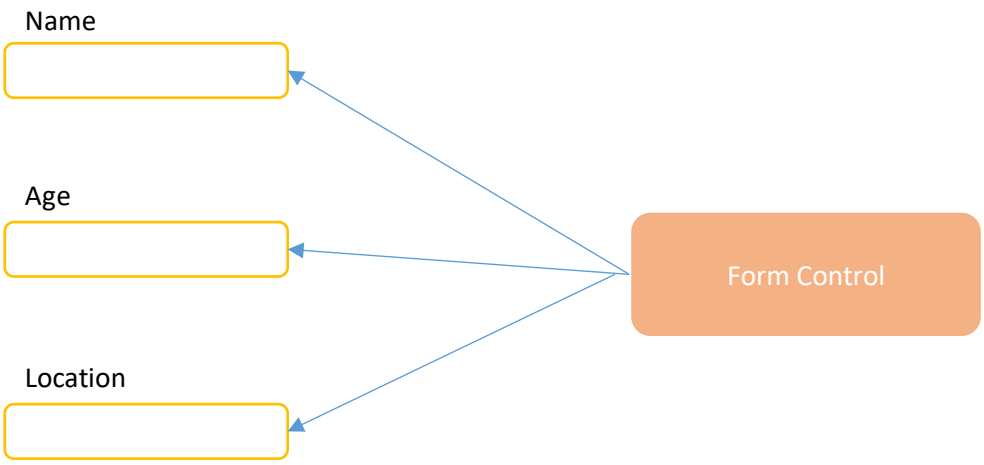
	REACTIVE	TEMPLATE-DRIVEN
Setup of form model	Explicit, created in component class	Implicit, created by directives
Data model	Structured and immutable	Unstructured and mutable
Data flow	Synchronous	Asynchronous
Form validation	Functions	Directives

Common form foundation classes

Both reactive and template-driven forms are built on the following base classes.

BASE CLASSES	DETAILS
FormControl	Tracks the value and validation status of an individual form control.
FormGroup	Tracks the same values and status for a collection of form controls.
FormArray	Tracks the same values and status for an array of form controls.
ControlValueAccessor	Creates a bridge between Angular FormControl instances and built-in DOM elements.

Form Control



Form Control is a class that enables validation. For each input field, an instance of this class is created. These instances help check the values of the field and see if they are touched, untouched, dirty, pristine, valid, invalid, and so on.

Explanation:

Consider a simple text input box:

```
First Name : <input type="text" name="firstname" />
```

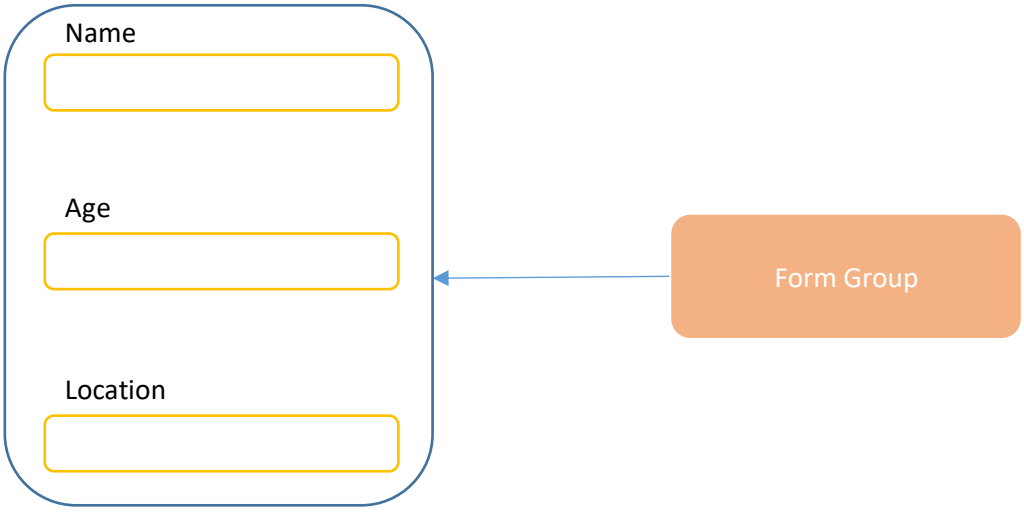
As a developer, you would like to know the current value in the text box. You would also be like to know if the value is valid or not. If the user has changed the value(dirty) or is it unchanged. You would like to be notified when the user changes its value.

The FormControl is an object that encapsulates all the information related to the single input element. It Tracks the value and validation status of each of these controls.

The FormControl is just a class. A FormControl is created for each form field. We can refer to them in our component class and inspect its properties and methods

We can use FormControl to set the value of the Form field. Find the status of form field like (valid/invalid, pristine/dirty, touched/untouched), etc. You can add validation rules to it.

Form Group



Explanation:

```
<form>
  First Name: <input type="text" name="firstname" />
  Last Name: <input type="text" name="lastname" />
  Email: <input type="text" name="email" />
</form>
```

We create a FormControl for each of these input fields. It tracks the value & validity of these elements. All of the above input fields are represented as the separate FormControl. If we wanted to check the validity of our form, we have to check the validity of each and every FormControl for validity. Imagine a form having a large no of fields. It is cumbersome to loop over large no of FormControls and check for validity.

The **FormGroup** solves this issue by providing a wrapper around a collection of FormControls It encapsulates all the information related to a group of form elements. It Tracks the value and validation status of each of these controls. We can use it to check the validity of the elements. set its values & listen for change events, add and run validations on the group, etc

The FormGroup is just a class. We create a FormGroup to organize and manage the related elements. For Example, form elements like address, city, state, pin code etc can be grouped together as a single FormGroup. It makes it easier to manage them. A FormGroup aggregates the values of each child FormControl into one object, with each control name

as the key. It calculates its status by reducing the status values of its children. For example, if one of the controls in a group is invalid, the entire group becomes invalid.

Reactive Forms

In Reactive Forms approach, it is our responsibility to build the Model using FormGroup, FormControl and FormArray.

To use FormControl, first, we need to import the FormControl from the @angular/forms

```
import { FormGroup, FormControl, Validators } from '@angular/forms'
```

Then create the top-level FormGroup. The first argument to FormGroup is the collection of FormControl. They are added using the FormControl method as shown below.

```
reactiveForm = new FormGroup({
  firstname: new FormControl('',[Validators.required]),
  lastname: new FormControl(),
  email: new FormControl(),
})
```

Or you can make use of the FormBuilder API

```
this.reactiveForm = this.formBuilder.group({
  firstname: ['',[Validators.required]],
  lastname: [''],
  email: [''],
});
```

Bind the form element with the template using the formControlName directive as shown below

```
<form [formGroup]="reactiveForm" (ngSubmit)="onSubmit()" novalidate>

  <p>
    <label for="firstname">First Name </label>
    <input type="text" id="firstname" name="firstname"
formControlName="firstname">
  </p>

  <p>
    <label for="lastname">Last Name </label>
    <input type="text" id="lastname" name="lastname" formControlName="lastname">
  </p>

  <p>
    <label for="email">Email </label>
    <input type="text" id="email" name="email" formControlName="email">
  </p>

  <p>
    <button type="submit">Submit</button>
  </p>

</form>
```

Template-driven forms

In template-driven forms, the FormControl is defined in the Template. The <Form> directive creates the top-level FormGroup. We use the ngModel directive on each Form element, which automatically creates the FormControl instance.

```
<form #templateForm="ngForm" (ngSubmit)="onSubmit(templateForm)" novalidate>

  <p>
    <label for="firstname">First Name</label>
    <input type="text" name="firstname" ngModel>
  </p>

  <p>
    <label for="lastname">Last Name</label>
    <input type="text" name="lastname" ngModel>
  </p>

  <p>
    <label for="email">Email </label>
    <input type="text" id="email" name="email" ngModel>
  </p>

  <p>
    <button type="submit">Submit</button>
  </p>

</form>
```

Activity Thirteen

Follow the instructions:

1. Open the Visual Studio code.
2. Create two components Personal-details and Registration.
3. Create a form in Personal-details component using template driven approach as shown below:

4. Create a form in Register component using reactive form approach as shown below:

Form State and Input State

Input fields have the following states:

- $untouched The field has not been touched yet
- $touched The field has been touched
- $pristine The field has not been modified yet
- $dirty The field has been modified
- $invalid The field content is not valid
- $valid The field content is valid

They are all properties of the input field, and are either true or false.

Forms have the following states:

- $pristine No fields have been modified yet
- $dirty One or more have been modified
- $invalid The form content is not valid
- $valid The form content is valid
- $submitted The form is submitted

They are all properties of the form, and are either true or false.

You can use these states to show meaningful messages to the user. Example, if a field is required, and the user leaves it blank, you should give the user a warning.

Control Status

The FormGroup tracks the validation status of all the FormControls, which is part of the FormGroup. That also includes the status of nested FormGroup or FormArray. If any of the control becomes invalid, then the entire FormGroup becomes invalid.

The following is the list of status-related properties

status

```
status: string
```

The Angular runs validation checks, whenever the value of a form control changes. Based on the result of the validation, the FormGroup can have four possible states.

- VALID: All the controls of the FormGroup has passed all validation checks.
- INVALID: At least one of the control has failed at least one validation check.
- PENDING: This Group is in the midst of conducting a validation check.
- DISABLED: This FormGroup is exempt from validation checks

```
//reactive forms
this.reactiveForm.status
```

Valid

A FormGroup is valid when it has passed all the validation checks and the FormGroup is not disabled.

invalid

invalid: boolean
A FormGroup is invalid when one of its controls has failed a validation check or the entire FormGroup is disabled.

pending

pending: boolean
A FormGroup is pending when it is in the midst of conducting a validation check.

disabled

disabled: boolean
A FormGroup is disabled when all of its controls are disabled.

enabled

enabled: boolean
A FormGroup is enabled as long one of its control is enabled.

pristine

pristine: boolean
A FormGroup is pristine if the user has not yet changed the value in the UI in any of the controls

dirty

dirty: boolean
A FormGroup is dirty if the user has changed the value in the UI in any one of the control.

touched

touched: boolean
True if the FomGroup is marked as touched. A FormGroup is marked as touched once the user has triggered a blur event on any one of the controls

untouched

untouched: boolean
True if the FormGroup has not been marked as touched. A FormGroup is untouched if the user has not yet triggered a blur event on any of its child controls.

Built-in Validators

We have a few built in validators in Angular:

- **minLength:** Validator that requires controls to have a value of a minimum length.
- **maxLength**: Validator that requires controls to have a value of a maximum length.
- **pattern:** Validator that requires a control to match a regex to its value. You can find more information about regex patterns in the PatternValidator reference.
- **email:** Validator that performs email validation.
- **compose:** is used when more than one validation is needed for the same form field.
- **required**: Validator that requires controls to have a non-empty value. It also validates that the value matches the input type. For example, if the input is of “email” type, then the input will be valid if it's not empty and if the value is of email type.

We can use these in two ways:

1. As functions we can pass to the FormControl constructor in model-driven forms.

```
new FormControl('', Validators.required)
```

The above creates a form control with a required validator function attached

2. As directives in template-driven forms.

```
<input name="fullName" ngModel required>
```

These required, minlength, maxlength and pattern attributes are already in the official HTML specification. They are a core part of HTML and we don't actually need Angular in order to use them. If they are present in a form then the browser will perform some default validation itself. However, we do need a way for Angular to recognise their presence and support the same validation logic in our own Angular forms.

If you remember template-driven forms are just model-driven forms but with the creation of the model driven by the template, they still have an underlying model. Therefore, just like model-driven forms we need to attach a validator function to the underlying model form control. Angular does this by secretly creating special validator directives which have selectors matching required, minlength, maxlength and pattern.

So, if you have imported FormsModule into your ngModule then anytime Angular sees a required tag in the HTML it will link it to an instance of a directive called RequiredValidator. This directive validator applies the same Validators.required function as we use in model-driven forms. That's how the built-in validators work.

Grouping Form Controls

We can group various FormControls together. For Example, fields such as street, city, country and Pincode each will have their own FormControl but can be grouped together as an address FormGroup.

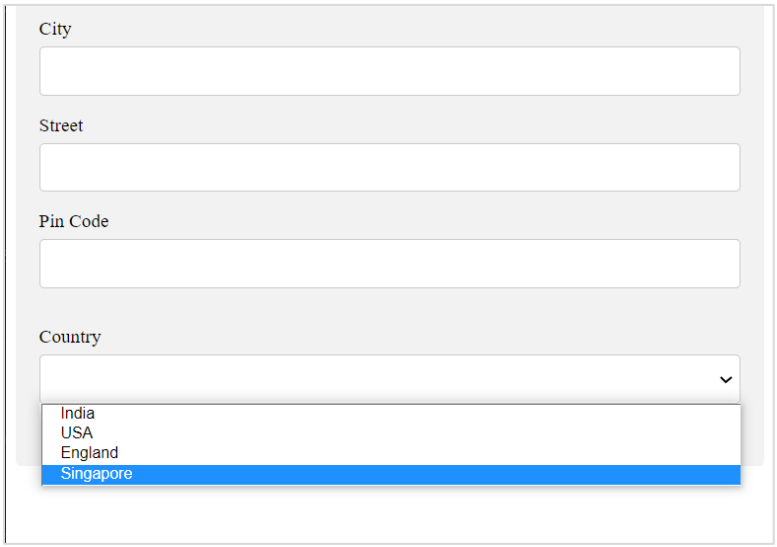
```
contactForm = new FormGroup({
  firstname: new FormControl(),
  lastname: new FormControl(),
  email: new FormControl(),
  gender: new FormControl(),
  isMarried: new FormControl(),
  address:new FormGroup({
    city: new FormControl(),
    street: new FormControl(),
    pincode:new FormControl(),
    country: new FormControl(),
  })
})
```

In the code above, we have created new FormGroup Address and added three form controls i.e city, street, country & Pincode.

In the template use the formGroupName directive to enclose the control using a div element as shown below

```
<div formGroupName="address">

    <div class="form-group">
        <label for="city">City</label>
        <input type="text" class="form-control" name="city" formControlName="city" >
    </div>

    <div class="form-group">
        <label for="street">Street</label>
        <input type="text" class="form-control" name="street"
formControlName="street" >
    </div>

    <div class="form-group">
        <label for="pincode">Pin Code</label>
        <input type="text" class="form-control" name="pincode"
formControlName="pincode">
    </div>

    <p>
      <label for="country">country </label>
      <select id="country" name="country" formControlName="country">
        <option value="1">India</option>
        <option value="2">USA</option>
        <option value="3">England</option>
        <option value="4">Singapore</option>
      </select>
    </p>

  </div>
```

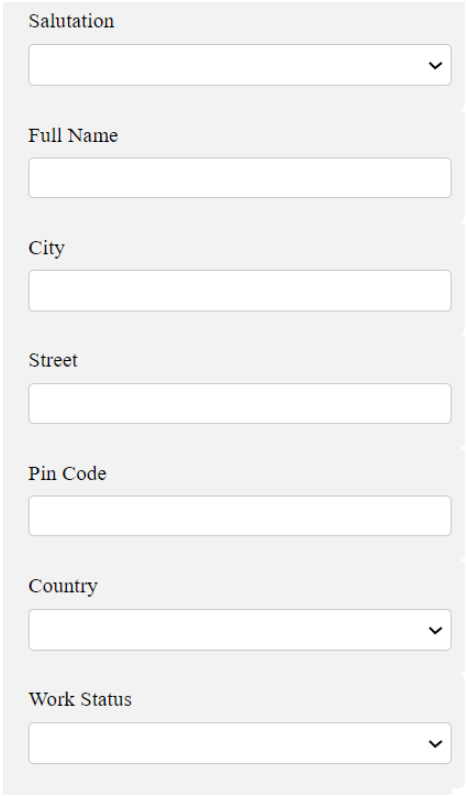
Output:



Activity Fourteen

Follow the instructions:

1. Open the Visual Studio code.
2. create the program as given above in grouping formcontrol topic.
3. Add following labels:
    - Salutation: Mr. Mrs, Ms
    - Full name
    - Work status: Employed, Unemployed, Student, Job Seeker.



Form Builder

The FormBuilder is used to create a big reactive form with minimum code in Angular application. The FormBuilder methods are group(), control() and array() that returns FormGroup, FormControl and FormArray respectively. Using FormBuilder we can directly pass object or array of object of a class to create the form. We need to create form HTML template with formArrayName by assigning the field name of that class.

Suppose we want to create a FormGroup with some fields, then first create a class with those fields and then pass the instance of that class to FormBuilder.group() and it will return FormGroup with form controls named as class fields. After form submit, we can get complete form value as an object of a class. In this way, with the help of FormBuilder, we can create form using object of a class, set values in the form using object of a class and also we can fetch form values as an object of a class after form submit. Now we will provide complete FormBuilder example step-by-step.

FormBuilder creates reactive form with minimum code using FormGroup, FormControl and FormArray. FormBuilder has following methods.

- **group():** Creates FormGroup.
- **control():** Creates FormControl.
- **array():** Creates FormArray.

Find the sample example to create a form with FormBuilder.

```
this.teamForm = this.formBuilder.group({
   teamName: ['', Validators.required ],
   teamManager: '',
   teamDept: this.formBuilder.group(new Department()),
   employees: this.formBuilder.array([])
});
```

Form Validation

Angular offers client-side form validation. Angular monitors the state of the form and input fields (input, textarea, select), and lets you notify the user about the current state. Angular also holds information about whether they have been touched, or modified, or not.

You can use standard HTML5 attributes to validate input, or you can make your own validation functions.

Validating input in template-driven forms

To add validation to a template-driven form, you add the same validation attributes as you would with native HTML form validation. Angular uses directives to match these attributes with validator functions in the framework.

Every time the value of a form control changes, Angular runs validation and generates either a list of validation errors that results in an INVALID status, or null, which results in a VALID status.

You can then inspect the control's state by exporting ngModel to a local template variable. The following example exports NgModel into a variable called name:

```
<input type="text" id="name" name="name" class="form-control" required
minlength="4" appForbiddenName="bob"[(ngModel)]="hero.name" #name="ngModel">

<div *ngIf="name.invalid && (name.dirty || name.touched)" class="alert">

  <div *ngIf="name.errors?.['required']"> Name is required.
  </div>
  <div *ngIf="name.errors?.['minlength']"> Name must be at least 4 characters
long.
  </div>
  <div *ngIf="name.errors?.['forbiddenName']"> Name cannot be Bob.
  </div>

</div>
```

Notice the following features illustrated by the example.

- The <input> element carries the HTML validation attributes: required and minlength. It also carries a custom validator directive, forbiddenName. For more information, see the Custom validators section.
- #name="ngModel" exports NgModel into a local variable called name. NgModel mirrors many of the properties of its underlying FormControl instance, so you can use this in the template to check for control states such as valid and dirty. For a full list of control properties, see the AbstractControl API reference.
    - The *ngIf on the element reveals a set of nested message dives but only if the name is invalid and the control is either dirty or touched.
    - Each nested can present a custom message for one of the possible validation errors. There are messages for required, minlength, and forbiddenName.

Validating input in reactive forms

In a reactive form, the source of truth is the component class. Instead of adding validators through attributes in the template, you add validator functions directly to the form control model in the component class. Angular then calls these functions whenever the value of the control changes.

Validator functions

Validator functions can be either synchronous or asynchronous.

VALIDATOR TYPE	DETAILS
Sync validators	Synchronous functions that take a control instance and immediately return either a set of validation errors or null. Pass these in as the second argument when you instantiate a FormControl.
Async validators	Asynchronous functions that take a control instance and return a Promise or Observable that later emits a set of validation errors or null. Pass these in as the third argument when you instantiate a FormControl.

For performance reasons, Angular only runs async validators if all sync validators pass. Each must complete before errors are set.

Built-in validator functions

You can choose to write your own validator functions, or you can use some of Angular's built-in validators.

The same built-in validators that are available as attributes in template-driven forms, such as required and minlength, are all available to use as functions from the Validators class. For a full list of built-in validators, see the following link-

https://angular.io/api/forms/Validators

Step 1: We need to first import the ReactiveFormsModule:

```
import { ReactiveFormsModule } from '@angular/forms';

@NgModule({
  imports: [BrowserModule, ReactiveFormsModule],
  ...
})
export class AppModule {}
```

Step 2: And can then build our form either using FormControl and FormGroup APIs:

hero-form-reactive.component.ts (validator functions)

```
ngOnInit(): void {
  this.heroForm = new FormGroup({
    name: new FormControl(this.hero.name, [
      Validators.required,
      Validators.minLength(4),
      forbiddenNameValidator(/bob/i) // <-- Here's how you pass in the custom
validator.
    ]),
    alterEgo: new FormControl(this.hero.alterEgo),
    power: new FormControl(this.hero.power, Validators.required)
  });

}

get name() { return this.heroForm.get('name'); }

get power() { return this.heroForm.get('power'); }
```

In this example, the name control sets up two built-in validators —Validators.required and Validators.minLength(4)— and one custom validator, forbiddenNameValidator.

All of these validators are synchronous, so they are passed as the second argument. Notice that you can support multiple validators by passing the functions in as an array.

This example also adds a few getter methods. In a reactive form, you can always access any form control through the get method on its parent group, but sometimes it's useful to define getters as shorthand for the template.

If you look at the template for the name input again, it is fairly similar to the template-driven example.

hero-form-reactive.component.html (name with error msg)

```
<input type="text" id="name" class="form-control"
      formControlName="name" required>

<div *ngIf="name.invalid && (name.dirty || name.touched)"
    class="alert alert-danger">

  <div *ngIf="name.errors?.['required']">
    Name is required.
  </div>
  <div *ngIf="name.errors?.['minlength']">
    Name must be at least 4 characters long.
  </div>
  <div *ngIf="name.errors?.['forbiddenName']">
    Name cannot be Bob.
  </div>
</div>
```

This form differs from the template-driven version in that it no longer exports any directives. Instead, it uses the name getter defined in the component class.

Notice that the required attribute is still present in the template. Although it's not necessary for validation, it should be retained to for accessibility purposes.

Defining custom validators

The built-in validators don't always match the exact use case of your application, so you sometimes need to create a custom validator.

Consider the forbiddenNameValidator function from previous reactive-form examples. Here's what the definition of that function looks like.

shared/forbidden-name.directive.ts (forbiddenNameValidator)

```
export function forbiddenNameValidator(nameRe: RegExp): ValidatorFn {
  return (control: AbstractControl): ValidationErrors | null => {
    const forbidden = nameRe.test(control.value);
    return forbidden ? {forbiddenName: {value: control.value}} : null;
  };
}
```

The function is a factory that takes a regular expression to detect a specific forbidden name and returns a validator function.

In this sample, the forbidden name is "bob", so the validator rejects any hero name containing "bob". Elsewhere it could reject "alice" or any name that the configuring regular expression matches.

The forbiddenNameValidator factory returns the configured validator function. That function takes an Angular control object and returns either null if the control value is valid or a validation error object. The validation error object typically has a property whose name is the validation key, 'forbiddenName', and whose value is an arbitrary dictionary of values that you could insert into an error message, {name}.

Custom async validators are similar to sync validators, but they must instead return a Promise or observable that later emits null or a validation error object. In the case of an observable, the observable must complete, at which point the form uses the last value emitted for validation.

Adding custom validators to reactive forms

In reactive forms, add a custom validator by passing the function directly to the FormControl

reactive/hero-form-reactive.component.ts (validator functions)

```
this.heroForm = new FormGroup({
  name: new FormControl(this.hero.name, [
    Validators.required,
    Validators.minLength(4),
    forbiddenNameValidator(/bob/i) // <-- Here's how you pass in the custom
validator.
  ]),
  alterEgo: new FormControl(this.hero.alterEgo),
  power: new FormControl(this.hero.power, Validators.required)
});
```

Adding custom validators to template driven forms

- In template-driven forms, add a directive to the template, where the directive wraps the validator function. For example, the corresponding ForbiddenValidatorDirective serves as a wrapper around the forbiddenNameValidator.
- Angular recognizes the directive's role in the validation process because the directive registers itself with the NG_VALIDATORS provider, as shown in the following example. NG_VALIDATORS is a predefined provider with an extensible collection of validators.

forbidden-name.directive.ts (providers)

```
providers: [{provide: NG_VALIDATORS, useExisting: ForbiddenValidatorDirective,
multi: true}]
```

forbidden-name.directive.ts (directive)

```
@Directive({
  selector: '[appForbiddenName]',
  providers: [{provide: NG_VALIDATORS, useExisting: ForbiddenValidatorDirective,
multi: true}]
})
export class ForbiddenValidatorDirective implements Validator {
  @Input('appForbiddenName') forbiddenName = '';

  validate(control: AbstractControl): ValidationErrors | null {
    return this.forbiddenName ? forbiddenNameValidator(new
RegExp(this.forbiddenName, 'i'))(control): null;
  }
}
```

template/hero-form-template.component.html (forbidden-name-input)

```
<input type="text" id="name" name="name" class="form-control"
      required minlength="4" appForbiddenName="bob"
      [(ngModel)]="hero.name" #name="ngModel">
```

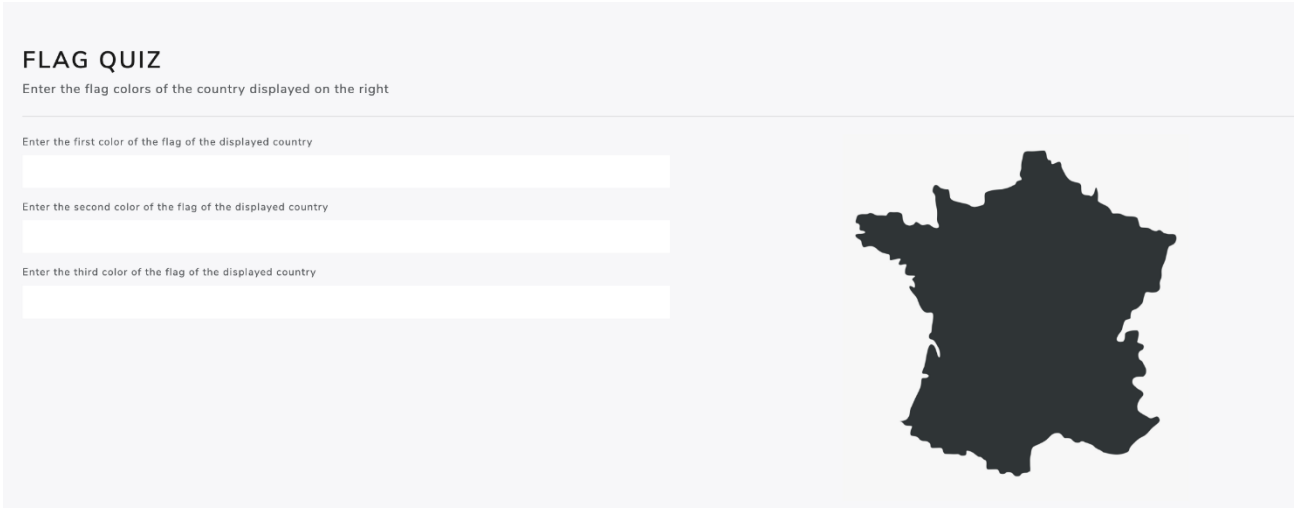
Activity Fifteen

Follow the instructions:

1. Open the Visual Studio code.
2. Follow the steps given in example 1 and perform the program.

Example 1 (Adding custom validators to reactive forms):

Let's say we want to implement a simple quiz where you need to guess the right colors of the flag of a country.



Do you know which country is displayed here?

Yes, it's France. The flag of France is blue, white, and red ?. We can now use custom validators to validate that the first input field must be blue, the second one white, and the third one red.

Let's go ahead and implement some custom Validators. Let's start with the Validator that validates that we entered blue as a color.

```
import {AbstractControl, ValidatorFn} from '@angular/forms';

export function blue(): ValidatorFn {
    return (control: AbstractControl): { [key: string]: any } | null =>
        control.value?.toLowerCase() === 'blue'
            ? null : {wrongColor: control.value};
}
```

The validator itself is just a function that accepts an AbstractControl and returns an Object containing the validation error or null if everything is valid. Once the blue validator is finished, we can import it in our Component and use it.

```
constructor(private fb: FormBuilder) {
}

ngOnInit(){
    this.flagQuiz = fb.group({
        firstColor: new FormControl('', blue()),
        secondColor: new FormControl(''),
        thirdColor: new FormControl('')
    }, {updateOn: 'blur'});
}
```

The firstColor input field is now validated. If it doesn't contain the value blue our validator will return an error object with the key wrongColor and the value we entered. We can then add the following lines in our HTML to print out a sweet error message.

```
<div *ngIf="flagQuiz.get('firstColor').errors?.wrongColor"
     class="invalid-feedback">
    Sorry, {{flagQuiz.get('firstColor')?.errors?.wrongColor}} is wrong
</div>
```

To make our custom validator accessible for template-driven forms, we need to implement our validator as a directive and provide it as NG_VALIDATORS.

```
@Directive({
    selector: '[blue]',
    providers: [{
        provide: NG_VALIDATORS,
        useExisting: BlueValidatorDirective,
        multi: true
    }]
})
export class BlueValidatorDirective implements Validator {

    validate(control: AbstractControl): { [key: string]: any } | null {
        return blue()(control);
    }
}
```

The Directive implements the Validator interface from @angular/forms which forces us to implement the validate method.

Note the similarity of the signature of our validate function and the validate method we implemented here. They are the same. Both accept an AbstractControl and return either an error object or null.

Of course, it doesn't make sense to duplicate the validation logic. Therefore, we are going to reuse our validation function in the validate function of our Directive.

```
import {
    AbstractControl,
    NG_VALIDATORS,
    Validator,
    ValidatorFn
} from '@angular/forms';
import {Directive} from '@angular/core';

export function blue(): ValidatorFn {
    return (control: AbstractControl): { [key: string]: any } | null =>
        control.value?.toLowerCase() === 'blue'
            ? null : {wrongColor: control.value};
}

@Directive({
    selector: '[blue]',
    providers: [{
        provide: NG_VALIDATORS,
        useExisting: BlueValidatorDirective,
        multi: true
    }]
})
export class BlueValidatorDirective implements Validator {

    validate(control: AbstractControl): { [key: string]: any } | null {
        return blue()(control);
    }
}
```

In our app.module.ts we can now add our Directive to the declarations and start to use it in our templates.

```
<label for="firstColor">
    Enter the first color of the flag of France
</label>
<input #firstColor="ngModel" blue name="firstColor" class="form-control"
id="firstColor" [(ngModel)]="flagQuizAnswers.firstColor" type="text"/>
<div *ngIf="firstColor.errors?.wrongColor" class="invalid-feedback">
    Sorry, {{firstColor?.errors?.wrongColor}} is wrong
</div>
```

We created a custom validator that is usable within reactive forms and template-driven forms. With the same approach, we could now also implement a validator for the other colors white and red.

Example 2 (Template Driven form):

Template driven forms is created using directives in the template. It is mainly used for creating a simple form application. Let's understand how to create template driven forms in brief.

Configure Forms

Before understanding forms, let us learn how to configure forms in an application. To enable template driven forms, first we need to import FormsModule in app.module.ts. It is given below –

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

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { FormcomponentComponent } from './formcomponent/formcomponent.component';

@NgModule({
  declarations: [
    AppComponent,
    FormcomponentComponent
  ],
  imports: [
```

```
    BrowserModule,
    AppRoutingModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Once, FormsModule is imported, the application will be ready for form programming.

Create simple form

Let us create a sample application (template-form-app) to learn the template driven form.

Open command prompt and create new Angular application using below command –

```
cd /go/to/workspace
ng new template-form-app
cd template-form-app
```

Create a test component using Angular CLI as mentioned below –

```
ng generate component FormComponent
```

The above create a new component and the output is as follows –

```
CREATE src/app/test/test.component.scss (0 bytes)
CREATE src/app/test/test.component.html (19 bytes)
CREATE src/app/test/test.component.spec.ts (614 bytes)
CREATE src/app/test/test.component.ts (262 bytes)
UPDATE src/app/app.module.ts (545 bytes)
```

Let's create a simple form to display user entered text.

Add the below code in test.component.html file as follows –

```
<div class="container">
```

```
    <h1>User Registration</h1>
    <form>
        <div class="form-group">
            <label for="firstname">First Name</label><br>
            <input type="text" name="firstname" class = "form-control" ngModel>
        </div>
        <pre></pre>
        <div class="form-group">
            <label for="lastname">Last Name</label><br>
            <input type="text" name="lastname" class = "form-control"    ngModel>
        </div>
        <pre></pre>
        <div class="form-group">
            <label for="email">Email ID</label><br>
            <input type="text" name="email" class = "form-control" ngModel>
        </div>
        <pre></pre>
        <div class="form-group">
            <label for="password">Password</label><br>
            <input type="text" name="password" class = "form-control" ngModel>
        </div>
        <pre></pre>
        <button class="submit" type="submit">Submit</button>
    </form>
    </form>
```

Here, we used ngModel attribute in input text field.

Create Submit() method inside formcomponent.component.ts file as shown below

```
import { Component, OnInit } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { NgModel } from '@angular/forms';
@Component({
  selector: 'app-formcomponent',
  templateUrl: './formcomponent.component.html',
  styleUrls: ['./formcomponent.component.css']
})
```

```
export class FormcomponentComponent {
  submit(){
    console.log("Form Submitted !")
  }
}
```

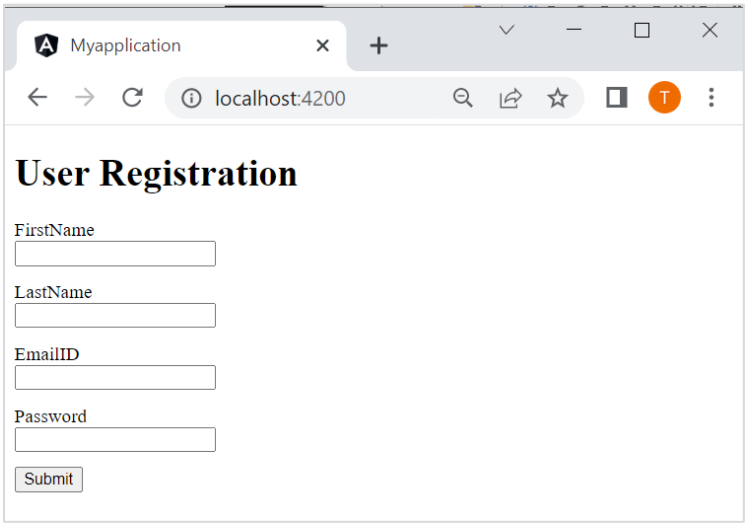
Open app.component.html and change the content as specified below –

```
<app-formcomponent></app-formcomponent>
```

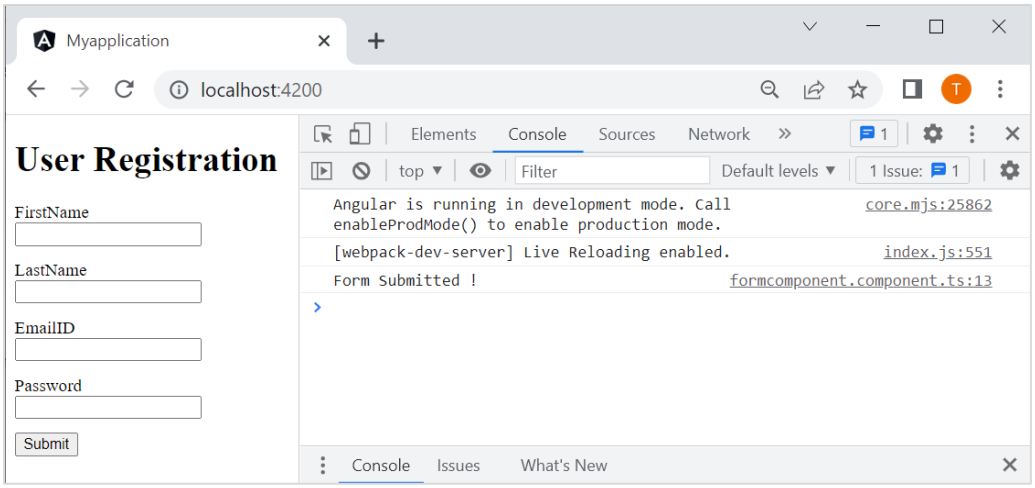
Finally, start your application (if not done already) using the below command –

```
ng serve
```

Now, run your application and you could see the below response –



Type the text in the text field and enter submit. The Submit() function will be called and the successful submission of the form will sent as an argument. The Submit() function will show the text as form submitted in the console as shown below-–



Example 2 (Reactive form):

Reactive Forms is created inside component class so it is also referred as model driven forms. Every form control will have an object in the component and this provides greater control and flexibility in the form programming. Reactive Form is based on structured data model. Let's understand how to use Reactive forms in angular.

Configure Reactive forms

To enable reactive forms, first we need to import ReactiveFormsModule in app.module.ts. It is defined below

```
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { BrowserModule } from '@angular/platform-browser';
import { ReactiveFormsModule } from '@angular/forms';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { FormcomponentComponent } from './formcomponent/formcomponent.component';

@NgModule({
  declarations: [
    AppComponent,
    FormcomponentComponent
  ],
```

```
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule,
    ReactiveFormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Open command prompt and create new Angular application using below command –

```
cd /go/to/workspace
ng new reactive-form-app
cd reactive-form-app
```

Create a test component using Angular CLI as mentioned below –

```
ng generate component formcomponent
```

The above create a new component and the output is as follows –

```
CREATE src/app/test/test.component.scss (0 bytes)
CREATE src/app/test/test.component.html (19 bytes)
CREATE src/app/test/test.component.spec.ts (614 bytes)
CREATE src/app/test/test.component.ts (262 bytes)
UPDATE src/app/app.module.ts (545 bytes)
```

Let's create a simple form to display user entered text.

We need to import FormGroup, FormControl classes in FormComponent.

Example:

We are creating a 'User Registration' form consisting of four fields, viz, Firstname, Lastname, Email ID, and Password.

Step 1: create a component using the Angular CLI and provide the name of your choice.

In this case, we've created a component called "formComponent." In the template-driven approach, the form's module needs to be imported into the app.module.ts file.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { NewComponentComponent } from './components/new-component/new-component.component';
import { FormComponentComponent } from './form-component/form-component.component';

@NgModule({
  declarations: [
    AppComponent,
    NewComponentComponent,
    FormComponentComponent,
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

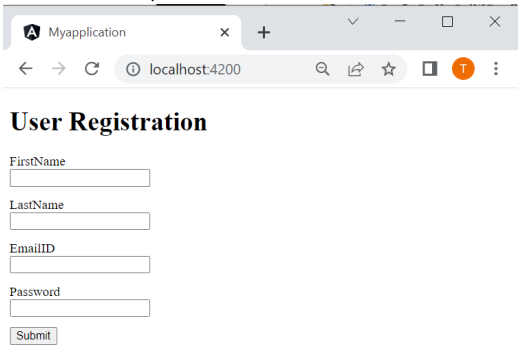
Step 2: Form Creation

Create a form tag within a div tag to create the four fields.

```
<div class="container">

    <h1>User Registration</h1>

    <form>
        <div class="form-group">
           <label for="firstname">First Name</label><br>
```

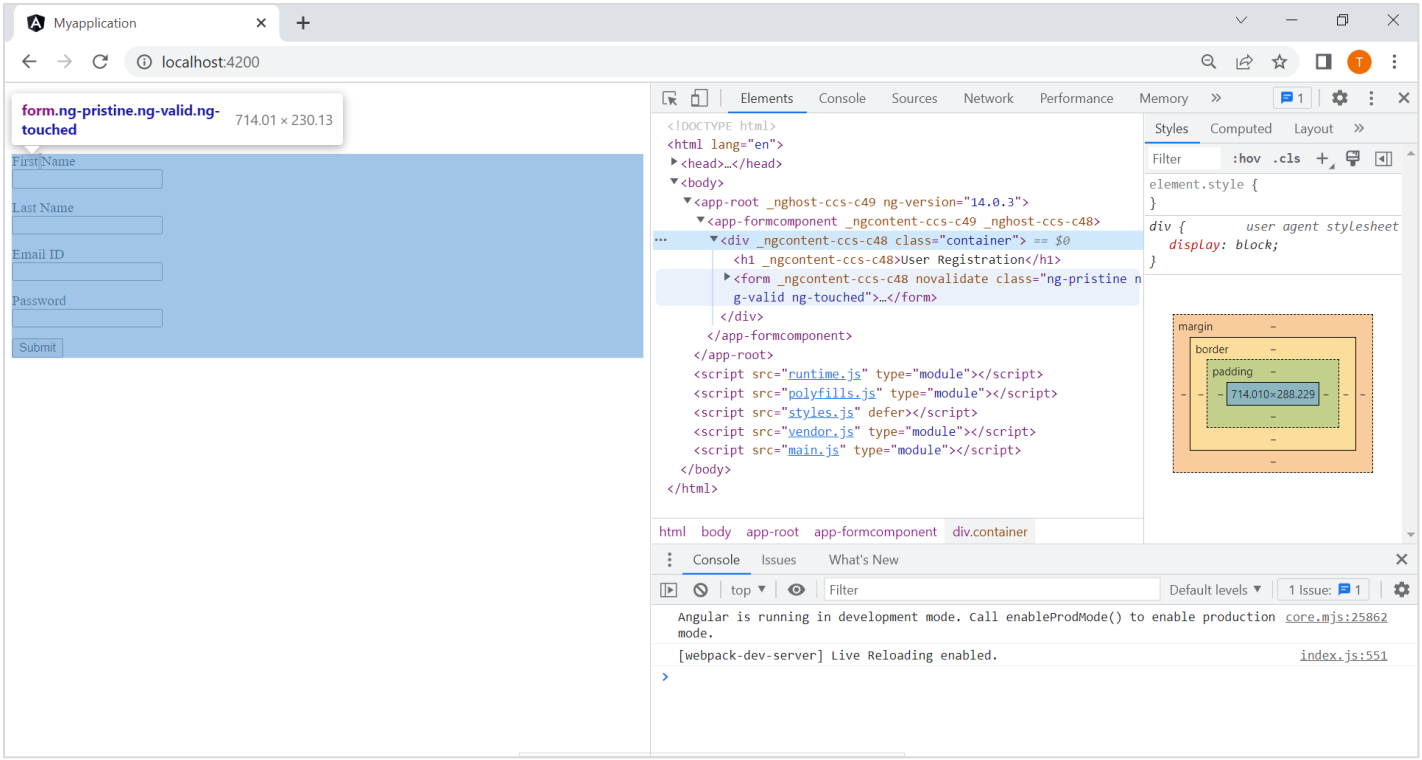
```
            <input type="text" name="firstname" class = "form-control" ngModel>
        </div>
        <pre></pre>
        <div class="form-group">
            <label for="lastname">Last Name</label><br>
            <input type="text" name="lastname" class = "form-control"    ngModel>
        </div>
        <pre></pre>
        <div class="form-group">
            <label for="email">Email ID</label><br>
            <input type="text" name="email" class = "form-control" ngModel>
        </div>
        <pre></pre>
        <div class="form-group">
            <label for="password">Password</label><br>
            <input type="text" name="password" class = "form-control" ngModel>
        </div>
        <pre></pre>
        <form (ngSubmit)="submit()">
        <button class="submit" type="submit">Submit</button>
    </form>
    </form>
```

Once served, the browser looks like this:



Step 2: Adding Angular Form Controls

When you inspect the form, classes like ng-untouched, ng-pristine, and ng-valid are added. This indicates that Angular has recognized the form tag and has added the classes to the form as well as the fields. By adding the ngModel directive in the input tag, form controls are added to every input field.



The form has an output property attached to it called ngSubmit. This property can be bound with a method called "Submit()." Once submitted, this method is called.

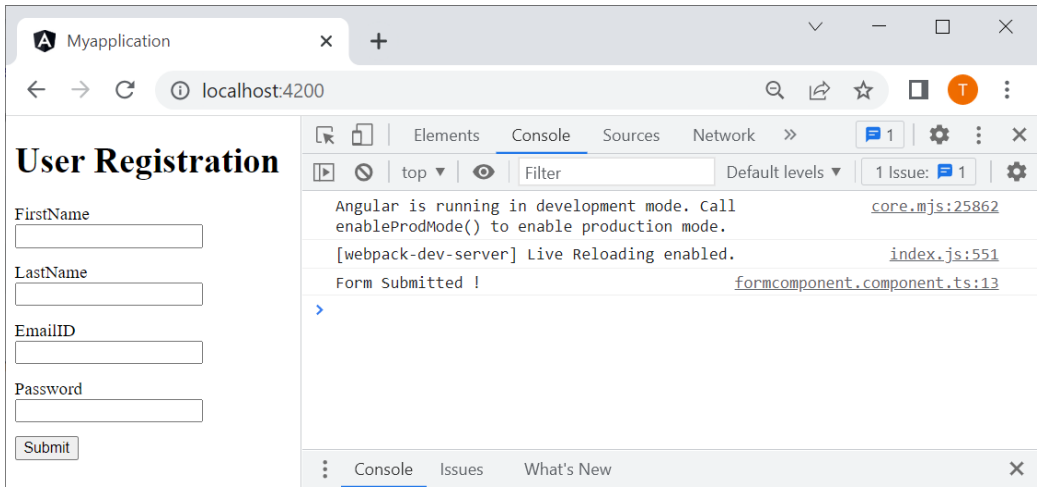
<form (ngSubmit)="submit()">

The submit method can be defined in the component, i.e., form-component.component.ts file.

```
import { Component, OnInit } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { NgModel } from '@angular/forms';
@Component({
  selector: 'app-formcomponent',
  templateUrl: './formcomponent.component.html',
  styleUrls: ['./formcomponent.component.css']
})
export class FormcomponentComponent {
  submit(){
```

```
    console.log("Form Submitted !")
  }
}
```

The output is seen on the console:

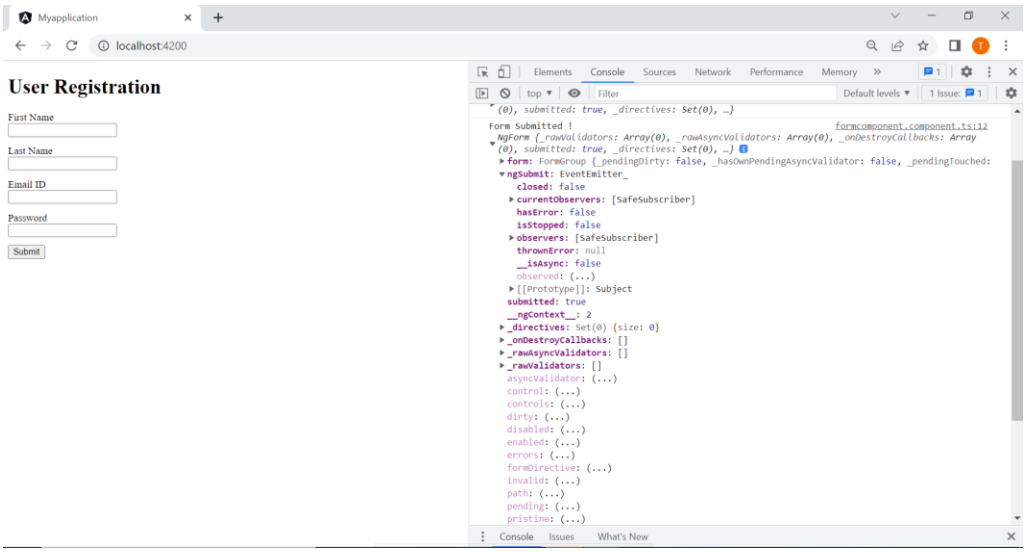


Step3: Getting the JavaScript Object Representation

To generate the JavaScript Representation, another directive called NgForm is assigned to a template variable.

<form #login = "ngForm" (ngSubmit)="submit(login)">

Consequently, update the submit() method in the .ts file as well.



Once submitted, the message is printed and along with it, the Ngform object is obtained indicating the JavaScript Representation of the form. Getting this representation makes form validation easy.

If you observe closely, the value object includes the form controls for the input fields.

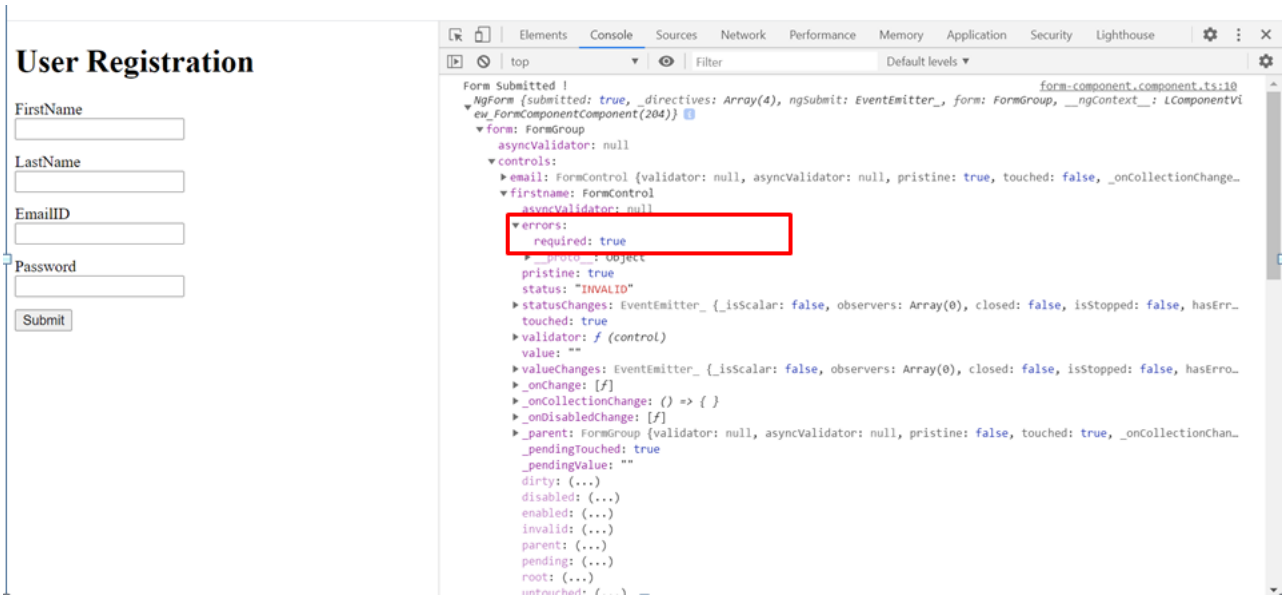
Step4: Angular Form Validation

One way to ensure that all the fields are filled correctly is by disabling the submit button in case the fields aren't filled. Apart from this, you can also specify certain properties in your input tag for the corresponding input field.

Let's say, that the fields can't be empty and the form only accepts names with a minimum length of 2 and a maximum length of 7.

<input required minlength="2" maxlength="7" type="text" name="firstname" class = "form-control" ngModel>

To check this, we've submitted the form without filling the FirstName field and evidently, an error can be seen.



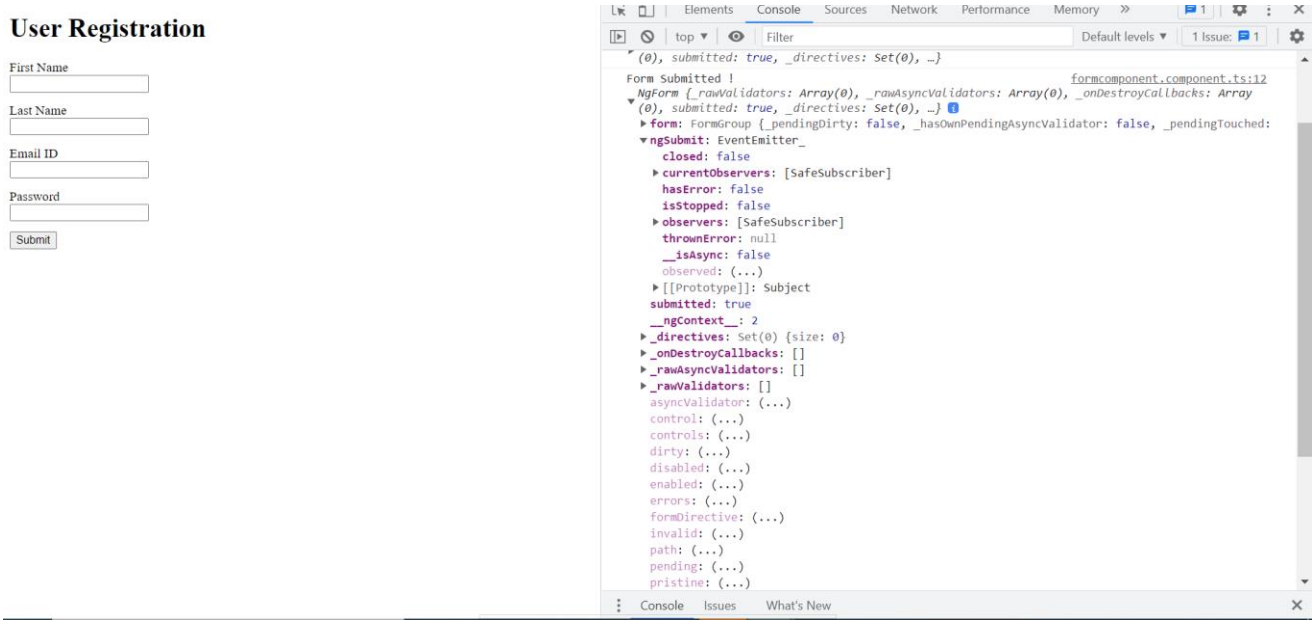
So you can leverage the form control objects to ensure field validation and display a message when an error occurs.

To do that, you need to access the form control objects and for that, create a template variable for the field and assign it to the form control object.

<input required minlength="2" maxlength="7" type="text" name="firstname" class = "form-control" #name="ngModel" ngModel>

Here, the variable name receives the form control object.

When submitted without filling the FirstName field, the invalid property is set to true.



This helps us use this property to alert the user with the help of the simple ‘if’ logic.

Module 6: Components Deep Dive / Routing

Component Life Cycle Hooks

A component instance has a lifecycle that starts when Angular instantiates the component class and renders the component view along with its child views. The lifecycle continues with change detection, as Angular checks to see when data-bound properties change, and updates both the view and the component instance as needed. The lifecycle ends when Angular destroys the component instance and removes its rendered template from the DOM. Directives have a similar lifecycle, as Angular creates, updates, and destroys instances in the course of execution.

Your application can use lifecycle hook methods to tap into key events in the lifecycle of a component or directive to initialize new instances, initiate change detection when needed, respond to updates during change detection, and clean up before deletion of instances.

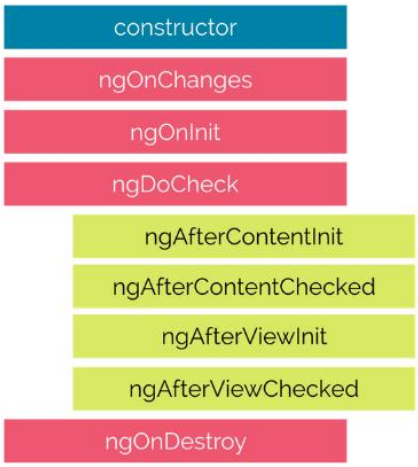
Phases

A component in Angular has a life-cycle, a number of different phases it goes through from birth to death.

We can hook into those different phases to get some pretty fine-grained control of our application.

To do this we add some specific methods to our component class which get called during each of these life-cycle phases, we call those methods hooks.

The hooks are executed in this order:



These phases are broadly split up into phases that are linked to the component itself and phases that are linked to the children of that component.

Hooks for the Component

- **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.
- **ngOnChanges**
  - The Angular invokes ngOnChanges life cycle hook whenever any data-bound input property of the component or directive changes. Initializing the Input properties is the first task that angular carries during the change detection cycle. And if it detects any change in property, then it raises the ngOnChanges hook. It does so during every change detection cycle. This hook is not raised if change detection does not detect any changes.
  - Input properties are those properties, which we define using the @Input decorator. It is one of the ways by which a parent communicates with the child component.

In the following example, the child component declares the property message as the input property

```
@Input() message:string
```

The parent can send the data to the child using the property binding as shown below.

```
<app-child [message]="message">
</app-child>
```

The change detector checks if such input properties of a component are changed by the parent component. If it is then it raises the ngOnChanges hook.

- **ngOnInit**
  - The Angular raises the ngOnInit hook, after it creates the component and updates its input properties. It raises it after the ngOnChanges hook.
  - This hook is fired only once and immediately after its creation (during the first change detection).
  - This is a perfect place where you want to add any initialisation logic for your component. Here you have access to every input property of the component. You can use them in http get requests to get the data from the back end server or run some initialization logic etc.
  - But note that none of child components or projected content are available at this point. Hence any properties we decorate with @ViewChild, @ViewChildren , @ContentChild & @ContentChildren will not be available to use.
- **ngDoCheck**
  - The Angular invokes the ngDoCheck hook event during every change detection cycle. This hook is invoked even if there is no change in any of the properties.
  - Angular invokes it after the ngOnChanges & ngOnInit hooks.
  - Use this hook to Implement a custom change detection, whenever Angular fails to detect the changes made to Input properties. This hook is particularly useful when you opt for the Onpush change detection strategy.
  - The Angular ngOnChanges hook does not detect all the changes made to the input properties.
- **ngOnDestroy**
  - This hook is called just before the Component/Directive instance is destroyed by Angular
  - You can Perform any cleanup logic for the Component here. This is the correct place where you would like to Unsubscribe Observables and detach event handlers to avoid memory leaks.

Hooks for the Component's Children

These hooks are only called for components and not directives.

- **ngAfterContentInit**

- ngAfterContentInit Life cycle hook is called after the Component's projected content has been fully initialized. Angular also updates the properties decorated with the ContentChild and ContentChildren before raising this hook. This hook is also raised, even if there is no content to project.
- The content here refers to the external content injected from the parent component via Content Projection.

The Angular Components can include the ng-content element, which acts as a placeholder for the content from the parent as shown below.

```
<h2>Child Component</h2>
<ng-content></ng-content>   <!-- placehodler for content from parent -->
```

Parent injects the content between the opening & closing element. Angular passes this content to the child component.

```
<h1>Parent Component</h1>
<app-child> This <b>content</b> is injected from parent</app-child>
```

During the change detection cycle, Angular checks if the injected content has changed and updates the DOM.

This is a component only hook.

- **ngAfterContentChecked**
  - ngAfterContentChecked Life cycle hook is called during every change detection cycle after Angular finishes checking of component's projected content. Angular also updates the properties decorated with the ContentChild and ContentChildren before raising this hook. Angular calls this hook even if there is no projected content in the component
  - This hook is very similar to the ngAfterContentInit hook. Both are called after the external content is initialized, checked & updated. Only difference is that ngAfterContentChecked is raised after every change detection cycle. While ngAfterContentInit during the first change detection cycle.
  - This is a component only hook.

- **ngAfterViewInit**
  - ngAfterViewInit hook is called after the Component's View & all its child views are fully initialized. Angular also updates the properties decorated with the ViewChild & ViewChildren properties before raising this hook.
  - The View here refers to the template of the current component and all its child components & directives.
  - This hook is called during the first change detection cycle, where angular initializes the view for the first time
  - At this point all the lifecycle hook methods & change detection of all child components & directives are processed & Component is completely ready
  - This is a component only hook.
- **ngAfterViewChecked**
  - The Angular fires this hook after it checks & updates the component's views and child views. This event is fired after the ngAfterViewInit and after that during every change detection cycle
  - This hook is very similar to the ngAfterViewInit hook. Both are called after all the child components & directives are initialized and updated. Only difference is that ngAfterViewChecked is raised during every change detection cycle. While ngAfterViewInit during the first change detection cycle.
  - This is a component only hook.

Activity Sixteen

Follow the instructions:

1. Open the Visual Studio code.
2. Demonstrate how to add hooks in angular with the following example.

In order to demonstrate how the hooks, let's change the appComponent so it hooks into all the phases.

All we need to do is to add functions to the component class matching the hook names, like so:

app.component.ts

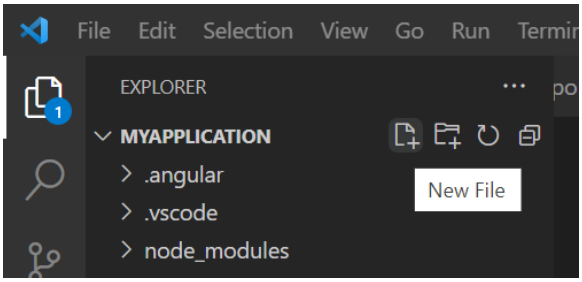
```
import { ChangeDetectionStrategy, Component, VERSION } from "@angular/core";

@Component({
  selector: "my-app",
  changeDetection:ChangeDetectionStrategy.Default,
  template: `
    <h1>Angular Life Cycle Hooks</h1>
    Reference :
    <a
      href=" https://angular.io/guide/lifecycle-hooks"
      >Angular Life Cycle Hooks</a
    >
    <h1>Root Component</h1>
    <br />
    <input
      type="text"
      name="message"
      [(ngModel)]="message"
      autocomplete="off"
    />
    <br />
    <input
      type="text"
      name="content"
      [(ngModel)]="content"
      autocomplete="off"
    />
    <br />
    hide child :
    <input
      type="checkbox"
      name="hideChild"
      [(ngModel)]="hideChild"
      autocomplete="off"
    />
```

```
    <br />
    <br />
    <app-child [message]="message" *ngIf="!hideChild">
      <!-- Injected Content -->
      <b> {{ content }} </b>
    </app-child>
  `
})
export class AppComponent {
  name = "Angular " + VERSION.major;

  message = "Hello";
  content = "Hello";
  hideChild=false;

  constructor() {
    console.log("AppComponent:Contructed");
  }

  ngOnChanges() {
    console.log("AppComponent:ngOnChanges");
  }

  ngOnInit() {
    console.log("AppComponent:ngOnInit");
  }

  ngDoCheck() {
    console.log("AppComponent:DoCheck");
  }

  ngAfterContentInit() {
    console.log("AppComponent:ngAfterContentInit");
  }

  ngAfterContentChecked() {
    console.log("AppComponent:AfterContentChecked");
  }
```

```
  ngAfterViewInit() {
    console.log("AppComponent:AfterViewInit");
  }

  ngAfterViewChecked() {
    console.log("AppComponent:AfterViewChecked");
  }

  ngOnDestroy() {
    console.log("AppComponent:ngOnDestroy");
  }
}
```

- We are listening to all the hooks and logging them to the console.
- There are two form fields message & content. We pass both to the child component. One as input property & the other via content projection
- Using the hideChild form field we an add or remove the ChildComponent from the DOM. We are making use of the ngIf directive.
- We pass the message property to ChildComponent using Property binding.
- The content property is passed as projected content.

Create a typescript file and name it 'customer.ts'



Type the code:

```
export class Customer {
    code:string | undefined
```

```
    name:string | undefined
  }
```

Child.component.ts

```
import { ChangeDetectionStrategy, Component, Input,  OnInit } from '@angular/core';
import { Customer } from './customer';

@Component({
  selector: 'app-child',
  changeDetection:ChangeDetectionStrategy.Default,
  template: `
      <h2>child component</h2>
      <br>
      <!-- Data as a input -->
      Message from Parent via @input {{message}}
      <br><br>
      <!-- Injected Content -->
      Message from Parent via content injection
      <ng-content></ng-content>
      <br><br><br>
      Code :
      <input type="text" name="code" [(ngModel)]="customer.code" autocomplete="off">
      <br><br>
      Name:
      <input type="text" name="name" [(ngModel)]="customer.name" autocomplete="off">

      <app-grand-child [customer]="customer"></app-grand-child>
  `
})
export class ChildComponent {
  @Input() message:string
  customer:Customer = new Customer()
constructor() {
    console.log("  ChildComponent:Contructed");
  }
  ngOnChanges() {
    console.log("  ChildComponent:ngOnChanges");
  }
```

```
  ngOnInit() {
    console.log("  ChildComponent:ngOnInit");
  }
  ngDoCheck() {
    console.log("  ChildComponent:DoCheck");
  }
  ngAfterContentInit() {
    console.log("  ChildComponent:ngAfterContentInit");
  }
  ngAfterContentChecked() {
    console.log("  ChildComponent:AfterContentChecked");
  }
  ngAfterViewInit() {
    console.log("  ChildComponent:AfterViewInit");
  }
  ngAfterViewChecked() {
    console.log("  ChildComponent:AfterViewChecked");
  }
  ngOnDestroy() {
    console.log("  ChildComponent:ngOnDestroy");
  }

}
```

- We are listening to all the hooks
- @Input decorator marks the message as input property. It will receive the data from the parent
- <ng-content></ng-content> is the place holder to receive the projected content from the parent.
- Two forms fields for customer object, which we pass it to the GrandChildComponent

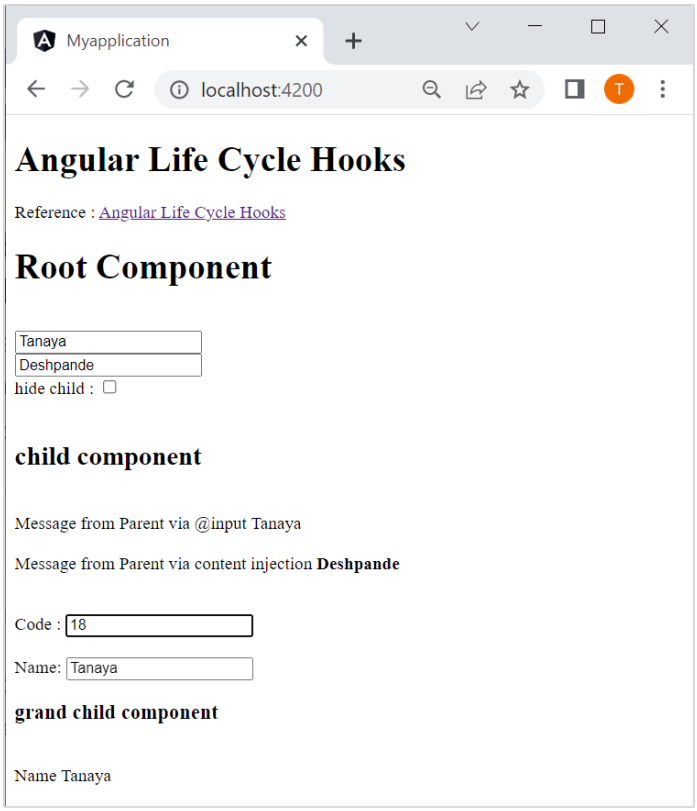
grandchild.component.ts

```
import { ChangeDetectionStrategy, Component, Input,  OnInit } from '@angular/core';
import { Customer } from './customer';

@Component({
  selector: 'app-grand-child',
  changeDetection:ChangeDetectionStrategy.Default,
  template: `
      <h3>grand child component </h3>
      <br>
```

```
      Name {{customer.name}}
  `,
})
export class GrandChildComponent {
  @Input() customer:Customer

  constructor() {
    console.log("    GrandChildComponent:Contructed");
  }
  ngOnChanges() {
    console.log("    GrandChildComponent:ngOnChanges");
  }
  ngOnInit() {
    console.log("    GrandChildComponent:ngOnInit");
  }
  ngDoCheck() {
    console.log("    GrandChildComponent:DoCheck");
  }
  ngAfterContentInit() {
    console.log("    GrandChildComponent:ngAfterContentInit");
  }
  ngAfterContentChecked() {
    console.log("    GrandChildComponent:AfterContentChecked");
  }
  ngAfterViewInit() {
    console.log("    GrandChildComponent:AfterViewInit");
  }
  ngAfterViewChecked() {
    console.log("    GrandChildComponent:AfterViewChecked");
  }
  ngOnDestroy() {
    console.log("    GrandChildComponent:ngOnDestroy");
  }
}
```

- We are listening to all the hooks
- @Input decorator marks the customer as input property. It will receive the data from the parent

Output:



Reusable components in angular

Introduction to Reusable Angular Components

Components that define a base structure/behaviour but still can be used in various contexts/ dynamic content. Every time you use a reusable component, you also have a parent component. This flexible content inside the reusable component comes from parent content and ends up in a dedicated slot inside the reusable component. So it is projected down to the parent component.

Examples: Buttons, Spinners/ Loaders, tab, Dropdowns, Search Bar, etc.

What are the levels of Reusability of a component?

- **Templating:** Reuse markups across components, Future changes are simple. Don't forget to update all the different implementations, e.g. Button
- **Configuration:** Props to create variation in their behavior [loading state, disabled state]
- **Adaptability:** Allow the child component to control our markup of the parent component (more flexible than props) [slot for content inside a component]
- **Inversion:** Tell the child how to render (customize render a list of items)
- **Extension:** Extend the behavior of a component in many ways
- **Nesting:** Apply for extension through the component hierarchy. Each component extends with more specific functionality.

Why do we use reusable components?

- **Efficiency:** Reuse markups across components, Future changes are simple
- **Consistency:** Updating reusable components get affected everywhere it is used
- **Easy to Test:** Things become easy to test when they follow SRP

How to create Reusable Angular Components with examples?

ng-template

- app-header.component.ts (reusable child component)
- homepage-header.component.ts (parent component)

Angular ngTemplateOutlet can be used to insert a common template in various sections of a view. A reusable component where anything from a parent can be inserted into the child view using templateRef. Therefore, you get customized/ modified changes from the parent component & reusable code from the child component.

Example:

Step 1: Add the data in app.component.ts as shown below.

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

@Component({
```

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

 data = [{
   name : 'Sam Johnson',
   dept : 'Electrical'
 },{
   name : 'Roy Thomas',
   dept : 'Mechanical'
 },{
   name : 'Jim Lasker',
   dept : 'Medical'
 }]
}
```

To make the application appealing, let's add bootstrap to the Angular project. From the Angular CLI execute the following command:

```
npm install bootstrap jquery popper.js
```

Now add the Bootstrap scripts and CSS files to the angular.json file.

```
"styles": [
   "src/styles.css",
   "node_modules/bootstrap/dist/css/bootstrap.min.css"
   ],
"scripts": [
   "node_modules/jquery/dist/jquery.min.js",
   "node_modules/popper.js/dist/umd/popper.min.js",
   "node_modules/bootstrap/dist/js/bootstrap.min.js"
   ]
```

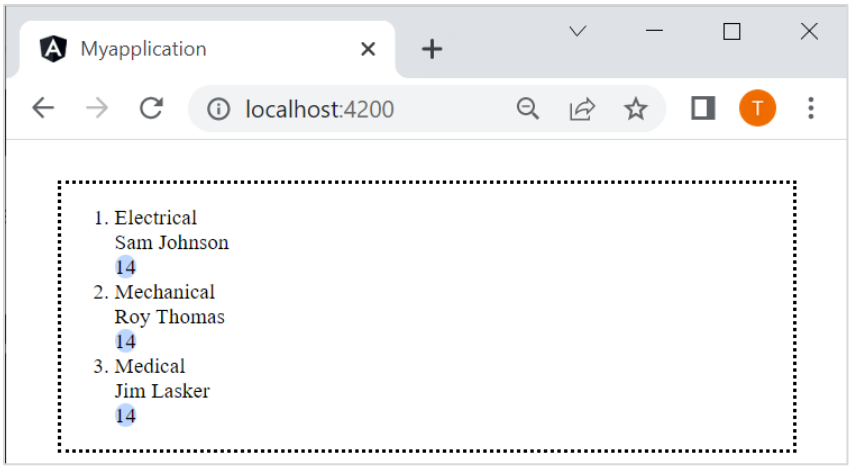
Let's render a list of the data you defined in app.component.ts file. Add the following HTML code to the app.component.html file.

```
<div class="container margin-30">
    <ol class="list-group list-group-numbered">
        <li *ngFor="let item of data" class="list-group-item d-flex justify-content-between
align-items-start">
          <div class="ms-2 me-auto">
            <div class="fw-bold font-fam">{{item.dept}}</div>
            {{item.name}}
          </div>
          <span class="badge bg-primary rounded-pill">14</span>
        </li>
      </ol>
      </div>
```

Add the following code to the app.component.css file.

```
<div class="container margin-30">
    <ol class="list-group list-group-numbered">
        <li *ngFor="let item of data" class="list-group-item d-flex justify-content-between
align-items-start">
          <div class="ms-2 me-auto">
            <div class="font-fam">{{item.dept}}</div>
            {{item.name}}
          </div>
          <span class="rounded-pill">14</span>
        </li>
      </ol>
      </div>
```

Save the above changes and you'll be able to see the list of employees based on the data in the app.component.ts file.



Creating a reusable Angular component

You just added some code to render a list of items. As you can see in the list presented above, the total count is presented next to each item. There might be scenarios where you might not be needing the count but rather the same list. So the same code works for you with a bit of modification.

Let's make the list rendering code reusable by creating a separate component altogether. The item count can be made configurable to display or hide as per need.

Create a new component using the Angular CLI.

```
ng g component dataList
```

It will create a new component inside app/data-list called DataListComponent.

Move the app.component.html and the app.component.css code to data-list.component.html and data-list.component.css respectively.

For the time being, let's also move the data to data-list.component.ts. Eventually, you'll be passing the data to the component using @Input decorator.

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

@Component({
  selector: 'app-data-list',
```

```
  templateUrl: './data-list.component.html',
  styleUrls: ['./data-list.component.css']
})
export class DataListComponent implements OnInit {

  data = [{
    name : 'Sam Johnson',
    dept : 'Electrical'
  },{
    name : 'Roy Thomas',
    dept : 'Mechanical'
  },{
    name : 'Jim Lasker',
    dept : 'Medical'
  }];

  constructor() { }

  ngOnInit(): void {

  }
}
```

Save the changes and reload your Angular application. You won't be able to see the data list rendered.

You created an Angular component to render the list. But for it to render you need to use the DataListComponent selector app-data-list in the app.component.html.

```
<app-data-list></app-data-list>
```

Save the changes and you'll be able to see the data list.

Using @Input and @output

A common pattern in Angular is sharing data between a parent component and one or more child components. You can implement this pattern by using the @[Input]() and @[Output]() directives.

Now, let's make it reusable by passing in data to the Angular component. You can use the @Input decorator for passing data to the component.

Let's start by defining the @Input decorator inside the DataList component.

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

@Component({
  selector: 'app-data-list',
  templateUrl: './data-list.component.html',
  styleUrls: ['./data-list.component.css']
})
export class DataListComponent implements OnInit {

  @Input() data;
  @Input() showCount = false;

  constructor() { }

  ngOnInit(): void {

  }
}
```

As seen in the above code, you have defined two @Input decorators, one for passing the data to render and another, a Boolean, to decide whether to show count info.

In the data-list.component.html modify the HTML to configure the data as per the @Input parameters.

```
<div class="container margin-30">
        <div class="card">
          <div class="card-body">
            <ol class="list-group list-group-numbered">
              <li *ngFor="let item of data" class="list-group-item d-flex justify-content-
between align-items-start">
                <div class="ms-2 me-auto">
                  <div class="fw-bold font-fam">{{item.dept}}</div>
                  {{item.name}}
                </div>
                <span *ngIf="showCount" class="badge bg-primary rounded-pill">14</span>
              </li>
            </ol>
```

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

Modify the app.component.html file to pass the @Input decorators to the reusable child component.

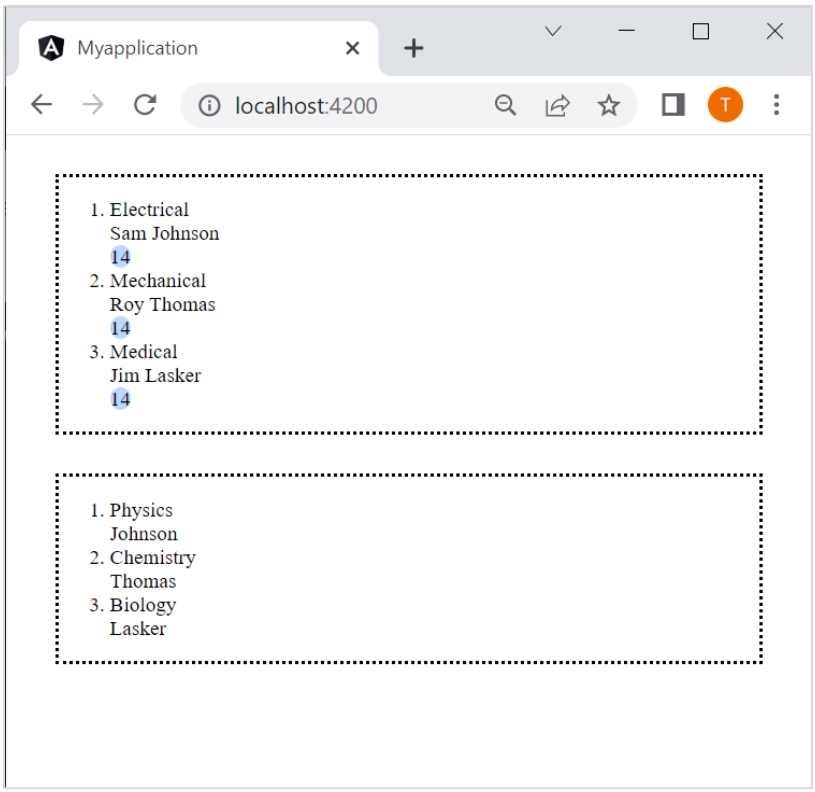
```
<app-data-list  [data]="data1"  [showCount]="true"></app-data-list>

<app-data-list  [data]="data2"></app-data-list>
```

Define data1 and data2 inside the app.component.ts file.

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

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

  data1 = [{
    name : 'Sam Johnson',
    dept : 'Electrical'
  },{
    name : 'Roy Thomas',
    dept : 'Mechanical'
  },{
    name : 'Jim Lasker',
    dept : 'Medical'
  }];

  data2 = [{
    name : 'Johnson',
    dept : 'Physics'
  },{
    name : 'Thomas',
    dept : 'Chemistry'
```

```
  },{
    name : 'Lasker',
    dept : 'Biology'
  }];

}
```

Save the above changes and you will be able to see two lists of data rendered.



- ng-template has the ability to hold content without rendering it. It is the underlying mechanism to all directives that modify the DOM structure and therefore a fundamental concept behind all structural Angular directives like *ngIf or *ngFor.
- ng-template is very powerful and can be used for multiple things, one of them is creating flexible components by passing a template to a host component.
- The base idea of reusable and flexible components is simple. You have a reusable host component. This component contains some standard behaviour and template which is always the same.

- Furthermore, it also defines some slots. Those slots define where the flexible content will end up. Depending on the use case, different content can be passed to the same component.

Activity Seventeen

Follow the instructions:

1. Open the Visual Studio code.
2. Demonstrate how to reuse components using <ng template>.
3. Follow the steps and create the angular application as shown above.

Ng-content

First, let's take a look at how we would design the button component without ng-content. We define an input value and using curly braces to display the text.

about.component.ts

```
import { Component, Input, NgModule, OnInit } from '@angular/core';
import { AbstractControl, FormBuilder, FormControl, FormGroup, ValidatorFn, Validators }
from '@angular/forms';
import { ActivatedRoute, Router } from '@angular/router';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';

@Component({
  selector: 'ng-button',
  template: `<button (click)="onButtonClicked()">{{text}} </button>
  `
})
export class AboutComponent implements OnInit {
 @Input()
  text!: string;

 public onButtonClicked(){

 }
```

```
  ngOnInit(): void {
  }

}
```

app.component.html

```
<ng-button [text]=" 'publish new post' "></ng-button> </ng-button>
```

When using curly braces, the text of the button is coming from the component itself. While this solution does the job, there is a better one out there: <ng-content></ng-content>

- **Using <ng-content>**

Instead of using curly braces and keeping track of the input value, we can pass the value into the component from its execution context using <ng-content>.

about.component.ts

```
import { Component, Input, NgModule, OnInit } from '@angular/core';
import { AbstractControl, FormBuilder, FormControl, FormGroup, ValidatorFn, Validators }
from '@angular/forms';
import { ActivatedRoute, Router } from '@angular/router';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';

@Component({
  selector: 'ng-button',
  template: `<button (click)="onButtonClicked">
    <ng-content></ng-content>
  </button>
  `
})
export class AboutComponent implements OnInit {
 @Input()
  text!: string;
```

```
 public onButtonClicked(){

 }

  ngOnInit(): void {
  }

}
```

ng-content is similar to ng-transclude in AngularJs. It works as a placeholder. Whatever content is placed between the button component will be displayed instead of ng-content.

app.component.html

```
<ng-button>publish new post</ng-button>
```

The use of ng-content removes complexity, making the code cleaner and more readable. The good thing is we are not limited to just one ng-content element.

- **Using multiple ng-content elements**

We can define multiple content slot using <ng-content></ng-content> and the Selector select. Let's create a card component with a header, content and footer section.

about.component.ts

```
import { Component, Input, NgModule, OnInit } from '@angular/core';
import { AbstractControl, FormBuilder, FormControl, FormGroup, ValidatorFn, Validators }
from '@angular/forms';
import { ActivatedRoute, Router } from '@angular/router';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';

@Component({
  selector: 'ng-button',
  template: `
  <div class="card">
    <header>
      <ng-content select="card-header"></ng-content>
</header>
<div class="card-content">
```

```
  <ng-content select="card-content"></ng-content>
</div>
<footer>
  <ng-content select="card-footer"></ng-content>
</footer>
  </div>
  `
})
export class AboutComponent implements OnInit {
}
```

app.component.html

```
<card>
     <card-header> post: ng-content in Angular</card-header>
     <card-content>
          how does ng-content work...
     </card-content>
     <card-footer>
          <ng-button>edit post</ng-button>
          <ng-button>publish post</ng-button>
</card-footer>
     </card>
</card>
```

If we are using the elements which we defined via select, Angular will throw the following error:

> *Template parse errors:*
>
> *'card-header' is not a known element:*
>
> *1. If 'card-header' is an Angular component, then verify that it is part of this module.*
>
> *2. If 'card-header' is a Web Component then add 'CUSTOM_ELEMENTS_SCHEMA' to the '@NgModule.schemas' of this component to suppress this message.*

That happens because Angular doesn't know our elements <card-header>, <card-content> and <card-footer>. There are either components, directive nor known HTML elements. We can solve this problem by adding the schema NO_ERRORS_SCHEMA to our AppModule.

Angular Routing

What is Angular Routing?

In Angular, routing plays a vital role since it is essentially used to create Single Page Applications. These applications are loaded just once, and new content is added dynamically. Applications like Google, Facebook, Twitter, and Gmail are a

few examples of SPA. The best advantage of SPA is that they provide an excellent user experience and you don't have to wait for pages to load, and by extension, this makes the SPA fast and gives a desktop-like feel.

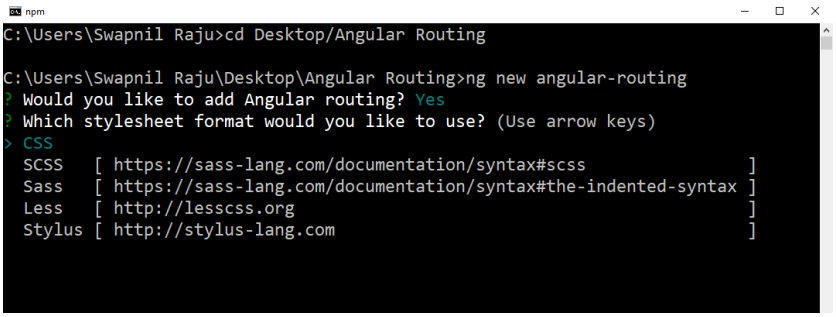
It generally specifies navigation with a forward slash followed by the path defining the new content.

Angular provides extensive set of navigation feature to accommodate simple scenario to complex scenario. The process of defining navigation element and the corresponding view is called Routing. Angular provides a separate module, RouterModule to set up the navigation in the Angular application.

Configure Routing

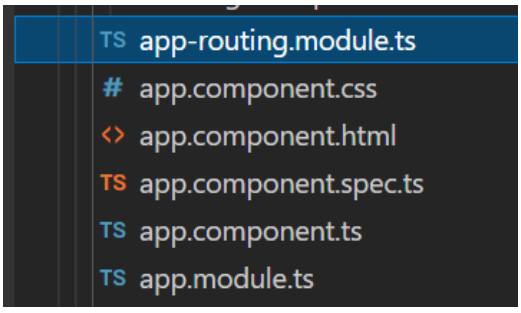
Angular CLI provides complete support to setup routing during the application creation process as well as during working an application. Let us create a new application with router enabled using below command –

ng new <name-application>



you get a question on the "Would you like to add Angular Routing?"

When you type yes, the app routing file is created along with the application folder.



Angular CLI generate a new module, AppRoutingModule for routing purpose. The generated code is as follows-

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

const routes: Routes = [];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Here,

- Imports RouterModule and Routes from @angular/router package.
- RouterModule provides functionality to configure and execute routing in the application.
- Routes is the type used to setup the navigation rules.
- Routes is the local variable (of type Routes) used to configure the actual navigation rules of the application.
- RouterModule.forRoot() method will setup the navigation rules configured in the routes variable.

Angular CLI include the generated AppRoutingModule in AppComponent as mentioned below –

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

Here,

AppComponent imports the AppRoutingModule module using imports meta data.

Angular CLI provides option to set routing in the existing application as well. The general command to include routing in an existing application is as follows –

```
ng generate module my-module --routing
```

This will generate new module with routing features enabled. To enable routing feature in the existing module (AppModule), we need to include extra option as specified below –

```
ng generate module app-routing --module app --flat
```

Here, module app configures the newly created routing module, AppRoutingModule in the AppModule module.

Let us configure the routing module in ExpenseManager application.

Open command prompt and go to project root folder.

```
cd /go/to/expense-manager
```

Generate routing module using below command –

```
ng generate module app-routing --module app --flat
```

There are two files created is mentioned below –

```
CREATE src/app/app-routing.module.ts (196 bytes)
UPDATE src/app/app.module.ts (785 bytes)
```

Here,

CLI generate AppRoutingModule and then, configures it in AppModule

Creating routes

Creating a route is simple and easy. The basic information to create a route is given below −

- Target component to be called.
- The path to access the target component.

The code to create a simple route is mentioned below −

```
const routes: Routes = [
{ path: 'about', component: AboutComponent },
];
```

Here,

- **Routes** is the variable in the AppRoutingModule.
- **about** is the path and AboutComponent is the target / destination component. When user requests http://localhost:4200/about url, the path matches with about rule and then AboutComponent will be called.

Accessing routes

Let us learn how to use the configured routes in the application.

Accessing the route is a two-step process.

1. Include **router-outlet** tag in the root component template.

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

2. Use **routerLink** and **routerLinkActive** property in the required place.

```
<a routerLink="/about" routerLinkActive="active">First Component</a>
```

Here,

- **routerLink** set the route to be called using the path.
- **routerLinkActive** set the CSS class to be used when the route is activated.

Sometime, we need to access routing inside the component instead of template. Then, we need to follow below steps:

- Inject instance of **Router** and **ActivatedRoute** in the corresponding component.

```
import { Router, ActivatedRoute } from '@angular/router';
constructor(private router: Router, private route: ActivatedRoute)
```

Here,

- **Router** provides the function to do **routing operations**.
- **Route** refers the current **activate route**.

Use router's navigate function.

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

Here,

navigate function expects an array with necessary path information.

Using relative path

Route path is similar to web page URL and it supports relative path as well. To access AboutComponent from another component, say HomePageComponent, simple use .. notation as in web url or folder path.

```
<a routerLink="../about">Relative Route to about component</a>
```

To access relative path in the component –

```
import { NavigationExtras } from '@angular/router';
this.router.navigate(['about'], { relativeTo: this.route });
```

Here,

relativeTo is available under **NavigationExtras** class.

Route ordering

Route ordering is very important in a route configuration. If same path is configured multiple times, then the first matched path will get called. If the first match fails due to some reason, then the second match will get called.

Redirect routes

Angular route allows a path to get redirected to another path. redirectTo is the option to set redirection path. The sample route is as follows −

```
const routes: Routes = [
{ path: '', redirectTo: '/about' },
];
```

Here,

redirectTo sets about as the redirection path if the actual path matches empty string.

Wildcard routes

Wildcard route will match any path. It is created using ** and will be used to handle non existing path in the application. Placing the wildcard route at the end of the configuration make it called when other path is not matched.

The sample code is as follows −

```
const routes: Routes = [
  { path: 'about', component: AboutComponent },
  { path: '',   redirectTo: '/about', pathMatch: 'full' },
  { path: '**', component: PageNotFoundComponent },  // Wildcard route for a 404
page
];
```

Here,

If a non existent page is called, then the first two route gets failed. But, the final wildcard route will succeed and the **PageNotFoundComponent** gets called.

Access Route parameters

In Angular, we can attach extra information in the path using parameter. The parameter can be accessed in the component by using paramMap interface. The syntax to create a new parameter in the route is as follows –

```
const routes: Routes = [
  { path: 'about', component: AboutComponent },
  { path: 'item/:id', component: ItemComponent },
  { path: '',   redirectTo: '/about', pathMatch: 'full' },
  { path: '**', component: PageNotFoundComponent },  // Wildcard route for a 404
page
];
```

Here, we have attached id in the path. id can be accessed in the **ItemComponent** using two techniques.

- Using Observable.
- Using snapshot (non-observable option).

Using Observable

Angular provides a special interface, paramMap to access the parameter of the path. parmaMap has following methods:

- **has(name)** – Returns true if the specified name is available in the path (parameter list).
- **get(name)** – Returns the value of the specified name in the path (parameter list).
- **getAll(name)** – Returns the multiple value of the specified name in the path. get() method returns only the first value when multiple values are available.
- **keys** – Returns all parameter available in the path.

Steps to access the parameter using **paramMap** are as follows –

- Import **paramMap** available in @angular/router package.
- Use **paramMap** in the **ngOnInit()** to access the parameter and set it to a local variable.

```
ngOnInit() {
    this.route.paramMap.subscribe(params => {
        this.id = params.get('id');
    });
}
```

We can use it directly in the rest service using pipe method.

```
this.item$ = this.route.paramMap.pipe(
    switchMap(params => {
      this.selectedId = Number(params.get('id'));
      return this.service.getItem(this.selectedId);
    })
);
```

Using snapshot

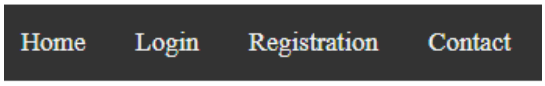
snapshot is similar to Observable except, it does not support observable and get the parameter value immediately.

```
let id = this.route.snapshot.paramMap.get('id');
```

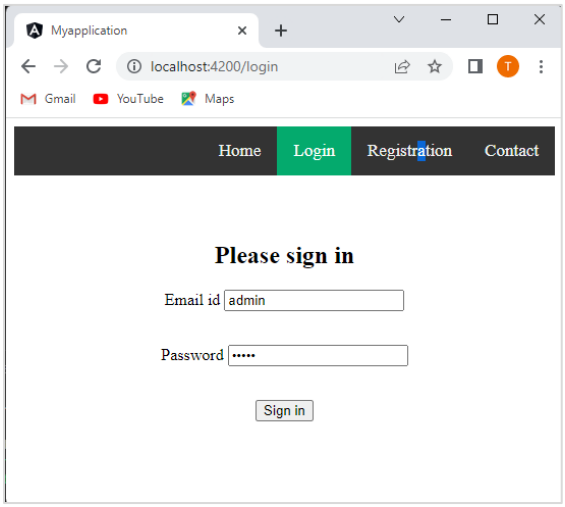
Activity Eighteen

Follow the instructions:

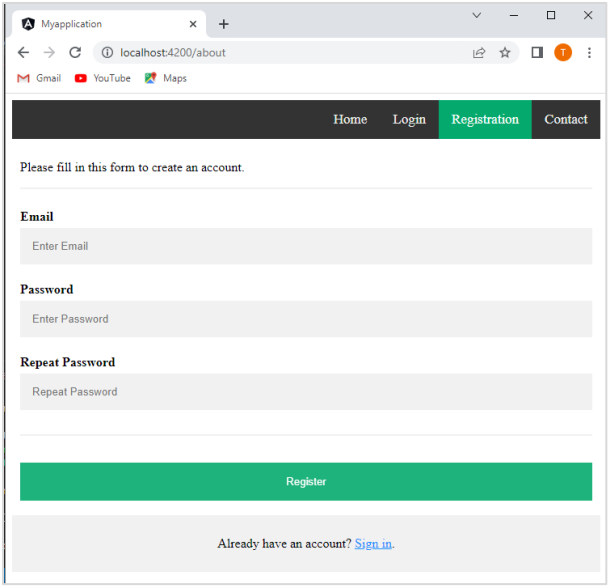
1. Create a website for a university.
2. Create components Home, Login, Registration, Contact.
3. Create a tab on the top of the website as shown below:



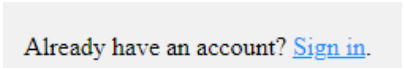
4. Using the routing option navigate the components and add information as given below:
   - In Home tab, there should information about the university.
   - In Login tab, there should login form as shown below:



- In Register tab, there should be registration form as shown below:

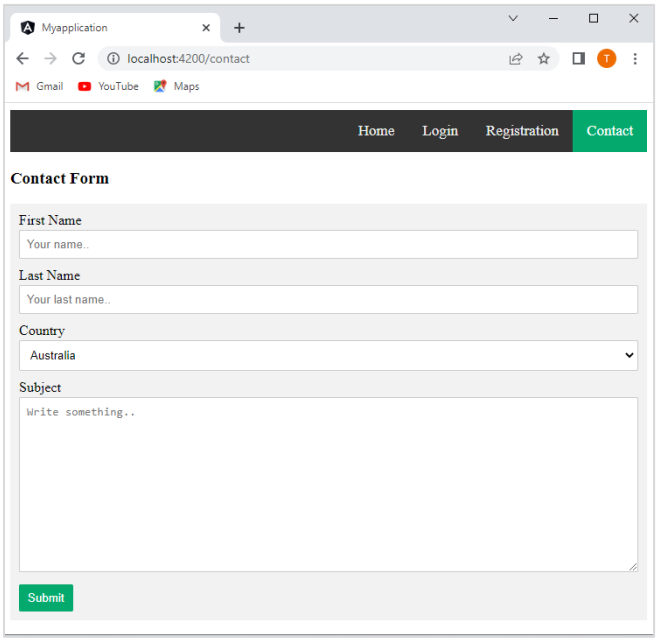


- The last line where we are asking them to sign in if already have an account, should connect to the login component.



(When clicked on Sign in, the login form should open)

- In Contact, add contact us form as shown below:



Nested routing

In general, router-outlet will be placed in root component (AppComponent) of the application. But, router-outlet can be used in any component. When router-outlet is used in a component other then, root component, the routes for the particular component has to be configured as the children of the parent component. This is called Nested routing.

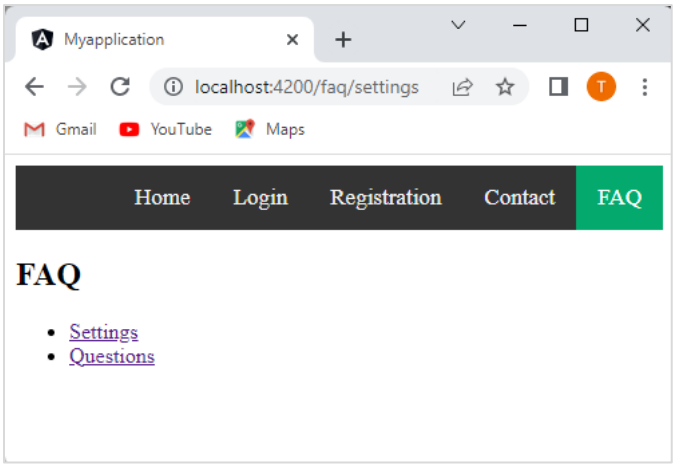
Let us consider a component, say **faqComponent** is configured with **router-outlet** and has two **routerLink** as specified below:

```
<h2>FAQ</h2>
<nav>
   <ul>
      <li><a routerLink="Settings">Settings</a></li>
      <li><a routerLink="questions">Questions</a></li>
   </ul>
</nav>
<router-outlet></router-outlet>
```

The route for the faqComponent has to be configured as **Nested routing** in app-routing.component.ts as specified below:

```
const routes: Routes = [
  {
    path: 'faq',

    component: FAQComponent,
    children: [
    {
       path: 'settings',
       component: SettingsComponent
    },
    {
    path: 'questions',
    component: QuestionsComponent
    }
    ]
 }]
```

Output:



Query Parameters

The query parameters feature in Angular lets you pass the optional parameters while navigating from one route to another. When you pass a query parameter to a specific route, it looks something like this,

http://localhost:4200/orders?category=information

As you can see in the above URL, the category is the query parameter and watches is its value.

How to pass the Query Parameters?

There are two primary ways to pass query parameters.

Using Router service

The Angular router service comes bundled with the navigate method which lets you move from one route to another. The first argument to this method is an array of URL fragments that represents a route. This method also accepts a second argument in form of an object. This object could be utilized to specify the query params.

Let's understand this in detail with the help of the URL mentioned in the above section. Let's say that we wish to navigate to the orders page with category as the query parameter and watches as its value. We can do this using the following code snippet:

```
home() {
    this.router.navigate(['/home], { queryParams: { category: 'information' } });
}
```

As you can see in the above code snippet, the parameters ['/home'] and { queryParams: { category: 'information' } } passed to the navigate method lets Angular know that it needs to navigate to the orders route with category = information in the query parameters.

You could also pass multiple parameters by adding multiple key-value pairs to queryParams object.

```
home(): void {
    this.router.navigate(['/home'], { queryParams: { category: 'information', type:
'analog' } });
}
```

The above code will pass two query parameters, category and type to the home route and the URL will look like this,

http://localhost:4200/home?category=information&type=analog

Preserving or merging query parameters

In our first example, when we moved to home, it sends a query parameter category to the route. However, this parameter is lost when we try to navigate away from this page.

However, if we wish to preserve; i.e. persist the query params we could do that by adding queryParamsHandling option to navigate the method of the next route.

```
home(): void {
    this.router.navigate(['/home'], { queryParamsHandling: 'preserve' });
}
```

The above code will result in the following URL,

http://localhost:4200/home?category=information

If the home route has its own query parameter, then we could use merge instead of preserve.

```
home() {
    this.router.navigate(['/home'], { queryParams: { message: 'hello' },
queryParamsHandling: 'merge' });
}
```

The resulting URL will look like this,

http://localhost:4200/home?category=information&message=hello

Using RouterLink Directive

You could also use queryParams directive to pass the query parameters, instead of Router service.

```
<a [routerLink]="['/home']" [queryParams]="{ category: 'information'}">
  Home
</a>
```

You could also use queryParamsHandling directive to specify the preserve or merge option.

```
<a
  [routerLink]="['/home']"
  [queryParams]="{ category: 'information'}"
  queryParamsHandling="preserve">
    Home
</a>
```

How to access the Query Parameters?

In order to access the parameters, we need to utilize the ActivatedRoute service. The ActivatedRoute service consists of queryParams property which holds an observable. Subscribing to this observable will give you access to query parameters passed in the URL. Let's understand this using the orders route example,

The first thing we need to do is to inject the ActivatedRoute route inside the component where we wish to access the parameters.

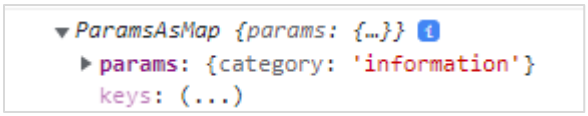
```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute, Router } from '@angular/router';
@Component({
  selector: 'app-home',
  templateUrl: './home.component.html',
  styleUrls: ['./home.component.css']
})
export class HomeComponent implements OnInit {

  constructor(private route: ActivatedRoute) {
   }
  ngOnInit(): void {
}
}
```

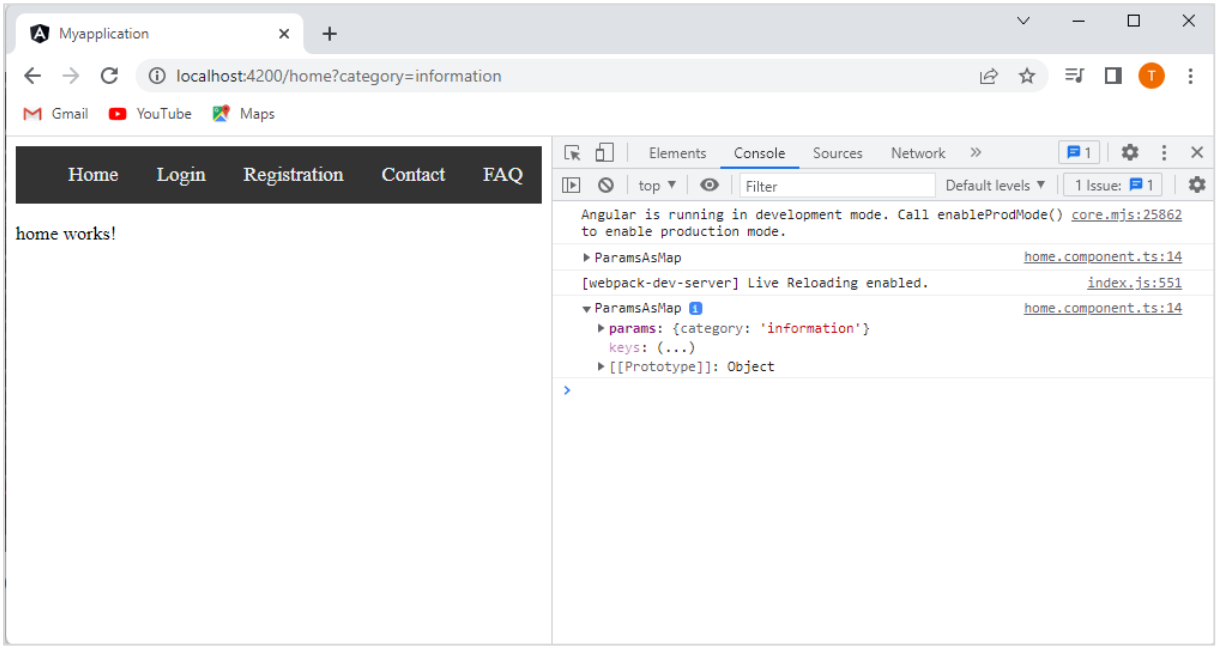
Next, we'll subscribe to the queryParams property,

```
  ngOnInit(): void {

  this.route.queryParamMap.subscribe((params => console.log(params));
  }
```

http://localhost:4200/orders?category=information

For the above URL output in the console will look something like this,



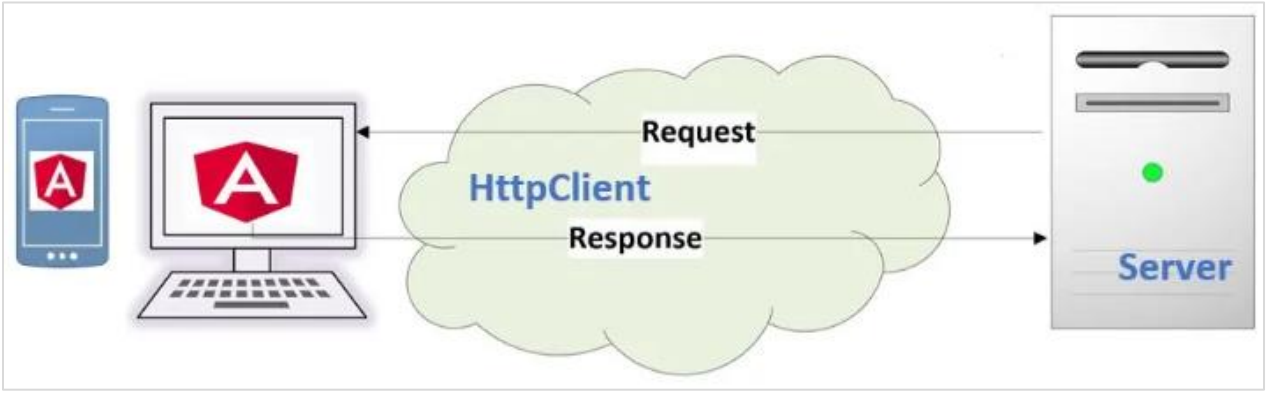


Module 7: Http Requests/ Observables

Most front-end applications need to communicate with a server over the HTTP protocol, to download or upload data and access other back-end services. Angular provides a client HTTP API for Angular applications, the HttpClient service class in @angular/common/http.

The HTTP client service offers the following major features.

- The ability to request typed response objects
- Streamlined error handling
- Testability features
- Request and response interception
- Supports RxJS observable-based APIs



Setup for server communication

Before you can use HttpClient, you need to import the Angular HttpClientModule. Most apps do so in the root AppModule.

app/app.module.ts

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

@NgModule({
imports: [
BrowserModule,
// import HttpClientModule after BrowserModule.
HttpClientModule,
],
declarations: [
AppComponent,
],
bootstrap: [ AppComponent ]
})
export class AppModule {}
```

You can then inject the HttpClient service as a dependency of an application class, as shown in the following LearnerService example. You can create a 'learner' service by using the following command:

```
Ng g service learner
```

app /learner.service.ts

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

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

```
  constructor(private http: HttpClient) { }

}
```

The HttpClient service makes use of observables for all transactions. You must import the RxJS observable and operator symbols that appear in the example snippets.

app/config/config.service.ts

```
import { Observable, throwError } from 'rxjs';
import { catchError, retry } from 'rxjs/operators';
```

What Is an RxJS Observable?

An observable is a unique object similar to Promise and it helps to manage async code. It's not from the JavaScript language, so to use it we need the most popular observable library, called RxJS (Reactive Extension for JavaScript). RxJS is a library for reactive programming using observables that makes it easier to compose asynchronous or callback-based code. Angular uses observables as an interface to handle the common asynchronous operations.

Requesting data from a server

Use the HttpClient.get() method to fetch data from a server. The asynchronous method sends an HTTP request, and returns an Observable that emits the requested data when the response is received. The return type varies based on the observe and responseType values that you pass to the call.

The get() method takes two arguments; the endpoint URL from which to fetch, and an options object that is used to configure the request.

```
options: {
  headers?: HttpHeaders | {[header: string]: string | string[]},
  observe?: 'body' | 'events' | 'response',
  params?: HttpParams|{[param: string]: string | number | boolean |
ReadonlyArray<string | number | boolean>},
  reportProgress?: boolean,
  responseType?: 'arraybuffer'|'blob'|'json'|'text',
  withCredentials?: boolean,
}
```

Important options include the observe and responseType properties.

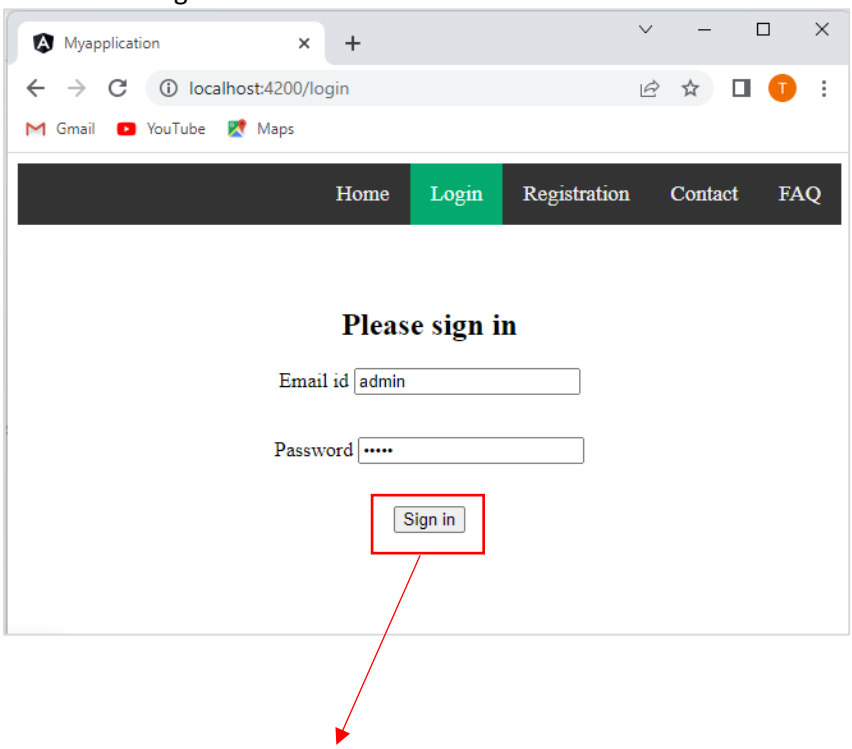
- The observe option specifies how much of the response to return
- The responseType option specifies the format in which to return data

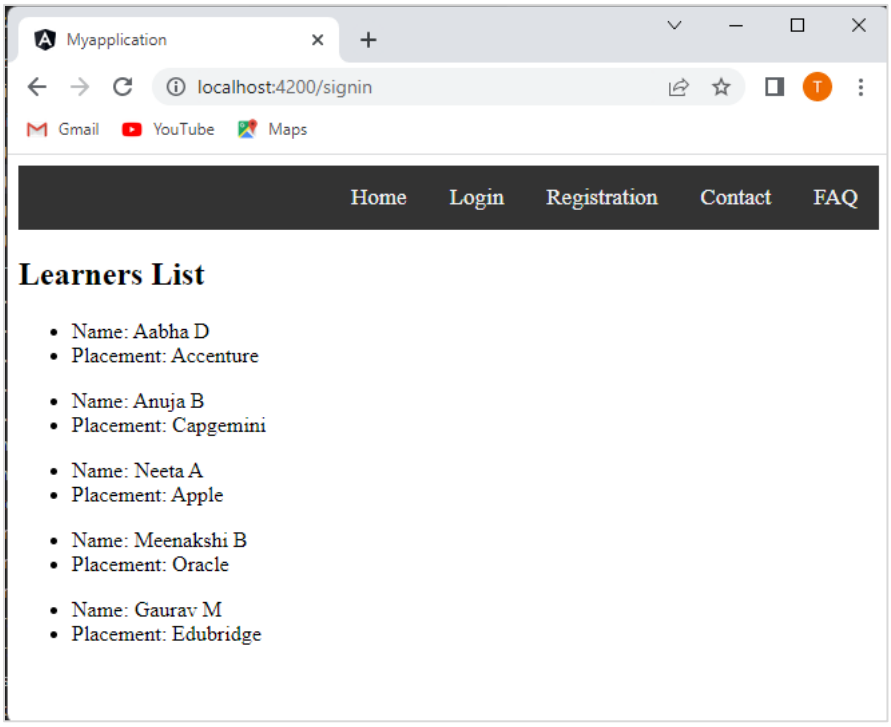
Note:

- Use the options object to configure various other aspects of an outgoing request. In Adding headers, for example, the service set the default headers using the headers option property.
- Use the params property to configure a request with HTTP URL parameters, and the reportProgress option to listen for progress events when transferring large amounts of data.

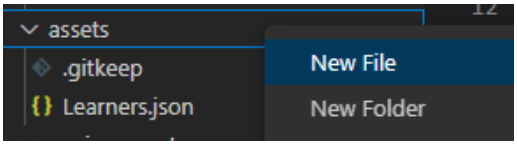
Example:

We will show the data of the learners when the user 'sign in' in our application. When the sign in button is clicked, it should be navigated to learners list as shown below.





Applications often request JSON data from a server. In the LearnerService example, the app needs a configuration file on the server, learner.json, that specifies resource URLs.
Right click assets and create a JSON file inside the folder.



Let us add the learner information in this file.

assets/learner.json

```
[
    {"id":1, "name": "Aabha D", "age": 30, "placement": "Accenture" },
    {"id":2, "name": "Anuja B", "age": 25, "placement": "Capgemini" },
    {"id":3, "name": "Neeta A", "age": 28, "placement": "Apple" },
    {"id":4, "name": "Meenakshi B", "age": 35, "placement": "Oracle" },
    {"id":5, "name": "Gaurav M", "age": 32, "placement": "Edubridge" }
]
```

To fetch this kind of data, the get() call needs the following options: {observe: 'body', responseType: 'json'}. These are

the default values for those options, so the following examples do not pass the options object.
The example confirms to the best practices for creating scalable solutions by defining a re-usable injectable service to perform the data-handling functionality.
In addition to fetching data, the service can post-process the data, add error handling, and add retry logic.
The LearnerService fetches this file using the HttpClient.get() method.

app/ learner.service.ts

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http'
import { Observable } from 'rxjs';
import { Ilearner } from './learner';

@Injectable({
  providedIn: 'root'
})
export class LearnerService {
  private _url: string ="/assets/Learners.json";

  constructor(private http: HttpClient) { }

getLearners(): Observable<Ilearner[]>{
  return this.http.get<Ilearner[]>(this._url);
```

The LearnerService gets and calls the getLearners service method.
Because the service method returns an Observable of configuration data, the component subscribes to the method's return value. The subscription callback performs minimal post-processing. It copies the data fields into the component's config object, which is data-bound in the component template for display.

Let us create a component using the following command:

```
Ng g c signin
```

app /learnerslist.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Ilearner } from '../learner';
import { LearnerService } from '../learner.service';

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

```
  styleUrls: ['./signin.component.css']
})
export class SigninComponent implements OnInit {
public learners=[] as any;
  constructor(private _learnerService: LearnerService) { }

  ngOnInit() {
    this._learnerService.getLearners()
    .subscribe(data => this.learners = data);
  }
}
```

learnerslist.component.html

```
<h2>Learners List</h2>
<ul *ngFor="let learner of learners">
    <li>Name: {{learner.name}}</li>
    <li>Placement: {{learner.placement}}</li>
</ul>
```

Now let us connect the sign in button to the learnerslist component.

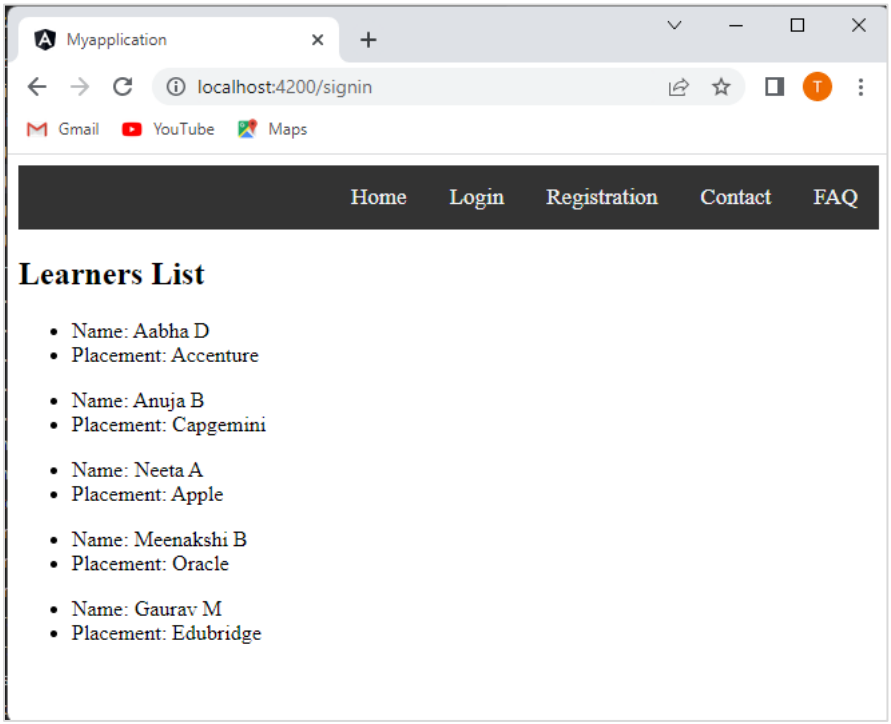
But for that we need to make modifications in the login component that we have created in the last session.

login.component.ts

```
import { Component, OnInit } from '@angular/core';
import { FormGroup, FormControl } from '@angular/forms';
import { AuthService } from '../auth.service';
import { Router } from '@angular/router';

@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.css']
})
export class LoginComponent implements OnInit {

  userName!: string;
  password!: string;
```

```
formData!: FormGroup;

constructor(private router : Router) {
 }

ngOnInit() {
   this.formData = new FormGroup({
      userName: new FormControl("admin"),
      password: new FormControl("admin"),

   });
}
onClickSubmit(data: any) {
        if(data) this.router.navigate(['/learnerslist']);
   };
}
```

Now when you click on Sign in button, the learners details will be shown on the screen.



Activity Nineteen

Follow the instructions:

1. Follow the steps to create an application as shown above.

Making a PUT request

- The HTTP PUT request method creates a new resource or replaces a representation of the target resource with the request payload.
- PUT updates the entire resource with data that is passed in the body payload. If there is no resource that matches the request, it will create a new resource.
- An app can send PUT requests using the HTTP client service.
- The following userService example, like the POST example, replaces a resource with updated data.

Example:

app/user.service.ts

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

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

  constructor(private http:HttpClient) { }

  userDetails(){
    const putBody={
      name: 'Tanaya',
      userId:1
    };
   return this.http.put('https://jsonplaceholder.typicode.com/users/1', putBody)
  }
}
```

This sends an HTTP PUT request to the JSONPlaceholder api which is a fake online REST api that includes a /users/1 route

that responds to PUT requests with the contents of the put request body and the post id property.

In the above example, we are charging the name of the userid 1.

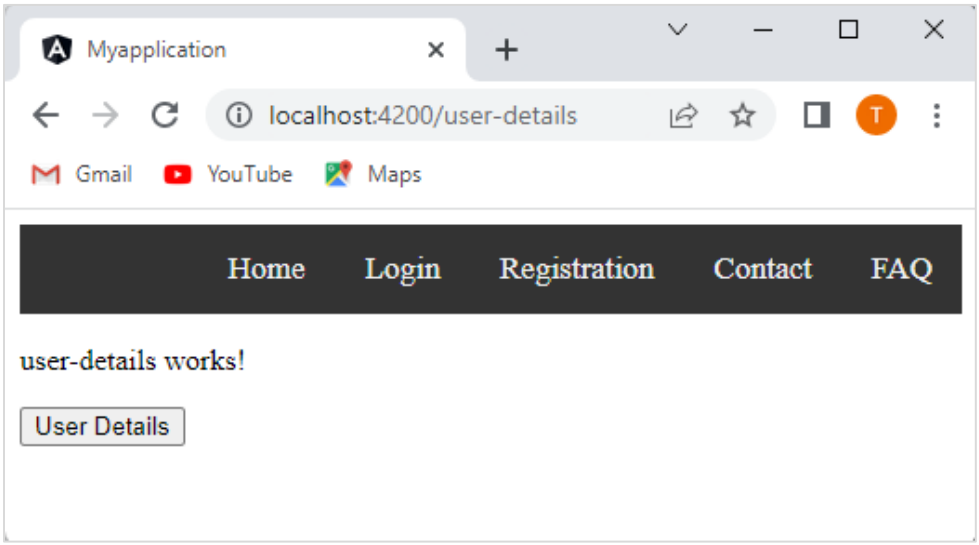
User-details.component.ts

```
import { Component, OnInit } from '@angular/core';
import { UserService } from '../user.service';

@Component({
  selector: 'app-user-details',
  templateUrl: './user-details.component.html',
  styleUrls: ['./user-details.component.css']
})
export class UserDetailsComponent implements OnInit {

  constructor(private userService: UserService) { }

  ngOnInit(): void {
    this.userService.userDetails();
  }
  userDetails(){
    this.userService.userDetails().subscribe(data =>{
      console.log(data);
    }, (err) => {
      console.log(err);
    })
  }
}
```

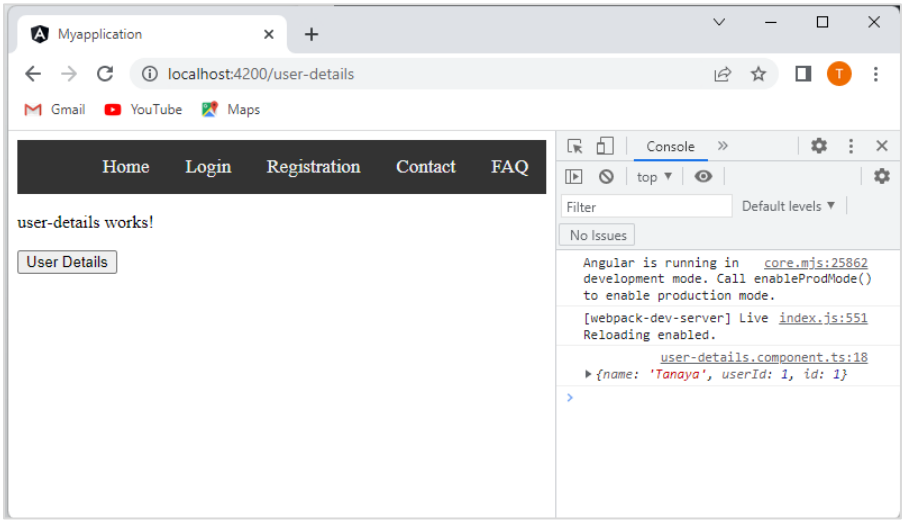
User-details.component.html

```
<p>user-details works!</p>
<button (click)="userDetails()">User Details</button>
```

Type http://localhost:4200/user-details . and see the output:



When you click on User-details, in the console, you will see the name and id.



Activity Twenty

Follow the instructions:

1. Follow the steps to create an application as shown above.

Handling request errors

If the request fails on the server, HttpClient returns an error object instead of a successful response.

The same service that performs your server transactions should also perform error inspection, interpretation, and resolution.

When an error occurs, you can obtain details of what failed in order to inform your user. In some cases, you might also automatically retry the request.

Getting error details

An app should give the user useful feedback when data access fails. A raw error object is not particularly useful as feedback. In addition to detecting that an error has occurred, you need to get error details and use those details to compose a user-friendly response.

Two types of errors can occur.

- The server backend might reject the request, returning an HTTP response with a status code such as 404 or 500. These are error responses.
- Something could go wrong on the client-side such as a network error that prevents the request from completing successfully or an exception thrown in an RxJS operator. These errors have status set to 0 and the error property contains a ProgressEvent object, whose type might provide further information.

HttpClient captures both kinds of errors in its HttpErrorResponse. Inspect that response to identify the error's cause.

The following example defines an error handler in the previously defined LearnerService.

app/learner.service.ts

```
private handleError(error: HttpErrorResponse) {
  if (error.status === 0) {
    // A client-side or network error occurred. Handle it accordingly.
    console.error('An error occurred:', error.error);
  } else {
    // The backend returned an unsuccessful response code.
    // The response body may contain clues as to what went wrong.
    console.error(
      `Backend returned code ${error.status}, body was: `, error.error);
  }
  // Return an observable with a user-facing error message.
  return throwError(() => new Error('Something bad happened; please try
again later.'));
}
```

The handler returns an RxJS ErrorObservable with a user-friendly error message. The following code updates the getLearners() method, using a pipe to send all observables returned by the HttpClient.get() call to the error handler.

app/signin.component.ts

```
getLearners() {
  return this.http.get<Ilearner>(this._Url)
    .pipe(
      catchError(this.handleError)
    );
}
```

Retrying a failed request

Sometimes the error is transient and goes away automatically if you try again. For example, network interruptions are common in mobile scenarios, and trying again can produce a successful result.

The RxJS library offers several retry operators. For example, the retry() operator automatically re-subscribes to a failed Observable a specified number of times. Re-subscribing to the result of an HttpClient method call has the effect of reissuing the HTTP request.

The following example shows how to pipe a failed request to the retry() operator before passing it to the error handler.

app /learner.service.ts

```
getLearners() {
  return this.http.get<Ilearner>(this._Url)
    .pipe(
      retry(3), // retry a failed request up to 3 times
      catchError(this.handleError) // then handle the error
    );
}
```

Basics of Observables & Promises

What is Promise in Angular?

Promises are a JavaScript construct that provides a way to handle asynchronous operations. They represent an eventual result of an asynchronous operation and can be in one of three states: pending, fulfilled, or rejected.

Promises can also be chained together to represent a series of dependent asynchronous operations.

Moreover, a Promise is a type of object that can produce a single value in the future, either a resolved value or a reason why it can't be resolved, or it can be pending.

Promise's drawbacks include:

- A failed call could not be retried.
- Promises become more difficult to manage as our application grows.
- The user could not cancel a request to the API.

What is Observe in Angular?

Angular Observables are more powerful than Promises because it has many advantages such as better performance and easier debugging.

An Observable can supply many values over time, similar to how streaming works, and it can be quickly terminated or retried if there are any errors. Observables can be coupled in various ways, or there might be a competition to see who can use the first one.

The RxJS Observables library is a powerful tool. Moreover, they are like event emitters that can emit unlimited values over time.

Observable's drawbacks include:

- You have to acquire a complicated framework of observable.
- You may find yourself using Observables in unexpected situations.

Let's discuss some of the differences between Promise and Observable.

Promise	**Observable**
Once a promise is fulfilled, its value does not alter. They are limited to emitting, rejecting and resolving a single value.	Multiple values can be emitted over some time using observables. It helps retrieve multiple values at once.
Lazy Loading does not apply to promises. So, Promises are immediately applied whenever they are formed.	Observables don't start executing until they're subscribed. This makes Observables handy whenever an action is required.
Instead of handling errors directly, Angular Promises always pass errors on to the child's promises.	Error handling is the responsibility of Angular Observables. When we utilize Observables, we can handle all errors in a single spot.
You can't back out of a promise once you've started it. The promise will be resolved or rejected based on the callback provided to the Promise.	With the help of the unsubscribe method, you can cancel observables at any time.

Example of Promise and Observable in Angular

Let's discuss an example that helps us understand the concept of promise and observable in detail.

Typescript code:

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

@Component({
  selector: 'my-app',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
    promiseTemp: Promise<boolean>;
  createPromise() {
    this.promiseTemp = this.testPromise(0);
  }
  observableTemp: Observable<boolean>;
  createObservable() {
    this.observableTemp = this.testObservable(0);
  }

  subscribeObservable() {
    this.observableTemp
      .subscribe(
        result => {
          console.log(`Observable Ok: ${result}`);
        },
        err => {
          console.error(err);
        });
  }
```

```
observableNextTemp: Observable<boolean>;
  createObservableNext() {
    this.observableNextTemp = this.testObservableNext(0);
  }
  promiseTemp2: Promise<boolean>;
  createToPromiseByObservable() {
    this.promiseTemp2 = this.testObservable(0).toPromise();
  }
  observableForkJoinTemp: Observable<boolean[]>;
  countForkJoin = 2;

  testObservable(value: number): Observable<boolean> {
    console.log("create Observable");
    return Observable.create((observer: Observer<boolean>) => {
      console.log("execute Observable");
      this.testPromise2(value)
        .then(result => {
          observer.next(result);
          observer.complete();
        })
        .catch(err => {
          observer.error(err);
          observer.complete();
        });
    });
  }
testObservableNext(value: number): Observable<boolean> {
    console.log("create Observable Next");
    return Observable.create((observer: Observer<boolean>) => {
      console.log("execute Observable Next");
      this.testPromise2(value)
        .then(result => {
          observer.next(result);

setTimeout(() => {
            observer.next(result);
            observer.complete();
          }, 5000);

        })
        .catch(err => {
          observer.error(err)
        });
    });
  }
```

```
  testPromise(value: number): Promise<boolean> {
    console.log("creation");
    return new Promise((resolve, reject) => {
      console.log("execution");
      this.testPromise2(value)
        .then(result => {
          resolve(result);
        })
        .catch(reject);
    });
  }

  testPromise2(value: number): Promise<boolean> {
    return new Promise((resolve, reject) => {
      if (value > 1) {
        reject(new Error("testing"));
      } else {
        resolve(true);
      }
    });
  }
}
```

HTML code:

```
<h2>Example of Promise and Observable </h2>

<br/>
<button (click)="createPromise()">Create Promise</button>
<br/>
<br/>
<button (click)="createObservable()">Create Observable</button>
<br/>
<br/>
<button (click)="subscribeObservable()">Subscribe Observable</button>
<br/>
<br/>
<button (click)="createObservableNext()">Create Observable Next</button>
<br/>
<br/>
<button (click)="createToPromiseByObservable()">Create Promise by
Observable</button>
<br/>
```

Promises are used in Angular to resolve asynchronous operations, and Observables are used when the data comes from an external source like an API.

Promises can be resolved with a single value or an array of values, and Observables emit one or more values over time. So, while managing an HTTP search, Promise can only handle a single response for the same request; however, if there

are numerous results to the same request, we must use Observable.

Activity Twenty-one

Follow the instructions:

1. Explain the concept of observable and promise.
2. Explain and demonstrate the example given above.

Module 8: Authentication and Route Protection

Authentication

When talking about authentication in web applications a developer's mind immediately jumps to the login where the user enters their username and password to prove that they're really who they claim to be. Although this certainly is the most important aspect of it and we will be spending most of this article on this topic, two other areas also need to be addressed: The session refresh and the logout.

Session refresh refers to the process of getting a fresh token for the user when the current one has expired. This feature is of paramount importance for the user experience — ideally, it saves the user having to enter their password every other hour. Users of modern web applications are used to not having to enter their password for a long time after initially logging in to the application. Think about it: When was the last time you entered your Facebook password when using its web application? Probably not for a very long time, unless you switched your browser or your computer.

On the other hand, a web application should also provide the user with the possibility to explicitly log out if they want to end their session. Although this is not a feature that will be used with high frequency, it's still important. Imagine a user wants to use two accounts in the same application for different purposes. I myself have been using two GitHub accounts for a while, one for work with my work email address and a separate one for private projects. To switch between accounts, I log out of one and then log into the other (don't tell Github, they don't seem to like this).

Sometimes the user simply wants to log out of your application because they're using a shared computer and don't want other people using the account.

How Authentication works in SPA

Authentication and authorization are important pieces on almost every serious application. Single Page Applications (SPAs) are no exception. The application may not expose all of its data and functionality to just any user. Users may have to authenticate themselves to see certain portions of the application, or to perform certain action on the application. To identify the user in the application, we need get the user logged in.

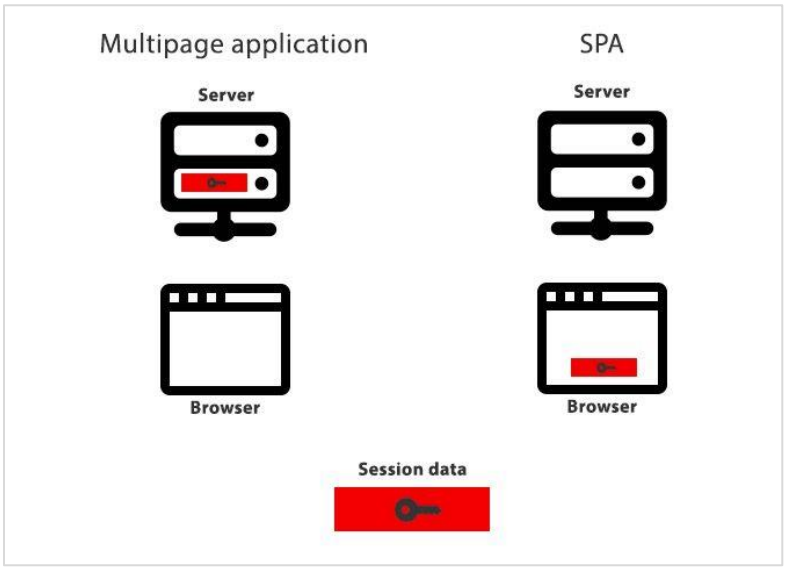
There is a difference in the way user management is implemented in traditional server-driven applications and Single Page Applications. The only way in which SPA can interact with its server components is through AJAX. This is true even for logging in and logging out.

The server responsible for identifying the user has to expose an authentication endpoint. The SPA will send the credentials entered by the user to this endpoint to for verification. In a typical token-based authentication system, the service may respond with an access token or with an object containing the name and role of the logged in user after validating the credentials. The client has to use this access token in all secured API requests made to the server.

As the access token will be used multiple times, it is better to store it on the client side. In Angular, we can store the value in a service or a value as they are singleton objects on the client side. But, if the user refreshes the page, the value in the service or value would be lost. In such case, it is better to store the token using one of the persistence mechanisms offered by the browser; preferably sessionStorage, as it is cleared once the browser is closed.

The Storing of Session Data: Stateful vs. Stateless Approach

The main difference between SPA and Multipage applications regarding authentication is whether session data is stored on the server or not. In other words, multipage applications have a so-called stateful authentication approach and SPA have a stateless one.



The REST API is the most widely used SPA data layer. One of the architectural restrictions of its design is that each request must hold all required data from the application state in order to change the server's resource state. The "ST" in REST – REpresentational State Transfer – refers to the fact that you are constantly transferring a state between the server and the client over the HTTP protocol.

Advantages of Single-Page Applications

- According to Google's research, every time a page load goes from 1 sec to 3 sec, the probability of bounce rate increases by 32%. For 10 sec, the chance rises to a whopping 123%.
- No matter what, faster's always better.
- One of the major benefits of single-page applications is velocity. The majority of the resources that SPAs need—viz.HTML, CSS, and Scripts are loaded along with the application. They don't need to be reloaded throughout the entire session.
- Data is the only entity that is oscillated between the server and the webpage. This makes the application efficient and incredibly responsive to any query.

What are JWTs?

- A JSON Web Token (or JWT) is simply a JSON payload containing a particular claim. The key property of JWTs is that in order to confirm if they are valid we only need to look at the token itself.
- We don't have to contact a third-party service or keep JWTs in-memory between requests to confirm that the claim they carry is valid - this is because they carry a Message Authentication Code or MAC (more on this later).
- JWT token consists of three parts:
  - payload
  - signature
  - header
- The key thing about JWTs is that in order to confirm if they are valid, we only need to inspect the token itself and validate the signature, without having to contact a separate server for that, or keeping the tokens in memory or in the database between requests.
- If JWTs are used for Authentication, they will contain at least a user ID and an expiration timestamp**.**

here is an example:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIzNTM0NTQzNTQzNTQzNTM0NTMiLCJleHAiOjE1MDQ2OTkyNTZ9.zG-2FvGegujxoLWwIQfNB5IT46D-xC4e8dEDYwi6aRM
```

To see it, let's head over to jwt.io and paste the complete JWT string into the validation tool, we will then see the JSON Payload:

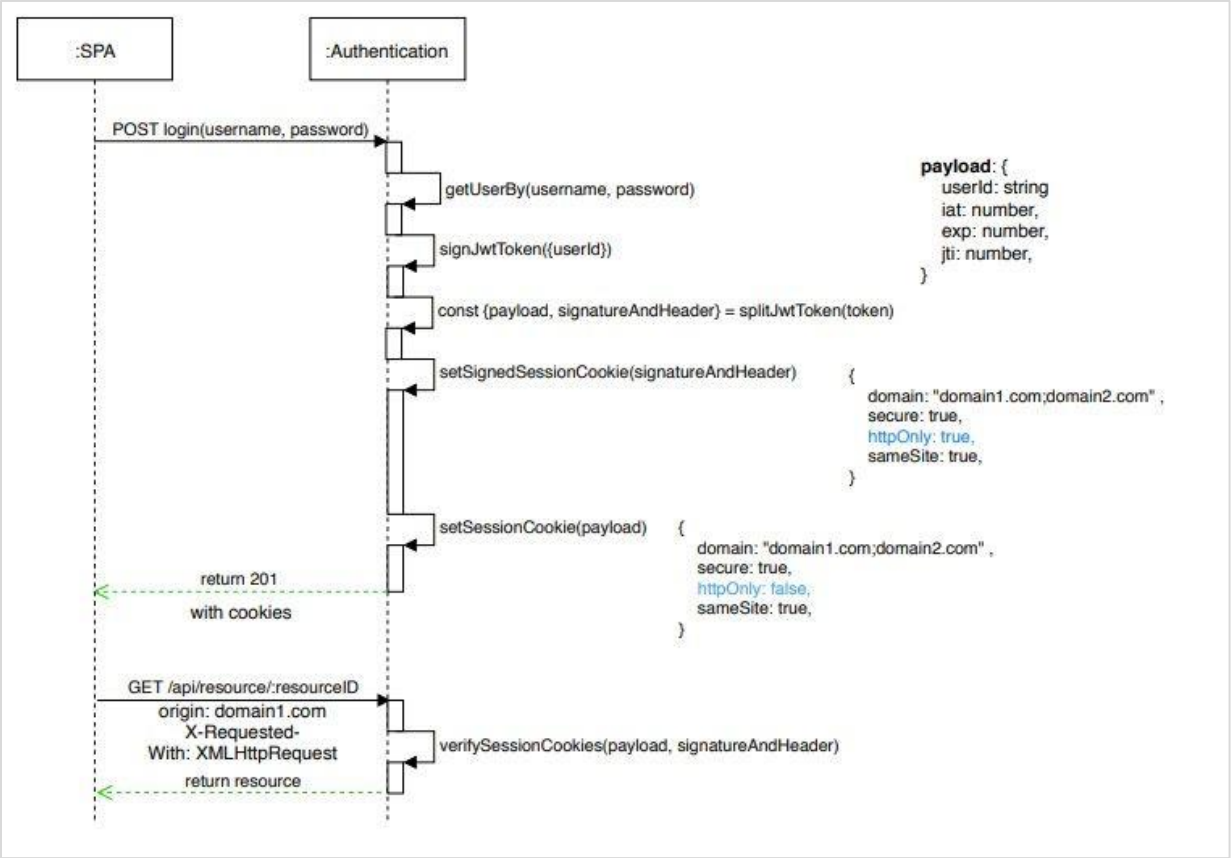
```
{
  "sub": "353454354354353453",
  "exp": 1504699256
}
```

The sub property contains the user identifier, and the exp property contains the expiration timestamp. This type of token is known as a Bearer Token, meaning that it identifies the user that owns it, and defines a user session.

If the payload has data that the front-end needs, we have to provide it somehow. A simple solution is splitting the JWT token into two cookies:

- one holding payload
- one with signature and header data

Payload cookie should have httpOnly flag set to false and signature.header cookie must have httpOnly flag set to true. Here is a diagram that shows the whole flow.



The key advantage of using this type of authentication system is improving the app's overall security. Another less obvious benefit is that the front-end developer will have a far better experience. He can now focus solely on how he will read the content of the payload cookie because he no longer needs to care about session data storage.

JWT Token – The Key Component

Stateless REST API requires storing session data on the client-side. A key component is a JWT token that holds authentication data that can be confidentially transmitted between clients.

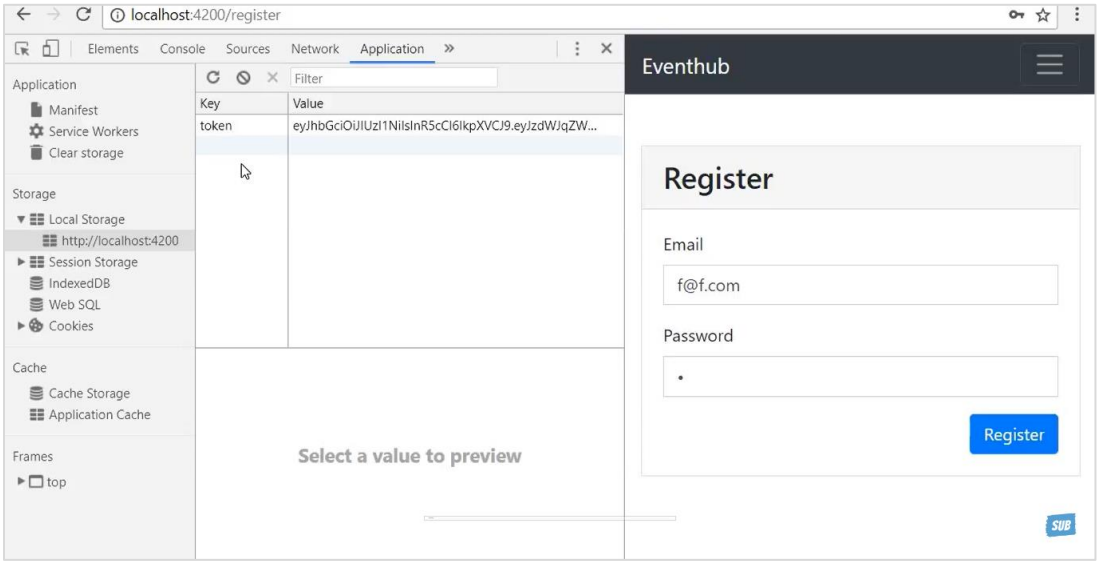
In SPA, developers commonly store the JWT token in the browser's local storage and include it in an authorization header for each request, possibly leading to security threats. Because local storage is readable from JavaScript, a simple cross-site-scripting attack or XSS could read the JWT token and open doors to impersonate a user.

Therefore, we need to find another solution to store the JWT token on the browser.

Example of JSON token

```
loginUser() {
  this._auth.loginUser(this.loginUserData)
    .subscribe(
      res => {
        console.log(res)
        localStorage.setItem('token', res.token)
      },
      err => console.log(err)
    )
}
```

Output:



Activity Twenty-two

Follow the instructions:

1. Demonstrate the steps given below to create an application for login, logout.

Example:

The Login Page

Authentication starts with a Login page, which can be hosted either in our domain or in a third-party domain. In an enterprise scenario, the login page is often hosted on a separate server, which is part of a company-wide Single Sign-On solution.

On the public Internet, the login page might also be:

- hosted by a third-party Authentication provider such as Auth0
- available directly in our single page application using a login screen route or a modal

A separately hosted login page is an improvement security-wise because this way the password is never directly handled by our application code in the first place.

The separately hosted login page can have minimal Javascript or even none at all, and it could be styled to make it look and feel as part of the whole application.

But still, logging in a user via a login screen inside our application is also a viable and commonly used solution, so let's cover that too.

1. Create a LoginComponent

Open LoginComponent template and include below template code.

```
<!-- Page Content -->
<div class="container">
<div class="row">
```

```
<div class="col-lg-12 text-center" style="padding-top: 20px;">
 <div class="container box" style="margin-top: 10px; padding-left: 0px; padding-right: 0px;">
<div class="row">
 <div class="col-12" style="text-align: center;">
 <form [formGroup]="formData" (ngSubmit)="onClickSubmit(formData.value)" class="form-
signin">
<h2 class="form-signin-heading">Please sign in</h2>
 <label for="inputEmail" class="sr-only">Email id </label>
 <input type="text" id="username" class="form-control" formControlName="userName"
placeholder="Username" required autofocus><pre></pre>
<label for="inputPassword" class="sr-only">Password </label>
<input type="password" id="inputPassword" class="form-control" formControlName="password"
placeholder="Password" required><pre></pre>
<button class="btn btn-lg btn-primary btn-block" type="submit">Sign in</button>
</form>
</div>
</div>
</div>
</div>
</div>
</div>
```

Here, we created a reactive form and designed a login form. We have attached the onClickSubmit method to the form submit action.

Open LoginComponent style and include below CSS Code. Here, some styles are added to design the login form.

```
.form-signin {
    max-width: 330px;

    padding: 15px;
    margin: 0 auto;
 }

 input {
    margin-bottom: 20px;
 }
```

Open **LoginComponent** and include below code –

```
import { Component, OnInit } from '@angular/core';
import { FormGroup, FormControl } from '@angular/forms';
import { AuthService } from '../auth.service';
import { Router } from '@angular/router';

@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.css']
})
export class LoginComponent implements OnInit {

  userName!: string;
  password!: string;
  formData!: FormGroup;

  constructor(private authService : AuthService, private router : Router) {
   }
  ngOnInit() {
     this.formData = new FormGroup({
        userName: new FormControl("admin"),
        password: new FormControl("admin"),
     });
  }
  onClickSubmit(data: any) {
     this.userName = data.userName;
     this.password = data.password;

     console.log("Login page: " + this.userName);
     console.log("Login page: " + this.password);

     this.authService.login(this.userName, this.password)
        .subscribe( (data: string) => {
           console.log("Is Login Success: " + data);

          if(data) this.router.navigate(['/signin']);
     });
  }
  }
```

Here,

- Used reactive forms.
- Imported AuthService and Router and configured it in constructor.
- Created an instance of FormGroup and included two instance of FormControl, one for user name and another for password.
- Created a onClickSubmit to validate the user using authService and if successful, navigate to signin .

2. Create a logout component

```
ng g c logout
```

Open LogoutComponent and include below code.

```
import { Component, OnInit } from '@angular/core';
import { Router } from '@angular/router';
import { AuthService } from '../auth.service';

@Component({
  selector: 'app-logout',
  templateUrl: './logout.component.html',
  styleUrls: ['./logout.component.css']
})
export class LogoutComponent implements OnInit {

  constructor(private authService : AuthService, private router: Router) { }

  ngOnInit() {
    this.authService.logout();
    this.router.navigate(['/']);
  }

}
```

Open LogoutComponent template

```
<p>Logged out Successfully!</p>
```

Here, we used logout method of AuthService. Once the user is logged out, the page will redirect to home page (/).

3. Create a service, AuthService to authenticate the user.

```
ng generate service auth
CREATE src/app/auth.service.spec.ts (323 bytes)
CREATE src/app/auth.service.ts (133 bytes)
```

Open AuthService and include below code.

```
import { Injectable } from '@angular/core';
import { delay, Observable, of, tap } from 'rxjs';

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

  isUserLoggedIn: boolean = false;

   login(userName: string, password: string): Observable<any> {
      console.log(userName);
      console.log(password);
      this.isUserLoggedIn = userName == 'admin' && password == 'admin';
      localStorage.setItem('isUserLoggedIn', this.isUserLoggedIn ? "true" : "false");

   return of(this.isUserLoggedIn).pipe(
      delay(1000),
      tap(val => {
         console.log("Is User Authentication is successful: " + val);
      })
   );
   }
   logout(): void {
   this.isUserLoggedIn = false;
      localStorage.removeItem('isUserLoggedIn');
   }
   constructor() { }
}
```

Here,

- We have written two methods, login and logout.
- The purpose of the login method is to validate the user and if the user successfully validated, it stores the information in localStorage and then returns true.
- Authentication validation is that the user name and password should be admin.
- We have not used any backend. Instead, we have simulated a delay of 1s using Observables.
- The purpose of the logout method is to invalidate the user and removes the information stored in localStorage.

Angular Guard

A route guard is an important feature of the Angular Router that allows or denies the user access to the route pages based on some logic, based on weather user is logged in or not.

- ROUTE-GUARDS are very much important in web app having login/logout scenarios.
- It's commonly used to check if a user is logged in and has the authorization to access a page.
- We can easily manage which page is allowed for logged in user and which for non-logged in users.

In a web application, a resource is referred by url. Every user in the system will be allowed access a set of urls. For example, an administrator may be assigned all the url coming under administration section.

As we know already, URLs are handled by Routing. Angular routing enables the urls to be guarded and restricted based on programming logic. So, a url may be denied for a normal user and allowed for an administrator.

Angular provides a concept called Router Guards which can be used to prevent unauthorised access to certain part of the application through routing. Angular provides multiple guards and they are as follows:

- CanActivate − Used to stop the access to a route.
- CanActivateChild − Used to stop the access to a child route.

- CanDeactivate − Used to stop ongoing process getting feedback from user. For example, delete process can be stop if the user replies in negative.
- Resolve − Used to pre-fetch the data before navigating to the route.
- CanLoad − Used to load assets.

CanActivate

The Angular CanActivate guard decides, if a route can be activated (or component gets rendered). We use this guard, when we want to check on some condition, before activating the component or showing it to the user. This allows us to cancel the navigation.

Interface that a class can implement to be a guard deciding if a route can be activated. If all guards return true, navigation continues. If any guard returns false, navigation is cancelled. If any guard returns a UrlTree, the current navigation is cancelled and a new navigation begins to the UrlTree returned from the guard.

Use cases for the CanActivate Guard

- Checking if a user has logged in
- Checking if a user has permission

One of the use case of this guard is to check if the user has logged in to the system. If user has not logged in, then the guard can redirect him to login page.

How to use CanActivate Guard

First, we need to create a Angular Service.

The service must import & implement the CanActivate Interface. This Interface is defined in the @angular/router module. The Interface has one method i.e. canActivate. We need to implement it in our Service. The details of the CanActivate interface is as shown below.

```
interface CanActivate {
  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):
Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree
}
```

The method gets the instance of the ActivatedRouteSnapshot & RouterStateSnapshot. We can use this to get access to the route parameter, query parameter etc.

The guard must return true/false or a UrlTree . The return value can be in the form of observable or a promise or a simple boolean value.

- A route can have more than one canActivate guard.
- If all guards returns true, navigation to the route will continue.
- If any one of the guard returns false, navigation will be cancelled.

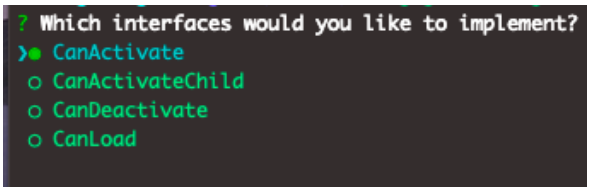
If any one of the guard returns a UrlTree, current navigation will be cancelled and a new navigation will be kicked off to the UrlTree returned from the guard.

Implementation of route guard

- We can add a route guard by implementing the CanActivate interface available from the @angular/router package and extends the canActivate() method which holds the logic to allow or deny access to the route.
- For example, the following guard will check value of userLoggedIn and allow access to a route accordingly:

4. Create a guard using below command

```
ng generate guard learners
```



Press Enter and the LearnersGuard is ready.

Open LearnersGuard and include below code –

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

```
import { ActivatedRouteSnapshot, CanActivate, RouterStateSnapshot, UrlTree } from
'@angular/router';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class LearnersGuard implements CanActivate {
  router: any;
  canActivate(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<boolean | UrlTree>
| boolean | UrlTree {
      let url: string = state.url;
      let val: string;

      return this.checkLogin(url);
  }
  checkLogin(url: string) {
    console.log("Url: "+URL)
    let val = localStorage.getItem('isUserLoggedIn');
    if(val != null && val == "true"){
      if(url == "/login")
         this.router.parseUrl('/expenses');
      else
         return true;
   } else {
      return this.router.parseUrl('/login');
   }
}
  }
```

Here,

- checkLogin will check whether the localStorage has the user information and if it is available, then it returns true.
- If the user is logged in and goes to login page, it will redirect the user to learnerlist page
- If the user is not logged in, then the user will be redirected to Home page.

Open AppRoutingModule (src/app/app-routing.module.ts) and update below code –

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { LoginComponent } from './login/login.component';
import { AboutComponent } from './about/about.component';
import { ContactComponent } from './contact/contact.component';
import { HomeComponent } from './home/home.component';
import { FAQComponent } from './faq/faq.component';
import { LearnersGuard } from './learners.guard';
import { LearnerslistComponent } from './learnerslist/learnerslist.component';
import { LearnerdetailsComponent } from './learnerdetails/learnerdetails.component';
import { LogoutComponent } from './logout/logout.component';

const routes: Routes = [
  {path:'learnerdetails', component: LearnerdetailsComponent},
  {path:'user-details', component: UserDetailsComponent},
  {path: 'login', component: LoginComponent},
  { path: 'logout', component: LogoutComponent },
  {path: 'about', component: AboutComponent},
  {path: 'settings', component: SettingsComponent},
  {path: 'contact', component: ContactComponent},
  {path: 'learnerslist', component: LearnerslistComponent, canActivate: [LearnersGuard]},
  {path: 'learnerslist/detail/:id', component: LearnerdetailsComponent, canActivate:
[LearnersGuard]},
  { path: '', redirectTo: 'signin', pathMatch: 'full' },
  {path: 'home', component: HomeComponent},
 },
]
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Here,

- We have used canActivate attribute to protect a route with the guard for learnerslistComponent and LearnersdetailsComponent.
- We can apply 'Route Guards' to an authenticated area of our web app, or an admin section that requires special permissions to be accessed.
- Once we attach this guard to a route, the canActivate() method will fire before the route is activated, means

before it opens the actual path or before it loads actual component.

- The logic will first check if the user is valid, and if not it will navigate to the login route.
- Note it also grabs the current URL and sets is as a query parameter, so it would be something like /login?return=%2Fusers%2Fabc123 (the URL is encoded). This is the case when we are trying to access user like this: /users/abc123 and the canActivate detects it unauthorized user.

5. Open AppComponent template and add two login and logout link.

```
<div class="topnav">
     <li *ngIf="isUserLoggedIn; else isLogOut"></li>
     <a class="nav-link" routerLink="/logout">Logout</a>
     <a routerLink="faq" routerLinkActive="active">FAQ</a> 
         <a routerLink="contact" routerLinkActive="active">Contact</a> 
         <a  routerLink="about" routerLinkActive="active">Registration</a> 
         <ng-template #isLogOut></ng-template>
         <a  routerLink="login" routerLinkActive="active">Login</a> 
         <a [routerLink]="['/home']" [queryParams]="{ category: 'information'}">
          Home
        </a>

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

6. Open AppComponent and update below code –

```
import { ChangeDetectionStrategy, Component, VERSION } from '@angular/core';
import { FormBuilder, FormControl, FormsModule } from '@angular/forms';
import { ReactiveFormsModule } from '@angular/forms';
import { AuthService } from './auth.service';

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

export class AppComponent {
  title = 'Angular';
  isUserLoggedIn = false;
```

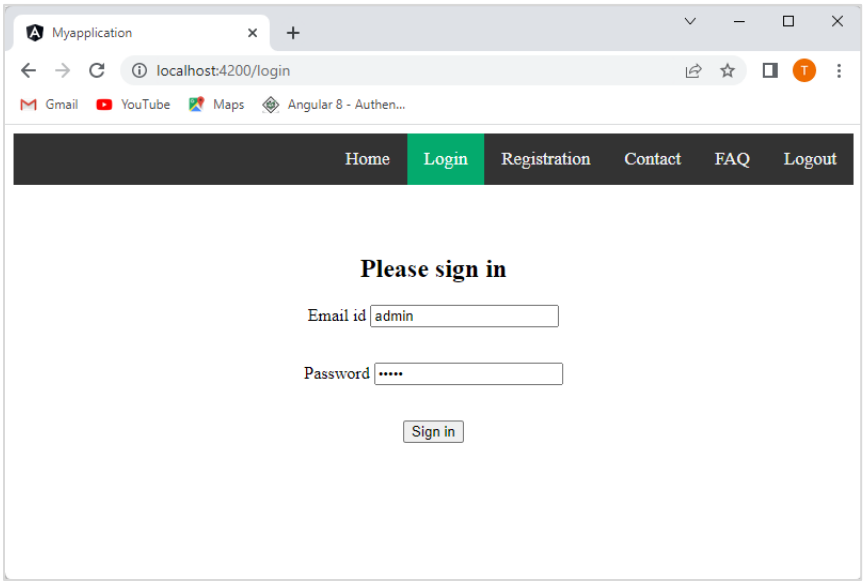
```
  constructor(private authService: AuthService) {}

  ngOnInit() {
     let storeData = localStorage.getItem("isUserLoggedIn");
     console.log("StoreData: " + storeData);

     if( storeData != null && storeData == "true")
        this.isUserLoggedIn = true;
     else
        this.isUserLoggedIn = false;
  }
}
```

Here, we have added the logic to identify the user status so that we can show login / logout functionality.

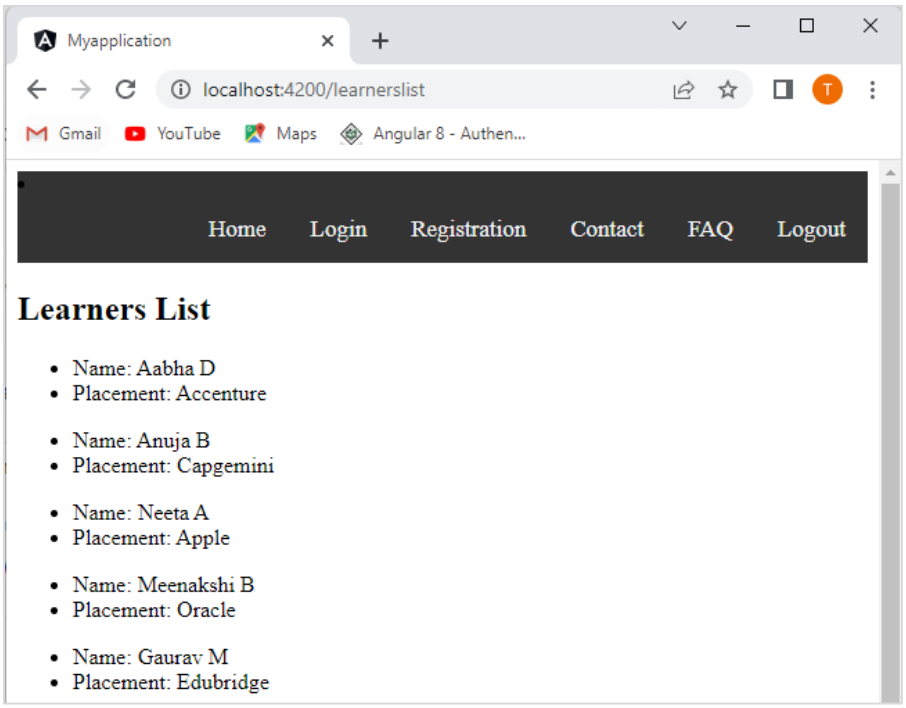
7. Open AppModule (src/app/app.module.ts) and configure ReactiveFormsModule

```
import { ReactiveFormsModule } from '@angular/forms';
imports: [
  ReactiveFormsModule
]
```

Now, run the application and the application opens the login page.



Enter admin and admin as username and password and then, click submit. The application process the login and redirects the user to learnerlist page as shown below –



Finally, you can click logout and exit the application.