

Angular JS Assignments

Contents

Assignment 1: Angular 2 Purpose	4
Assignment 2: Setting up Angular 2 in Visual Studio	6
Assignment 3: Run angular 2 app using f5 from visual studio	13
Assignment 4: Angular 2 Components.....	16
Assignment 5: Angular template vs templateUrl	18
Assignment 6: Angular 2 nested components	20
Assignment 7: Styling angular 2 components.....	26
Assignment 8: Angular interpolation	31
Assignment 9: Property binding in Angular 2	34
Assignment 10: Property binding in Angular 2	36
Assignment 11: Angular attribute binding.....	40
Assignment 12: Class binding in angular 2.....	43
Assignment 13: Style binding in angular 2.....	47
Assignment 14: Angular2 event binding.....	49
Assignment 15: Two way data binding in angular 2	53
Assignment 16: Angular ngFor directive.....	55
Assignment 17: Angular ngFor trackBy.....	59
Assignment 18: Angular pipes.....	67
Assignment 19: Angular custom pipe	70
Assignment 20: Angular 2 container and nested components.....	72
Assignment 21: Angular component input properties	79
Assignment 22: Angular component output properties.....	84
Assignment 23: Interfaces in Angular 2	90
Assignment 24: Angular component lifecycle hooks.....	94
Assignment 25: Angular services tutorial	98
Assignment 26: Angular and ASP.NET Web API.....	102
Assignment 27: Angular 2 http service tutorial	107
Assignment 28: Angular 2 http error handling	114
Assignment 29: Using Bootstrap with Angular 2	120
Assignment 30: Angular 2 routing tutorial.....	126
Assignment 31: Angular 2 route parameters.....	130
Assignment 32: Angular dependency injection	136
Assignment 33: Why dependency injection	139
Assignment 34: Angular singleton service	142
Assignment 35: Angular Injector.....	150
Assignment 36: Angular root injector	153
Assignment 37: Angular router navigate method	156
Assignment 38: Promises in angular 2 example	158
Assignment 39: Angular promises vs observables.....	161
Assignment 40: Observable retry on error	167
Assignment 41: Angular observable unsubscribe	174
Assignment 42: Angular services tutorial	178
Assignment 43: CRUD Operations Sample Project	181

Assignment 1: Angular 2 Purpose

Angular 1 was released in October 2010, and by far the most popular JavaScript framework available for creating web applications. Many developers are already using Angular 1, so the obvious question that comes to our mind is why we should use Angular 2.

Angular 2 is not a simple upgrade from angular 1. Angular 2 is completely rewritten, so it has lot of improvements when compared with Angular 1. Let's look at a few of these improvements.

Performance: From a performance standpoint, Angular 2 has faster initial loads, change detection, and improved rendering time. Not just performance, we also have improved modularity, Dependency injection and testability. According to angular conference meetup, Angular 2 is 5 times faster compared to AngularJS 1.

Mobile Support: Angular 1 was not built for mobile devices. It is possible to run Angular 1 on mobile but we will have to use other frameworks. Angular 2 on the other hand is designed from the ground up with mobile support. Mobile device features and limitations like touch interfaces, limited screen real estate, and mobile hardware have all been considered in Angular 2. So with Angular 2 we can build a single application that works across mobile and desktop devices.

Component Based Development: Component based web development is the future of web development. In Angular 2, "everything is a component". Components are the building blocks of an Angular application. The advantage of the component-based approach is that, it facilitates greater code reuse. From unit testing standpoint, the use of components make Angular2 more testable.

More language choices : There are several languages that we can use to develop Angular applications. To name a few, we have

1. ECMAScript 5
2. ECMAScript 6 (also called ES 2015)
3. TypeScript etc.

Besides these 3 languages we can also use Dart, PureScript, Elm, etc, but among all these, TypeScript is the most popular language.

Angular 2 itself, is built using TypeScript. TypeScript has great support of ECMAScript 6 standard. So the obvious questions that come to our mind at this point are

1. What is ECMAScript
2. What is Type Script

What is ECMAScript : The JavaScript language standard is officially called ECMAScript. Over the past several years many versions of ECMAScript were released starting with ECMAScript version 1 all the way till ECMAScript version 7.

Most of the modern browsers available today support ECMAScript 5. The browser support for ECMAScript 6 is still incomplete. However, using a process called Transpilation, ECMAScript 6 can be converted to ECMAScript 5 which is supported by all the modern browsers. ECMAScript 6 is officially known as ECMAScript 2015. ECMAScript 2015 introduced several new features like classes, modules, arrow functions etc.

If you are interested in reading more about the ECMAScript standard and what these different versions of ECMAScript have to offer, please refer to the the following Wikipedia article.

<https://en.wikipedia.org/wiki/ECMAScript>

What is TypeScript : TypeScript is a free and open-source programming language developed by Microsoft. It is a superset of JavaScript and compiles to JavaScript through a process called transpilation. Using TypeScript to build angular applications provides several benefits.

1. Intellisense
2. Autocompletion
3. Code navigation
4. Advanced refactoring
5. Strong Typing
6. Supports ES 2015 (also called ES 6) features like classes, interfaces and inheritance. If you have any experience with object oriented programming languages like C# and Java, learning TypeScript is easy.

Because of all these benefits writing, maintaining and refactoring applications can be an enjoyable experience. So obviously TypeScript has become the number one choice of many developers for developing Angular applications.

For this course we will be using Visual Studio as the code editor. Besides Visual Studio, TypeScript is supported by several other editors like

1. Visual Studio Code
2. Eclipse
3. WebStorm
4. Atom
5. Sublime Text etc.

So you can use any favourite editor of your choice to build Angular 2 applications using TypeScript.

Assignment 2: Setting up Angular 2 in Visual Studio

In this assignment we will discuss **how to set up Angular 2 in Visual Studio**.

Step 1 : The first step is to install Node.js and npm. It is recommended that you have node version 4.6.x or greater and npm 3.x.x or greater. To check the versions that you have on your machine type the following commands in a command window.

```
node -v  
npm -v
```

You can get the latest version of Node.js from the following website. Click on the correct download link depending on the Operating System you have.

<https://nodejs.org/en/download/>

To find out if you have 32 - bit or 64 - bit operating system

1. Click the Start icon ,
2. Type "System" in the Start Search box, and then click System Information in the Programs list.
3. Select "System Summary"
4. In the right pane if :
 - System type is x64-based PC, then you have 64-bit operating system
 - System type is x86-based PC, then you have 32-bit operating system

Step 2 : Make sure you have Visual Studio 2015 Update 3 installed. To check the version of Visual Studio you have click on the "**Help**" menu and then select "**About Microsoft Visual Studio**". The following are the download links if you don't have Visual Studio 2015 Update 3.

[Visual Studio Enterprise 2015 - Update 3](#)

[Visual Studio Professional 2015 - Update 3](#)

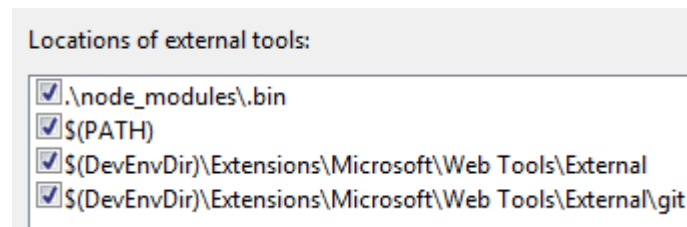
[Visual Studio Community 2015 - Update 3](#)



Microsoft Visual Studio Enterprise 2015
Version 14.0.25425.01 Update 3
© 2016 Microsoft Corporation.
All rights reserved.

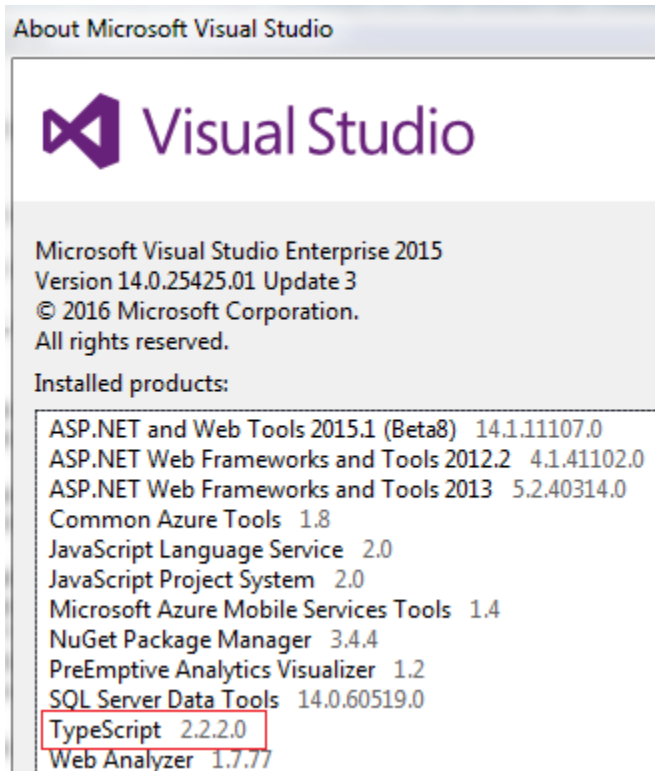
Step 3 : Configure environment settings for node and npm in Visual Studio

1. In Visual Studio click on Tools - Options.
2. In the "Options" window, expand "Projects and Solutions" and select "External Web Tools"
3. In the right pane, move the global \$(PATH) entry to be above the internal path \$(DevEnvDir) entries. This tells Visual Studio to look for external tools (like npm) in the global path before the internal path.
4. Click "OK" to close the "Options" window and then restart Visual Studio for the changes to take effect



Step 4 : Install TypeScript for Visual Studio 2015

1. To develop Angular applications you need TypeScript 2.2.0 or later
2. To check the version of TypeScript, click on the "Help" menu in Visual Studio and select "About Microsoft Visual Studio"



3. Download and install the latest version of TypeScript for Visual Studio 2015 from the following URL <https://www.microsoft.com/en-us/download/details.aspx?id=48593>
4. After installing TypeScript, the installation wizard prompts you to restart Visual Studio. So, please restart Visual Studio for the changes to take effect.

Step 5 : Create Empty ASP.NET Web Application project

1. Run Visual Studio as Administrator
2. Click on File - New Project
3. Select "Web" under "Visual C#". From the right pane select "ASP.NET Web Application"
4. Name the project "Angular2Demo"
5. On the next screen, select "Empty" template and click "OK"

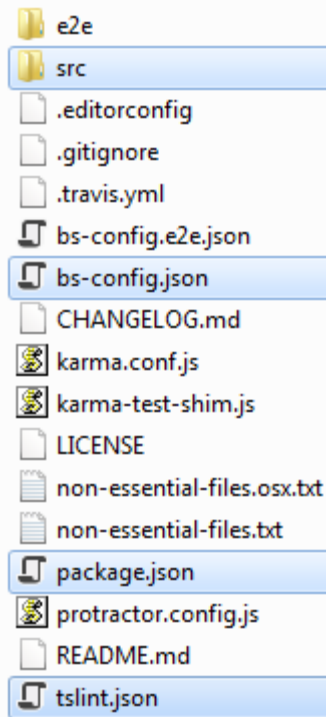
Step 6 : Download the "**Quick Start Files**" from the Angular web site using the link below. Extract the contents of the downloaded .ZIP folder.

<https://github.com/angular/quickstart>

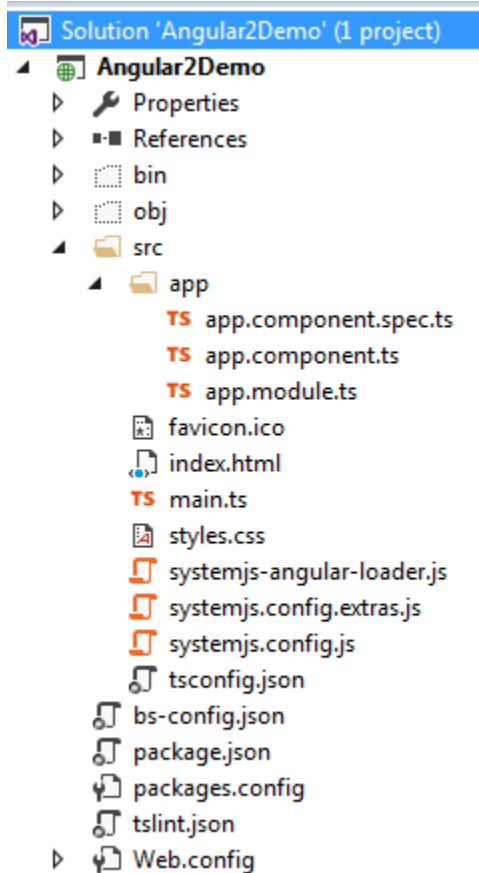
Step 7 : Copy the required "Starter files" to the web application project

We do not need all the starter files that we downloaded. As you can see from the image below, we need 4 folders/files

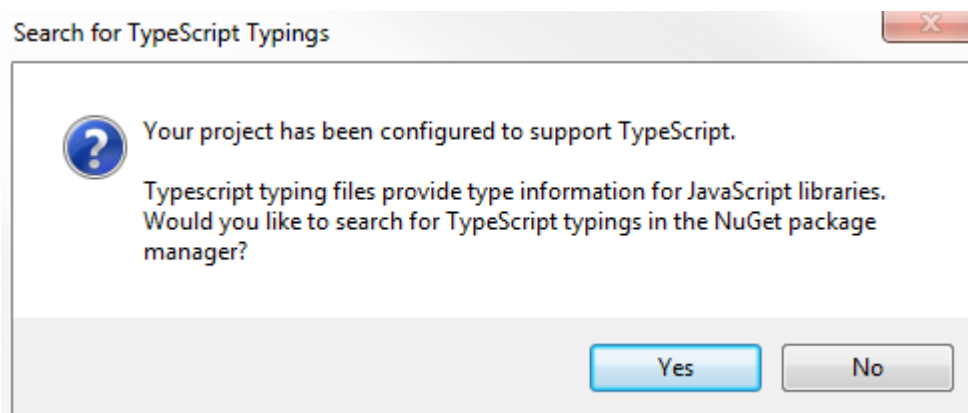
- src folder and it's contents
- bs-config.json
- package.json
- tslint.json



Copy the above files/folders and paste them in the root directory of "Angular2Demo" web application project. Now click "Show All File" icon in "Solution Explorer" and include all the copied files/folders in the project. At this stage your project structure in Visual Studio should be as shown below.

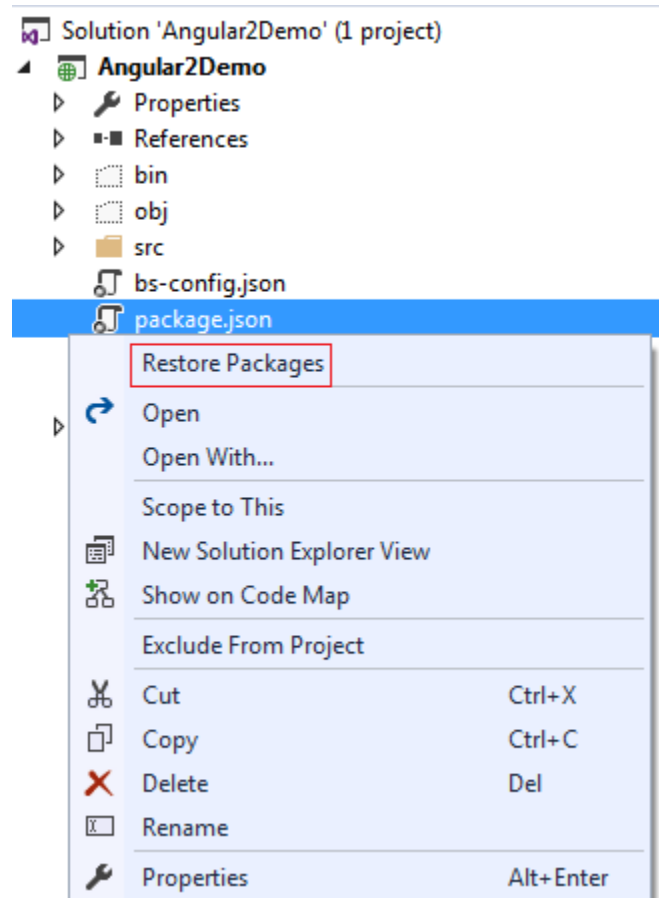


When including the files in the project if you get a prompt to **"Search for Typescript Typings"** click "No"



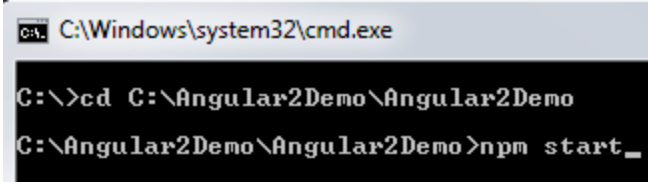
Step 8 : Restore the required packages.

In the "Solution Explorer" right click on "package.json" file and select "Restore Packages" from the context menu. This takes a few minutes to load all the modules. You can see the status in "Visual Studio Output" window. After the restoration is complete, you will see a message "Installing Packages Complete". To see all the installed node modules, click on "Show all Files" icon in Solution Explorer. DO NOT include "node_modules" folder in the project.



Step 9 : Run the project

1. In the "RUN" window type "cmd" and press enter
2. Change the directory in the command prompt to the directory where you have the web application project. The web application project in "C:\Angular2Demo\Angular2Demo". So type CD C:\Angular2Demo\Angular2Demo and upon pressing the enter key in the root folder.
3. Type "npm start" and press "Enter" key



```
C:\Windows\system32\cmd.exe
C:\>cd C:\Angular2Demo\Angular2Demo
C:\Angular2Demo\Angular2Demo>npm start_
```

4. This launches the TypeScript compiler (tsc) which compile the application and wait for changes. It also starts the lite-server and launches the browser where you will see the output - Hello Angular.
5. At this point, open "app.component.ts" file from "Solution Explorer". This file is present in "app" folder in "src" folder.
6. Change "name" value from "Angular" to "Angular 2!" and you will see the changes reflected on the web page automatically.

At the moment we do not have the capability to run the project by pressing F5 or CTRL + F5.

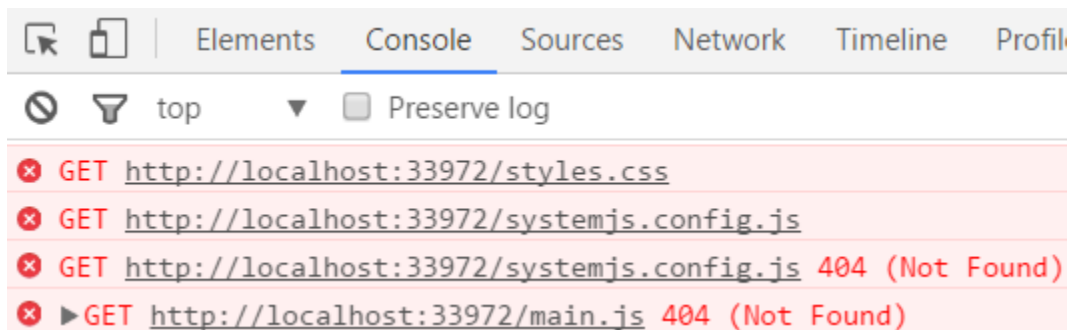
Assignment 3: Run angular 2 app using f5 from visual studio

In this assignment we will discuss how to run angular 2 application from visual studio using F5 or CTRL + F5.

At the moment, if we run the application from Visual Studio, using F5 or CTRL+F5, we get the message "Loading AppComponent content here ..." but nothing happens beyond that. To be able to run the application using F5 or CTRL+F5 we need to make the following changes.

1. Launch browser developers tools by pressing F12. Notice we have "404 Not Found" errors for the following files.

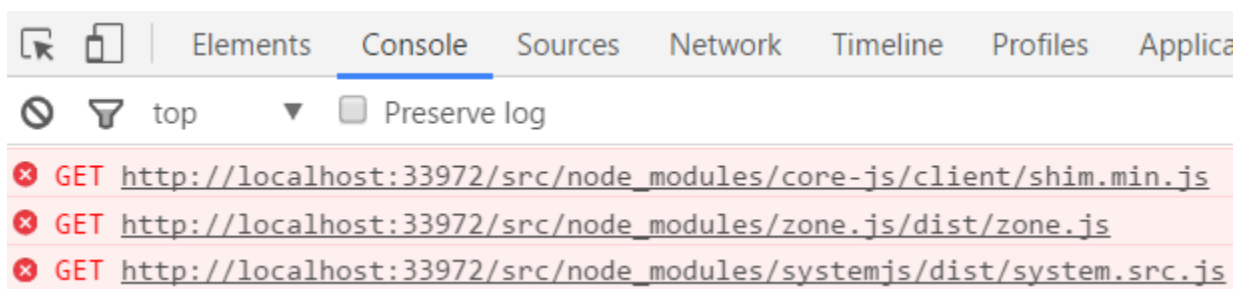
- styles.css
- systemjs.config.js
- main.js



All these files are present in "src" folder. So to fix these "404 Not Found" errors, in index.html file, change `<base href="/">` to `<base href="/src/">`

2. Save the changes and reload the page. At this point we get another set of "404 Not Found" errors for the following files.

- shim.min.js
- zone.js
- system.src.js



```
<script src="node_modules/core-js/client/shim.min.js"></script>
<script src="node_modules/zone.js/dist/zone.js"></script>
<script src="node_modules/systemjs/dist/system.src.js"></script>
```

To fix these errors, in index.html change the above script references as shown below. Notice, we have included "/" just before node_modules

```
<script src="/node_modules/core-js/client/shim.min.js"></script>
<script src="/node_modules/zone.js/dist/zone.js"></script>
<script src="/node_modules/systemjs/dist/system.src.js"></script>
```

Also in systemjs.config.js file, CHANGE
'npm:': 'node_modules/' TO 'npm:': '/node_modules/'

At this point reload the page and you will see "Hello Angular" message without any errors.

One important point to keep in mind is that, now we will not be able to run the application using "npm start" command.

We still have one more issue. Let us first understand the issue.

1. Expand "app" folder. This folder is inside "src" folder
2. Open "app.component.ts" file
3. Set name="Angular 2!" from name="Angular"
4. Save the changes and reload the web page
5. Notice, we do not see the changes on the web page
6. However, if we run the application by pressing F5 or CTRL+F5 from visual studio we see the changes in the browser.

So what is the issue?

TypeScript is not compiled to JavaScript when we save the file and this the reason we do not see the changes in the browser. However, when we run the application by pressing F5 or CTRL+F5 from visual studio TypeScript is compiled to JavaScript and we see the changes.

If you want Visual Studio to compile TypeScript to JavaScript when the changes are saved, we need to turn this feature "ON" by including the following setting in tsconfig.json file. You will find this file in "src" folder. Notice we have set "compileOnSave" to true. With this change tsconfig.json file looks as shown below.

```
{
  "compileOnSave": true,
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "moduleResolution": "node",
    "sourceMap": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "lib": [ "es2015", "dom" ],
    "noImplicitAny": true,
    "suppressImplicitAnyIndexErrors": true
  }
}
```

```
}  
}
```

At this point TypeScript is automatically compiled to JavaScript when the file is saved, so the changes are reflected in the browser when the page is reloaded.

At the moment, we are using Visual Studio built-in IIS express server. In a later assignment in this course we will discuss how to use full blown IIS instead of Visual Studi built-in IIS express.

Should I learn AngularJS1 before learning Angular 2?

NO, Angular 2 is completely rewritten and very different from AngularJS1, so there is no need to learn AngularJS 1 before learning Angular 2.

Assignment 4: Angular 2 Components

In this assignment tutorial we will discuss - **What is a component in Angular 2**

What is a component in Angular 2?

A component in Angular is a class with a template and a decorator. So in simple terms a component in Angular is composed of these 3 things

- **Template** - Defines the user interface. Contains the HTML, directives and bindings.
- **Class** - Contains the code required for template. Just like a class in any object oriented programming language like C# or Java, a class in angular can contain methods and properties. Properties contain the data that we want to display in the view template and methods contain the logic for the view. We use TypeScript to create the class.
- **Decorator** - We use the Component decorator provided by Angular to add metadata to the class. A class becomes an Angular component, when it is decorated with the Component decorator.

Component Example : We have downloaded quick start files from the Angular Website. One of the files in these quick start files, is the app.component.ts file. You can find this file in the "app" folder. This file contain a component. The name of the component is **AppComponent**. The AppComponent is the root component of the application.

The commented code in the following example and it should be self-explanatory.

```
// Component decorator is provided by the Angular core library, so we
// have to import it before using it. The import keyword is similar to
// using keyword in C#. Any exported member can be imported using import
// keyword.
import { Component } from '@angular/core';

// The class is decorated with Component decorator which adds metadata
// to the class. We use the @ symbol to apply a decorator to the class
// Applying a decorator on a class is similar to applying an attribute
// to a class in C# or other programming languages. Component is just
// one of the several built-in decorators provided by angular. We will
// discuss the other decorators provided by angular in upcoming assignments
@Component({
  // component has several properties. Here we are using just 2. For
  // the full list of properties refer to the following URL
  // https://angular.io/docs/ts/latest/api/core/index/Component-decorator.html
  // To use this component on any HTML page we specify the selector
  // This selector becomes the directive <my-app> on the HTML page
  // At run time, the directive <my-app> is replaced by the template
  // HTML specified below
  selector: 'my-app',
  // The template contains the HTML to render. Notice in the HTML
  // we have a data-binding expression specified by double curly
  // braces. We have a default value "Angular" assigned to "name"
```

```

    // property in the AppComponent class. This will be used at runtime
    // inplace of the data-binding expression
    template: `<h1>Hello {{name}}</h1>`,
  })
  // export keyword allows this class to be exported, so other components
  // in the application can import and use it if required
  export class AppComponent {
    // name is a property and the data type is string and
    // has a default value "angular"
    name: string = 'Angular';
  }

```

Notice in the **index.html** page, we have used the **AppComponent** using the directive `<my-app>`. At runtime `<my-app>` directive is replaced with the HTML we specified using the selector property in the component decorator.

```

<!DOCTYPE html>
<html>
<head>
  <title>Angular QuickStart</title>
  <base href="/src/">
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" href="styles.css">

  <!-- Polyfill(s) for older browsers -->
  <script src="/node_modules/core-js/client/shim.min.js"></script>
  <script src="/node_modules/zone.js/dist/zone.js"></script>
  <script src="/node_modules/systemjs/dist/system.src.js"></script>

  <script src="systemjs.config.js"></script>
  <script>
    System.import('main.js').catch(function (err) { console.error(err); });
  </script>
</head>
<body>
  <my-app>Loading AppComponent content here ...</my-app>
</body>
</html>

```

When we build the project in Visual Studio TypeScript is compiled to JavaScript which the browser understands and renders. Our TypeScript code for this component is present in `app.component.ts` file. Notice a corresponding **app.component.js** file is generated on build. To see the generated `.js` file click on show-all-files icon in solution explorer. Besides `.js` files, there are several other files.

Assignment 5: Angular template vs templateUrl

In this assignment we will discuss **template** and **templateurl** properties of the Component decorator.

Notice the code we have implemented in app.component.ts file. We have embedded the view template inline in the .ts file.

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

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

The view template is inline in a pair of backtick characters. The first question that comes to our mind is can't we include the HTML in a pair of single or double quotes. The answer is "YES" we can as long as the HTML is in a single line. So this means the above code can be rewritten using a pair of single quotes as shown below.

```
template: '<h1>Hello {{name }}</h1>'
```

We can also replace the pair of single quotes with a pair of double quotes as shown below, and the application still continues to work exactly the same way as before.

```
template: "<h1>Hello {{name }}</h1>"
```

The obvious next question that comes to our mind is when should we use backticks instead of single or doublequotes

If you have the HTML in more than one line, then you have to use backticks instead of single or double quotes as shown below. If you use single or double quotes instead of backticks you will get an error.

```
template: `<h1>
  Hello {{name }}
</h1>`
```

Instead of using an inline view template, you can have it in a separate HTML file. Here are the steps to have the view template in a separate HTML file

Step 1 : Right click on the **"app"** folder and add a new HTML file. Name it **"app.component.html"**.

Step 2 : Include the following HTML in **"app.component.html"**

```
<h1>
  Hello {{name}}
</h1>
```

Step 3 : In **"app.component.ts"**, reference the external view template using **templateUrl** property as shown below. Notice instead of the **"template"** property we are using **"templateUrl"** property. Please note that **templateUrl** path is relative to index.html

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

```
@Component({
  selector: 'my-app',
  templateUrl: 'app/app.component.html'
})
export class AppComponent {
  name: string = "Angular";
}
```

What are the differences between template and templateUrl properties and when to use one over the other

Angular2 recommends to extract templates into a separate file, if the view template is longer than 3 lines. Let's understand why is it better to extract a view template into a separate file, if it is longer than 3 lines.

With an inline template

1. We lose Visual Studio editor intellisense, code-completion and formatting features.
2. TypeScript code is not easier to read and understand when it is mixed with the inline template HTML.

With an external view template

1. We have Visual Studio editor intellisense, code-completion and formatting features and
2. Not only the code in **"app.component.ts"** is clean, it is also easier to read and understand

Assignment 6: Angular 2 nested components

In this assignment we will discuss nesting angular components i.e including a component inside another component.

As we already know Angular 2 is all about components. A component in Angular allows us to create a reusable UI widget. A component can be used by any other component. Let's look at a simple example of nesting a component inside another component.

Here is what we want to do. Create a page that displays Employee details as shown below.

Employee Details

First Name	Tom
Last Name	Hopkins
Gender	Male
Age	20

As you can see from the image below we want to create 2 components

- **AppComponent** - This component is the root component and displays just the page header
- **EmployeeComponent** - This component is the child component and displays the Employee details table. This child component will be nested inside the root AppComponent

Page Header "**Employee Details**" comes from the root component - **AppComponent**

Employee Details

First Name	Tom
Last Name	Hopkins
Gender	Male
Age	20

Employee Details table comes from another component called- **EmployeeComponent**

Step 1 : Right click on the "App" folder and add a new folder. Name it "employee". We will create our EmployeeComponent in this folder.

Step 2 : Right click on the "employee" folder and add a new HTML page. Name it employee.component.html. Copy and paste the following HTML.

```
<table>
  <tr>
    <td>First Name</td>
    <td>{{firstName}}</td>
  </tr>
  <tr>
    <td>Last Name</td>
    <td>{{lastName}}</td>
  </tr>
  <tr>
    <td>Gender</td>
    <td>{{gender}}</td>
  </tr>
  <tr>
    <td>Age</td>
    <td>{{age}}</td>
  </tr>
```

</table>

Step 3 : Right click on the "**employee**" folder and add a new TypeScript file. Name it employee.component.ts. Copy and paste the following code in it. At this point we have our child component EmployeeComponent created. Next let's create the root component - AppComponent.

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

@Component({
  selector: 'my-employee',
  templateUrl: 'app/employee/employee.component.html'
})
export class EmployeeComponent {
  firstName: string = 'Tom';
  lastName: string = 'Hopkins';
  gender: string = 'Male';
  age: number = 20;
}
```

Step 4 : We are going to use the root component to just display the page header. So in "app.component.ts" file, include the following code. Notice, since the View Template HTML is just 3 lines we have used an inline template instead of an external template. Angular2 recommends to extract templates into a separate file, if the view template is longer than 3 lines.

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

@Component({
  selector: 'my-app',
  template: `<div>
    <h1>{{pageHeader}}</h1>
  </div>`
})
export class AppComponent {
  pageHeader: string = 'Employee Details';
}
```

At this point if we run the application, we only see the page header - "Employee Details", but not the table which has the employee details. To be able to display employee details table along with the page header, we will have to nest EmployeeComponent inside AppComponent. There are 2 simple steps to achieve this.

Step 1 : In "app.module.ts" file we need to do 2 things as shown below.

- Import EmployeeComponent
- Add EmployeeComponent to the declarations array

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

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

@NgModule({
  imports: [BrowserModule],
  declarations: [AppComponent, EmployeeComponent],
  bootstrap: [AppComponent]
})

export class AppModule { }
```

What is AppModule

AppModule is the root module which bootstraps and launches the angular application. You can name it anything you want, but by convention it is named AppModule.

It imports 2 system modules - BrowserModule and NgModule

- BrowserModule - Every application that runs in a browser needs this module. In a later assignment in this course we will discuss NgIf and NgFor directives which are also provided by this module.
- NgModule - @component decorator adds metadata to an angular component class, similarly @NgModule decorator adds metadata to the angular module class.

We discussed that if a class is decorated @component decorator then that class becomes an angular component. Similarly if a class is decorated with @NgModule decorator then that class becomes an angular module.

Properties of the @NgModule decorator

- imports - Imports the BrowserModule required for an angular application to run in a web browser
- declarations - Contains the components registered with this module. In our case we have two - AppComponent and EmployeeComponent
- bootstrap - Contains the root component that Angular creates and inserts into the index.html host web page

Step 2 : In "app.component.ts" file include "my-employee" as a directive. Remember in "employee.component.ts" file we used "my-employee" as a selector.

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

```
@Component({
  selector: 'my-app',
  template: `<div>
    <h1>{{pageHeader}}</h1>
    <my-employee></my-employee>
  </div>`
})
export class AppComponent {
  pageHeader: string = 'Employee Details';
}
```

Run the application and you will see both page header and the employee details table. The employee details table is not styled very well. To style the table, include the following styles for <td> and <table> elements in styles.css file.

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

```
td {
  border: 1px solid black;
}
```

Run the application and you will now see the employee details table with the specified styles applied. At this point, launch browser developers tools and click on the "Elements" tab and notice <my-app> and <my-employee> directives in the rendered HTML.

```
..<!DOCTYPE html> == $0
<html>
  ▶ <head>...</head>
  ▼ <body>
    ▼ <my-app ng-version="4.0.3">
      <h1>Employee Details</h1>
      ▼ <my-employee>
        ▼ <table>
          ▼ <tbody>
            ▶ <tr>...</tr>
            ▶ <tr>...</tr>
            ▶ <tr>...</tr>
            ▶ <tr>...</tr>
          </tbody>
        </table>
      </my-employee>
    </my-app>
  </body>
</html>
```


Assignment 7: Styling angular 2 components

In this assignment we will discuss the different options available to apply styles to Angular Components.

In our previous assignment we have built "employee" component which displays employee details in a table as shown below.

First Name	Tom
Last Name	Hopkins
Gender	Male
Age	20

employee.component.ts

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

@Component({
  selector: 'my-employee',
  templateUrl: 'app/employee/employee.component.html'
})
export class EmployeeComponent {
  firstName: string = 'Tom';
  lastName: string = 'Hopkins';
  gender: string = 'Male';
  age: number = 20;
}
```

employee.component.html

```
<table>
  <tr>
    <td>First Name</td>
    <td>{{firstName}}</td>
  </tr>
  <tr>
    <td>Last Name</td>
    <td>{{lastName}}</td>
  </tr>
  <tr>
    <td>Gender</td>
    <td>{{gender}}</td>
  </tr>
  <tr>
    <td>Age</td>
    <td>{{age}}</td>
  </tr>
</table>
```

```
</tr>
</table>
```

The following are the different options available to style this "**employee component**"

Option 1: Specify the following <table> and <td> styles in external stylesheet - **styles.css**

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

td {
  border: 1px solid black;
}
```

Advantages :

1. Visual Studio editor features (Intellisense, Code completion & formatting) are available.
2. Application maintenance is also easy as we only have to change the styles in one place if we need to change them for any reason.

Disadvantages :

1. The Stylesheet that contains the styles must be referenced for the component to be reused.
2. Since styles.css is referenced in index.html page, these styles may affect the table and td elements in other components, and you may or may not want this behaviour.

Option 2 : Specify the styles inline in the component HTML file as shown below.

```
<table style="color: #369;font-family: Arial, Helvetica, sans-serif;
  font-size:large;border-collapse: collapse;">
  <tr>
    <td style="border: 1px solid black;">First Name</td>
    <td style="border: 1px solid black;">{{firstName}}</td>
  </tr>
  <tr>
    <td style="border: 1px solid black;">Last Name</td>
    <td style="border: 1px solid black;">{{lastName}}</td>
  </tr>
  <tr>
    <td style="border: 1px solid black;">Gender</td>
    <td style="border: 1px solid black;">{{gender}}</td>
  </tr>
  <tr>
    <td style="border: 1px solid black;">Age</td>
    <td style="border: 1px solid black;">{{age}}</td>
```

```
</tr>
</table>
```

Advantages :

1. Visual Studio editor features (Intellisense, Code completion & formatting) are available.
2. Component can be easily reused as the styles are defined inline
3. Styles specified using this approach are local to the component and don't collide with styles used elsewhere in the application.

Disadvantages :

1. Application maintenance is difficult. For example, if we want to change the <td> border colour to red we have to change it in several places.

Option 3 : Specify the styles in the component html file using <style> tag as shown below

```
<style>
  table {
    color: #369;
    font-family: Arial, Helvetica, sans-serif;
    font-size: large;
    border-collapse: collapse;
  }

  td {
    border: 1px solid black;
  }
</style>
<table>
  <tr>
    <td>First Name</td>
    <td>{{firstName}}</td>
  </tr>
  <tr>
    <td>Last Name</td>
    <td>{{lastName}}</td>
  </tr>
  <tr>
    <td>Gender</td>
    <td>{{gender}}</td>
  </tr>
  <tr>
    <td>Age</td>
    <td>{{age}}</td>
  </tr>
</table>
```

Advantages :

1. Component can be easily reused as the styles are defined inline with in the component itself
2. Application maintenance is also easy as we only have to change the styles in one place
3. Visual Studio editor features (Intellisense, Code completion & formatting) are available
4. Styles specified using this approach are local to the component and don't collide with styles used elsewhere in the application.

Option 4 : Specify the styles in the component TypeScript file using the `@component` decorator styles property as shown below. Notice the styles property takes an array of strings containing your styles.

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

```
@Component({
  selector: 'my-employee',
  templateUrl: 'app/employee/employee.component.html',
  styles: ['table { color: #369; font-family: Arial, Helvetica, sans-serif; font-size: large; border-collapse: collapse;}', 'td {border: 1px solid black; }']
})
export class EmployeeComponent {
  firstName: string = 'Tom';
  lastName: string = 'Hopkins';
  gender: string = 'Male';
  age: number = 20;
}
```

Advantages :

1. Component can be easily reused as the styles are defined inline with in the component itself
2. Application maintenance is also easy as we only have to change the styles in one place for this component if we need to change them for any reason.
3. Styles specified using this approach are local to the component and don't collide with styles used elsewhere in the application.

Disadvantages :

1. Visual Studio editor features (Intellisense, Code completion & formatting) are not available.

Option 5 : Specify the styles using the @component decorator styleUrls property. The styleUrls property is an array of strings containing stylesheet URLs.

Step 1 : Right click on the "employee" folder and add a new StyleSheet. Name it employee.component.css

Step 2 : Copy and paste the following styles for <table> and <td> elements in employee.component.css

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

Step 3 : In employee.component.ts file reference employee.component.css stylesheet using styleUrls property as shown below. Please note, the stylesheet path is relative to index.html file.

```
import { Component } from '@angular/core';  
  
@Component({  
  selector: 'my-employee',  
  templateUrl: 'app/employee/employee.component.html',  
  styleUrls: ['app/employee/employee.component.css']  
})  
export class EmployeeComponent {  
  firstName: string = 'Tom';  
  lastName: string = 'Hopkins';  
  gender: string = 'Male';  
  age: number = 20;  
}
```

Advantages :

1. Component can be easily reused as both the stylesheet itself and it's path are included with in the component
2. Application maintenance is also easy as we only have to change the styles in one place
3. Visual Studio editor features (Intellisense, Code completion & formatting) are available
4. Styles specified using this approach are local to the component and don't collide with styles used elsewhere in the application.

Assignment 8: Angular interpolation

In this assignment we will discuss the concept of Interpolation in Angular.

Interpolation is all about data binding. In Angular data-binding can be broadly classified into 3 categories

Data Binding	Description
One way data-binding	From Component to View Template
One way data-binding	From View Template to Component
Two way data-binding	From Component to View Template & From View template to Component

In this assignment, we will discuss the first one way data-binding i.e From Component to View Template. We achieve this using interpolation. We will discuss the rest of the 2 data-binding techniques in our upcoming assignments.

One way data-binding - From Component to View Template : To display read-only data on a view template we use one-way data binding technique interpolation. With interpolation, we place the component property name in the view template, enclosed in double curly braces: `{{propertyName}}`.

In the following example, Angular pulls the value of the `firstName` property from the component and inserts it between the opening and closing `<h1>` element.

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

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

Output :

Tom

It is also possible to concatenate a hard-coded string with the property value
`<h1>{{'Name = ' + firstName}}</h1>`

The above expression displays **"Name = Tom"** in the browser.

You can specify any valid expression in double curly braces. For example you can have

```
<h1>{{ 10 + 20 + 30 }}</h1>
```

The above expression evaluates to 60

The expression that is enclosed in double curly braces is commonly called as Template Expression. This template expression can also be a ternary operator as shown in the example below. Since firstName property has a value 'Tom', we see it in the browser.

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

@Component({
  selector: 'my-app',
  template: `
    <h1>{{firstName ? firstName : 'No name specified'}}</h1>
  `
})
export class AppComponent {
  firstName: string = 'Tom';
}
```

If we set firstName = null as shown below. The value **'No name specified'** is displayed in the browser
firstName: string = null;

You can also use interpolation to set src as shown in the example below.

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

@Component({
  selector: 'my-app',
  template: `<div>
    <h1>{{pageHeader}}</h1>
    <img src='{{imagePath}}'/>
  </div>`
})
export class AppComponent {
  pageHeader: string = 'Employee Details';
  imagePath: string = 'http://pragimtech.com/images/logo.jpg';
}
```

Output :

Employee Details



We can also call class methods using interpolation as shown below.

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

@Component({
  selector: 'my-app',
  template: `<div>
    <h1>{{'Full Name = ' + getFullName()}}</h1>
  </div>`
})
export class AppComponent {
  firstName: string = 'Tom';
  lastName: string = 'Hopkins';

  getFullName(): string {
    return this.firstName + ' ' + this.lastName;
  }
}
```

Output : Full Name = Tom Hopkins

Assignment 9: Property binding in Angular 2

In this assignment we will discuss **Property binding in Angular** with examples.

Previously, we discussed we can use interpolation to bind component class properties to view template. Another option to achieve exactly the same thing is by using **Property binding**.

In our previous assignment we have bound imagePath property of the component class to element src property using interpolation as shown below.

```
<img src='{{imagePath}}'/>
```

We can rewrite the above example, using property binding as shown below. Notice the element src property is in a pair of square brackets, and the component class property is in quotes.

```
<img [src]='imagePath'/>
```

Both **Interpolation** and **Property binding** flows a value in one direction, i.e from a component's data property into a target element property.

What is the difference between Interpolation and Property binding

Interpolation is a special syntax that Angular converts into a property binding.

Interpolation is just a convenient alternative to property binding.

In some cases like when we need to concatenate strings we have to use interpolation instead of property binding as shown in the example below.

```
<img src='http://www.pragimtech.com/{{imagePath}}' />
```

When setting an element property to a non-string data value, you must use property binding. In the following example, we are disabling a button by binding to the boolean property isDisabled.

```
<button [disabled]='isDisabled'>Click me</button>
```

If we use interpolation instead of property binding, the button is always disabled irrespective of isDisabled class property value

```
<button disabled='{{isDisabled}}'>Click me</button>
```

Some important points to keep in mind when using Property binding

Remember to enclose the property name with a pair of square brackets. If you omit the brackets, Angular treats the string as a constant and initializes the target property with that string.

```
<span [innerHTML]='pageHeader'></span>
```

With Property binding we enclose the element property name in square brackets

```
<button [disabled]='isDisabled'>Click me</button>
```

We can also use the alternate syntax with bind- prefix. This is known as canonical form
`<button bind-disabled='isDisabled'>Click me</button>`

From security standpoint, Angular data binding sanitizes malicious content before displaying it on the browser. Both interpolation and property binding protects us from malicious content.

In the example below we are using interpolation. Notice the malicious usage of `<script>` tag.

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

@Component({
  selector: 'my-app',
  template: '<div>{{badHtml}}</div>'
})
export class AppComponent {
  badHtml: string = 'Hello <script>alert("Hacked");</script> World';
}
```

Angular interpolation sanitizes the malicious content and displays the following in the browser
Hello `<script>alert("Hacked");</script>` World

In this example below we are using property binding.
`<div [innerHTML]="badHtml"></div>`

Property binding sanitizes the malicious content slightly differently and we get the following output, but the important point to keep in mind is both the techniques protect us from malicious content and render the content harmlessly.

Hello `alert("Hacked");` World

Assignment 10: Property binding in Angular 2

In this assignment we will discuss the **difference between HTML attribute and DOM property**.

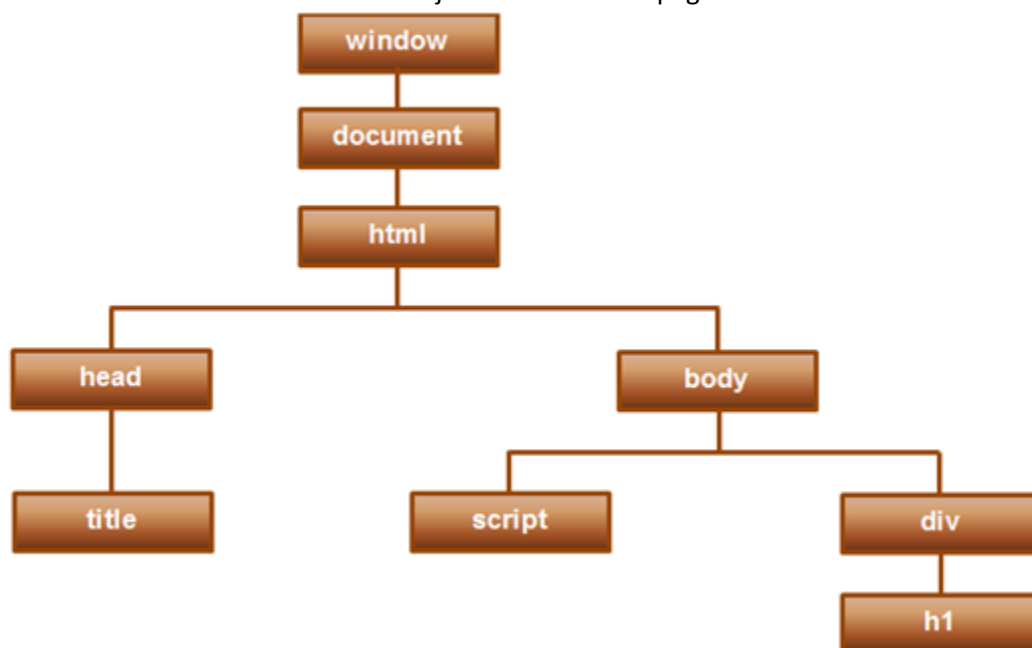
What is DOM

DOM stands for Document Object Model. When a browser loads a web page, the browser creates a Document Object Model of that page.

For example, for the HTML below,

```
<html>
  <head>
    <title>My Page Title</title>
  </head>
  <body>
    <script type="text/javascript">
    </script>
    <div>
      <h1>This is browser DOM</h1>
    </div>
  </body>
</html>
```

The browser creates a Document Object Model of that page as shown below



So in simple terms you can think of the DOM as an application programming interface (API) for HTML, and we can use programming languages like JavaScript, or JavaScript frameworks like Angular to access and manipulate the HTML using their corresponding DOM objects.

In other words DOM contains the HTML elements as objects, their properties, methods and events and it is a standard for accessing, modifying, adding or deleting HTML elements.

In the previous 2 assignments we discussed interpolation and property binding in Angular.

Interpolation example

```
<button disabled='{{isDisabled}}'>Click Me</button>
```

Property binding example

```
<button [disabled]='isDisabled'>Click Me</button>
```

If you notice the above 2 examples, it looks like we are binding to the Button's disabled attribute. This is not true. We are actually binding to the disabled property of the button object. Angular data-binding is all about binding to DOM object properties and not HTML element attributes.

What is the difference between HTML element attribute and DOM property

1. Attributes are defined by HTML, where as properties are defined by the DOM.
2. Attributes initialize DOM properties. Once the initialization complete, the attributes job is done.
3. Property values can change, where as attribute values can't.

Let's prove this point - Property values change, but the attribute values don't with an example.

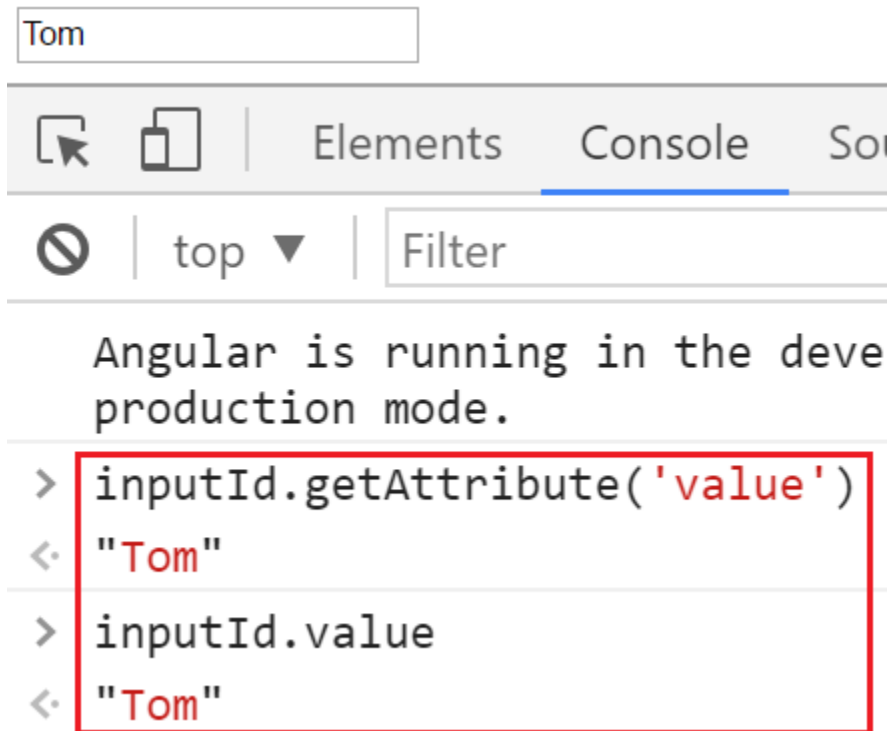
In the example below, we have set the value attribute of the input element to Tom.

```
<input id='inputId' type='text' value='Tom'>
```

At this point, run the web page and in the textbox you will see 'Tom' as the value.

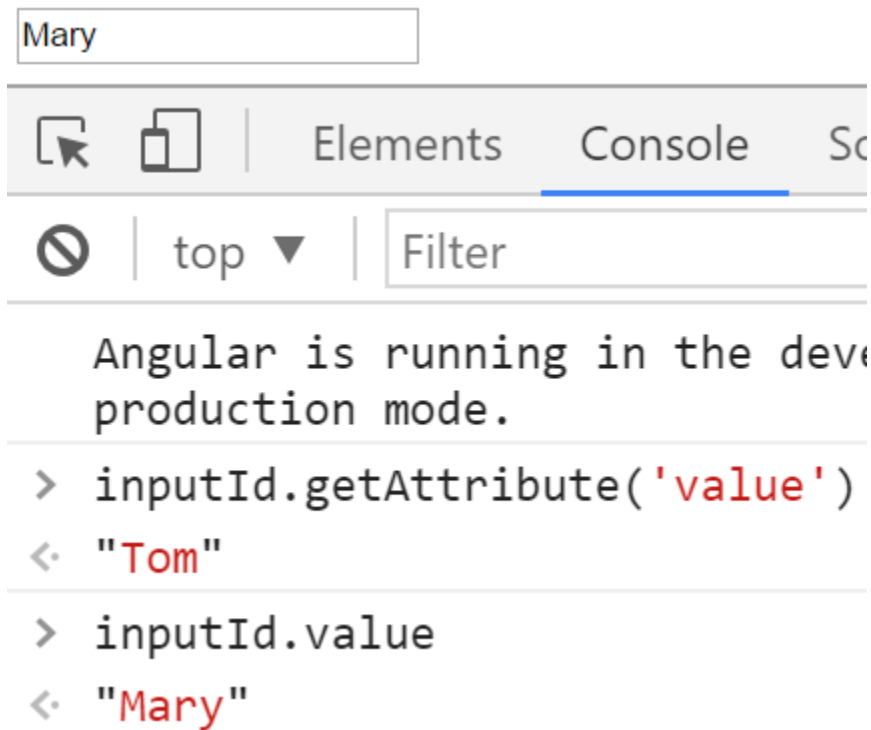
Launch the browser developer tools.

On the 'Console' tab, use the `getAttribute()` method and `value` property of the input element to get the attribute and property values. Notice at the moment both have the value 'Tom'



Change the value in the textbox to Mary.

Notice now, when we query for the attribute and property values, the attribute value is still Tom but the property value is Mary. So this proves the point - Property values change, whereas attribute values don't.



So it is important to keep in mind that,

1. HTML attributes and the DOM properties are different things.
2. Angular binding works with properties and events, and not attributes.
3. The role of attributes is to initialize element properties and their job is done.

Assignment 11: Angular attribute binding

In this assignment we will discuss **Attribute Binding in Angular**

In our previous assignments we discussed **Interpolation** and **Property binding** that deal with binding Component class properties to HTML element properties and not Attributes.

However, in some situations we want to be able to bind to HTML element attributes. For example, **colspan** and **aria** attributes does not have corresponding DOM properties. So in this case we want to be able to bind to HTML element attributes.

To make this happen, Angular provides attribute binding. With **attribute binding** we can set the value of an attribute directly. Angular team recommends to use Property binding when possible and use attribute binding only when there is no corresponding element property to bind.

Let us understand **Attribute Binding in Angular** with an example. We want to display Employee Details in a table as shown below.

Employee Details	
First Name	Tom
Last Name	Hopkins
Gender	Male
Age	20

In our root component - **app.component.ts** we have the following code

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

```
@Component({
  selector: 'my-app',
  template: `
    <div>
      <my-employee></my-employee>
    </div>
  `
})
export class AppComponent {
}
```

Code in employee.component.ts file

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

@Component({
  selector: 'my-employee',
  templateUrl: 'app/employee/employee.component.html',
  styleUrls: ['app/employee/employee.component.css']
})
export class EmployeeComponent {
  imagePath: string = 'Tom.png';
  firstName: string = 'Tom';
  lastName: string = 'Hopkins';
  gender: string = 'Male';
  age: number = 20;
}
```

Code in employee.component.css file

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

td {
  border: 1px solid black;
}

thead{
  border: 1px solid black;
}
```

Code in employee.component.html file

```
<table>
  <thead>
    <tr>
      <th colspan="2">
        Employee Details
      </th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>First Name</td>
      <td>{{firstName}}</td>
    </tr>
  </tbody>
</table>
```



```

        <td>Last Name</td>
        <td>{{lastName}}</td>
    </tr>
    <tr>
        <td>Gender</td>
        <td>{{gender}}</td>
    </tr>
    <tr>
        <td>Age</td>
        <td>{{age}}</td>
    </tr>
</tbody>
</table>

```

Notice at the moment we have hard-coded **colspan** attribute value to 2 in the HTML. Instead we want to bind to a component class property. So in the `employee.component.ts` file include 'columnSpan' property as shown below.

```

export class EmployeeComponent {
    columnSpan: number = 2;
    imagePath: string = 'Tom.png';
    firstName: string = 'Tom';
    lastName: string = 'Hopkins';
    gender: string = 'Male';
    age: number = 20;
}

```

If we use interpolation to bind `columnSpan` property of the component class to `colspan` attribute of the `<th>` element we get the error - Can't bind to 'colspan' since it isn't a known property of 'th'

```
<th colspan="{{columnSpan}}">
```

We get the same error if we use Property Binding

```
<th [colspan]="columnSpan">
```

This error is because we do not have a corresponding property in the DOM for `colspan` attribute. To fix this we have to use attribute-binding in Angular, which sets the `colspan` attribute. To tell angular framework that we are setting an attribute value we have to prefix the attribute name with `attr` and a DOT as shown below.

```
<th [attr.colspan]="columnSpan">
```

The same is true when using interpolation

```
<th attr.colspan="{{columnSpan}}">
```

Assignment 12: Class binding in angular 2

In this assignment we will discuss **CSS Class binding in Angular** with examples.

For the demos in this assignment, we will use same example we have been working with so far in this assignment series. In **styles.css** file include the following 3 CSS classes. If you recollect styles.css is already referenced in our host page - index.html.

```
.boldClass{
  font-weight:bold;
}
```

```
.italicsClass{
  font-style:italic;
}
```

```
.colorClass{
  color:red;
}
```

In **app.component.ts**, include a button element as shown below. Notice we have set the class attribute of the button element to 'colorClass'.

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

@Component({
  selector: 'my-app',
  template: `
    <button class='colorClass'>My Button</button>
  `
})
export class AppComponent {
}
```

At this point, run the application and notice that the 'colorClass' is added to the button element as expected.

Replace all the existing css classes with one or more classes

Modify the code in app.component.ts as shown below.

1. We have introduced a property 'classesToApply' in **AppComponent** class
2. We have also specified class binding for the button element. The word 'class' is in a pair of square brackets and it is binded to the property 'classesToApply'
3. This will replace the existing css classes of the button with classes specified in the class binding

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

```
@Component({
  selector: 'my-app',
  template: `
    <button class='colorClass' [class]='classesToApply'>My Button</button>
  `
})
export class AppComponent {
  classesToApply: string = 'italicsClass boldClass';
}
```

Run the application and notice 'colorClass' is removed and these classes (italicsClass & boldClass) are added.

Adding or removing a single class : To add or remove a single class, include the prefix 'class' in a pair of square brackets, followed by a DOT and then the name of the class that you want to add or remove. The following example adds boldClass to the button element. Notice it does not remove the existing colorClass already added using the class attribute. If you change applyBoldClass property to false or remove the property altogether from the AppComponent class, css class boldClass is not added to the button element.

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

```
@Component({
  selector: 'my-app',
  template: `
    <button class='colorClass' [class.boldClass]='applyBoldClass'>My Button</button>
  `
})
export class AppComponent {
  applyBoldClass: boolean = true;
}
```

With class binding we can also use ! symbol. Notice in the example below applyBoldClass is set to false. Since we have used ! in the class binding the class is added as expected.

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

```
@Component({
  selector: 'my-app',
  template: `
    <button class='colorClass' [class.boldClass]='!applyBoldClass'>My Button</button>
  `
})
export class AppComponent {
  applyBoldClass: boolean = false;
}
```

You can also removed an existing class that is already applied. Consider the following example. Notice we have 3 classes (colorClass, boldClass & italicsClass) added to the button element using the class attribute. The class binding removes the boldClass.

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

@Component({
  selector: 'my-app',
  template: `
    <button class='colorClass boldClass italicsClass'
      [class.boldClass]='applyBoldClass'>My Button</button>
  `
})
export class AppComponent {
  applyBoldClass: boolean = false;
}
```

To add or remove multiple classes use ngClass directive as shown in the example below.

1. Notice the colorClass is added using the class attribute
2. ngClass is binded to addClasses() method of the AppComponent class
3. addClasses() method returns an object with 2 key/value pairs. The key is a CSS class name. The value can be true or false. True to add the class and false to remove the class.
4. Since both the keys (boldClass & italicsClass) are set to true, both classes will be added to the button element
5. let is a new type of variable declaration in JavaScript.
6. let is similar to var in some respects but allows us to avoid some of the common gotchas that we run into when using var.
7. The differences between let and var are beyond the scope of this assignment. For our example, var also works fine.
8. As TypeScript is a superset of JavaScript, it supports let

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

@Component({
  selector: 'my-app',
  template: `
    <button class='colorClass' [ngClass]='addClasses()'>My Button</button>
  `
})
export class AppComponent {
  applyBoldClass: boolean = true;
  applyItalicsClass: boolean = true;

  addClasses() {
    let classes = {
```

```

        boldClass: this.applyBoldClass,
        italicsClass: this.applyItalicsClass
    };

    return classes;
}
}

```

We have included our css classes in a external stylesheet - **styles.css**. Please note we can also include these classes in the styles property instead of a separate stylesheet as shown below.

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

```

@Component({
  selector: 'my-app',
  template: `
    <button class='colorClass' [ngClass]='addClasses()'>My Button</button>
  `,
  styles: [
    .boldClass{
      font-weight:bold;
    }

    .italicsClass{
      font-style:italic;
    }

    .colorClass{
      color:red;
    }
  ]
})
export class AppComponent {
  applyBoldClass: boolean = true;
  applyItalicsClass: boolean = true;

  addClasses() {
    let classes = {
      boldClass: this.applyBoldClass,
      italicsClass: this.applyItalicsClass
    };

    return classes;
  }
}

```

Assignment 13: Style binding in angular 2

In this assignment we will discuss **Style binding in Angular** with examples.

Setting inline styles with style binding is very similar to setting CSS classes with class binding.

Notice in the example below, we have set the font color of the button using the style attribute.

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

@Component({
  selector: 'my-app',
  template: `
    <button style="color:red">My Button</button>
  `
})
export class AppComponent {
}
```

The following example sets a single style (font-weight). If the property **'isBold'** is true, then font-weight style is set to bold else normal.

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

@Component({
  selector: 'my-app',
  template: `
    <button style='color:red'
      [style.font-weight]="isBold ? 'bold' : 'normal'">My Button
    </button>
  `
})
export class AppComponent {
  isBold: boolean = true;
}
```

style property name can be written in either dash-case or camelCase. For example, font-weight style can also be written using camel case - fontWeight.

Some styles like font-size have a unit extension. To set font-size in pixels use the following syntax. This example sets font-size to 30 pixels.

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

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

```

template: `
  <button style='color:red'
    [style.font-size.px]="fontSize">My Button
  </button>
`
})
export class AppComponent {
  fontSize: number = 30;
}

```

To set multiple inline styles use NgStyle directive as shown below

- Notice the color style is added using the style attribute
- ngStyle is binded to **addStyles()** method of the AppComponent class
- **addStyles()** method returns an object with 2 key/value pairs. The key is a style name, and the value is a value for the respective style property or an expression that returns the style value.
- let is a new type of variable declaration in JavaScript.
- let is similar to var in some respects but allows us to avoid some of the common gotchas that we run into when using var.
- The differences between let and var are beyond the scope of this assignment. For our example, var also works fine.
- As TypeScript is a superset of JavaScript, it supports let

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

```

@Component({
  selector: 'my-app',
  template: ` <button style='color:red' [ngStyle]="addStyles()">My Button</button>
`
})
export class AppComponent {
  isBold: boolean = true;
  fontSize: number = 30;
  isItalic: boolean = true;

  addStyles() {
    let styles = {
      'font-weight': this.isBold ? 'bold' : 'normal',
      'font-style': this.isItalic ? 'italic' : 'normal',
      'font-size.px': this.fontSize
    };

    return styles;
  }
}

```

Assignment 14: Angular2 event binding

In this assignment we will discuss **Event Binding in Angular** with examples.

The following bindings that we have discussed so far in this assignment series flow data in one direction i.e from a component class property to an HTML element property.

1. Interpolation
2. Property Binding
3. Attribute Binding
4. Class Binding
5. Style Binding

How about flowing data in the opposite direction i.e from an HTML element to a component. When a user performs any action like clicking on a button, hovering over an element, selecting from a dropdownlist, typing in a textbox etc, then the corresponding event for that action is raised. We need to know when user performs these actions. We can use angular event binding to get notified when these events occur.

For example the following is the syntax for binding to the click event of a button. Within parentheses on the left of the equal sign we have the target event, (click) in this case and on the right we have the template statement. In this case the **onClick()** method of the component class is called when the click event occurs.

```
<button (click)="onClick()">Click me</button>
```

With event binding we can also use the on- prefix alternative as shown below. This is known as the canonical form

```
<button on-click="onClick()">Click me</button>
```

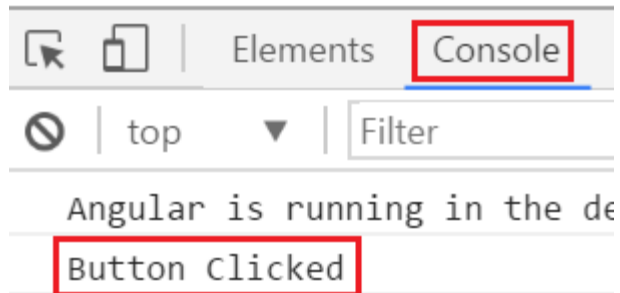
Event Binding Example :

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

@Component({
  selector: 'my-app',
  template: `<button (click)="onClick()" >Click me</button>`
})
export class AppComponent {
  onClick(): void {
    console.log('Button Clicked');
  }
}
```

Every time we click the button, 'Button Clicked' message is logged to the console. You can see this message under the Console tab, in the browser developer tools.

Click me



Another Example : Initially when the page loads we want to display only the First Name and Last of Employee. We also want to display "Show Details" button.

Employee Details	
First Name	Tom
Last Name	Hopkins

Show Details

When we click "**Show Details**" button, we want to display "**Gender**" and "**Age**" as well. The text on the button should be changed to "**Hide Details**".

When we click "**Hide Details**" button, "**Gender**" and "**Age**" should be hidden and the button text should be changed to "**Show Details**".

Employee Details	
First Name	Tom
Last Name	Hopkins
Gender	Male
Age	20

Hide Details

To achieve this we will make use of event binding in Angular. We will also make use of one of the structural directives "ngIf" in angular.

Code in **employee.component.ts** : Notice we have introduced "showDetails" boolean property. The default value is false, so when the page first loads, we will have "Gender" and "Age" hidden. We also have a method, toggleDetails(), which toggles the value of showDetails.

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

@Component({
  selector: 'my-employee',
  templateUrl: 'app/employee/employee.component.html',
  styleUrls: ['app/employee/employee.component.css']
})
export class EmployeeComponent {
  columnSpan: number = 2;
  firstName: string = 'Tom';
  lastName: string = 'Hopkins';
  gender: string = 'Male';
  age: number = 20;
  showDetails: boolean = false;

  toggleDetails(): void {
    this.showDetails = !this.showDetails;
  }
}
```

Code in **employee.component.html** :

- Notice the click event of the button is binded to toggleDetails() method
- To dynamicall change the text on the button, we are using ternary operator - {{showDetails ? 'Hide' : 'Show'}} Details
- We used ngIf structural directive on "Gender" and "Age" <tr> elements
- The * prefix before a directive indicates, it is a structural directive
- Besides ngIf, there are other structural directives which we will discuss in our upcoming assignments
- The ngIf directive conditionally adds or removes content from the DOM based on whether or not an expression is true or false
- If "showDetails" is true, "Gender" and "Age" <tr> elements are added to the DOM, else removed

```

<table>
  <thead>
    <tr>
      <th attr.colspan="{{columnSpan}}">
        Employee Details
      </th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>First Name</td>
      <td>{{firstName}}</td>
    </tr>
    <tr>
      <td>Last Name</td>
      <td>{{lastName}}</td>
    </tr>
    <tr *ngIf='showDetails'>
      <td>Gender</td>
      <td>{{gender}}</td>
    </tr>
    <tr *ngIf='showDetails'>
      <td>Age</td>
      <td>{{age}}</td>
    </tr>
  </tbody>
</table>
<br />
<button (click)='toggleDetails()'>
  {{showDetails ? 'Hide' : 'Show'}} Details
</button>

```

Assignment 15: Two way data binding in angular 2

In this assignment we will discuss **Two way Data Binding in Angular 2**.

Consider the following code in **app.component.ts**

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

```
@Component({  
  selector: 'my-app',  
  template: `  
    Name : <input [value]='name'>  
    <br>  
    You entered : {{name}}  
  `
```

```
})
```

```
export class AppComponent {  
  name: string = 'Tom';  
}
```

- `<input [value]='name'>` : Binds component class "name" property to the input element's value property
- `You entered : {{name}}` : Interpolation displays the value we have in "name" property on the web page

Here is the output

Name :
You entered : Tom

At the moment when we change the value in the textbox, that changed value is not reflected in the browser. One way to achieve this is by binding to the input event of the input control as shown below.

```
Name : <input [value]='name' (input)='name = $event.target.value'>  
<br>  
You entered : {{name}}
```

At this point, as we type in the textbox, the changed value is displayed on the page.

Name :
You entered : Tom Hopkins

So let's understand what is happening here. Consider this code

Name : `<input [value]='name' (input)='name = $event.target.value'>`

You entered : `{{name}}`

- `[value]='name'` : This property binding flows data from the component class to element property
- `(input)='name = $event.target.value'` : This event binding flows data in the opposite direction i.e from the element to component class property "name"
- `$event` - Is exposed by angular event binding, and contains the event data. To retrieve the value from the input element use - `$event.target.value`.
- `name = $event.target.value` - This expression updates the value in the name property in the component class
- You entered : `{{name}}` - This interpolation expression will then display the value on the web page.

So in short two-way data binding in Angular is a combination of both Property Binding and Event Binding. To save a few keystrokes and simplify two-way data binding angular has provided `ngModel` directive. So with `ngModel` directive we can rewrite the following line of code
Name : `<input [value]='name' (input)='name = $event.target.value'>`

Like this : Name : `<input [(ngModel)]='name'>`

At this point if you view the page in the browser, you will get the following error
Template parse errors:
Can't bind to 'ngModel' since it isn't a known property of 'input'

This is because `ngModel` directive is, in an Angular system module called `FormsModule`. For us to be able to use `ngModel` directive in our root module - `AppModule`, we will have to import `FormsModule` first.

Here are the steps to import `FormsModule` into our `AppModule`

1. Open `app.module.ts` file
2. Include the following import statement in it
`import { FormsModule } from '@angular/forms';`
3. Also, include `FormsModule` in the 'imports' array of `@NgModule`
`imports: [BrowserModule, FormsModule]`

With these changes, reload the web page and it will work as expected.

So here is the syntax for using two-way data binding in Angular
`<input [(ngModel)]='name'>`

- The square brackets on the outside are for property binding
- The parentheses on the inside are for event binding
- To easily remember this syntax, compare it to a banana in a box `[]()`

Assignment 16: Angular ngFor directive

Previously, we discussed **ngIf** structural directive which conditionally adds or removes DOM elements. In this assignment we will discuss another structural directive - **ngFor** in Angular.

Let us understand **ngFor** structural directive with an example. Consider the following array of Employee objects.

```
employees: any[] = [
  {
    code: 'emp101', name: 'Tom', gender: 'Male',
    annualSalary: 5500, dateOfBirth: '25/6/1988'
  },
  {
    code: 'emp102', name: 'Alex', gender: 'Male',
    annualSalary: 5700.95, dateOfBirth: '9/6/1982'
  },
  {
    code: 'emp103', name: 'Mike', gender: 'Male',
    annualSalary: 5900, dateOfBirth: '12/8/1979'
  },
  {
    code: 'emp104', name: 'Mary', gender: 'Female',
    annualSalary: 6500.826, dateOfBirth: '14/10/1980'
  },
];
```

We want to display these employees in a table on a web page as shown below.

Code	Name	Gender	Annual Salary	Date of Birth
emp101	Tom	Male	5500	25/6/1988
emp102	Alex	Male	5700.95	9/6/1982
emp103	Mike	Male	5900	12/8/1979
emp104	Mary	Female	6500.826	14/10/1980

We will use the same project that we have been working with so far in this assignment series.

Step 1 : Add a new **TypeScript** file to the **"employee"** folder. Name it **employeeList.component.ts**. Copy and paste the following code in it.

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

@Component({
```

```

    selector: 'list-employee',
    templateUrl: 'app/employee/employeeList.component.html',
    styleUrls: ['app/employee/employeeList.component.css']
  })
  export class EmployeeListComponent {
    employees: any[] = [
      {
        code: 'emp101', name: 'Tom', gender: 'Male',
        annualSalary: 5500, dateOfBirth: '25/6/1988'
      },
      {
        code: 'emp102', name: 'Alex', gender: 'Male',
        annualSalary: 5700.95, dateOfBirth: '9/6/1982'
      },
      {
        code: 'emp103', name: 'Mike', gender: 'Male',
        annualSalary: 5900, dateOfBirth: '12/8/1979'
      },
      {
        code: 'emp104', name: 'Mary', gender: 'Female',
        annualSalary: 6500.826, dateOfBirth: '14/10/1980'
      }
    ];
  }
}

```

Step 2 : Add a new HTML page to the "**employee**" folder. Name it **employeeList.component.html**. Copy and paste the following code in it.

Please note :

- **ngFor** is usually used to display an array of items
- Since **ngFor** is a structural directive it is prefixed with *
- ***ngFor='let employee of employees'** - In this example 'employee' is called template input variable, which can be accessed by the <tr> element and any of it's child elements.
- **ngIf** structural directive displays the row "**No employees to display**" when employees property does not exist or when there are ZERO employees in the array.

```

<table>
  <thead>
    <tr>
      <th>Code</th>
      <th>Name</th>
      <th>Gender</th>
      <th>Annual Salary</th>
      <th>Date of Birth</th>
    </tr>
  </thead>
  <tbody>

```

```
|
|  |

```

Step 3 : Add a new **StyleSheet** to the "**employee**" folder. Name it **employeeList.component.css**. Copy and paste the following code in it.

```

table {
  color: #369;
  font-family: Arial, Helvetica, sans-serif;
  font-size: large;
  border-collapse: collapse;
}

td {
  border: 1px solid #369;
  padding: 5px;
}

th {
  border: 1px solid #369;
  padding: 5px;
}

```

Step 4 : In the root module, i.e **app.module.ts**, import employeeList component and add it to the declarations array as shown below.

```

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

import { AppComponent } from './app.component';
import { EmployeeComponent } from './employee/employee.component';
import { EmployeeListComponent } from './employee/employeeList.component';

@NgModule({
  imports: [BrowserModule],
  declarations: [AppComponent, EmployeeComponent, EmployeeListComponent],

```



```
bootstrap: [AppComponent]
})
```

```
export class AppModule { }
```

Step 5 : In the root component, i.e **app.component.ts** use employeeList component as a directive as shown below.

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

```
@Component({
  selector: 'my-app',
  template: '<list-employee></list-employee>'
})
```

```
export class AppComponent { }
```

At this point, run the application and notice the employees are displayed in the table as expected.

Assignment 17: Angular ngFor trackBy

In this assignment we will discuss

1. Using trackBy with ngFor directive
2. How to get the index of an item in a collection
3. Identifying the first and the last elements in a collection
4. Identifying even and odd elements in a collection

Using trackBy with ngFor directive :

- ngFor directive may perform poorly with large lists
- A small change to the list like, adding a new item or removing an existing item may trigger a cascade of DOM manipulations

For example, consider this code in **employeeList.component.ts**

The constructor() initialises the employees property with 4 employee objects
getEmployees() method returns another list of 5 employee objects (The 4 existing employees plus a new employee object)

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

@Component({
  selector: 'list-employee',
  templateUrl: 'app/employee/employeeList.component.html',
  styleUrls: ['app/employee/employeeList.component.css']
})
export class EmployeeListComponent {
  employees: any[];

  constructor() {
    this.employees = [
      {
        code: 'emp101', name: 'Tom', gender: 'Male',
        annualSalary: 5500, dateOfBirth: '25/6/1988'
      },
      {
        code: 'emp102', name: 'Alex', gender: 'Male',
        annualSalary: 5700.95, dateOfBirth: '9/6/1982'
      },
      {
        code: 'emp103', name: 'Mike', gender: 'Male',
        annualSalary: 5900, dateOfBirth: '12/8/1979'
      },
      {
        code: 'emp104', name: 'Mary', gender: 'Female',
```

```

        annualSalary: 6500.826, dateOfBirth: '14/10/1980'
    },
];
}

getEmployees(): void {
    this.employees = [
        {
            code: 'emp101', name: 'Tom', gender: 'Male',
            annualSalary: 5500, dateOfBirth: '25/6/1988'
        },
        {
            code: 'emp102', name: 'Alex', gender: 'Male',
            annualSalary: 5700.95, dateOfBirth: '9/6/1982'
        },
        {
            code: 'emp103', name: 'Mike', gender: 'Male',
            annualSalary: 5900, dateOfBirth: '12/8/1979'
        },
        {
            code: 'emp104', name: 'Mary', gender: 'Female',
            annualSalary: 6500.826, dateOfBirth: '14/10/1980'
        },
        {
            code: 'emp105', name: 'Nancy', gender: 'Female',
            annualSalary: 6700.826, dateOfBirth: '15/12/1982'
        },
    ];
}
}

```

Now look at this code in **employeeList.component.html**

```

<table>
  <thead>
    <tr>
      <th>Code</th>
      <th>Name</th>
      <th>Gender</th>
      <th>Annual Salary</th>
      <th>Date of Birth</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor='let employee of employees'>
      <td>{{employee.code}}</td>
      <td>{{employee.name}}</td>
      <td>{{employee.gender}}</td>
      <td>{{employee.annualSalary}}</td>

```

```
        <td>{{employee.dateOfBirth}}</td>
    </tr>
    <tr *ngIf="!employees || employees.length==0">
        <td colspan="5">
            No employees to display
        </td>
    </tr>
</tbody>
</table>
<br />
<button (click)='getEmployees()'>Refresh Employees</button>
```

At the moment we are not using trackBy with ngFor directive

1. When the page initially loads we see the 4 employees
2. When we click **"Refresh Employees"** button we see the fifth employee as well
3. It looks like it just added the additional row for the fifth employee. That's not true, it effectively teared down all the <tr> and <td> elements of all the employees and recreated them.
4. To confirm this launch browser developer tools by pressing F12.
5. Click on the "Elements" tab and expand the <table> and then <tbody> elements
6. At this point click the **"Refresh Employees"** button and you will notice all the <tr>elements are briefly highlighted indicating they are teared down and recreated.

Code	Name	Gender	Annual Salary	Date of Birth
emp101	Tom	Male	5500	25/6/1988
emp102	Alex	Male	5700.95	9/6/1982
emp103	Mike	Male	5900	12/8/1979
emp104	Mary	Female	6500.826	14/10/1980
emp105	Nancy	Female	6700.826	15/12/1982

Refresh Employees

```

<!DOCTYPE html>
<html>
  <head>...</head>
  <body> == $0
    <my-app ng-version="4.0.3">
      <list-employee _ngghost-c0>
        <table _ngcontent-c0>
          <thead _ngcontent-c0>...</thead>
          <tbody _ngcontent-c0>
            <!--bindings={
              "ng-reflect-ng-for-of": "[object Object]"
            }-->
            <tr _ngcontent-c0>...</tr>
            <tr _ngcontent-c0>...</tr>
            <tr _ngcontent-c0>...</tr>
            <tr _ngcontent-c0>...</tr>
            <tr _ngcontent-c0>...</tr>

```

This happens because Angular by default keeps track of objects using the object references. When we click "**Refresh Employees**" button we get different object references and as a result Angular has no choice but to tear down all the old DOM elements and insert the new DOM elements.

Angular can avoid this churn with `trackBy`. The `trackBy` function takes the index and the current item as arguments and returns the unique identifier by which that item should be tracked. In our case we are tracking by Employee code. Add this method to **employeeList.component.ts**.

```

trackByEmpCode(index: number, employee: any): string {
    return employee.code;
}

```

Make the following change in **employeeList.component.html** : Notice along with ngFor we also specified trackBy

```
<tr *ngFor='let employee of employees; trackBy:trackByEmpCode'>
```

At this point, run the application and launch developer tools. When you click "**Refresh Employees**" first time, only the row of the fifth employee is highlighted indicating only that <tr> element is added. On subsequent clicks, nothing is highlighted meaning none of the <tr> elements are teared down or added as the employees collection has not changed. Even now we get different object references when we click "**Refresh Employees**" button, but as Angular is now tracking employee objects using the employee code instead of object references, the respective DOM elements are not affected.

How to get the index of an item in a collection : Notice in the example below, we are using the index property of the ngFor directive to store the index in a template input variable "i". The variable is then used in the <td> element where we want to display the index. We used the let keyword to create the template input variable "i".

The index of an element is extremely useful when creating the HTML elements dynamically. We will discuss creating HTML elements dynamically in our upcoming assignments.

```

<table>
  <thead>
    <tr>
      <th>Code</th>
      <th>Name</th>
      <th>Gender</th>
      <th>Annual Salary</th>
      <th>Date of Birth</th>
      <th>Index</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor='let employee of employees; let i=index'>
      <td>{{employee.code}}</td>
      <td>{{employee.name}}</td>
      <td>{{employee.gender}}</td>
      <td>{{employee.annualSalary}}</td>
      <td>{{employee.dateOfBirth}}</td>
      <td>{{i}}</td>
    </tr>
    <tr *ngIf='!employees || employees.length==0'>
      <td colspan="5">
        No employees to display
      </td>

```

```

    </tr>
  </tbody>
</table>

```

Notice the index of the element is displayed in the last <td> element

Code	Name	Gender	Annual Salary	Date of Birth	Index
emp101	Tom	Male	5500	25/6/1988	0
emp102	Alex	Male	5700.95	9/6/1982	1
emp103	Mike	Male	5900	12/8/1979	2
emp104	Mary	Female	6500.826	14/10/1980	3

Identifying the first and the last element in a collection : Use the first and last properties of the ngFor directive to find if an element is the first or last element respectively.

Modify the code in **employeeList.component.html** as shown below.

```

<table>
  <thead>
    <tr>
      <th>Code</th>
      <th>Name</th>
      <th>Gender</th>
      <th>Annual Salary</th>
      <th>Date of Birth</th>
      <th>Is First</th>
      <th>Is Last</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor='let employee of employees; let isFirst = first; let isLast = last'>
      <td>{{employee.code}}</td>
      <td>{{employee.name}}</td>
      <td>{{employee.gender}}</td>
      <td>{{employee.annualSalary}}</td>
      <td>{{employee.dateOfBirth}}</td>
      <td>{{isFirst}}</td>
      <td>{{isLast}}</td>
    </tr>
    <tr *ngIf='!employees || employees.length==0'>
      <td colspan="5">
        No employees to display
      </td>
    </tr>
  </tbody>
</table>

```

```

    </tbody>
</table>
<br />
<button (click)='getEmployees()'>Refresh Employees</button>

```

The output is shown below

Code	Name	Gender	Annual Salary	Date of Birth	Is First	Is Last
emp101	Tom	Male	5500	25/6/1988	true	false
emp102	Alex	Male	5700.95	9/6/1982	false	false
emp103	Mike	Male	5900	12/8/1979	false	false
emp104	Mary	Female	6500.826	14/10/1980	false	true

Identifying even and odd element in a collection : This is similar to identifying first and last element in a collection. Instead of using first and last properties, use even and odd properties.

Here is the code in **employeeList.component.html**

```

<table>
  <thead>
    <tr>
      <th>Code</th>
      <th>Name</th>
      <th>Gender</th>
      <th>Annual Salary</th>
      <th>Date of Birth</th>
      <th>Is Even</th>
      <th>Is Odd</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor='let employee of employees; let isEven = even; let isOdd = odd'>
      <td>{{employee.code}}</td>
      <td>{{employee.name}}</td>
      <td>{{employee.gender}}</td>
      <td>{{employee.annualSalary}}</td>
      <td>{{employee.dateOfBirth}}</td>
      <td>{{isEven}}</td>
      <td>{{isOdd}}</td>
    </tr>
    <tr *ngIf='!employees || employees.length==0'>
      <td colspan='5'>
        No employees to display
      </td>

```



```
</tr>
</tbody>
</table>
<br />
<button (click)='getEmployees()'>Refresh Employees</button>
```

The output is shown below

Code	Name	Gender	Annual Salary	Date of Birth	Is Even	Is Odd
emp101	Tom	Male	5500	25/6/1988	true	false
emp102	Alex	Male	5700.95	9/6/1982	false	true
emp103	Mike	Male	5900	12/8/1979	true	false
emp104	Mary	Female	6500.826	14/10/1980	false	true

Assignment 18: Angular pipes

In this assignment we will discuss **Pipes in Angular with examples.**

Pipes in Angular

- Transform data before display
- Built in pipes include lowercase, uppercase, decimal, date, percent, currency etc
- To apply a pipe on a bound property use the pipe character " | "
`<td>{{employee.code | uppercase}}</td>`
- We can also chain pipes `<td>{{employee.dateOfBirth | date:'fullDate' | uppercase }}</td>`
- Pass parameters to pipe using colon " : "
`<td>{{employee.annualSalary | currency:'USD':true:'1.3-3'}}</td>`
`<td>{{employee.dateOfBirth | date:'fullDate'}}</td>`
`<td>{{employee.dateOfBirth | date:'dd/MM/y'}}</td>`
- Custom pipes can be created
- To read more about angular built-in pipes

Pipe	URL
Date	https://angular.io/api/common/DatePipe
Decimal	https://angular.io/api/common/DecimalPipe
Currency	https://angular.io/api/common/CurrencyPipe
Percent	https://angular.io/api/common/PercentPipe

Please note : If you get the following error, chances are that your date is not in mm/dd/yyyy format. To fix this error please change the date format to mm/dd/yyyy or create a custom pipe
InvalidPipeArgument: '14/10/1980' for pipe 'DatePipe'

Angular Pipe Examples:

uppercase pipe in this example converts employee code to uppercase

```
<td>{{employee.code | uppercase}}</td>
```

Output :

Code
EMP101
EMP102
EMP103
EMP104
EMP105

In this example, we have chained date and uppercase pipes.

```
<td>{{employee.dateOfBirth | date:'fullDate' | uppercase }}</td>
```

Output :

Date of Birth
SATURDAY, JUNE 25, 1988
MONDAY, SEPTEMBER 6, 1982
SATURDAY, DECEMBER 8, 1979
TUESDAY, OCTOBER 14, 1980

In this example we are passing a single parameter to date pipe. With the parameter we specified we want the date format to be dd/mm/yyyy

```
<td>{{employee.dateOfBirth | date:'dd/MM/y'}}</td>
```

Output :

Date of Birth
25/06/1988
06/09/1982
08/12/1979
14/10/1980

For the list of date pipe parameter values please check the following article
<https://angular.io/api/common/DatePipe>

In this example we are passing 3 parameters to the currency pipe

<td>{{employee.annualSalary | currency:'USD':true:'1.3-3'}}</td>

1. The first parameter is the currencyCode
2. The second parameter is boolean - True to display currency symbol, false to display currency code
3. The third parameter ('1.3-3') specifies the number of integer and fractional digits

Output :

Annual Salary
\$5,500.000
\$5,700.950
\$5,900.000
\$6,500.826

Assignment 19: Angular custom pipe

In this assignment we will discuss **creating a Custom Pipe in Angular**. Let us understand this with an example.

Here is what we want to do. Depending on the gender of the employee, we want to display Mr. or Miss. prefixed to the employee name as shown below.

Code	Name	Gender	Annual Salary	Date of Birth
emp101	Mr.Tom	Male	5500	25/6/1988
emp102	Mr.Alex	Male	5700.95	9/6/1982
emp103	Mr.Mike	Male	5900	12/8/1979
emp104	Miss.Mary	Female	6500.826	14/10/1980
emp105	Miss.Nancy	Female	6700.826	15/12/1982

Step 1 : To achieve this let's create a custom pipe called employeeTitlePipe. Right click on the "employee" folder and add a new TypeScript file. Name it "employeeTitle.pipe.ts". Copy and paste the following code.

Code Explanation :

- Import Pipe decorator and PipeTransform interface from Angular core
- Notice "EmployeeTitlePipe" class is decorated with Pipe decorator to make it an Angular pipe
- name property of the pipe decorator is set to employeeTitle. This name can then be used on any HTML page where you want this pipe functionality.
- EmployeeTitlePipe class implements the PipeTransform interface. This interface has one method transform() which needs to be implemented.
- Notice the transform method has 2 parameters. value parameter will receive the name of the employee and gender parameter receives the gender of the employee. The method returns a string i.e Mr. or Miss. prefixed to the name of the employee depending on their gender.

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

```
@Pipe({
  name: 'employeeTitle'
})
export class EmployeeTitlePipe implements PipeTransform {
  transform(value: string, gender: string): string {
    if (gender.toLowerCase() == "male")
      return "Mr." + value;
    else
      return "Miss." + value;
  }
}
```

```
}  
}
```

Step 2 : Register "EmployeeTitlePipe" in the angular module where we need it. In our case we need it in the root module. So in app.module.ts file, import the EmployeeTitlePipe and include it in the "declarations" array of NgModule decorator

```
import { EmployeeTitlePipe } from './employee/employeeTitle.pipe'
```

```
@NgModule({  
  imports: [BrowserModule],  
  declarations: [AppComponent, EmployeeComponent,  
    EmployeeListComponent, EmployeeTitlePipe],  
  bootstrap: [AppComponent]  
})
```

```
export class AppModule { }
```

Step 3 : In "employeeList.component.html" use the "EmployeeTitlePipe" as shown below. Notice we are passing employee gender as an argument for the gender parameter of our custom pipe. Employee name gets passed automatically.

```
<tr *ngFor='let employee of employees;'>  
  <td>{{employee.code}}</td>  
  <td>{{employee.name | employeeTitle:employee.gender}}</td>  
  <td>{{employee.gender}}</td>  
  <td>{{employee.annualSalary}}</td>  
  <td>{{employee.dateOfBirth}}</td>  
</tr>
```

Assignment 20: Angular 2 container and nested components

In this assignment and in the next few assignments we will discuss

- What is a nested component
- What is a container component
- Passing data from the nested component to container component
- Passing data from the container component to nested component
- Along the way we will discuss component input and output properties
- Creating custom events using EventEmitter class
- What is ng-container directive and its use

We will use the following example to understand all of the above concepts. Notice we have a table that displays a list of employees.

Show : ☒ All(5) ☐ Male(3) ☐ Female(2)

Code	Name	Gender	Annual Salary	Date of Birth
EMP101	Mr.Tom	Male	\$5,500.000	25/06/1988
EMP102	Mr.Alex	Male	\$5,700.950	06/09/1982
EMP103	Mr.Mike	Male	\$5,900.000	08/12/1979
EMP104	Miss.Mary	Female	\$6,500.826	14/10/1980
EMP105	Miss.Nancy	Female	\$6,700.826	15/12/1982

Above the table we have 3 radio buttons. Next to each radio button we also have the count of employees.

Show : ☒ All(5) ☐ Male(3) ☐ Female(2)

All(5) radio button has the total count of employees. Male(3) radio button has the total count of male employees and the Female(2) radio button has the total count of female employees.

At the moment All(5) radio button is selected, so all the employees are displayed in the table. If we select Male(3) radio button, then only the 3 male employees should be displayed in the table. Along the same lines, if Female(2) radio button is selected only the female employees should be displayed.

As we develop this example, we will understand all the following concepts.

- What is a container and nested component
- Passing data from the nested component to container component and vice-versa
- Component input and output properties
- Creating custom events using EventEmitter class
- What is ng-container directive and it's use

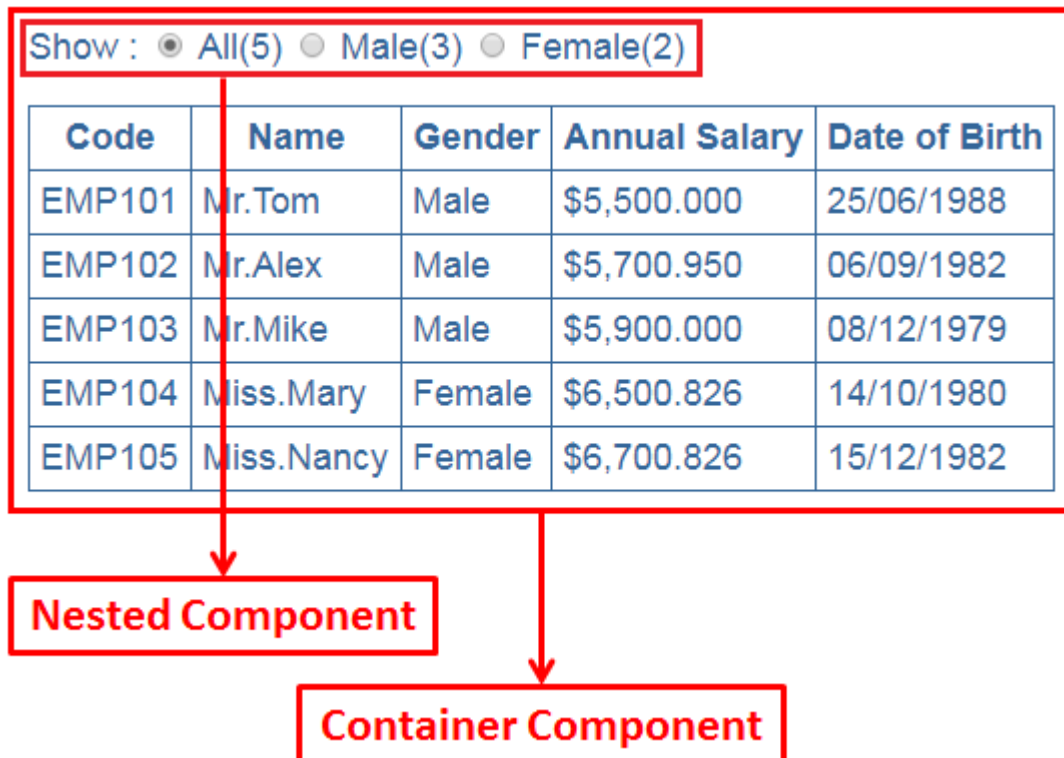
What is a container and nested component : In the example below, we have 2 components

One of the component displays the list of employees. We have already built this component in our previous assignments in this series. We named this component EmployeeListComponent.

The other component displays the radio buttons and the count of employees. We have not created this component yet. We will create it in this assignment. We will call this component EmployeeCountComponent.

We will nest EmployeeCountComponent in EmployeeListComponent.

So EmployeeCountComponent becomes the nested component or child component and EmployeeListComponent becomes the container component or parent component.



We have already implemented the required code for EmployeeListComponent in our previous assignments as shown below.

employeeList.component.ts

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

@Component({
  selector: 'list-employee',
  templateUrl: 'app/employee/employeeList.component.html',
  styleUrls: ['app/employee/employeeList.component.css']
})
export class EmployeeListComponent {
  employees: any[];

  constructor() {
    this.employees = [
      {
        code: 'emp101', name: 'Tom', gender: 'Male',
        annualSalary: 5500, dateOfBirth: '6/25/1988'
      },
      {
        code: 'emp102', name: 'Alex', gender: 'Male',
        annualSalary: 5700.95, dateOfBirth: '9/6/1982'
      },
      {
        code: 'emp103', name: 'Mike', gender: 'Male',
        annualSalary: 5900, dateOfBirth: '12/8/1979'
      },
      {
        code: 'emp104', name: 'Mary', gender: 'Female',
        annualSalary: 6500.826, dateOfBirth: '10/14/1980'
      },
      {
        code: 'emp105', name: 'Nancy', gender: 'Female',
        annualSalary: 6700.826, dateOfBirth: '12/15/1982'
      },
    ];
  }
}
```

employeeList.component.html

```
<table>
  <thead>
    <tr>
      <th>Code</th>
      <th>Name</th>
      <th>Gender</th>
      <th>Annual Salary</th>
      <th>Date of Birth</th>
```

```

    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let employee of employees;">
      <td>{{employee.code | uppercase}}</td>
      <td>{{employee.name | employeeTitle:employee.gender }}</td>
      <td>{{employee.gender}}</td>
      <td>{{employee.annualSalary | currency:'USD':true:'1.3-3'}}</td>
      <td>{{employee.dateOfBirth | date:'dd/MM/y'}}</td>
    </tr>
    <tr *ngIf="!employees || employees.length==0">
      <td colspan="5">
        No employees to display
      </td>
    </tr>
  </tbody>
</table>

```

employeeList.component.css

```

table {
  color: #369;
  font-family: Arial, Helvetica, sans-serif;
  font-size: large;
  border-collapse: collapse;
}

td {
  border: 1px solid #369;
  padding: 5px;
}

th {
  border: 1px solid #369;
  padding: 5px;
}

```

Now let's create the EmployeeCountComponent. Add a new TypeScript file to the employee folder. Name it employeeCount.component.ts. Copy and paste the following code.

employeeCount.component.ts

- We have set select='employee-count'. We can use this selector as a directive where we want to use this component. We are going to nest this component inside EmployeeListComponent using employee-count selector as a directive.

- We have 3 properties (all, male and Female). At the moment we have hard coded the values. In our next assignment we will discuss how to pass the values for these properties from the container component i.e from the EmployeeListComponent.

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

@Component({
  selector: 'employee-count',
  templateUrl: 'app/employee/employeeCount.component.html',
  styleUrls: ['app/employee/employeeCount.component.css']
})
export class EmployeeCountComponent {
  all: number = 10;
  male: number = 5;
  female: number = 5;
}
```

Add a new StyleSheet to the employee folder. Name it employeeCount.component.css. Copy and paste the following style class.

employeeCount.component.css

```
.radioClass {
  color: #369;
  font-family: Arial, Helvetica, sans-serif;
  font-size: large;
}
```

Now let's add the view template for EmployeeCountComponent. Add a new HTML page to the employee folder. Name it employeeCount.component.html. Copy and paste the following html.

employeeCount.component.html

Notice we have 3 radio buttons and bound them to the 3 properties (all, male, female) we have in the component class. We are using interpolation for data-binding.

```
<span class="radioClass">Show : </span>

<input type="radio" name="options" />
<span class="radioClass">{{"All(" + all + ")"}}</span>

<input name="options" type="radio">
<span class="radioClass">{{"Male(" + male + ")"}}</span>

<input name="options" type="radio">
<span class="radioClass">{{"Female(" + female + ")"}}</span>
```

Nest EmployeeCountComponent in EmployeeListComponent component. To do this, use EmployeeCountComponent selector (employee-count) as a directive <employee-count></employee-count> on EmployeeListComponent component as shown below.

```
<employee-count></employee-count>
<br /><br />
<table>
  <thead>
    <tr>
      <th>Code</th>
      <th>Name</th>
      <th>Gender</th>
      <th>Annual Salary</th>
      <th>Date of Birth</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let employee of employees;">
      <td>{{employee.code | uppercase}}</td>
      <td>{{employee.name | employeeTitle:employee.gender }}</td>
      <td>{{employee.gender}}</td>
      <td>{{employee.annualSalary | currency:'USD':true:'1.3-3'}}</td>
      <td>{{employee.dateOfBirth | date:'dd/MM/y'}}</td>
    </tr>
    <tr *ngIf="!employees || employees.length==0">
      <td colspan="5">
        No employees to display
      </td>
    </tr>
  </tbody>
</table>
```

Finally, declare EmployeeCountComponent in module.ts file. Please make sure you import the component first and then add it to the declarations array of @NgModule decorator.

```
import { EmployeeCountComponent } from './employee/employeeCount.component';
```

```
@NgModule({
  imports: [BrowserModule, FormsModule],
  declarations: [AppComponent, EmployeeComponent,
    EmployeeListComponent, EmployeeTitlePipe, EmployeeCountComponent],
  bootstrap: [AppComponent]
})
```

```
export class AppModule { }
```

At this point, run the project and you should see employee count radio buttons and the employee list.

At the moment the employee counts are hard coded with in the EmployeeCountComponent. In our next assignment we will discuss how to pass the count values from the container component i.e from the EmployeeListComponent to the nested component i.e EmployeeCountComponent.

Assignment 21: Angular component input properties

In this assignment we will discuss **how to pass data from the container component to the nested component using input properties.**

At the moment the count of employees displayed against each radio button are hard-coded with in the EmployeeCountComponent.

Show : ☐ All(10) ☐ Male(5) ☒ Female(5)

Code	Name	Gender	Annual Salary	Date of Birth
EMP101	Mr.Tom	Male	\$5,500.000	25/06/1988
EMP102	Mr.Alex	Male	\$5,700.950	06/09/1982
EMP103	Mr.Mike	Male	\$5,900.000	08/12/1979
EMP104	Miss.Mary	Female	\$6,500.826	14/10/1980
EMP105	Miss.Nancy	Female	\$6,700.826	15/12/1982

Here is the code in **employeeCount.component.ts** file : Notice the values for the 3 properties (all, male, female) are hard-coded. We want the values for these 3 properties to be passed from the container component i.e EmployeeListComponent.

```
export class EmployeeCountComponent {  
  all: number = 10;  
  male: number = 5;  
  female: number = 5;  
}
```

Convert a component property to an input property using @Input decorator : To be able to pass the values for these 3 properties from the container component to the nested component we need to decorate the properties with @Input() decorator. Decorating a property with @Input() decorator makes the property an input property. Notice we have also removed the default hard-coded values, as we will be passing the values from the parent component i.e EmployeeListComponent. To be able to use the @Input() decorator we will have to first import it from @angular/core.

```

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

@Component({
  selector: 'employee-count',
  templateUrl: 'app/employee/employeeCount.component.html',
  styleUrls: ['app/employee/employeeCount.component.css']
})
export class EmployeeCountComponent {
  @Input()
  all: number;

  @Input()
  male: number;

  @Input()
  female: number;
}

```

Passing data from the parent component to the child component : There are 2 modifications that we need to do in EmployeeListComponent to be able to pass values from the parent component i.e EmployeeListComponent to the child component i.e EmployeeCountComponent. The first change is in EmployeeListComponent TypeScript file as shown below. Notice I have introduced 3 methods that return male employees count, female employees count and total employees count.

```

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

@Component({
  selector: 'list-employee',
  templateUrl: 'app/employee/employeeList.component.html',
  styleUrls: ['app/employee/employeeList.component.css']
})
export class EmployeeListComponent {
  employees: any[];

  constructor() {
    this.employees = [
      {
        code: 'emp101', name: 'Tom', gender: 'Male',
        annualSalary: 5500, dateOfBirth: '6/25/1988'
      },
      {
        code: 'emp102', name: 'Alex', gender: 'Male',
        annualSalary: 5700.95, dateOfBirth: '9/6/1982'
      },
      {
        code: 'emp103', name: 'Mike', gender: 'Male',
        annualSalary: 5900, dateOfBirth: '12/8/1979'
      },
    ],
  }
}

```

```

    {
      code: 'emp104', name: 'Mary', gender: 'Female',
      annualSalary: 6500.826, dateOfBirth: '10/14/1980'
    },
    {
      code: 'emp105', name: 'Nancy', gender: 'Female',
      annualSalary: 6700.826, dateOfBirth: '12/15/1982'
    },
  ];
}

getTotalEmployeesCount(): number {
  return this.employees.length;
}

getMaleEmployeesCount(): number {
  return this.employees.filter(e => e.gender === 'Male').length;
}

getFemaleEmployeesCount(): number {
  return this.employees.filter(e => e.gender === 'Female').length;
}
}

```

Please note : In the filter method we are using tripple equals (===) instead of double equals (==). The table below explains single, double and tripple equals in TypeScript.

Operator	Use to
=	Assign a value
==	Compare two values
===	Compare two values and their types

The second change is in the view template of EmployeeListComponent i.e **employeeList.component.html** file. Notice with in <employee-count> directive we are using property binding to bind the properties (all, male, female) of the nested component (EmployeeCountComponent) with the 3 methods in the container component (EmployeeListComponent).

```

<employee-count [all]="getTotalEmployeesCount()"
               [male]="getMaleEmployeesCount()"
               [female]="getFemaleEmployeesCount()">
</employee-count>
<br /><br />
<table>
  <thead>
    <tr>

```



```

        <th>Code</th>
        <th>Name</th>
        <th>Gender</th>
        <th>Annual Salary</th>
        <th>Date of Birth</th>
    </tr>
</thead>
<tbody>
    <tr *ngFor="let employee of employees;">
        <td>{{employee.code | uppercase}}</td>
        <td>{{employee.name | employeeTitle:employee.gender }}</td>
        <td>{{employee.gender}}</td>
        <td>{{employee.annualSalary | currency:'USD':true:'1.3-3'}}</td>
        <td>{{employee.dateOfBirth | date:'dd/MM/y'}}</td>
    </tr>
    <tr *ngIf="!employees || employees.length==0">
        <td colspan="5">
            No employees to display
        </td>
    </tr>
</tbody>
</table>

```

At this point, save all the changes and run the application and we see the correct count of employee next to each radio button.

Show : ☒ All(5) ☐ Male(3) ☐ Female(2)

Code	Name	Gender	Annual Salary	Date of Birth
EMP101	Mr.Tom	Male	\$5,500.000	25/06/1988
EMP102	Mr.Alex	Male	\$5,700.950	06/09/1982
EMP103	Mr.Mike	Male	\$5,900.000	08/12/1979
EMP104	Miss.Mary	Female	\$6,500.826	14/10/1980
EMP105	Miss.Nancy	Female	\$6,700.826	15/12/1982

Now, let's add the following new employee object to the to the employees array in EmployeeListComponent.

```

{
  code: 'emp106', name: 'Steve', gender: 'Male',
  annualSalary: 7700.481, dateOfBirth: '11/18/1979'
}

```

Save changes and reload the web page and notice that All count and Male count is increased by 1 as expected.

Show : ☒ All(6) ☐ Male(4) ☐ Female(2)

Code	Name	Gender	Annual Salary	Date of Birth
EMP101	Mr.Tom	Male	\$5,500.000	25/06/1988
EMP102	Mr.Alex	Male	\$5,700.950	06/09/1982
EMP103	Mr.Mike	Male	\$5,900.000	08/12/1979
EMP104	Miss.Mary	Female	\$6,500.826	14/10/1980
EMP105	Miss.Nancy	Female	\$6,700.826	15/12/1982
EMP106	Mr.Steve	Male	\$7,700.481	18/11/1979

At the moment when we click the radio buttons nothing happens. In our next assignment we will discuss **how to pass data from the child component to the parent component**.i.e when a radio button checked event is raised in the child component, we want to know about it in the parent component so we can react and decide which employees to show in the table depending on the selection of the radio button.

Assignment 22: Angular component output properties

In this assignment we will discuss

- How to pass user actions or user entered values or selections from the child component to the parent component using output properties.
- Along the way we will discuss creating custom events using angular EventEmitter class
- Finally what is ng-container directive and it's use

Show : ☒ All(5) ☐ Male(3) ☐ Female(2)

Code	Name	Gender	Annual Salary	Date of Birth
EMP101	Mr.Tom	Male	\$5,500.000	25/06/1988
EMP102	Mr.Alex	Male	\$5,700.950	06/09/1982
EMP103	Mr.Mike	Male	\$5,900.000	08/12/1979
EMP104	Miss.Mary	Female	\$6,500.826	14/10/1980
EMP105	Miss.Nancy	Female	\$6,700.826	15/12/1982

At the moment when we click the radio buttons, nothing happens. Here is what we want to do.

User Action	What should happen
All(6) radio button is clicked	Display all the employees in the table
Male(4) radio button is clicked	Display the 4 Male employees in the table
Female(2) radio button is clicked	Display the 2 Female employees in the table

To achieve this we are going to make use of component output properties. First let's look at the changes required in the nested component i.e EmployeeCountComponent.

The changes required in employeeCount.component.ts are commented and self-explanatory

```

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

@Component({
  selector: 'employee-count',
  templateUrl: 'app/employee/employeeCount.component.html',
  styleUrls: ['app/employee/employeeCount.component.css']
})
export class EmployeeCountComponent {
  @Input()
  all: number;

  @Input()
  male: number;

  @Input()
  female: number;

  // Holds the selected value of the radio button
  selectedRadioButtonValue: string = 'All';

  // The Output decorator makes the property an Output property
  // EventEmitter class is used to create the custom event
  // When the radio button selection changes, the selected
  // radio button value which is a string gets passed to the
  // event handler method. Hence, the event payload is string.
  @Output()
  countRadioButtonSelectionChanged: EventEmitter<string> =
    new EventEmitter<string>();

  // This method raises the custom event. We will bind this
  // method to the change event of all the 3 radio buttons
  onRadioButtonSelectionChange() {
    this.countRadioButtonSelectionChanged
      .emit(this.selectedRadioButtonValue);
  }
}

```

The following are the changes required in the view template of EmployeeCountComponent i.e employeeCount.component.html. Notice we have made 3 changes on each radio button

1. value attribute is set to (All, Male or Female)
2. Implemented 2 way data-binding using the ngModel directive. Notice ngModel is bound to selectedRadioButtonValue property in the component class. This 2 way data-binding ensures whenever the radio button selection changes, the selectedRadioButtonValue property is updated with the value of the selected radio button.

- onRadioButtonSelectionChange() method is binded to "change" event of the radio button. So this means whenever, the selection of the radio button changes, onRadioButtonSelectionChange() method raises the custom event "countRadioButtonSelectionChanged". We defined this custom event using Angular EventEmitter class.

```
<span class="radioClass">Show : </span>
```

```
<input name='options' type='radio' value="All"  
  [(ngModel)]="selectedRadioButtonValue"  
  (change)="onRadioButtonSelectionChange()">  
<span class="radioClass">{{'All(' + all + ')'}}</span>
```

```
<input name="options" type="radio" value="Male"  
  [(ngModel)]="selectedRadioButtonValue"  
  (change)="onRadioButtonSelectionChange()">  
<span class="radioClass">{{"Male(" + male + ")"}}</span>
```

```
<input name="options" type="radio" value="Female"  
  [(ngModel)]="selectedRadioButtonValue"  
  (change)="onRadioButtonSelectionChange()">
```

```
<span class="radioClass">{{"Female(" + female + ")"}}</span>
```

Now let's look at the changes required in the parent component i.e EmployeeListComponent

The following are the changes required in the EmployeeListComponent class i.e employeeList.component.ts. The changes are commented and self-explanatory

```
import { Component } from '@angular/core';  
  
@Component({  
  selector: 'list-employee',  
  templateUrl: 'app/employee/employeeList.component.html',  
  styleUrls: ['app/employee/employeeList.component.css']  
})  
  
export class EmployeeListComponent {  
  employees: any[];  
  
  // This property keeps track of which radio button is selected  
  // We have set the default value to All, so all the employees  
  // are displayed in the table by default  
  selectedEmployeeCountRadioButton: string = 'All';  
  
  constructor() {  
    this.employees = [
```

```

        {
            code: 'emp101', name: 'Tom', gender: 'Male',
            annualSalary: 5500, dateOfBirth: '6/25/1988'
        },
        {
            code: 'emp102', name: 'Alex', gender: 'Male',
            annualSalary: 5700.95, dateOfBirth: '9/6/1982'
        },
        {
            code: 'emp103', name: 'Mike', gender: 'Male',
            annualSalary: 5900, dateOfBirth: '12/8/1979'
        },
        {
            code: 'emp104', name: 'Mary', gender: 'Female',
            annualSalary: 6500.826, dateOfBirth: '10/14/1980'
        },
        {
            code: 'emp105', name: 'Nancy', gender: 'Female',
            annualSalary: 6700.826, dateOfBirth: '12/15/1982'
        },
        {
            code: 'emp106', name: 'Steve', gender: 'Male',
            annualSalary: 7700.481, dateOfBirth: '11/18/1979'
        },
    ];
}

getTotalEmployeesCount(): number {
    return this.employees.length;
}

getMaleEmployeesCount(): number {
    return this.employees.filter(e => e.gender === 'Male').length;
}

getFemaleEmployeesCount(): number {
    return this.employees.filter(e => e.gender === 'Female').length;
}

// Depending on which radio button is selected, this method updates
// selectedEmployeeCountRadioButton property declared above
// This method is called when the child component (EmployeeCountComponent)
// raises the custom event - countRadioButtonSelectionChanged
// The event binding is specified in employeeList.component.html
onEmployeeCountRadioButtonChange(selectedRadioButtonValue: string): void {
    this.selectedEmployeeCountRadioButton = selectedRadioButtonValue;
}
}

```

The following are the changes required in the view template of EmployeeListComponent i.e employeeList.component.html.

1. onEmployeeCountRadioButtonChange(\$event) method is bound to the custom event - countRadioButtonSelectionChanged. The \$event object will have the selected radio button value as that is what is passed as the event payload from the nested component. The event handler method (onEmployeeCountRadioButtonChange()) in the component class updates the property "selectedEmployeeCountRadioButton". This property is then used along with *ngIf structural directive to decide which employee objects to display in the table.

2. On the <tr> element, we are using "ngIf" directive along with selectedEmployeeCountRadioButton property which controls the employee objects to display. Notice, just above the <tr> element, we have introduced <ng-container> element and the "ngFor" directive is placed on this element. If you are wondering why we have done this, Angular does not allow multiple structural directives to be placed on one element as shown below.

```
<tr *ngFor="let employee of employees;"
    *ngIf="selectedEmployeeCountRadioButton=='All'
    || selectedEmployeeCountRadioButton==employee.gender">
```

The above line of code raises the following error

Can't have multiple template bindings on one element. Use only one attribute named 'template' or prefixed with *.

employeeList.component.html

```
<employee-count [all]="getTotalEmployeesCount()"
    [male]="getMaleEmployeesCount()"
    [female]="getFemaleEmployeesCount()"
    (countRadioButtonSelectionChanged)="onEmployeeCountRadioButtonChange($event)">
</employee-count>
<br /><br />
<table>
  <thead>
    <tr>
      <th>Code</th>
      <th>Name</th>
      <th>Gender</th>
      <th>Annual Salary</th>
      <th>Date of Birth</th>
    </tr>
  </thead>
  <tbody>
    <ng-container *ngFor="let employee of employees;">
      <tr *ngIf="selectedEmployeeCountRadioButton=='All' ||
        selectedEmployeeCountRadioButton==employee.gender">
        <td>{{employee.code | uppercase}}</td>
```

```

        <td>{{employee.name | employeeTitle:employee.gender }}</td>
        <td>{{employee.gender}}</td>
        <td>{{employee.annualSalary | currency:'USD':true:'1.3-3'}}</td>
        <td>{{employee.dateOfBirth | date:'dd/MM/y'}}</td>
    </tr>
</ng-container>
<tr *ngIf="!employees || employees.length==0">
    <td colspan="5">
        No employees to display
    </td>
</tr>
</tbody>
</table>

```

At this point, run the application and test. Notice, the correct set of employees are displayed based on the selection of the radio button.

Assignment 23: Interfaces in Angular 2

In this assignment we will discuss, **what are interfaces and when to use them in Angular**.

What is an Interface in TypeScript

If you have experience with any object oriented programming language like C#, Java or C++, then you know an interface is an abstract type. It only contain declarations of properties, methods and events. The implementation for the interface members is provided by a class that implements the interface.

If a class that implements the interface fails to provide implementation for all the interface members, the language compiler raises an error alerting the developer that something has been missed.

In general, an interface defines a contract (i.e the shape or structure of an API), that an implementing class must adhere to. Even in TypeScript we use the interface for the same purpose.

We know TypeScript is a strongly typed language. This means every property we define in a TypeScript class has a type associated with it. Similarly every method parameter and return type has a type. As you know by now, TypeScript has several built-in pre-defined types like string, number, boolean etc.

However, the business objects that we usually create in real-world applications like Employee, Customer, Order, Invoice etc, does not have a pre-defined type. In this case, we can use an interface to create a custom type for our business object.

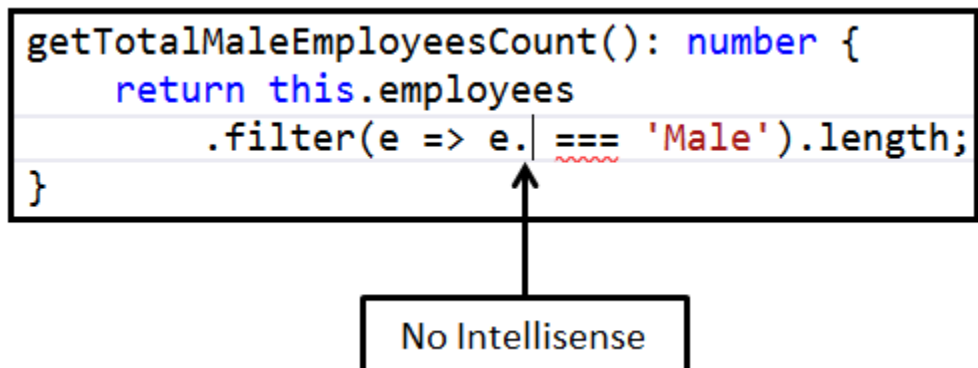
Notice the below line of code in EmployeeListComponent class. The property "**employees**" is defined as an array of any type.

```
employees: any[];
```

Since we do not have a Type for employee object, we specified the type as any.

There are 2 problems with the above line of code

1. For the object properties in the array we do not get intellisense



2. Since we do not get intellisense, we are prone to making typographical errors and the compiler will not be able to flag them as errors. We will come to know about these errors only at runtime.

Let's create a Type for employee using an interface as shown below. Add a new TypeScript file to the employee folder. Name it **employee.ts**. Copy and paste the following code.

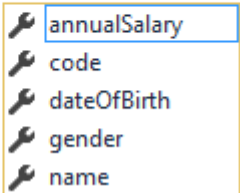
```
export interface IEmployee {  
  code: string;  
  name: string;  
  gender: string;  
  annualSalary: number;  
  dateOfBirth: string;  
}
```

With this IEmployee interface in place, we can now import and use the interface type as the type for "employees" property. The code in EmployeeListComponent class is shown below.

```
import { IEmployee } from './employee';  
  
export class EmployeeListComponent {  
  employees: IEmployee[];  
}
```

Since we have specified a type for the "employees" property, we now get intellisense for the object properties in the array

```
getTotalMaleEmployeesCount(): number {  
  return this.employees  
    .filter(e => e. === 'Male').length;  
}
```



If we make any typographical errors with the property names, we will get to know these errors right away at compile time.

While we are here, let's discuss some of the concepts related to TypeScript interfaces. Consider the following interface. The code is commented and self-explanatory.

```
export interface IEmployee {  
  code: string;  
  name: string;  
  gender: string;  
  annualSalary: number;  
  dateOfBirth: string;  
  // To make a property optional use a ?  
  // A class that implements this interface need
```

```

    // not provide implementation for this property
    department?: string;

    computeMonthlySalary(annualSalary: number): number;
}

export class Employee implements IEmployee {
    // All the interface mandatory properties are defined
    public code: string;
    public name: string;
    public gender: string;
    public annualSalary: number;
    public dateOfBirth: string;

    // The above class properties are then initialized
    // using the constructor parameters. To do something
    // like this, TypeScript has a shorthand syntax which
    // reduces the amount of code we have to write
    constructor(code: string, name: string, gender: string,
        annualSalary: number, dateOfBirth: string) {
        this.code = code;
        this.name = name;
        this.gender = gender;
        this.annualSalary = annualSalary;
        this.dateOfBirth = dateOfBirth;
    }

    // Implementation of the interface method
    computeMonthlySalary(annualSalary: number): number {
        return annualSalary / 12;
    }
}

```

Here is the shorthand syntax to initialise class properties with constructor parameters

```

export interface IEmployee {
    code: string;
    name: string;
    gender: string;
    annualSalary: number;
    dateOfBirth: string;
}

export class Employee implements IEmployee {

    constructor(public code: string, public name: string, public gender: string,
        public annualSalary: number, public dateOfBirth: string) {
    }
}

```

```
}
```

The above shorthand syntax is not limited to public class properties. We can also use it with private class properties. In the example below we have 2 private properties (firstName & lastName) which are initialised with class constructor.

```
export class Employee {  
  private firstName: string;  
  private lastName: string;  
  
  constructor(firstName: string, lastName: string) {  
    this.firstName = firstName;  
    this.lastName = lastName;  
  }  
}
```

We can rewrite the above code using shorthand syntax as shown below. In both the cases the generated JavaScript code is the same.

```
export class Employee {  
  constructor(private firstName: string, private lastName: string) {  
  }  
}
```

Interfaces in TypeScript

1. Use interface keyword to create an interface
2. It is common to prefix the interface name with capital letter "I". However, some interfaces in Angular does not have the prefix "I". For example, OnInit interface
3. Interface members are public by default and does not require explicit access modifiers. It is a compile time error to include an explicit access modifier. You will see an error message like - public modifier cannot appear on a type member.
4. A class that implements an interface must provide implementation for all the interface members unless the members are marked as optional using the ?operator
5. Use the implements keyword to make a class implement an interface
6. TypeScript interfaces exist for developer convenience and are not used by Angular at runtime. During transpilation, no JavaScript code is generated for an interface. It is only used by Typescript for type checking during development.
7. To reduce the amount of code you have to write, consider using short-hand syntax to initialise class properties with constructor parameters

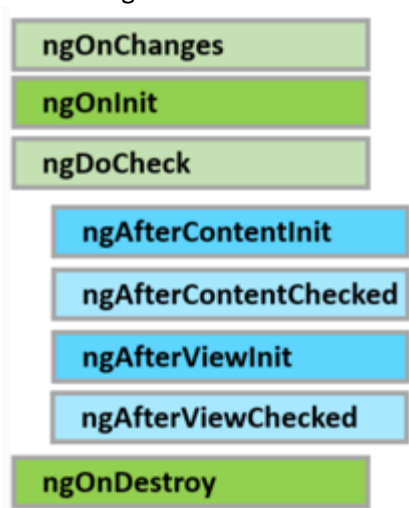
Assignment 24: Angular component lifecycle hooks

In this assignment we will discuss **Angular component lifecycle hooks**.

A component has a lifecycle managed by Angular. Angular

1. Creates the component
2. Renders the component
3. Creates and renders the component children
4. Checks when the component data-bound properties change, and
5. Destroys the component before removing it from the DOM

To tap into and react when these life cycle events occur, angular offers several lifecycle hooks as shown in the image below.



The 3 most commonly used hooks are

Life Cycle Hook	Purpose
ngOnChanges	Executes, every time the value of an input property changes. The hook method receives a SimpleChanges object containing current and previous property values. This is called before ngOnInit
ngOnInit	Executes after the constructor and after ngOnChange hook for the first time. It is most commonly used for component initialisation and retrieving data from a database
ngOnDestroy	Executes just before angular destroys the component and generally used for performing cleanup

There are 3 simple steps to use the Life Cycle Hooks

Step 1 : Import the Life Cycle Hook interface. For example, to use `ngOnInit()` life cycle hook, import `OnInit` interface.

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

Step 2 : Make the component class implement the Life Cycle Hook interface, using the `implements` keyword as shown below. This step is optional, but good to have so you will get editor support and flags errors at compile time if you incorrectly implement the interface method or make any typographical errors.

```
export class SimpleComponent implements OnInit { }
```

Step 3 : Write the implementation code for the life cycle interface method. Each interface has a single hook method whose name is the interface name prefixed with `ng`.

```
ngOnInit() {  
    console.log('OnInit Life Cycle Hook');  
}
```

Let's understand `ngOnChanges` life cycle hook with a simple example. Here is what we want to do. As soon as the user starts typing into the text box, we want to capture the current and previous value and log it to the browser console as shown below. We can very easily achieve this by using the `ngOnChanges` life cycle hook.

Your Text :

You entered : PragimTech



`ngOnChanges`, is called every time the value of an input property of a component changes. So first let's create a `SimpleComponent` with an input property as shown below. We will continue with the example we worked with in our previous assignment. Add a new folder in the `App` folder and name it `Others`. Add a new TypeScript file to this folder and name it `simple.component.ts`. Copy and paste the following code which is commented and self explanatory.

```

// Step 1 : Import OnChanges and SimpleChanges
import { Component, Input, OnChanges, SimpleChanges } from '@angular/core';

// The selector "simple" will be used as the directive
// where we want to use this component. Notice we are
// also using the simpleInput property with interpolation
// to display the value it receives from the parent
// component
@Component({
  selector: 'simple',
  template: `You entered : {{simpleInput}}`
})
// Step 2 : Implement OnChanges Life Cycle Hook interface
export class SimpleComponent implements OnChanges {
  // Input property. As and when this property changes
  // ngOnChanges life cycle hook method is called
  @Input() simpleInput: string;

  // Step 3 : Implementation for the hook method
  // This code logs the current and previous value
  // to the console.
  ngOnChanges(changes: SimpleChanges) {
    for (let propertyName in changes) {
      let change = changes[propertyName];
      let current = JSON.stringify(change.currentValue);
      let previous = JSON.stringify(change.previousValue);
      console.log(propertyName + ': currentValue = '
        + current + ', previousValue = ' + previous);
      // The above line can be rewritten using
      // placeholder syntax as shown below
      // console.log(`${propertyName}: currentValue
      // = ${current }, previousValue = ${previous }`);
    }
  }
}

```

Now copy and paste the following code in our root component - app.component.ts

```

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

// Notice we have placed the text box in this root component
// To keep the value in the textbox and the component property
// value "userText" in sync we are using 2 way data binding
// We have also bound userText property of this component
// to the input property of the SimpleComponent
@Component({
  selector: 'my-app',

```

```
    template: `Your Text : <input type='text' [(ngModel)]='userText'/>
      <br/><br/>
      <simple [simpleInput]='userText'></simple>
    ,
  })
  export class AppComponent {
    userText: string = 'Pragim';
  }
```

Please note : Do not forget to import and declare SimpleComponent in the root module - app.module.ts

At this point run the application and launch browser developer tools. As you start typing into the text box, the changes are logged to the console by the ngOnChanges life cycle hook method.

The steps for implementing the other component life cycle hook methods are very similar.

Assignment 25: Angular services tutorial

In this assignment we will discuss

- Why we need a service in Angular
- Creating a service in Angular
- Injecting and using the service
- Difference between constructor and ngOnInit

Why do we need a service in Angular

A service in Angular is generally used when you need to reuse data or logic across multiple components. Anytime you see logic or data-access duplicated across multiple components, think about refactoring that piece of logic or data-access code into a service. Using a service ensures we are not violating one of the Software principles - DRY ((Don't repeat yourself). The logic or data access is implemented once in a service, and the service can be used across all the components in our application.

Without the service you would have to repeat your code in each component. Imagine the overhead in terms of time and effort required to develop, debug, test and maintain the duplicated code across multiple places instead of having that duplicated code at one central place like a service and reusing that service where required.

Creating a service in Angular : We will be working with the same example that we have been working with so far in this assignment series. Add a new TypeScript file to the "employee" folder and name it employee.service.ts. Copy and paste the following code. At the moment we have the data hard-coded in the service method. In a later assignment we will discuss retrieving data from a remote server using HTTP.

```
import { Injectable } from '@angular/core';
import { IEmployee } from './employee';

// The @Injectable() decorator is used to inject other dependencies
// into this service. As our service does not have any dependencies
// at the moment, we may remove the @Injectable() decorator and the
// service works exactly the same way. However, Angular recommends
// to always use @Injectable() decorator to ensure consistency
@Injectable()
export class EmployeeService {
  getEmployees(): IEmployee[] {
    return [
      {
        code: 'emp101', name: 'Tom', gender: 'Male',
        annualSalary: 5500, dateOfBirth: '6/25/1988'
      },
      {
        code: 'emp102', name: 'Alex', gender: 'Male',
```

```

        annualSalary: 5700.95, dateOfBirth: '9/6/1982'
    },
    {
        code: 'emp103', name: 'Mike', gender: 'Male',
        annualSalary: 5900, dateOfBirth: '12/8/1979'
    },
    {
        code: 'emp104', name: 'Mary', gender: 'Female',
        annualSalary: 6500.826, dateOfBirth: '10/14/1980'
    },
    {
        code: 'emp105', name: 'Nancy', gender: 'Female',
        annualSalary: 6700.826, dateOfBirth: '12/15/1982'
    },
    {
        code: 'emp106', name: 'Steve', gender: 'Male',
        annualSalary: 7700.481, dateOfBirth: '11/18/1979'
    },
    ],
};
}
}

```

Injecting and using the service : We need the employee service we created above in EmployeeListComponent. So let's import, register and use the Employee service in EmployeeListComponent as shown below.

```

// Import OnInit Life Cycle Hook interface
import { Component, OnInit } from '@angular/core';
import { IEmployee } from './employee';
// Import EmployeeService
import { EmployeeService } from './employee.service';

@Component({
    selector: 'list-employee',
    templateUrl: 'app/employee/employeeList.component.html',
    styleUrls: ['app/employee/employeeList.component.css'],
    // Register EmployeeService in this component by
    // declaring it in the providers array
    providers: [EmployeeService]
})
// Make the class implement OnInit interface
export class EmployeeListComponent implements OnInit {
    employees: IEmployee[];

    selectedEmployeeCountRadioButton: string = 'All';

    // Inject EmployeeService using the constructor
    // The private variable _employeeService which points to

```

```

// EmployeeService singleton instance is then available
// throughout this class
constructor(private _employeeService: EmployeeService) {
}

// In ngOnInit() life cycle hook call the getEmployees()
// service method of EmployeeService using the private
// variable _employeeService
ngOnInit() {
  this.employees = this._employeeService.getEmployees();
}

getTotalEmployeesCount(): number {
  return this.employees.length;
}

getTotalMaleEmployeesCount(): number {
  return this.employees
    .filter(e => e.gender === 'Male').length;
}

getTotalFemaleEmployeesCount(): number {
  return this.employees.filter(e => e.gender === 'Female').length;
}

onEmployeeCountRadioButtonChange(selectedRadioButtonValue: string): void {
  this.selectedEmployeeCountRadioButton = selectedRadioButtonValue;
}
}

```

Please do not forget to use the EmployeeListComponent selector (list-employee) as a directive in the root component - AppComponent (app.component.ts) as shown below.

```

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

@Component({
  selector: 'my-app',
  template: `<list-employee></list-employee>`
})
export class AppComponent {
}

```

At this point, run the application and notice it works exactly the same way as before.

The following line of code which calls the service, can be placed even in the constructor and the application still works exactly the same way as before. So what is the difference between a constructor and ngOnInit life cycle hook, and when to use one over the other.

```
this.employees = this._employeeService.getEmployees();
```

Difference between constructor and ngOnInit

A class constructor is automatically called when an instance of the class is created. It is generally used to initialise the fields of the class and its sub classes.

ngOnInit is a life cycle hook method provided by Angular. ngOnInit is called after the constructor and is generally used to perform tasks related to Angular bindings. For example, ngOnInit is the right place to call a service method to fetch data from a remote server. We can also do the same using a class constructor, but the general rule of thumb is, tasks that are time consuming should use ngOnInit instead of the constructor. As fetching data from a remote server is time consuming, the better place for calling the service method is ngOnInit.

So coming back to our example, the dependency injection is done using the class constructor and the actual service method call is issued from ngOnInit life cycle hook as shown below

```
constructor(private _employeeService: EmployeeService) { }
```

```
ngOnInit() {  
    this.employees = this._employeeService.getEmployees();  
}
```

In our next assignment we will discuss **retrieving data from a remote server using HTTP**.

Assignment 26: Angular and ASP.NET Web API

In this assignment we will discuss **creating ASP.NET Web API service** that retrieves employees data from a database table. In our next assignment we will discuss, how to call this ASP.NET Web API service using Angular

Step 1 : Execute the following SQL Script using SQL Server Management studio. This script creates

- EmployeeDB database
- Creates the Employees table and populate it with sample data

Create Database EmployeeDB
Go

Use EmployeeDB
Go

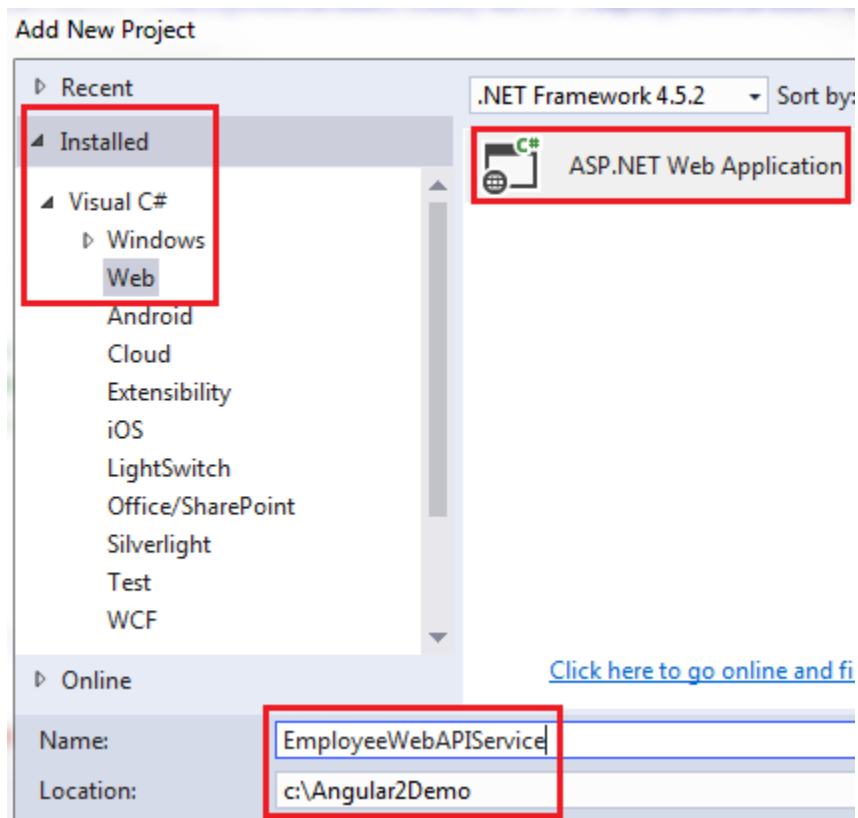
Create table Employees
(
 code nvarchar(50) primary key,
 name nvarchar(50),
 gender nvarchar(50),
 annualSalary decimal(18,3),
 dateOfBirth nvarchar(50)
)
Go

Insert into Employees values ('emp101', 'Tom', 'Male', 5500, '6/25/1988')
Insert into Employees values ('emp102', 'Alex', 'Male', 5700.95, '9/6/1982')
Insert into Employees values ('emp103', 'Mike', 'Male', 5900, '12/8/1979')
Insert into Employees values ('emp104', 'Mary', 'Female', 6500.826, '10/14/1980')
Insert into Employees values ('emp105', 'Nancy', 'Female', 6700.826, '12/15/1982')
Insert into Employees values ('emp106', 'Steve', 'Male', 7700.481, '11/18/1979')

Step 2 : To keep Angular and Web API projects separate, let's create a new project for our Web API Service. Right click on "**Angular2Demo**" solution in the Solution Explorer and select Add - New Project.

Step 3 : In the Add New Project window

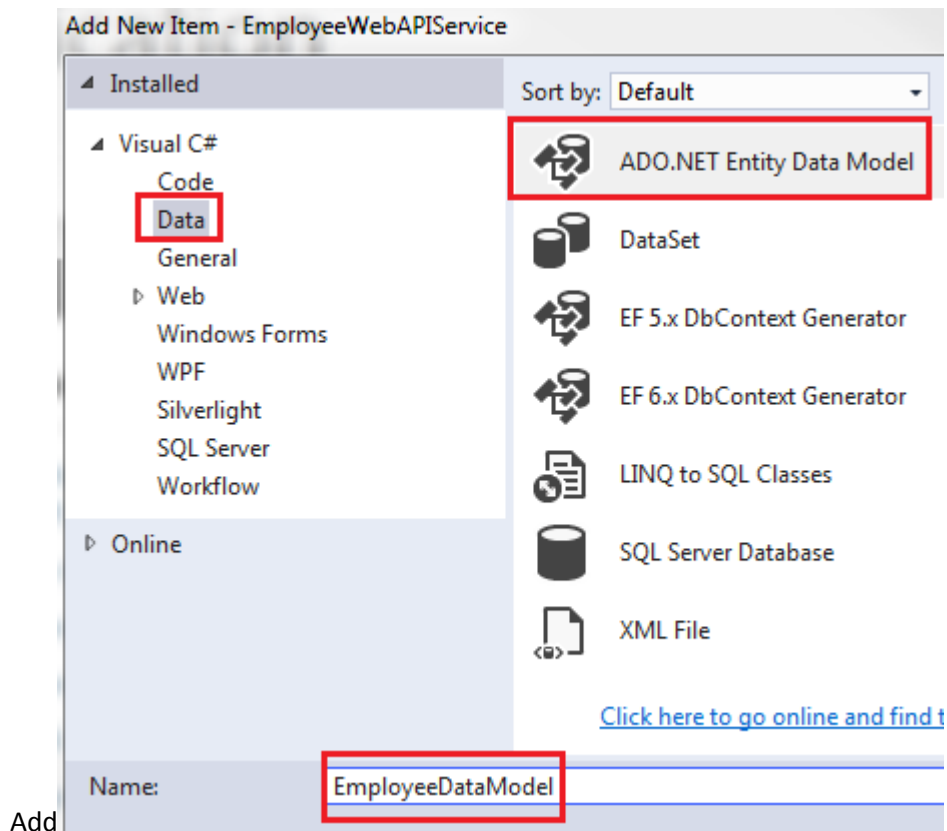
Select "Visual C#" under "Installed - Templates"
From the middle pane select, ASP.NET Web Application
Name the project "EmployeeWebAPIService" and click OK



Step 4 : On the next window, select "Web API" and click "OK". At this point you should have the Web API project created.

Step 5 : Add ADO.NET Entity Data Model to retrieve data from the database. Right click on "EmployeeWebAPIService" project and select Add - New Item

1. In the "Add New Item" window
 2. Select "Data" from the left pane
 3. Select ADO.NET Entity Data Model from the middle pane
- In the Name text box, type EmployeeDataModel



Step 6 : On the Entity Data Model Wizard, select "EF Designer from database" option and click next

Step 7 : On the next screen, click "New Connection" button

Step 8 : On "Connection Properties" window, set

1. Server Name = (local)
2. Authentication = Windows Authentication
3. Select or enter a database name = EmployeeDB
4. Click OK and then click Next

Step 9 : On the nex screen, select "Employees" table and click Finish.

Adding Web API Controller

1. Right click on the Controllers folder in EmployeeWebAPIService project and select Add - Controller
2. Select "Web API 2 Controller - Empty" and click "Add"
3. On the next screen set the Controller Name = EmployeesController and click Add
4. Copy and paste the following code in EmployeesController.cs

```

using System.Collections.Generic;
using System.Linq;
using System.Web.Http;

namespace EmployeeWebAPIService.Controllers
{
    public class EmployeesController : ApiController
    {
        public IEnumerable<Employee> Get()
        {
            using(EmployeeDBEntities entities = new EmployeeDBEntities())
            {
                return entities.Employees.ToList();
            }
        }

        public Employee Get(string code)
        {
            using (EmployeeDBEntities entities = new EmployeeDBEntities())
            {
                return entities.Employees.FirstOrDefault(e => e.code == code);
            }
        }
    }
}

```

At this point when you navigate to /api/employees you will see all the employees as expected. However, when you navigate to /api/employees/emp101, we expect to see employee whose employee code is emp101, but we still see the list of all employees.

This is because the parameter name for the Get() method in EmployeesController is "code"

```

public Employee Get(string code)
{
    using (EmployeeDBEntities entities = new EmployeeDBEntities())
    {
        return entities.Employees.FirstOrDefault(e => e.code == code);
    }
}

```

but in the default Web API route in WebApiConfig.cs file the parameter name is {id}. Change this to "code" as shown below

```

config.Routes.MapHttpRoute(
    name: "DefaultApi",
    routeTemplate: "api/{controller}/{code}",
    defaults: new { code = RouteParameter.Optional }
);

```


With this change if we navigate to `/api/employees/emp101` we see just that employee whose employee code is "emp101"

Assignment 27: Angular 2 http service tutorial

In this assignment we will discuss

How to call ASP.NET Web API service using Angular 2 http service. Though this example demonstrates calling ASP.NET Web API service, we can use this same approach to call any web service built using any server side technology. We will also briefly discuss Observable pattern

In our previous assignment we have created ASP.NET EmployeeWebAPIService. Here are the steps to call the Web API service using the Angular builtin http service.

Step 1 - Import the angular HTTP module : The first step is to import `HttpModule` which is present in a separate javascript file - `@angular/http`. After the `HttpModule` is imported, include it in the imports array of the `NgModule()` decorator of our root module "AppModule" which is in "app.module.ts" file. With this change we can now start using the angular built-in http service throughout our application to access web services over HTTP.

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

```
@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule],
  declarations: [
    AppComponent,
    EmployeeComponent,
    EmployeeListComponent,
    EmployeeTitlePipe,
    EmployeeCountComponent,
    SimpleComponent
  ],
  bootstrap: [AppComponent]
})
```

```
export class AppModule { }
```

Step 2 - Modify angular `EmployeeService` to issue a GET request using the builtin http service : The angular `EmployeeService` is in `employee.service.ts` file.

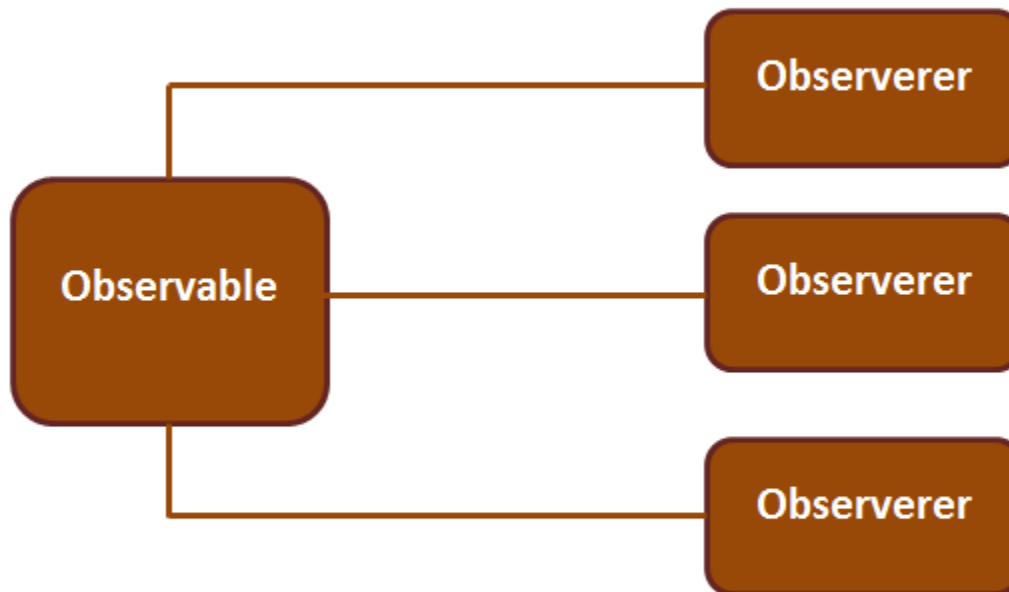
- Use the `EmployeeService` class constructor to inject Angular Http service. The injected http service can then be used anywhere in this class to call a web service over http.
- Since this Angular `EmployeeService` class has an injected dependency, `@Injectable()` decorator is required on this class. If there are no injectable dependencies then we may omit the `@Injectable()` decorator, but angular strongly recommends to use the `@Injectable()` decorator irrespective of there are injectible dependencies or not for consistency and future proof.

- Notice in the `getEmployees()` method, we are using the `get()` method of the angular http service to issue a get request over http. If you right click on `get()` method and go to it's definition you will notice that this method return `Observable<Response>`.
- `Observable<Response>` is not that useful to us, so we have set the return type of `getEmployees()` method to `Observable<IEmployee[]>`
- To convert `Observable<Response>` to `Observable<IEmployee[]>` we are using the `map` operator provided by rxjs.
- At the moment, we are not handling exceptions. We will discuss how to handle exceptions in our upcoming assignments.

What is an Observable

- Observable is an asynchronous pattern. In the Observable pattern we have an Observable and an Observer. Observer observes the Observable. In many implementations an Observer is also called as a Subscriber.
- An Observable can have many Observers (also called Subscribers).
- Observable emits items or notifications over time to which an Observer (also called Subscriber) can subscribe.
- When a subscriber subscribes to an Observable, the subscriber also specifies a callback function.
- This subscriber callback function is notified as and when the Observable emits items or notifications.
- Within this callback function we write code to handle data itmes or notifications received from the Observable.

The callback function is notified when the Observable emits data



Observers (also called Subscribers) subscribe to Observable with a callback function

```
import { Injectable } from '@angular/core';
import { IEmployee } from './employee';
// Import Http & Response from angular HTTP module
import { Http, Response } from '@angular/http';
// Import Observable from rxjs/Observable
import { Observable } from 'rxjs/Observable';
// Import the map operator
import 'rxjs/add/operator/map';

@Injectable()
export class EmployeeService {

  // Inject Angular http service
  constructor(private _http: Http) { }

  // Notice the method return type is Observable<IEmployee[]>
  getEmployees(): Observable<IEmployee[]> {
```

```

// To convert Observable<Response> to Observable<IEmployee[]>
// we are using the map operator
return this._http.get('http://localhost:24535/api/employees')
    .map((response: Response) => <IEmployee[]>response.json());
}
}

```

Step 3 - Subscribe to the Observable returned by angular EmployeeService : EmployeeListComponent needs the employees data returned by the service. So in the ngOnInit() method of "employeeList.component.ts" use the subscribe method as shown below.

```

ngOnInit() {
    this._employeeService.getEmployees()
        .subscribe(employeesData => this.employees = employeesData);
}

```

Notice to the subscribe() function we are passing an other arrow function as a parameter. This arrow function is called when the Observable emits an item. In our case the Observable emits an array of IEmployee objects. employeesData parameter receives the array of IEmployee objects, which we are then using to initialise employees property of the EmployeeListComponent class. We can specify upto 3 callback functions as parameters to the subscribe() method as shown below.

Callback Method	Purpose
onNext	The Observable calls this method whenever the Observable emits an item. The emitted item is passed as a parameter to this method
onError	The Observable calls this method if there is an error
onCompleted	The Observable calls this method after it has emitted all items, i.e after it has called onNext for the final time

At the moment, in our example, we only have used onNext() callback method. In our upcoming assignments we will discuss using onError() method to handle exceptions. Using Visual Studio intellisense you can see these 3 callback functions of the subscribe() method.

At this point, run the application. The data may not display and you see the following error in browser developer tools.

Cannot read property 'length' of undefined - at EmployeeListComponent.getTotalEmployeesCount

Reason for the above error : Consider this HTML in employeeList.component.html. Notice we are binding to the input properties (all, male and female) of <employee-count> component.

```

<employee-count [all]="getTotalEmployeesCount()"
    [male]="getTotalMaleEmployeesCount()"
    [female]="getTotalFemaleEmployeesCount()"
    (countRadioButtonSelectionChanged)=
    "onEmployeeCountRadioButtonChange($event)">

```

```
</employee-count>
```

Here is the `getTotalEmployeesCount()` method which is binded to "all" property of `<employee-count>` component.

```
getTotalEmployeesCount(): number {  
    return this.employees.length;  
}
```

The Web API call to retrieve is in `ngOnInit()`. Before the service call can initialise "employees" property of the `EmployeeListComponent` class, `getTotalEmployeesCount()` method is trying to access "length" property of "employees" property. "employees" property is still undefined at that point, so we get the error - Cannot read property 'length' of undefined.

```
ngOnInit() {  
    this._employeeService.getEmployees()  
        .subscribe(employeesData => this.employees = employeesData);  
}
```

To fix this error use angular structural directive `*ngIf` as shown below. This will delay the initialization of `employee-count` component until "employees" property is initialised.

```
<employee-count *ngIf="employees" [all]="getTotalEmployeesCount()"  
    [male]="getTotalMaleEmployeesCount()"  
    [female]="getTotalFemaleEmployeesCount()"  
    (countRadioButtonSelectionChanged)=  
    "onEmployeeCountRadioButtonChange($event)">  
</employee-count>
```

At this point if you run the application, you will see no employees on the web page and in the browser developer tools you will see the following error message
No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin 'http://localhost:12345' is therefore not allowed access.

We get this error because our Angular application and Web API service are in different projects. Because they are present in different projects the port numbers are different. Since the port numbers are different, the request from the angular project to the web api project is a cross domain request which violates the same origin policy and as a result the browser blocks the request for security reasons.

To fix this error include the following setting in `web.config` file of the Web API project

```
<system.webServer>  
  <httpProtocol>  
    <customHeaders>  
      <add name="Access-Control-Allow-Origin" value="*" />  
      <add name="Access-Control-Allow-Headers" value="Content-Type" />  
      <add name="Access-Control-Allow-Methods" value="GET, POST, PUT, DELETE, OPTIONS" />
```

```

    </customHeaders>
  </httpProtocol>
</system.webServer>

```

Here is the complete **employeeList.component.html**

```

<employee-count *ngIf="employees" [all]="getTotalEmployeesCount()"
  [male]="getTotalMaleEmployeesCount()"
  [female]="getTotalFemaleEmployeesCount()"
  (countRadioButtonSelectionChanged)=
    "onEmployeeCountRadioButtonChange($event)">
</employee-count>
<br />
<br />
<table>
  <thead>
    <tr>
      <th>Code</th>
      <th>Name</th>
      <th>Gender</th>
      <th>Annual Salary</th>
      <th>Date of Birth</th>
    </tr>
  </thead>
  <tbody>
    <ng-container *ngFor="let employee of employees;">
      <tr *ngIf="selectedEmployeeCountRadioButton=='All' ||
        selectedEmployeeCountRadioButton==employee.gender">
        <td>{{employee.code | uppercase}}</td>
        <td>{{employee.name | employeeTitle:employee.gender }}</td>
        <td>{{employee.gender}}</td>
        <td>{{employee.annualSalary | currency:'USD':true:'1.3-3'}}</td>
        <td>{{employee.dateOfBirth | date:'dd/MM/y'}}</td>
      </tr>
    </ng-container>
    <!--If the web service takes time to return data, the message in this <tr>
    is displayed. When the service call returns this message disappears
    and the employees data is displayed-->
    <tr *ngIf="!employees">
      <td colspan="5">
        Loading data. Please wait...
      </td>
    </tr>
    <!--This message is displayed if the web services does not return any data-->
    <tr *ngIf="employees && employees.length==0">
      <td colspan="5">
        No employee records to display
      </td>
    </tr>
  </tbody>
</table>

```

```
</tr>  
</tbody>  
</table>
```


Assignment 28: Angular 2 http error handling

In this assignment we will discuss **error handling in Angular**.

When using http, to call a web service, errors may occur. When they do occur we want to handle these errors.

1. We use the catch operator to catch any exceptions that occur.
2. Before we use the catch operator we have to import it, just like how we imported map operator.
`import 'rxjs/add/operator/catch';`

3. The catch operator can then be chained to the map operator.
`return this._http.get('http://localhost:24535/api/employeees')
 .map((response: Response) => <IEmployee[]>response.json())
 .catch(this.handleError);`

4. To the catch operator we are passing another method (handleError). This handleError() method is called when there is an exception.

```
handleError(error: Response) {  
  console.error(error);  
  return Observable.throw(error);  
}
```

5. In a real world application we may pass the error to a logging service to log the error to a file or a database table, so the developers can see these errors and fix them if required.
6. In our case, to keep things simple we are logging to the browser console.
7. Since we want the error message color to be red so it stands out, we are using console.error() method instead of console.log() method to log the error to the browser console.
8. After we log the error, we are throwing the error back, so the components that use this service are notified about the error condition, so they can display a meaningful error message to the user.
9. To use throw, we will have to import it from rxjs
`import 'rxjs/add/Observable/throw';`

Here is the complete code in **employee.service.ts**

```
import { Injectable } from '@angular/core';
import { IEmployee } from './employee';
import { Http, Response } from '@angular/http';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/catch';
import 'rxjs/add/Observable/throw';

@Injectable()
export class EmployeeService {

  constructor(private _http: Http) { }

  getEmployees(): Observable<IEmployee[]> {
    return this._http.get('http://localhost:24535/api/employees')
      .map((response: Response) => <IEmployee[]>response.json())
      .catch(this.handleError);
  }

  handleError(error: Response) {
    console.error(error);
    return Observable.throw(error);
  }
}
```

We are calling this service from EmployeeListComponent. So we need to handle the error we have thrown from the service and display a meaningful message to the user. We are subscribing to the service, in ngOnInit() life cycle hook of EmployeeListComponent.

Notice there are 2 parameters to the subscribe() function. Both these parameters are arrow functions. The first arrow function is called when the Observable successfully emits an item. The second arrow function is called, when there is an error.

```
ngOnInit() {
  this._employeeService.getEmployees()
    .subscribe(
      employeesData => this.employees = employeesData,
      error => {
        console.error(error);
        this.statusMessage = 'Problem with the service. Please try again after sometime';
      });
}
```

Here is the complete code in employeeList.component.ts : Notice the code that is relevant to exception handling is commented and self-explanatory

```

import { Component, OnInit } from '@angular/core';
import { IEmployee } from './employee';
import { EmployeeService } from './employee.service';

@Component({
  selector: 'list-employee',
  templateUrl: 'app/employee/employeeList.component.html',
  styleUrls: ['app/employee/employeeList.component.css'],
  providers: [EmployeeService]
})

export class EmployeeListComponent implements OnInit {
  employees: IEmployee[];

  // The view template will bind to this property to display
  // "Loading data. Please wait..." message when the data is
  // being loaded. If there is an error the second arrow
  // function in the subscribe method sets this property to
  // "Problem with the service. Please try again after sometime"
  statusMessage: string = 'Loading data. Please wait...';

  selectedEmployeeCountRadioButton: string = 'All';

  constructor(private _employeeService: EmployeeService) {

  }

  ngOnInit() {
    // The second arrow function sets the statusMessage property
    // to a meaningful message that can be displayed to the user
    this._employeeService.getEmployees()
      .subscribe(
        employeesData => this.employees = employeesData,
        error => {
          this.statusMessage =
            'Problem with the service. Please try again after sometime';
        });
  }

  getTotalEmployeesCount(): number {
    return this.employees.length;
  }

  getTotalMaleEmployeesCount(): number {
    return this.employees
      .filter(e => e.gender === 'Male').length;
  }
}

```

```

getTotalFemaleEmployeesCount(): number {
  return this.employees.filter(e => e.gender === 'Female').length;
}

onEmployeeCountRadioButtonChange(selectedRadioButtonValue: string): void {
  this.selectedEmployeeCountRadioButton = selectedRadioButtonValue;
}
}

```

At this point run the application, and notice since we do not have any errors, that data is loaded as expected. Now let's introduce an error. In `employee.component.ts` file

Change the url from
`http://localhost:24535/api/employees`

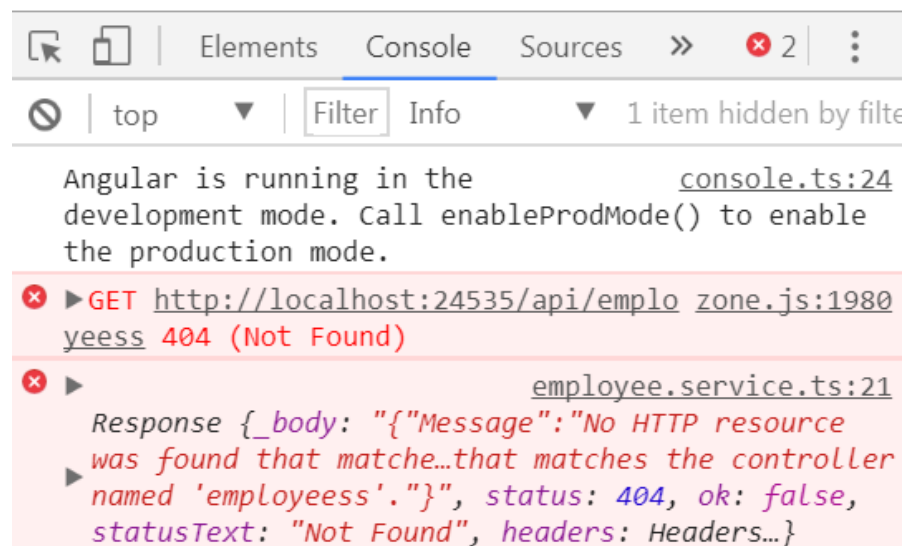
To (Notice the extra "s" at the end)
`http://localhost:24535/api/employeeess`

At this point when you reload the page in the browser, you will see the message "Loading data. Please wait...", but the data never loads.

Code	Name	Gender	Annual Salary	Date of Birth
Loading data. Please wait...				

Now launch browser developer tools and you will see the error message logged to the console by the service.

Code	Name	Gender	Annual Salary	Date of Birth
Loading data. Please wait...				



This message - "Loading data. Please wait..." is misleading in this case. Instead we should be displaying a meaningful message like - "Problem with the service. Please try again after sometime".

To do this in the view template of EmployeeListComponent (employeeList.component.html) bind to statusMessage property of the EmployeeListComponent class as shown below.

```
<tr *ngIf="!employees">
  <td colspan="5">
    {{statusMessage}}
  </td>
</tr>
```

With the above change, while the service is busy retrieving data, we see the message "Loading data. Please wait..." and if there is an error we see the message "Problem with the service. Please try again after sometime"

If you comment the following import statement in employee.service.ts, the exception handling still works as expected.

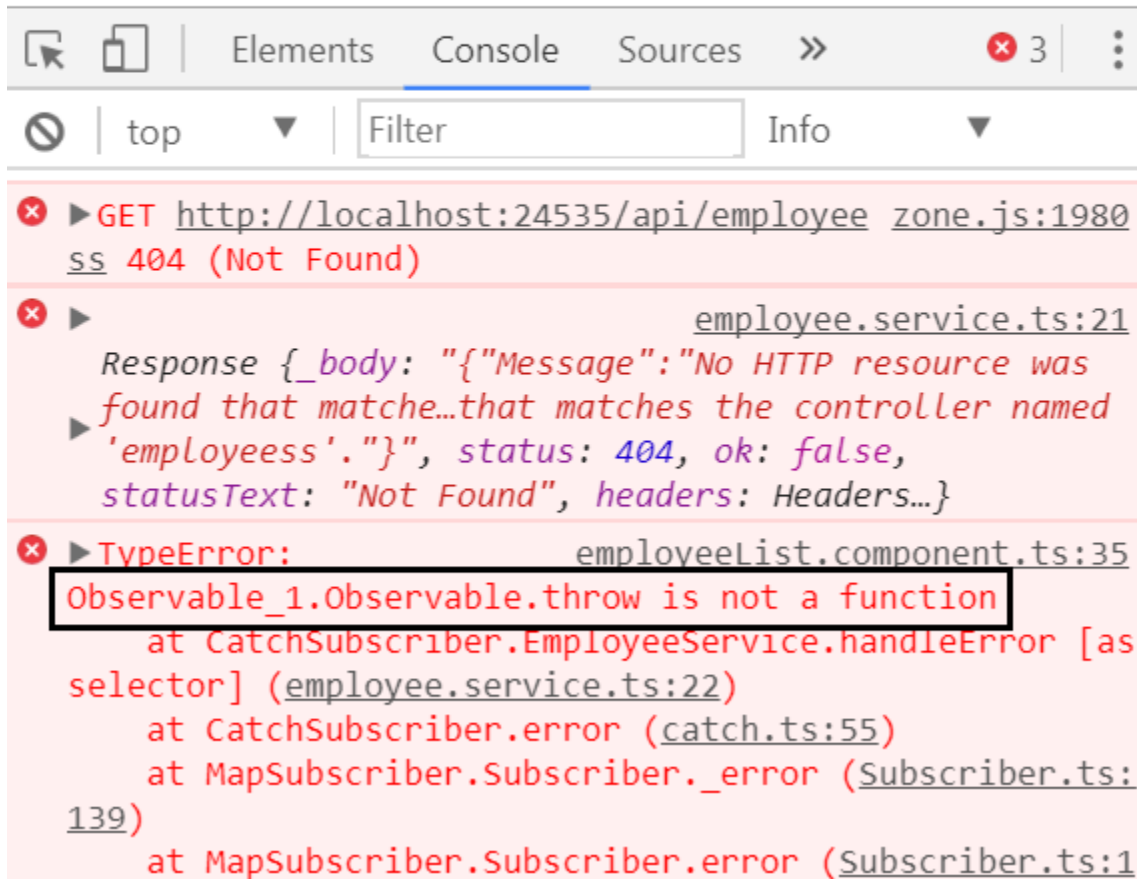
```
import 'rxjs/add/Observable/throw';
```

However, without the above import statement in employee.service.ts file, if you try to log the error object to the console in ngOnInit() method of EmployeeListComponent as shown below, the logging does not work as expected.

```
ngOnInit() {
  this._employeeService.getEmployees()
    .subscribe(
      employeesData => this.employees = employeesData,
      error => {
        this.statusMessage =
          'Problem with the service. Please try again after sometime';
        // Notice here we are logging the error to the browser console
        console.error(error);
      }
    );
}
```

With the above change notice the second error message logged to the console. You will see a message stating - observable_1.observable.throw is not a function, which is not the error message we expected. Angular is complaining that it cannot find throw.

Code	Name	Gender	Annual Salary	Date of Birth
Problem with the service. Please try again after sometime				



If you uncomment the import statement in `employee.service.ts` file, then we see the error message we expect.

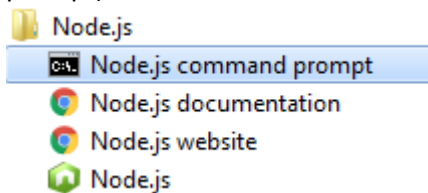
Assignment 29: Using Bootstrap with Angular 2

In this assignment we will discuss

1. How to install and use Bootstrap with Angular
2. How to enable intellisense for bootstrap in Visual Studio

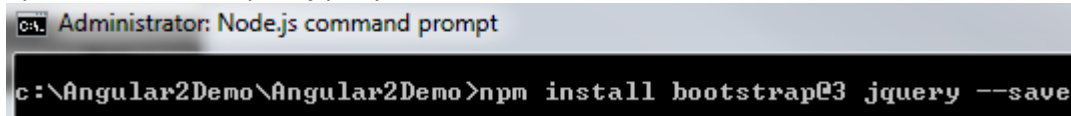
How to install and use Bootstrap with Angular : There are several ways to install Bootstrap. One way is by using Node.js command prompt.

Step 1 : Open node.js command prompt window. (Click windows start button, expand Node.js folder and right click on "node.js command prompt" and select "Run as administrator" from the command prompt)

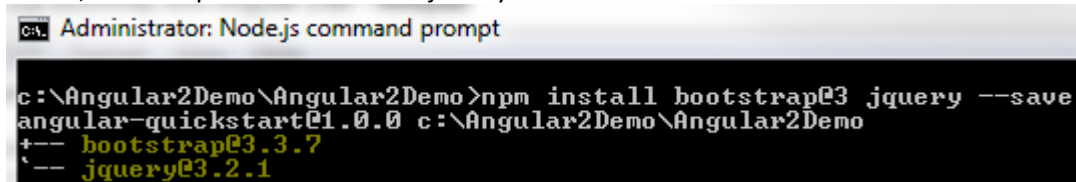


Step 2 : In the command prompt navigate to the folder that contains your angular project and type the following command and press enter key. As Bootstrap has a dependency on jQuery, we are installing jQuery as well.

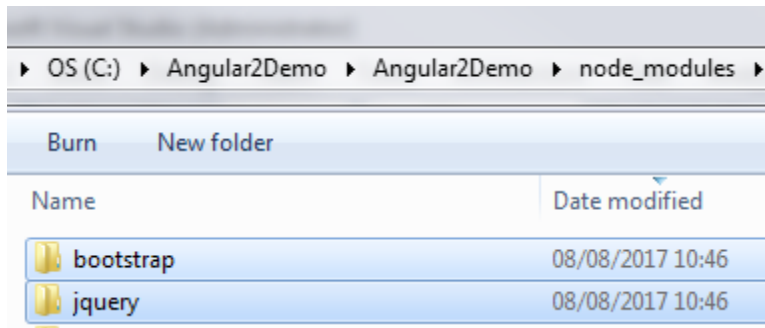
`npm install bootstrap@3 jquery --save`



This installs both Bootstrap and jQuery for use with our Angular project. Notice from the screenshot below, Bootstrap version 3.3.7 and jQuery version 3.2.1 are installed.



The required Bootstrap and jQuery files are copied into the **node_modules** folder within the project directory.



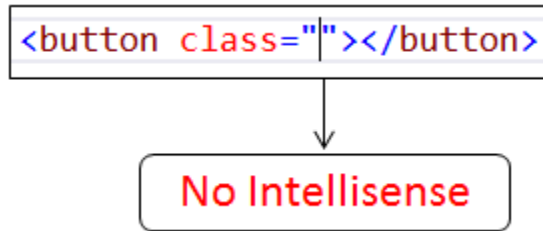
Also, notice package.json file is also automatically updated with both the dependencies (Bootstrap and jQuery)

```
"dependencies": {
  "@angular/common": "~4.0.0",
  "@angular/compiler": "~4.0.0",
  "@angular/core": "~4.0.0",
  "@angular/forms": "~4.0.0",
  "@angular/http": "~4.0.0",
  "@angular/platform-browser": "~4.0.0",
  "@angular/platform-browser-dynamic": "~4.0.0",
  "@angular/router": "~4.0.0",
  "angular-in-memory-web-api": "~0.3.0",
  "bootstrap": "^3.3.7",
  "core-js": "^2.4.1",
  "jquery": "^3.2.1",
  "rxjs": "5.0.1",
  "systemjs": "0.19.40",
  "zone.js": "^0.8.4"
},
```

Step 3 : Finally in index.html file add the following script and link elements to reference the required CSS and JS files. As you can see, these files are being served from the node_modules folder.

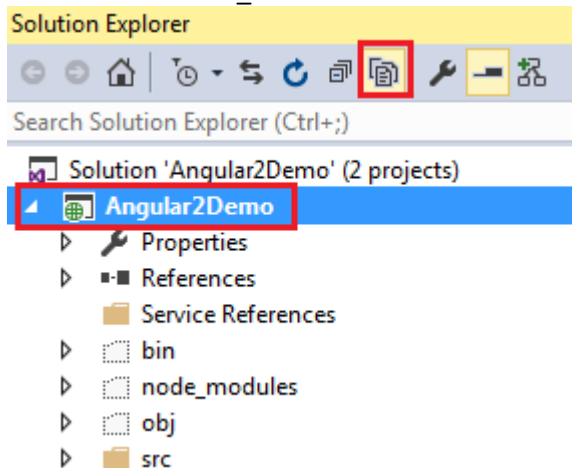
```
<script src="/node_modules/jquery/dist/jquery.min.js"></script>
<link href="/node_modules/bootstrap/dist/css/bootstrap.min.css" rel="stylesheet" />
<script src="/node_modules/bootstrap/dist/js/bootstrap.min.js"></script>
```

The problem at the moment is intellisense for Bootstrap CSS classes is not working in index.html, or any of the components html or stylesheet files.

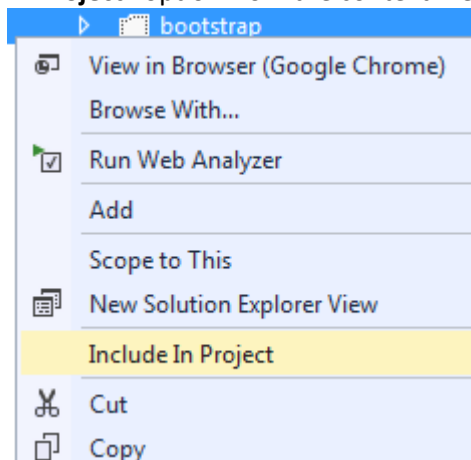


How to enable intellisense for bootstrap in Visual Studio : Use the following workaround to enable intellisense for bootstrap in Visual Studio.

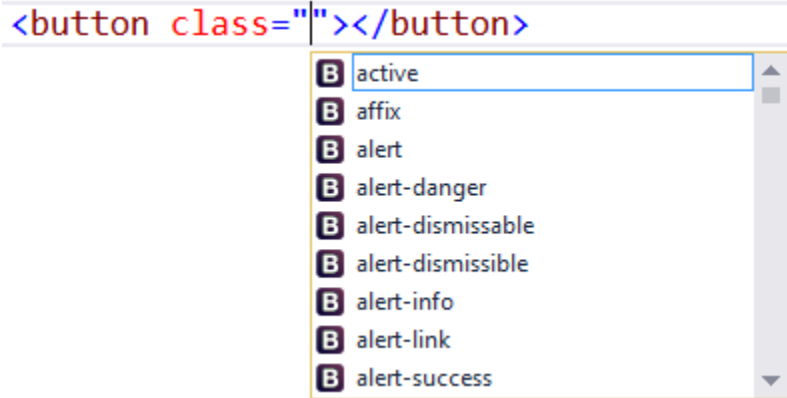
Step 1 : In Visual Studio, click to select the Angular project and then click on "Show All Files" icon. You will then see "node_modules" folder.



Step 2 : Expand "node_modules" folder. Locate "bootstrap" folder. Right click on it and select "Include In Project" option from the context menu.

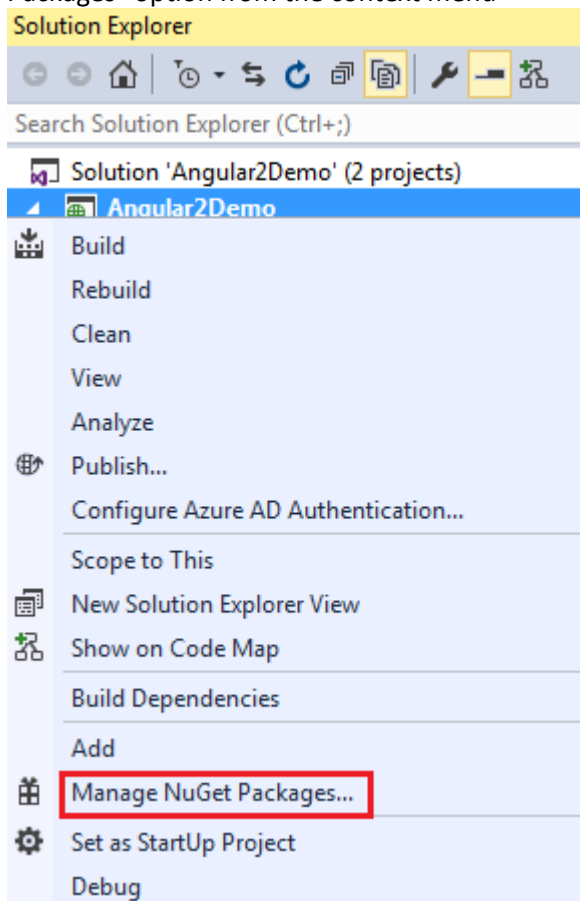


At this point you should get bootstrap intellisense in index.html and all angular component html files. If you still do not get bootstrap intellisense, please restart Visual Studio and you will get intellisense.



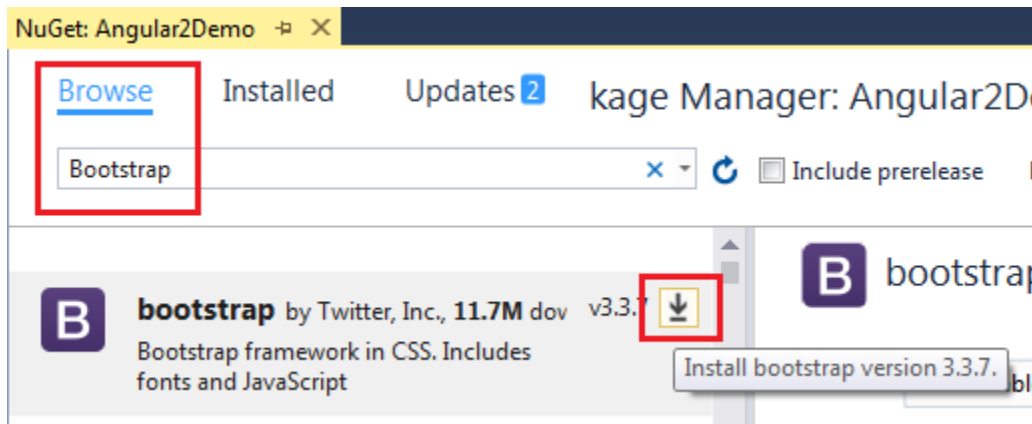
As WE said before, there are several ways to install bootstrap. One way is by using Node.js command prompt. The other way is by using Visual Studio NuGet package manager. To install Bootstrap using NuGet package manager follow these steps.

Step 1 : In Visual Studio Solution Explorer, right click on the Angular project and select "Manage NuGet Packages" option from the context menu

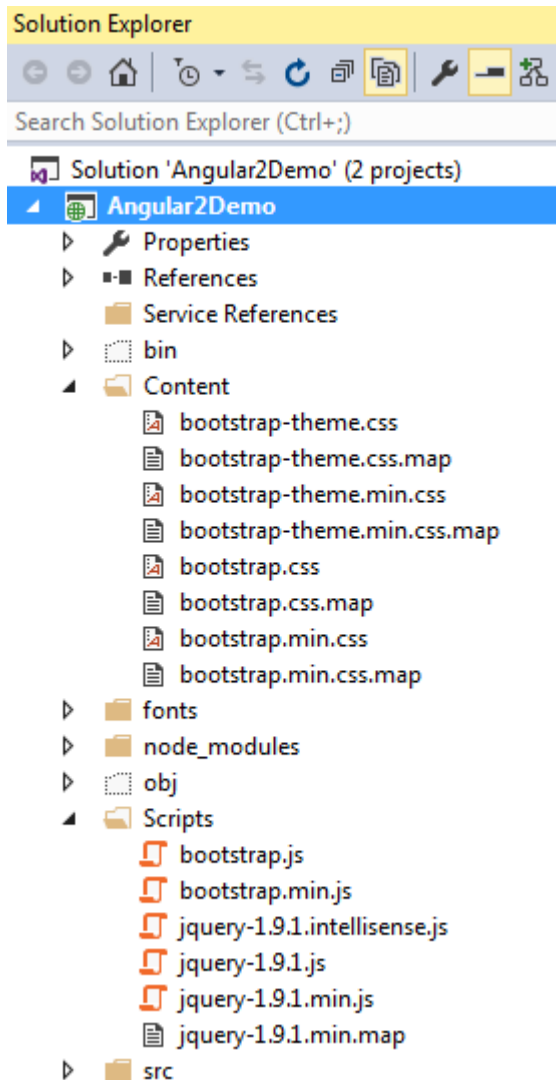


Step 2 : In the "NuGet Package Manager" window

1. Click on the "Browse" link
2. In the textbox, type "Bootstrap" and press "enter" key
3. Click on the first item which says - Bootstrap
4. Click on the arrow pointing downwards to install



The required CSS and JS files are placed in Content and Script folders. Finally place the required CSS and JS files in index.html page to start using Bootstrap. If you install Bootstrap using NuGet you don't have to do anything else to get bootstrap intellisense.



If you do not want to download bootstrap, you can use their CDN links. You can find the official Bootstrap CDN links at the following page.

<http://getbootstrap.com/getting-started/>

Besides these 3 ways, there are other ways of installing Bootstrap. Which way to use depends on your project needs.

Assignment 30: Angular 2 routing tutorial

In this assignment we will discuss the basics of **routing in Angular 2**. Routing allows users to navigate from one view to another view.

At the moment, we have EmployeeListComponent in our application. Let's create another simple HomeComponent so we can see how to navigate from HomeComponent to EmployeeListComponent and vice-versa

Creating HomeComponent

1. Right click on the "app" folder and add a new folder. Name it "home". Right click on the "home" folder and add a new TypeScript file. Name it **home.component.ts**. Copy and paste the following code in it. Notice we have not included the 'selector' property in the @component decorator. The selector is only required if we are going to embed this component inside another component using the selector as a directive. Instead we are going to use the router to navigate to this component.

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

@Component({
  template: '<h1>This is the home page</h1>'
})
export class HomeComponent {
}
```

2. In the application root module (app.module.ts) import HomeComponent and include it in the declarations array of @NgModule decorator.

```
import { HomeComponent } from './home/home.component';

@NgModule({
  imports: [BrowserModule, FormsModule, HttpClientModule],
  declarations: [AppComponent, HomeComponent, ...],
  bootstrap: [AppComponent]
})

export class AppModule { }
```

If the user tries to navigate to a route that does not exist, we want to route the user to PageNotFoundComponent. So let's create this component as well.

Right click on the "others" folder and add a new TypeScript file. Name it pageNotFound.component.ts. Copy and paste the following code in it.

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

```
@Component({
  template: '<h1>The page you are looking for does not exist</h1>'
})
export class PageNotFoundComponent {
}
```

Next, in the application root module (app.module.ts) import PageNotFoundComponent and include it in the declarations array of @NgModule decorator.

```
import { PageNotFoundComponent } from './Others/pageNotFound.component';
```

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

Here are the steps to implement routing in Angular 2 applications.

Step 1 : Set <base href> in the application host page which is index.html. The <base href> tells the angular router how to compose navigation URLs.

```
<base href="/src/">
```

Step 2 : In our angular application root module (app.module.ts), import RouterModule and Routes array and define routes as shown below.

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

// Routes is an array of Route objects
// Each route maps a URL path to a component
// The 3rd route specifies the route to redirect to if the path
// is empty. In our case we are redirecting to /home
// The 4th route (**) is the wildcard route. This route is used
// if the requested URL doesn't match any other routes already defined
const appRoutes: Routes = [
  { path: 'home', component: HomeComponent },
  { path: 'employees', component: EmployeeListComponent },
  { path: '', redirectTo: '/home', pathMatch: 'full' },
  { path: '**', component: PageNotFoundComponent }
];

// To let the router know about the routes defined above,
// pass "appRoutes" constant to forRoot(appRoutes) method
@NgModule({
  imports: [
```

```

    BrowserModule, FormsModule, HttpModule,
    RouterModule.forRoot(appRoutes)
  ],
  declarations: [AppComponent, HomeComponent, ...],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

Important: The order of the routes is very important. When matching routes, Angular router uses first-match wins strategy. So more specific routes should be placed above less specific routes. In the configuration above, routes with a static path are listed first, followed by an empty path route, that matches the default route. The wildcard route comes last because it matches every URL and should be selected only if no other routes are matched first.

Step 3 : Tie the routes to application menu. Modify the root component (app.component.ts) as shown below. The only change we made is in the inline template.

- We are using Bootstrap nav component to create the menu. We discussed Bootstrap nav component in Part 27 of Bootstrap tutorial.
- The routerLink directive tells the router where to navigate when the user clicks the link.
- The routerLinkActive directive is used to add the active bootstrap class to the HTML navigation element whose route matches the active route.
- The router-outlet directive is used to specify the location where we want the routed component's view template to be displayed.
- The routerLink, routerLinkActive and router-outlet directives are provided by the RouterModule which we have imported in our application root module.
- If you forget to specify the router-outlet directive, you will get an error stating - cannot find primary outlet to load component.

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

```

@Component({
  selector: 'my-app',
  template: `
    <div style="padding:5px">
      <ul class="nav nav-tabs">
        <li routerLinkActive="active">
          <a routerLink="home">Home</a>
        </li>
        <li routerLinkActive="active">
          <a routerLink="employees">Employees</a>
        </li>
      </ul>
      <br/>
      <router-outlet></router-outlet>
    </div>
  `
})

```

```
}}  
export class AppComponent {  
}
```

Step 4 : Finally in web.config file of our angular application include the following url-rewrite rule to tell IIS how to handle routes. The match url, <match url="*" />, will rewrite every request. The URL in <action type="Rewrite" url="/src/" /> should match the base href in index.html.

```
<system.webServer>  
  <rewrite>  
    <rules>  
      <rule name="Angular Routes" stopProcessing="true">  
        <match url="*" />  
        <conditions logicalGrouping="MatchAll">  
          <add input="{REQUEST_FILENAME}" matchType="IsFile" negate="true" />  
          <add input="{REQUEST_FILENAME}" matchType="IsDirectory" negate="true" />  
        </conditions>  
        <action type="Rewrite" url="/src/" />  
      </rule>  
    </rules>  
  </rewrite>  
</system.webServer>
```

If you do not have the above url rewrite rule, when you referesh the page you will 404 page not found error.

To use "hash style" urls instead of HTML5 style url's, you just need to make one change in app.module.ts file. Set useHash property to true and pass it to the forRoot() method as shown below.

```
RouterModule.forRoot(appRoutes, { useHash: true })
```

If you are using "hash style" routes, we don't need the URL rewrite rule in web.config file.

Assignment 31: Angular 2 route parameters

In this assignment we will discuss **passing parameters to routes in Angular**.

Let us understand this with an example. We want to make Employee Code on Employee List component clickable.

[Home](#) [Employees](#)

Show : ☒ All(6) ☐ Male(4) ☐ Female(2)

Code	Name	Gender	Annual Salary	Date of Birth
EMP101	Mr.Tom	Male	\$5,500.000	25/06/1988
EMP102	Mr.Alex	Male	\$5,700.950	06/09/1982
EMP103	Mr.Mike	Male	\$5,900.000	08/12/1979
EMP104	Miss.Mary	Female	\$6,500.826	14/10/1980
EMP105	Miss.Nancy	Female	\$6,700.826	15/12/1982
EMP106	Mr.Steve	Male	\$7,700.481	18/11/1979

When we click on an **Employee code**, we want to redirect the user to Employee Component which displays that specific employee details. In the URL we will pass the employee code as a parameter. So clicking on **EMP101** will redirect the user to URL (<http://localhost/employees/emp101>). The Employee component will then read the parameter value from the URL and retrieves that specific employee details by calling the server side web service.

[Home](#) [Employees](#)

Employee Details	
Employee Code	emp101
Name	Tom
Gender	Male
Annual Salary	5500
Date of Birth	6/25/1988

In our previous assignment we have modified the code in **app.module.ts** file to use hash style routing. Let's remove useHash property, so we are using HTML5 style routing instead of hash style routing. Notice from the `forRoot()` method WE have removed useHash property.

```
RouterModule.forRoot(appRoutes)
```

For HTML5 routing to work correctly, uncomment the following URL rewrite rule in web.config file of our Angular application.

```
<system.webServer>
  <rewrite>
    <rules>
      <rule name="Angular Routes" stopProcessing="true">
        <match url="*" />
        <conditions logicalGrouping="MatchAll">
          <add input="{REQUEST_FILENAME}" matchType="IsFile" negate="true" />
          <add input="{REQUEST_FILENAME}" matchType="IsDirectory" negate="true" />
        </conditions>
        <action type="Rewrite" url="/src/" />
      </rule>
    </rules>
  </rewrite>
</system.webServer>
```

In the root application module (`app.module.ts`), include the following route. When the user navigates to a URL like (`http://localhost:12345/employees/EMP101`), we want to display `EmployeeComponent`.

Notice the code parameter specified using colon (:).

```
{ path: 'employees/:code', component: EmployeeComponent }
```

Include the above route in `appRoutes` collection in `app.module.ts` file as shown below. Remember the order of the routes matter.

```
const appRoutes: Routes = [
  { path: 'home', component: HomeComponent },
  { path: 'employees', component: EmployeeListComponent },
  { path: 'employees/:code', component: EmployeeComponent },
  { path: '', redirectTo: '/home', pathMatch: 'full' },
  { path: '**', component: PageNotFoundComponent }
];
```

Next, in `EmployeeListComponent`, modify the `<td>` element that displays employee code to bind it to the route we created above using the `routerLink` directive as shown below.

```
<td>
  <a [routerLink]="['/employees',employee.code]">
    {{employee.code | uppercase}}
  </a>
```

</td>

Explanation of the above code:

- Notice in this example we are binding routerLink directive to an array.
- This array is called link parameters array.
- The first element in the array is the path of the route to the destination component.
- The second element in the array is the route parameter, in our case the employee code.

In the Angular EmployeeService (employee.service.ts), introduce the following `getEmployeeByCode()` method.

```
getEmployeeByCode(empCode: string): Observable<IEmployee> {  
  return this._http.get("http://localhost:31324/api/employees/" + empCode)  
    .map((response: Response) => <IEmployee>response.json())  
    .catch(this.handleError);  
}
```

Explanation of the above code:

- This method takes employee code as a parameter and returns that employee object (IEmployee).
- This method issues a GET request to the Web API service.
- Once the Web API service returns the employee object, this method maps it to IEmployee type and returns it.

In one of our previous assignments we have created EmployeeComponent to display employee details. When we created this component, we have hard coded employee data in the component itself. Now let's modify it

- To retrieve employee details by calling the Angular EmployeeService method **getEmployeeByCode()** we created above.
- This method calls the server side Web API service which retrieves that specific employee details from the database.
- The employee code parameter is in the URL
- To retrieve the parameter from the URL we are using the ActivatedRoute service provided by Angular
- Since ActivatedRoute is provided as a service inject it into the constructor just like how we have injected EmployeeService

employee.component.ts

```
import { Component, OnInit } from '@angular/core';  
import { IEmployee } from './employee';  
import { EmployeeService } from './employee.service';
```

```

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

@Component({
  selector: 'my-employee',
  templateUrl: 'app/employee/employee.component.html',
  styleUrls: ['app/employee/employee.component.css']
})
export class EmployeeComponent implements OnInit {
  employee: IEmployee;
  statusMessage: string = 'Loading data. Please wait...';

  constructor(private _employeeService: EmployeeService,
    private _activatedRoute: ActivatedRoute) { }

  ngOnInit() {
    let empCode: string = this._activatedRoute.snapshot.params['code'];
    this._employeeService.getEmployeeByCode(empCode)
      .subscribe((employeeData) => {
        if (employeeData == null) {
          this.statusMessage =
            'Employee with the specified Employee Code does not exist';
        }
        else {
          this.employee = employeeData;
        }
      },
      (error) => {
        this.statusMessage =
          'Problem with the service. Please try again after sometime';
        console.error(error);
      });
  }
}

```

employee.component.html

```

<table *ngIf="employee">
  <thead>
    <tr>
      <th colspan="2">
        Employee Details
      </th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>Employee Code</td>
      <td>{{employee.code}}</td>
    </tr>
  </tbody>
</table>

```

```

    </tr>
    <tr>
      <td>Name</td>
      <td>{{employee.name}}</td>
    </tr>
    <tr>
      <td>Gender</td>
      <td>{{employee.gender}}</td>
    </tr>
    <tr>
      <td>Annual Salary</td>
      <td>{{employee.annualSalary}}</td>
    </tr>
    <tr>
      <td>Date of Birth</td>
      <td>{{employee.dateOfBirth}}</td>
    </tr>
  </tbody>
</table>
<div *ngIf="!employee">
  {{statusMessage}}
</div>

```

There are different approaches to retrieve route parameters values. We will discuss all the different approaches and when to use what in our upcoming assignments.

Since we need EmployeeService both in EmployeeListComponent and EmployeeComponent, let's register it in the root module so we get a singleton instead of multiple instances of the service.

We will discuss what is a Singleton in Angular and why is it important in our next assignment. For now, let's remove the EmployeeService registration from EmployeeListComponent by removing the following providers property in employeeList.component.ts file

providers: [EmployeeService]

Now register the EmployeeService in application root module (app.module.ts) by including it in the providers property of @NgModule decorator as shown below

```

@NgModule({
  imports: [
    BrowserModule,
    OtherSystemModules..
  ],
  declarations: [
    AppComponent,
    OtherComponents..
  ],

```

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

Assignment 32: Angular dependency injection

In this assignment we will discuss

1. What is Dependency Injection
2. How dependency injection works in angular

Let us understand Dependency Injection in Angular with an example. Consider this piece of code in EmployeeListComponent.

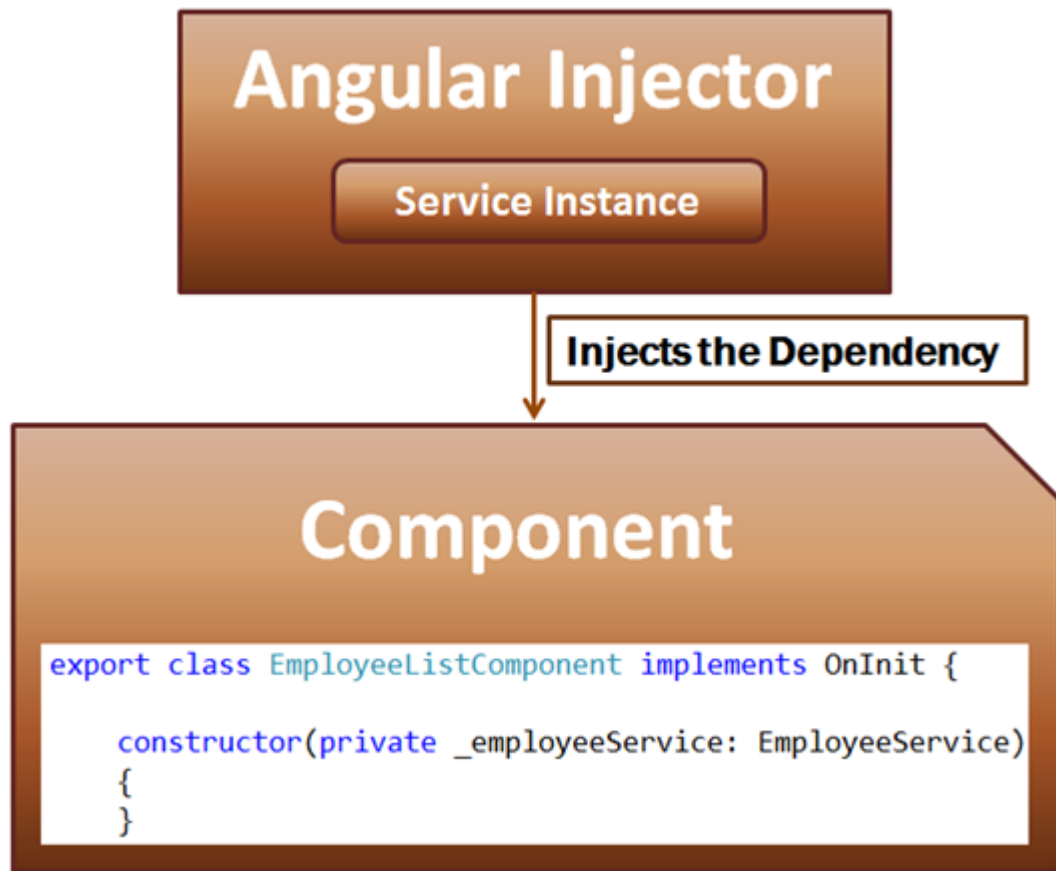
```
export class EmployeeListComponent implements OnInit {  
  
    private _employeeService: EmployeeService;  
  
    constructor(_employeeService: EmployeeService) {  
        this._employeeService = _employeeService;  
    }  
  
    ngOnInit() {  
        this._employeeService.getEmployees()  
            .subscribe(  
                employeesData => this.employees = employeesData,  
                error => this.statusMessage = 'Error');  
    }  
  
    // Rest of the code  
}
```

1. In EmployeeListComponent we need an instance of EmployeeService, so we could use that instance to call getEmployees() method of the service which retrieves the list of employees which this component needs.
2. But if you look at this code, the EmployeeListComponent is not creating an instance of EmployeeService.
3. We declared a private field _employeeService of type EmployeeService. The constructor also has a parameter _employeeService of type EmployeeService. The constructor is then initializing the private class field _employeeService with its parameter.
4. We are then using this private field _employeeService to call the service method getEmployees()
5. The obvious question that we get at this point is how are we getting an instance of the EmployeeService class.
6. We know for sure the EmployeeListComponent is not creating the instance of the EmployeeService class.
7. From looking at the code we can understand that the constructor is provided with an instance of EmployeeService class, and then the constructor is assigning that instance to the private field _employeeService.
8. At this point, the next question that comes to our mind is who is creating and providing the instance to the constructor.

9. The answer to this question is the Angular Injector. When an instance of EmployeeListComponent is created, the angular injector creates an instance of the EmployeeService class and provides it to the EmployeeListComponent constructor. The constructor then assigns that instance to the private field _employeeService. We then use this private field _employeeService to call the EmployeeService method getEmployees().
10. Another question that comes to our mind is - How does the angular injector know about EmployeeService.
11. For the Angular injector to be able to create and provide an instance of EmployeeService, we will have to first register the EmployeeService with the Angular Injector. We register a service with the angular injector by using the providers property of @Component decorator or @NgModule decorator.
12. We already know we decorate an angular component with @Component decorator and an angular module with @NgModule decorator. So this means if we are registering a service using the providers property of the @Component decorator then we are registering the service with an angular injector at the component level. The service is then available to that component and all of its children.
13. On the other hand if we register the service using the providers property of the @NgModule decorator then we are registering the service with an angular injector at the module level which is the root injector. The service registered with the root injector is then available to all the components across the entire application.
14. We will discuss these hierarchical injectors in Angular in detail with an example in our upcoming assignments.

Now let's quickly recap what we have discussed so far

- We register a service with the angular injector by using the providers property of @Component decorator or @NgModule decorator.
- When a component in Angular needs a service instance, it does not explicitly create it. Instead it just specifies it has a dependency on a service and needs an instance of it by including the service as a constructor parameter.
- When an instance of the component is created, the angular injector creates an instance of the service class and provides it to component constructor.
- So the component which is dependent on a service instance, receives the instance from an external source rather than creating it itself. This is called Dependency Injection.



What is Dependency Injection

It's a coding pattern in which a class receives its dependencies from an external source rather than creating them itself.

So if we relate this definition to our example, `EmployeeListComponent` has a dependency on `EmployeeService`. The `EmployeeListComponent` receives the dependency instance (i.e `EmployeeService` instance) from the external source (i.e the angular injector) rather than creating the instance itself.

1. Why should we use Dependency Injection?
2. What benefits it provide?
3. Why can't we explicitly create an instance of the `EmployeeService` class using the `new` keyword and use that instance instead in our `EmployeeListComponent`?

We will answer these questions in our next assignment.

Assignment 33: Why dependency injection

In this assignment we will discuss **why should we use dependency injection and the benefits it provide**. Let us understand this with a very simple example.

Let us say we want to build a Computer. In reality to build a computer we need several objects like processor, ram, hard-disk drive etc. To keep this example simple let's say we just need a processor object to build a computer.

Our Computer and Processor classes are as shown below. Notice at the moment, we are not using dependency injection. To build a Computer we need a Processor object and the Computer class is creating an instance of the Processor class it needs. This is the kind of programming style that most of us are used to and it is easy to understand as well. But there are 3 fundamental problems with this code

1. This code is difficult to maintain over time
2. Instances of dependencies created by a class that needs those dependencies are local to the class and cannot share data and logic.
3. Hard to unit test

```
export class Computer {  
  
  private processor: Processor;  
  
  constructor() {  
    this.processor = new Processor();  
  }  
}  
  
export class Processor {  
  
  constructor() {  
  }  
  
}
```

Now let us understand why this code is difficult to maintain. Let us say, the Processor class needs to know the speed of the processor to be able to create an instance of it. One way to address this requirement is by passing the processor speed as a parameter to the constructor of the Processor class as shown below.

```
export class Processor {  
  
  constructor(speed: number) {  
  }  
  
}
```

This change in the Processor class breaks the Computer class. So every time the Processor class changes, the Computer class also needs to be changed. At the moment, the Computer class has only one dependency. In reality it may have many dependencies and those dependencies in turn may have other dependencies. So when any of these dependencies change, the Computer class may also need to be changed. Hence this code is difficult to maintain.

The reason we have this problem is because the Computer class itself is creating the instance of the Processor class. Instead if an external source can create the processor instance and provide it to the computer class, then this problem can be very easily solved and that's exactly what dependency injection does. We have rewritten the above code using dependency injection, DI for short as shown below.

```
export class Computer {  
  
  private processor: Processor;  
  
  constructor(processor: Processor) {  
    this.processor = processor;  
  }  
}  
  
export class Processor {  
  
  constructor(speed: number) {  
  }  
  
}
```

Notice with DI, the Computer class is not creating the instance of the Processor class itself. Instead we have specified that the Computer class has a dependency on Processor class using the constructor. Now, when we create an instance of the Computer class, an external source i.e the Angular Injector will provide the instance of the Processor class to the Computer class. Since now the the Angular injector is creating the dependency instance, the Computer class need not change when the Processor class changes.

Now, let us understand the second problem - Instances of dependencies created by a class that needs those dependencies are local to the class and cannot share data and logic. The Processor class instance created in the Computer class is local to the Computer class and cannot be shared. Sharing a processor instance does not make that much sense, so let's understand this with another example.

Let us say we have a service called UserPreferencesService which keeps track of the user preferences like colour, font-size etc. We want this data to be shared with all the other components in our application. Now if we create an instance of this UserPreferenceService class in every component class like we did in the Computerclass, the service instance is local to the component in which we have created it and the data cannot be shared with other components. So if we need this UserPreferencesService in 10 different components, we end up creating 10 instances of the service, one for each component. As the service instance is local to the component that has created it, the data

that local service instance has cannot be shared by other components. If this does not make sense at the moment, please do not worry, we will discuss it with a working example in our next assignment.

On the other hand if we use Dependency Injection (DI), the angular injector provides a Singleton i.e a single instance of the service so the data and logic can be shared very easily across all the components.

From unit testing standpoint, it is difficult to mock the processor object, so unit testing Computer class can get complex. In this example, the Computer class has just one dependency (i.e the dependency on the Processor object).

In a real world application, a given object may have a dependency on several other objects, and those dependencies in turn may have dependencies on other objects. Just imagine, how complicated unit testing can become with all these hierarchies of dependencies if we do not have the ability to mock the dependencies.

With Dependency Injection it is very easy to mock objects when unit testing. This is one of the greatest benefits of DI.

If you are new to unit testing and mocking, it may be difficult for you to understand why unit testing can get difficult and complicated if we do not have the ability to mock dependencies. In our upcoming assignments we will discuss unit testing and mocking and it should be much clear at that point.

So in summary DI provides these benefits

1. Create applications that are easy to write and maintain over time as the application evolves
2. Easy to share data and functionality as the angular injector provides a Singleton i.e a single instance of the service
3. Easy to write and maintain unit tests as the dependencies can be mocked

Assignment 34: Angular singleton service

In our previous assignment we discussed why should we use dependency injection and the benefits it provides. One of the benefits of dependency injection is that it allows us to share data and functionality easily as the angular injector provides a **Singleton** i.e a single instance of the service.

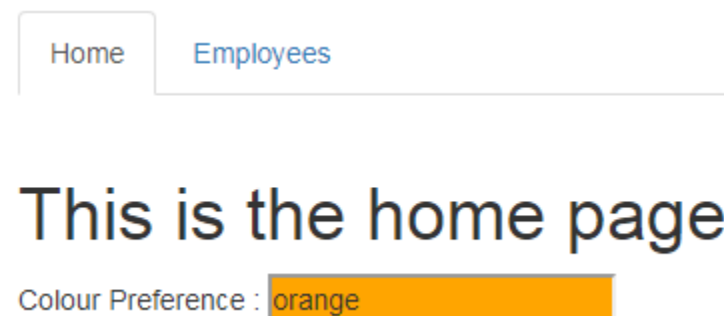
So in this assignments let us see how we can use Angular services and dependency injection to create a **Singleton** i.e a single instance of the service which enables us to share data and functionality across multiple components in our application.

Create a simple UserPreferencesService. Add a new file to the "employee" folder. Name it "userPreferences.service.ts". Copy and paste the following code

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

@Injectable()
export class UserPreferencesService {
  colourPreference: string = 'orange';
}
```

Notice this service has a single property "colourPreference" which is defaulted to "orange". We want to retrieve and use this colour in both "HomeComponent" and "EmployeeListComponent". Notice the textbox displays the default colour (orange) and background is also set to orange. Also both these components should have the ability to set the "colourPreference" property of the service to a different value using the textbox.



[Home](#)[Employees](#)Show : ☒ All(6) ☐ Male(4) ☐ Female(2)

Code	Name	Gender	Annual Salary	Date of Birth
EMP101	Mr.Tom	Male	\$5,500.000	25/06/1988
EMP102	Mr.Alex	Male	\$5,700.950	06/09/1982
EMP103	Mr.Mike	Male	\$5,900.000	08/12/1979
EMP104	Miss.Mary	Female	\$6,500.826	14/10/1980
EMP105	Miss.Nancy	Female	\$6,700.826	15/12/1982
EMP106	Mr.Steve	Male	\$7,700.481	18/11/1979

Colour Preference :

Modify the code in home.component.ts file as shown below. The code is commented and self-explanatory.

```
import { Component } from '@angular/core';
import { UserPreferencesService } from '../employee/userPreferences.service';

// Notice the colour property is bound to the textbox using angular two-way
// databinding. We are also using style binding to set the background colour
// of the textbox
@Component({
  template: `
    <h1>This is the home page</h1>
    <div>
      Colour Preference :
      <input type='text' [(ngModel)]='colour' [style.background]='colour' />
    </div>`
})
export class HomeComponent {

  // Create a private variable to hold an instance of the UserPreferencesService
  private _userPreferencesService: UserPreferencesService;

  // In the constructor we are creating an instance of the UserPreferencesService
  // using the new keyword. So this instance is local to this component and we
```

```

// cannot use it share data with other components. Later we will modify this
// code to use dependency injection, which creates a Singleton so the colour
// data can be shared with other components.
constructor() {
  this._userPreferencesService = new UserPreferencesService();
}

// Implement a getter to retrieve the colourPreference value
// from the service
get colour(): string {
  return this._userPreferencesService.colourPreference;
}

// Implement a setter to change the colourPreference value
// of the service
set colour(value: string) {
  this._userPreferencesService.colourPreference = value;
}
}

```

Now we need to make similar changes in "employeeList.component.html" and "employeeList.component.ts" files. First make the following change in "employeeList.component.html"

Include a <div> element after the table in the file. These are the similar changes we made in the inline view template of HomeComponent.

```

<div>
  Colour Preference :
  <input type="text" [(ngModel)]="colour" [style.background]="colour" />
</div>

```

Now make the following changes in "employeeList.component.ts"

1. Introduce a private field to hold an instance of UserPreferencesService
2. In the constructor create a new instance of UserPreferencesService. Again this instance is local to the EmployeeListComponent and cannot be used to share colour data with the other components. We will discuss how to solve this issue shortly using dependency injection.
3. Finally, introduce a getter and a setter to get and set "colourPreference" property of the UserPreferencesService

```

import { UserPreferencesService } from './userPreferences.service';

private _userPreferencesService: UserPreferencesService;

constructor(private _employeeService: EmployeeService) {
  this._userPreferencesService = new UserPreferencesService();
}

```

```
get colour(): string {  
    return this._userPreferencesService.colourPreference;  
}
```

```
set colour(value: string) {  
    this._userPreferencesService.colourPreference = value;  
}
```

Now run the application and notice that on both "HomeComponent" and "EmployeeListComponent" you will see the service colourPreference property default value "orange" displayed in the textbox. The background colour of the textbox is also set to "orange" as expected.

It looks like the "colourPreference" property of the service is being shared by both the components. But that is not true. Since we are using the new keyword to create an instance of the UserPreferencesService, the instance created in each component is local to that component. Let's prove this.

Introduce a constructor in UserPreferencesService. Notice in the constructor we are logging a message to the console stating that a "New Instance of Service Created". Here is the code in "userPreferences.service.ts"

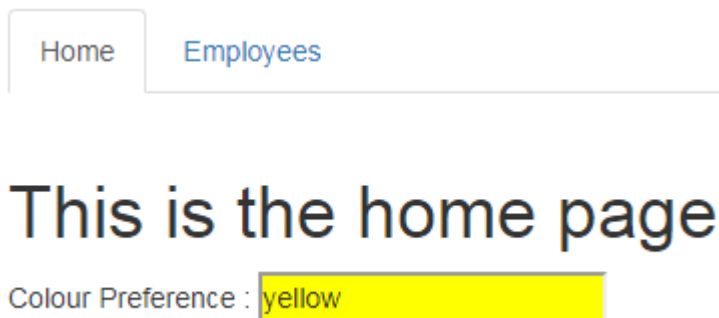
```
import { Injectable } from '@angular/core';  
  
@Injectable()  
export class UserPreferencesService {  
  
    constructor() {  
        console.log('New Instance of Service Created');  
    }  
  
    colourPreference: string = 'orange';  
}
```

Run the application again and launch browser developer tools and click on the Console tab. Notice in the Console you will see the message - "New Instance of Service Created"

Now click on "Employees" menu. This will take you to EmployeeListComponent. Take another look in the "Console" tab, you will see "New Instance of Service Created" logged second time. So this proves each component is creating an instance that is local to that component and sharing data using these local instances is not possible.



Now let us see what happens when each of the components change the "colourPreference" property value of the UserPreferencesService. Navigate to the "HomeComponent" and type "yellow" in the textbox. Notice the background colour of the textbox is changed to yellow as expected.



Now navigate to "EmployeeListComponent" by clicking on the "Employees" tab. Notice the "colourPreference" property value is still orange. This is because when we navigate to EmployeeListComponent it has created a new local instance of UserPreferencesService and the default value "orange" is being used.

[Home](#)[Employees](#)Show : ☒ All(6) ☐ Male(4) ☐ Female(2)

Code	Name	Gender	Annual Salary	Date of Birth
EMP101	Mr.Tom	Male	\$5,500.000	25/06/1988
EMP102	Mr.Alex	Male	\$5,700.950	06/09/1982
EMP103	Mr.Mike	Male	\$5,900.000	08/12/1979
EMP104	Miss.Mary	Female	\$6,500.826	14/10/1980
EMP105	Miss.Nancy	Female	\$6,700.826	15/12/1982
EMP106	Mr.Steve	Male	\$7,700.481	18/11/1979

Colour Preference : orange

At this point navigate back to the HomeComponent, notice the "colourPreference" property value on the HomeComponent is also "orange" now. This is because when we navigated away from the HomeComponent the previous local instance of the UserPreferencesService it has created is destroyed and when came back to the HomeComponent it created another new local instance of the UserPreferencesService and we got it's default colourPreference property value which is orange.

[Home](#)[Employees](#)

This is the home page

Colour Preference : orange

So bottom line, because each component is creating a local instance of the UserPreferencesService we are not able to share data i.e we are not able to see the changes made by one component in the other component.

Now let us see **how to create a Singleton** i.e a single instance of the UserPreferencesService and use that single instance to share data (colourPreference property value) between the 2 components (HomeComponent & EmployeeListComponent) using dependency injection.

Step 1 : Register the service using the Providers property of @NgModule() decorator in app.module.ts file

```
import { UserPreferencesService } from './employee/userPreferences.service';
```

```
@NgModule({  
  providers: [UserPreferencesService]  
})  
export class AppModule { }
```

Step 2 : Specify the dependency on UserPreferencesService using the constructor of HomeComponent & EmployeeListComponent.

Modify the HomeComponent class constructor as shown below. Notice now we are using dependency injection, instead of creating a local instance of the UserPreferencesService using the new keyword.

```
export class HomeComponent {  
  
  constructor(private _userPreferencesService: UserPreferencesService) {  
  }  
}
```

Along the samelines, modify EmployeeListComponent class as well.

```
export class EmployeeListComponent implements OnInit {  
  // Other code  
  
  constructor(private _employeeService: EmployeeService,  
    private _userPreferencesService: UserPreferencesService)  
  { }  
  
  // Other code  
}
```

With all these changes in place, run the application one more time and navigate to HomeComponent. Notice we have the default "colourPreference" property value orange. At this point change the colour in the textbox to yellow. Notice the background colour changes to yellow as expected.

[Home](#)[Employees](#)

This is the home page

Colour Preference : yellow

Now navigate to the EmployeeListComponent. Notice we see the yellow colour which the HomeComponent has set. Also notice in the "Console" table of the browser developer tools, this message (New Instance of Service Created) is logged only once when we navigate between HomeComponent and EmployeeListComponent which proves that only a single instance of the service (ie. a singleton) is created. This singleton enables data sharing between both the components. That's how we are able to see the changes made by one component in the other component.

[Home](#)[Employees](#)Show : ☒ All(6) ☐ Male(4) ☐ Female(2)

Code	Name	Gender	Annual Salary	Date of Birth
EMP101	Mr.Tom	Male	\$5,500.000	25/06/1988
EMP102	Mr.Alex	Male	\$5,700.950	06/09/1982
EMP103	Mr.Mike	Male	\$5,900.000	08/12/1979
EMP104	Miss.Mary	Female	\$6,500.826	14/10/1980
EMP105	Miss.Nancy	Female	\$6,700.826	15/12/1982
EMP106	Mr.Steve	Male	\$7,700.481	18/11/1979

Colour Preference : yellow

So bottom line, with dependency injection it is easy to share data and functionality as the angular injector provides a Singleton i.e a single instance of the service.

Assignment 35: Angular Injector

In this assignment we will discuss **Angular Injectors**.

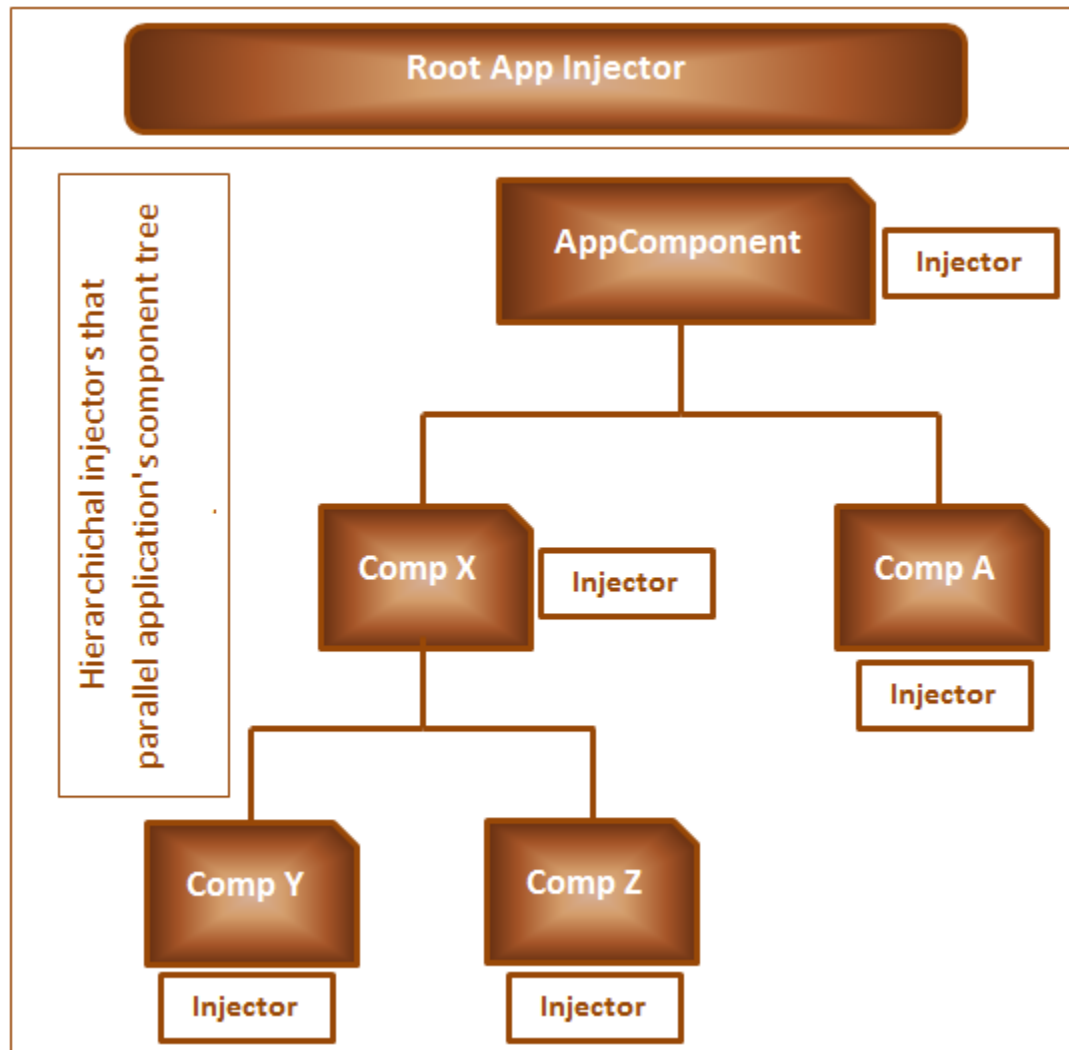
There are 2 simple steps to use dependency injection in Angular. For example, if you have a component that depends on a service and needs an instance of it,

Step 1 : First register the service with the angular injector

Step 2 : Specify a dependency using the constructor of your component class.

With these 2 steps in place, the angular injector will automatically inject an instance of the the service into the component's constructor when an instance of the component is created.

Angular 1 has only one global injector but in Angular 2 we have one injector at the application root level and hierarchical injectors that parallel an application's component tree. This means in Angular 2, we have one root injector plus an injector at every component level as you can see from the diagram below.



If a service is registered with the root injector then that service is available for all components in our entire application including the components in lazy loaded modules. We will discuss lazy loaded modules in a later assignment in this series.

If a service is registered with the injector at the Application Root component level, then that service is available for all the components in the application except the components in lazy loaded modules.

If a service is registered with a specific component injector, then that service is available for that component and any of its children. For example, if we register a service with Component X injector, then that service is available for Components X, Y & Z but not for Component A. Similarly if we register a service with Component A injector, then that service is available only for Component A but not for Components X, Y & Z.

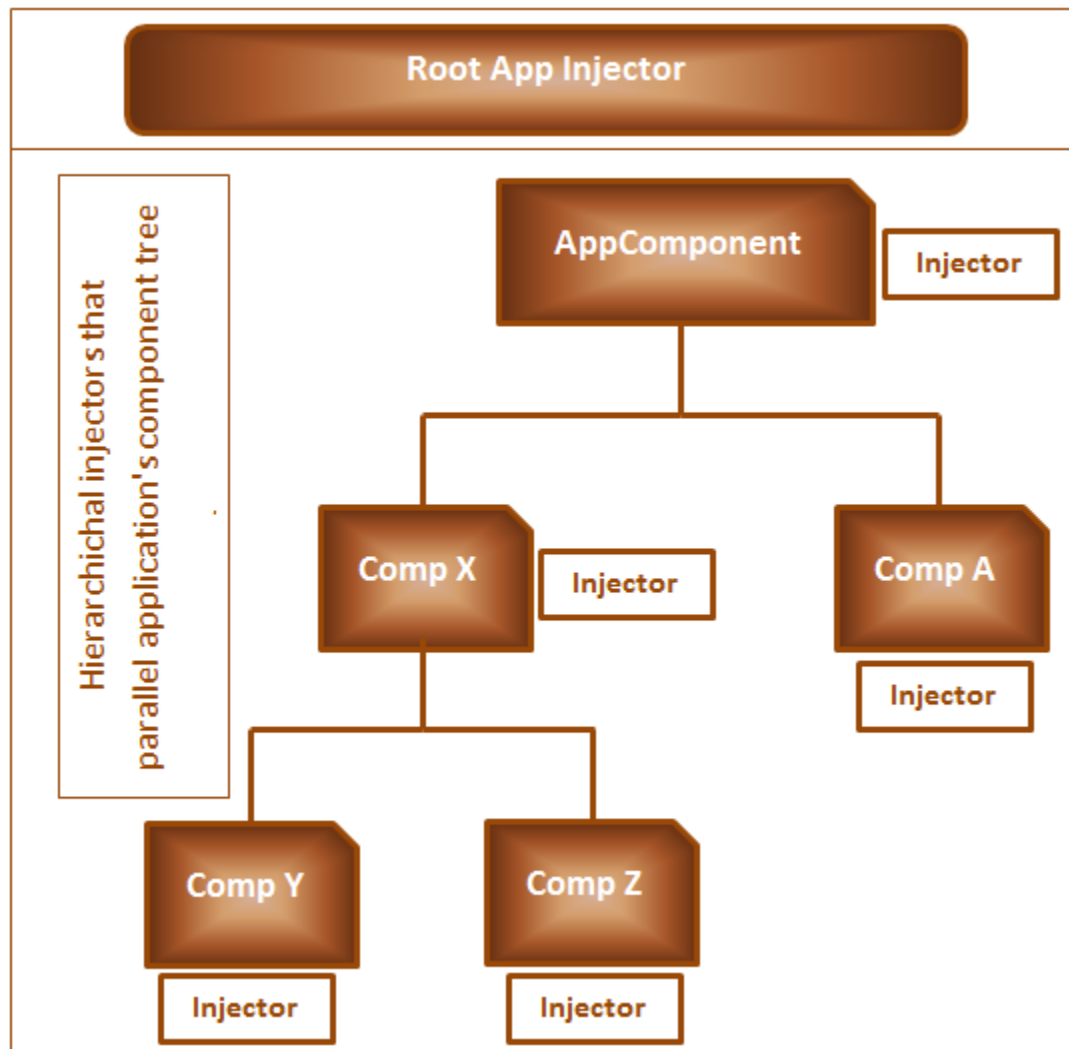
We can register a service with the angular injector using the providers property of `@Component` decorator or `@NgModule` decorator.

To register a service with the root injector use providers property of @NgModule decorator of root module (app.module.ts) or any feature module

To register a service with the injector at a component level use providers property of @Component decorator .

Assignment 36: Angular root injector

In this assignment we will discuss the **Angular root injector**. In our previous assignment we discussed that, in Angular 2 we have one root injector at the application level plus an injector at every component level as you can see from the diagram below.



To register a service with the root injector we use providers property of `@NgModule` decorator and to register a service with the injector at a component level use providers property of `@Component` decorator.

At the moment in our application we have just one module which is the **application root module**. In a real world application it is very common to have more than one module. For example, if you are building an HR management system in addition to the root application module (`AppModule`) we may

have **feature modules** like Employee module, Department Module, Admin Module, Reporting Module etc.

In our previous assignment we have used the `@NgModule()` decorator of the root module to register our service with the root injector. We can also use any of the feature module's `@NgModule()` decorator to register our service with the root injector. Let's prove this.

At the moment in our application we do not have any feature modules. So let's create a simple TestModule and then use that module's `@NgModule()` decorator to register our service with the root injector.

To create a TestModule

1. Add a new TypeScript file to the "app" folder. Name it "test.module.ts"

2. Copy and paste the following code in it. As you can see from the code below creating a module is very similar to creating a component. To create a component, we first create a class, import `@Component()` decorator and decorate the class with it.

This makes the TypeScript class an angular component. Along the same lines, to create a module, we first create a class, import `@NgModule()` decorator and decorate the class with it. This makes the TypeScript class an angular module. So as you can see, Angular provides consistent set of patterns for creating components, pipes, directives and modules.

Notice, we are registering our `UserPreferencesService` with the root injector using the `@NgModule()` decorator of this TestModule.

```
import { NgModule } from '@angular/core';
import { UserPreferencesService } from './employee/userPreferences.service';
```

```
@NgModule({
  providers: [UserPreferencesService]
})
export class TestModule { }
```

3. Just like how we have imported angular system modules (like `BrowserModule`, `FormsModule`, `HttpModule` etc) in our root module (`app.module.ts`), we need to import the TestModule to be able to use it in our application. So include the required import statement and make the TestModule part of imports array of `@NgModule()` decorator in the root module (i.e `app.module.ts` file).

Finally remove "UserPreferencesService" from the providers property. We have already registered our `UserPreferencesService` with the root injector using the `@NgModule()` decorator of the TestModule.

```
import { TestModule } from './test.module';
```

```
@NgModule({
  imports: [
    BrowserModule,
```

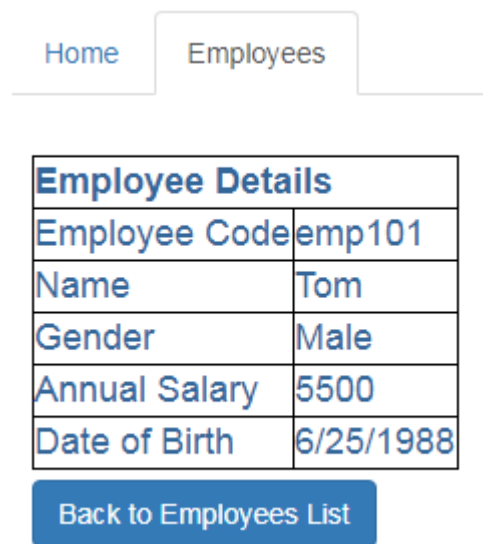
```
    FormsModule,  
    HttpModule,  
    TestModule,  
    RouterModule.forRoot(appRoutes)  
  ],  
  declarations: [  
    AppComponent,  
    EmployeeComponent,  
    EmployeeListComponent,  
    EmployeeTitlePipe,  
    EmployeeCountComponent,  
    SimpleComponent,  
    HomeComponent,  
    PageNotFoundComponent  
  ],  
  bootstrap: [AppComponent],  
  providers: [EmployeeService]  
})  
export class AppModule { }
```

With all these changes run the application and test. It works exactly the same way as before. So this proves that we can use any module provider property to register a service with the root injector. It can be a feature module or the root module.

Assignment 37: Angular router navigate method

In this assignment we will discuss the use of **Angular Router service navigate() method**. This method is useful when you want to navigate to another page programmatically.

Let us understand this with an example. Here is what we want to do. On the **EmployeeComponent** that displays specific employee details, we want to include a Button as shown in the image below. When we click the button we want to redirect the user to **EmployeeListComponent**.



Step 1 : Include the following HTML markup for the button on **employee.component.html**. Notice we are using Bootstrap button css classes (btn & btn-primary) to style the button. We are using event binding to bind click event of the button to bind to "onBackButtonClick()" method in the EmployeeComponent class.

```
<div style="margin-top:5px">
  <input type="button" class="btn btn-primary" value="Back to Employees List"
    (click)="onBackButtonClick()"/>
</div>
```

Step 2 : In the EmployeeComponent class ie. in employee.component.ts file make the following changes

Include the following Import statement to import Router service from '@angular/router'

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

Specify a dependency on the Router service using the EmployeeComponent class constructor. The angular injector will automatically inject an instance of the Router service when an instance of EmployeeComponent is created.

```
constructor(private _router: Router) {  
}
```

Finally include, "onBackButtonClick()" method. Notice we are using the Router service navigate() method to navigate to "/employees" route. This route is already defined in the root module (app.module.ts) which redirects the user to EmployeeListComponent.

```
onBackButtonClick() :void {  
    this._router.navigate(['/employees']);  
}
```

Assignment 38: Promises in angular 2 example

In this assignment we will discuss **using Promises instead of Observables in Angular**.

In Angular we can use either **Promises** or **Observables**. By default the Angular Http service returns an Observable. To prove this, hover the mouse over the `get()` method of the Http service in `employee.service.ts` file. Notice from the intellisense, that it returns `Observable<Response>`

```
getEmployeeByCode(empCode: string): Observable<IEmployee> {  
    return this._http.get("http://localhost:24535/api/employees/" + empCode)  
        .map((response: Response) => <IEmployee>response.json())  
        .catch(this.handleError);  
}
```

An IntelliSense tooltip is shown for the `get()` method of the `Http` service. The tooltip text is: `(method) Http.get(url: string, options?: RequestOptionsArgs): Observable<Response>`. The return type `Observable<Response>` is highlighted with a red box. Below the text, it says "Performs a request with 'get' http method."

We discussed Observables in [Part 27 of Angular 2 tutorial](#). To use Promises instead of Observables we will have to first make a change to the service to return a Promise instead of an Observable.

In `employee.service.ts` file modify `getEmployeeByCode()` method as shown below. The changes are commented so they are self-explanatory

```
import { Injectable } from '@angular/core';  
import { IEmployee } from './employee';  
import { Http, Response } from '@angular/http';  
import { Observable } from 'rxjs/Observable';  
import 'rxjs/add/operator/map';  
import 'rxjs/add/operator/catch';  
import 'rxjs/add/Observable/throw';  
// import toPromise operator  
import 'rxjs/add/operator/toPromise';  
  
@Injectable()  
export class EmployeeService {  
  
    constructor(private _http: Http) { }  
  
    getEmployees(): Observable<IEmployee[]> {  
        return this._http.get('http://localhost:24535/api/employees')  
            .map((response: Response) => <IEmployee[]>response.json())  
            .catch(this.handleError);  
    }  
  
    // Notice we changed the return type of the method to Promise<IEmployee>  
    // from Observable<IEmployee>. We are using toPromise() operator to  
    // return a Promise. When an exception is thrown handlePromiseError()
```

```

// logs the error to the console and throws the exception again
getEmployeeByCode(empCode: string): Promise<IEmployee> {
  return this._http.get("http://localhost:24535/api/employees/" + empCode)
    .map((response: Response) => <IEmployee>response.json())
    .toPromise()
    .catch(this.handlePromiseError);
}

// This method is introduced to handle exceptions
handlePromiseError(error: Response) {
  console.error(error);
  throw (error);
}

handleError(error: Response) {
  console.error(error);
  return Observable.throw(error);
}
}

```

Modify the code in **employee.component.ts** file as shown below. The code that we have changed is commented and is self-explanatory.

```

import { Component, OnInit } from '@angular/core';
import { IEmployee } from './employee';
import { EmployeeService } from './employee.service';
import { ActivatedRoute } from '@angular/router';
import { Router } from '@angular/router';

@Component({
  selector: 'my-employee',
  templateUrl: 'app/employee/employee.component.html',
  styleUrls: ['app/employee/employee.component.css']
})
export class EmployeeComponent implements OnInit {
  employee: IEmployee;
  statusMessage: string = 'Loading data. Please wait...';

  constructor(private _employeeService: EmployeeService,
    private _activatedRoute: ActivatedRoute,
    private _router: Router) { }

  ngOnInit() {
    let empCode: string = this._activatedRoute.snapshot.params['code'];
    // The only change that we need to make here is use
    // then() method instead of subscribe() method
    this._employeeService.getEmployeeByCode(empCode)
      .then((employeeData) => {

```

```

        if (employeeData == null) {
            this.statusMessage =
                'Employee with the specified Employee Code does not exist';
        }
        else {
            this.employee = employeeData;
        }
    },
    (error) => {
        this.statusMessage =
            'Problem with the service. Please try again after sometime';
        console.error(error);
    });
}

onBackButtonClick(): void {
    this._router.navigate(['/employees']);
}
}

```

With the above changes, we are now using a Promise instead of an Observable and the application works the same way as before.

There are several differences between Observables and Promises. We will discuss these differences in our next assignment.

Assignment 39: Angular promises vs observables

In this assignment we will discuss the **differences between promises and observables**. In Angular 2, to work with asynchronous data we can use either Promises or Observables. In our previous assignments in this series, we discussed using both Observables and Promises. There are several differences between Promises and Observables. In this assignment let's discuss these differences.

As a quick summary the differences are shown in the table below

Promise	Observable
Emits a single value	Emits multiple values over a period of time
Not Lazy	Lazy. An Observable is not called until we subscribe to the Observable
Cannot be cancelled	Can be cancelled using the unsubscribe() method
	Observable provides operators like map, forEach, filter, reduce, retry, retryWhen etc.

A Promise emits a single value where as an Observable emits multiple values over a period of time.

You can think of an Observable like a stream which emits multiple items over a period of time and the same callback function is called for each item emitted. So with an Observable we can use the same API to handle asynchronous data whether that data is emitted as a single value or multiple values over a period of time.

A Promise is not lazy where as an Observable is Lazy. Let's prove this with an example. Consider this method `getEmployeeByCode()` in `employee.service.ts`. Notice this method returns an Observable.

```
getEmployeeByCode(empCode: string): Observable<IEmployee> {  
    return this._http.get("http://localhost:31324/api/employees/" + empCode)  
        .map((response: Response) => <IEmployee>response.json())  
        .catch(this.handleError);  
}
```

Here is the consumer code of the above service. In our example, this code is in `employee.component.ts` in `ngOnInit()` method. Notice we are subscribing to the Observable using the `subscribe()` method. An Observable is lazy because, it is not called and hence will not return any data until we subscribe using the `subscribe()` method. At the moment we are using the `subscribe()` method. So the service method should be called and we should receive data.

```
ngOnInit() {  
    let empCode: string = this._activatedRoute.snapshot.params['code'];  
  
    this._employeeService.getEmployeeByCode(empCode)  
        .subscribe((employeeData) => {  
            if (employeeData == null) {  
                this.statusMessage =
```



```
        'Employee with the specified Employee Code does not exist';
    }
    else {
        this.employee = employeeData;
    }
},
(error) => {
    this.statusMessage =
        'Problem with the service. Please try again after sometime';
    console.error(error);
});
}
```

To prove this

1. Launch Browser developer tools by pressing F12 while you are on the browser.
2. Navigate to /src/employees/emp101
3. Click on the **Network Tab**
4. In the Filter textbox, type "api/employees"
5. At this point, you should only see the request issued to EmployeeService in the table below
- 6 Hover the mouse over "emp101" under "Name" column in the table
7. Notice the request to employee service (/api/employees/emp101) is issued

← → ↻ ⓘ localhost:33912/src/employees/emp101

Home Employees

Employee Details	
Employee Code	emp101
Name	Tom
Gender	Male
Annual Salary	5500
Date of Birth	6/25/1988

🔍 📄 | Elements Console Sources **Network**

🔴 🚫 | 📺 🔍 | View: 📄 📊 ☐ Group by frame

api/employees ☐ Regex ☐ Hide data URLs

All XHR JS CSS Img Media Font Doc WS Manifest

500 ms 1000 ms 1500 ms 2000 ms

Name	Method	St...	T...	Initia...	Si...	Ti...	Wate
<input type="checkbox"/> emp101	GET	200	xhr	zone...	6...	4 ...	

http://localhost:24535/api/employees/emp101

Instead of hovering over the request, if you click on it, you can see the response data as shown below. Make sure you select the "preview" tab.

Employee Details	
Employee Code	emp101
Name	Tom
Gender	Male
Annual Salary	5500
Date of Birth	6/25/1988

[Back to Employees List](#)

The screenshot shows the Chrome DevTools Network tab. The filter is set to 'api/employees'. A request to 'api/employees/emp101' is selected. The 'Preview' tab is active, showing the response body as a JSON object:

```
{
  "code": "emp101",
  "name": "Tom",
  "annualSalary": 5500,
  "dateOfBirth": "6/25/1988",
  "gender": "Male"
}
```

Now in **employee.component.ts** file, comment the subscribe() method code block as shown below. Notice we are still calling the `getEmployeeByCode()` method of the `EmployeeService`. Since we have not subscribed to the Observable, a call to the `EmployeeService` will not be issued over the network.

```
ngOnInit() {
  let empCode: string = this._activatedRoute.snapshot.params['code'];

  this._employeeService.getEmployeeByCode(empCode)
    // .subscribe((employeeData) => {
    //   if (employeeData == null) {
```

```

    //    this.statusMessage =
    //        'Employee with the specified Employee Code does not exist';
    // }
    // else {
    //     this.employee = employeeData;
    // }
    //},
    //(error) => {
    //    this.statusMessage =
    //        'Problem with the service. Please try again after sometime';
    //    console.error(error);
    //});
}

```

With the above change in place, reload the web page. Notice no request is issued over the network to the EmployeeService. You can confirm this by looking at the network tab in the browser developer tools. So this proves that an Observable is lazy and won't be called unless we subscribe using the subscribe() method.

Now, let's prove that a Promise is NOT Lazy. We have modified the code in employee.service.ts to return a Promise instead of an Observable. The modified code is shown below.

```

getEmployeeByCode(empCode: string): Promise<IEmployee> {
    return this._http.get("http://localhost:31324/api/employees/" + empCode)
        .map((response: Response) => <IEmployee>response.json())
        .toPromise()
        .catch(this.handlePromiseError);
}

```

Here is the consumer code of the above service. In our example, this code is in employee.component.ts in ngOnInit() method.

```

ngOnInit() {
    let empCode: string = this._activatedRoute.snapshot.params['code'];

    this._employeeService.getEmployeeByCode(empCode)
        .then((employeeData) => {
            if (employeeData == null) {
                this.statusMessage =
                    'Employee with the specified Employee Code does not exist';
            }
            else {
                this.employee = employeeData;
            }
        },
        (error) => {
            this.statusMessage =
                'Problem with the service. Please try again after sometime';
        }
    );
}

```

```
        console.error(error);  
    });  
}
```

Because a promise is eager(not lazy), calling `employeeService.getEmployeeByCode(empCode)` will immediately fire off a request across the network to the `EmployeeService`. We can confirm this by looking at the network tab in the browser tools.

Now you may be thinking, `then()` method in this case is similar to `subscribe()` method. If we comment `then()` method code block, will the service still be called. The answer is YES. Since a Promise is NOT LAZY, Irrespective of whether you have `then()` method or not, calling `employeeService.getEmployeeByCode(empCode)` will immediately fire off a request across the network to the `EmployeeService`. You can prove this by commenting `then()` method code block and reissuing the request.

In our next assignment, we will discuss how to retry when a request fails and in the assignment after that we will discuss how cancel an Observable.

Assignment 40: Observable retry on error

In this assignment we will **how to resubscribe and retry an Observable if there is an error.**

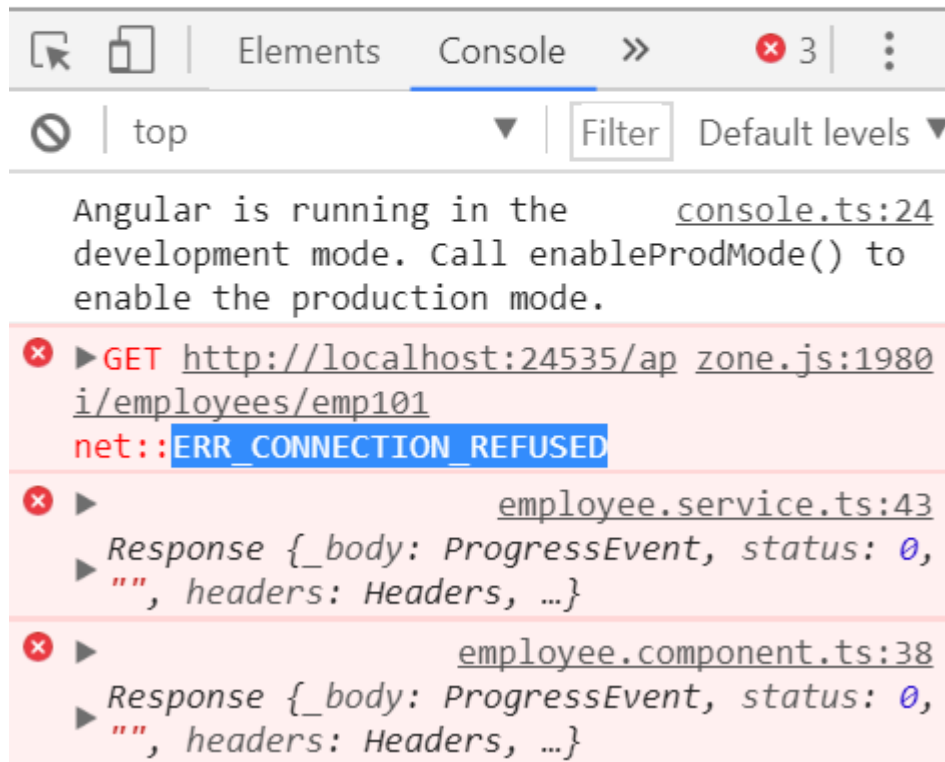
Please change **getEmployeeByCode()** method in employee.service.ts file to return an Observable instead of a Promise as shown below.

```
getEmployeeByCode(empCode: string): Observable<IEmployee> {  
    return this._http.get("http://localhost:24535/api/employees/" + empCode)  
        .map((response: Response) => <IEmployee>response.json())  
        .catch(this.handleError);  
}
```

To cause an error, stop the Web API Service. At this point if you navigate to `http://localhost:12345/src/employees/emp101`, the application display a message stating - Problem with the service. Please try again after sometime. In the "Console" tab of the "Browser Developer Tools" you will see - `ERR_CONNECTION_REFUSED`.

[Home](#)[Employees](#)

Problem with the service. Please try again after sometime

[Back to Employees List](#)

To resubscribe to the Observable and retry, use the rxjs retry operator. The changes in employee.component.ts file are commented and self explanatory.

```
import { Component, OnInit } from '@angular/core';
import { IEmployee } from './employee';
import { EmployeeService } from './employee.service';
import { ActivatedRoute } from '@angular/router';
import { Router } from '@angular/router';
```

```
// Import rxjs retry operator
import 'rxjs/add/operator/retry';
```

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

```

})
export class EmployeeComponent implements OnInit {
  employee: IEmployee;
  statusMessage: string = 'Loading data. Please wait...';
  retryCount: number = 1;

  constructor(private _employeeService: EmployeeService,
    private _activatedRoute: ActivatedRoute,
    private _router: Router) { }

  ngOnInit() {
    let empCode: string = this._activatedRoute.snapshot.params['code'];

    this._employeeService.getEmployeeByCode(empCode)
      // Chain the retry operator to retry on error.
      .retry()
      .subscribe((employeeData) => {
        if (employeeData == null) {
          this.statusMessage =
            'Employee with the specified Employee Code does not exist';
        }
        else {
          this.employee = employeeData;
        }
      },
      (error) => {
        this.statusMessage =
          'Problem with the service. Please try again after sometime';
        console.error(error);
      });
  }

  onBackButtonClick(): void {
    this._router.navigate(['/employees']);
  }
}

```

The downside of this approach is that the application keeps on retrying forever. If we start the Web API service, the call succeeds and the observable completes with employee data displayed on the web page.

Now if your requirement is not to retry forever, but only retry for a specific number of times if there is an error then we can use another variation of retry as shown below. Notice in this case we are passing number 3 to the retry operator indicating that we only want to retry 3 times. After the 3rd attempt the observable completes with an error.


```

ngOnInit() {
  let empCode: string = this._activatedRoute.snapshot.params['code'];

  this._employeeService.getEmployeeByCode(empCode)
    // Retry only 3 times if there is an error
    .retry(3)
    .subscribe((employeeData) => {
      if (employeeData == null) {
        this.statusMessage =
          'Employee with the specified Employee Code does not exist';
      }
      else {
        this.employee = employeeData;
      }
    },
    (error) => {
      this.statusMessage =
        'Problem with the service. Please try again after sometime';
      console.error(error);
    });
}

```

The problem with the retry operator is that, it immediately retries when there is an error. In our case it is a connection issue with the service. Retrying again immediately in our case does not make much sense, as most likely it might fail again. So in situations like this we may want to retry after a short delay, may be after a second or so. This is when we use the retryWhen rxjs operator. retryWhen operator allows us to specify delay in milli-seconds and can be used as shown below. Please donot forget to import retryWhen and delay operators.

```

import 'rxjs/add/operator/retrywhen';
import 'rxjs/add/operator/delay';

ngOnInit() {
  let empCode: string = this._activatedRoute.snapshot.params['code'];

  this._employeeService.getEmployeeByCode(empCode)
    // Retry with a delay of 1000 milliseconds (i.e 1 second)
    .retryWhen((err) => err.delay(1000))
    .subscribe((employeeData) => {
      if (employeeData == null) {
        this.statusMessage =
          'Employee with the specified Employee Code does not exist';
      }
      else {
        this.employee = employeeData;
      }
    },
    (error) => {

```

```

        this.statusMessage =
            'Problem with the service. Please try again after sometime';
        console.error(error);
    });
}

```

Now if you are wondering can't we use delay operator with retry operator, the answer is NO, we can't. The delay operator in the following example will not work as expected. As you can see from the browser console, it immediately retries instead of waiting for 5 seconds before a retry attempt.

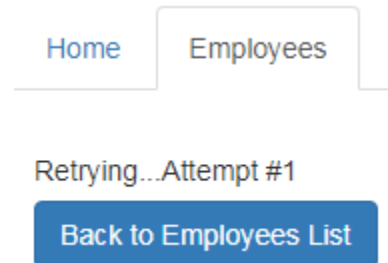
```

ngOnInit() {
    let empCode: string = this._activatedRoute.snapshot.params['code'];

    this._employeeService.getEmployeeByCode(empCode)
        // The delay operator will not work with retry
        .retry().delay(5000)
        .subscribe((employeeData) => {
            if (employeeData == null) {
                this.statusMessage =
                    'Employee with the specified Employee Code does not exist';
            }
            else {
                this.employee = employeeData;
            }
        },
        (error) => {
            this.statusMessage =
                'Problem with the service. Please try again after sometime';
            console.error(error);
        });
}

```

If you want to retry every 1000 milli-seconds only for a maximum of 5 times then we can use rxjs scan operator along with the take operator. While retrying we also want to show the retry attempt number to the user on the web page as shown below.



After all the retry attempts are exhausted, the application should stop retrying and display the error message to the user as shown below.

Problem with the service. Please try again after sometime

[Back to Employees List](#)

If the connection becomes available between the retry attempts, the application should display employee data.

Here is the complete code.

```
import { Component, OnInit } from '@angular/core';
import { IEmployee } from './employee';
import { EmployeeService } from './employee.service';
import { ActivatedRoute } from '@angular/router';
import { Router } from '@angular/router';

import 'rxjs/add/operator/retrywhen';
import 'rxjs/add/operator/delay';
import 'rxjs/add/operator/scan';

@Component({
  selector: 'my-employee',
  templateUrl: 'app/employee/employee.component.html',
  styleUrls: ['app/employee/employee.component.css']
})
export class EmployeeComponent implements OnInit {
  employee: IEmployee;
  statusMessage: string = 'Loading data. Please wait...';
  retryCount: number = 1;

  constructor(private _employeeService: EmployeeService,
    private _activatedRoute: ActivatedRoute,
    private _router: Router) { }

  ngOnInit() {
    let empCode: string = this._activatedRoute.snapshot.params['code'];

    this._employeeService.getEmployeeByCode(empCode)
      // Retry 5 times maximum with a delay of 1 second
      // between each retry attempt
      .retryWhen((err) => {
        return err.scan((retryCount, val) => {
          retryCount += 1;
          if (retryCount < 6) {
```

```

        this.statusMessage = 'Retrying...Attempt #' + retryCount;
        return retryCount;
    }
    else {
        throw (err);
    }
}, 0).delay(1000)
})
.subscribe((employeeData) => {
    if (employeeData == null) {
        this.statusMessage =
            'Employee with the specified Employee Code does not exist';
    }
    else {
        this.employee = employeeData;
    }
},
(error) => {
    this.statusMessage =
        'Problem with the service. Please try again after sometime';
    console.error(error);
});
}

onBackButtonClick(): void {
    this._router.navigate(['/employees']);
}
}

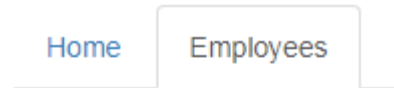
```

In our next assignment, we will discuss how to allow the end user to cancel retry attempts at any point using the unsubscribe method.

Assignment 41: Angular observable unsubscribe

In this assignment, we will discuss **how to cancel an Observable using the unsubscribe method**.

Let us understand this with an example. Here is what we want to do. When the request to the server is in progress we want to display "**Cancel Request**" button.



Retrying...Attempt #1

Back to Employees List

Cancel Request

The "**Cancel Request**" button should only be visible to the user when there is a request in progress. When the request fails permanently after exhausting all the retry attempts the button should disappear.



Problem with the service. Please try again after sometime

Back to Employees List

Along the same lines if the request completes successfully, then also the button should disappear.

[Home](#)[Employees](#)

Employee Details	
Employee Code	emp101
Name	Tom
Gender	Male
Annual Salary	5500
Date of Birth	6/25/1988

[Back to Employees List](#)

When the user cancels the request, by clicking the "Cancel Request" button, the request should be cancelled and button should disappear.

[Home](#)[Employees](#)

Request cancelled

[Back to Employees List](#)

To achieve this, include the following HTML in employee.component.html file. This HTML is for the "Cancel Request" button. Please pay attention to the structural directive **ngIf** on the <div> element that contains the button. As you can see we are showing the "Cancel Request" button only if the subscription is not closed (!subscription.closed), meaning only when there is a request to the Observable is in progress.

The **ngIf** directive will take care of showing and hiding the <div> element depending on whether the subscription is closed or not. We are also binding the click event of the button to "onCancelButtonClick()" method. We will create both the subscription object and the "onCancelButtonClick()" method in the component class.

```
<div style="margin-top:5px" *ngIf="!subscription.closed">
  <input type="button" class="btn btn-primary" value="Cancel Request"
    (click)="onCancelButtonClick()" />
</div>
```

Now modify the code in employee.component.ts file as shown below. The relevant code is commented and self-explanatory.

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

```

import { IEmployee } from './employee';
import { EmployeeService } from './employee.service';
import { ActivatedRoute } from '@angular/router';
import { Router } from '@angular/router';

import 'rxjs/add/operator/retrywhen';
import 'rxjs/add/operator/delay';
import 'rxjs/add/operator/scan';
// import ISubscription.
import { ISubscription } from "rxjs/Subscription";

@Component({
  selector: 'my-employee',
  templateUrl: 'app/employee/employee.component.html',
  styleUrls: ['app/employee/employee.component.css']
})
export class EmployeeComponent implements OnInit {
  employee: IEmployee;
  statusMessage: string = 'Loading data. Please wait...';
  retryCount: number = 1;

  // Create a class property of type ISubscription
  // The ISubscription interface has closed property
  // The ngIf directive in the HTML binds to this property
  // Go to the definition of ISubscription interface to
  // see the closed property
  subscription: ISubscription;

  constructor(private _employeeService: EmployeeService,
    private _activatedRoute: ActivatedRoute,
    private _router: Router) { }

  ngOnInit() {
    let empCode: string = this._activatedRoute.snapshot.params['code'];

    // Use the subscription property created above to hold on to the
    // subscription. We use this object in the onCancelButtonClick()
    // method to unsubscribe and cancel the request
    this.subscription = this._employeeService.getEmployeeByCode(empCode)
      .retryWhen((err) => {
        return err.scan((retryCount, val) => {
          retryCount += 1;
          if (retryCount < 4) {
            this.statusMessage = 'Retrying...Attempt #' + retryCount;
            return retryCount;
          }
          else {
            throw (err);

```

```

        }
    }, 0).delay(1000)
})
.subscribe((employeeData) => {
    if (employeeData == null) {
        this.statusMessage =
            'Employee with the specified Employee Code does not exist';
    }
    else {
        this.employee = employeeData;
    }
},
(error) => {
    this.statusMessage =
        'Problem with the service. Please try again after sometime';
    console.error(error);
});
}

onBackButtonClick(): void {
    this._router.navigate(['/employees']);
}

// This method is bound to the click event of the "Cancel Request" button
// Notice we are using the unsubscribe() method of the subscription object
// to unsubscribe from the observable to cancel the request. We are also
// setting the status message property of the class to "Request Cancelled"
// This message is displayed to the user to indicate that the request is cancelled
onCancelButtonClick(): void {
    this.statusMessage = 'Request cancelled';
    this.subscription.unsubscribe();
}
}

```


Assignment 42: Angular services tutorial

So far Google has released 3 versions of Angular and you can see the timelines below.

Version	Year
AngularJS	2010
Angular 2	2016
Angular 4	2017

In this assignment, let's discuss the differences between these versions.

What is the difference between AngularJS and Angular 2

The first version of Angular is called AngularJS and was released in the year 2010. Some people call it Angular 1, but it is officially called AngularJS.

Angular 2 is released in the year 2016. The most important thing to keep in mind is that, Angular 2 is not a simple upgrade from angular 1.

Angular 2 is completely rewritten from the ground up and as a result the way we write applications with AngularJS and Angular 2 is very different.

AngularJS applications revolve around the concept of controllers. To glue a controller and a view in AngularJS we use `$scope`. With Angular 2 both controllers and `$scope` are gone. Angular 2 is entirely component based, which means we create a set of independent or loosely coupled components and put them together to create an Angular 2 application. So components are the building blocks of an Angular 2 application. The advantage of the component-based approach is that, it facilitates greater code reuse. For example, if you have a rating component, you may use it on an employee page to rate an employee or on a product page to rate a product. From unit testing standpoint, the use of components make Angular2 more testable.

From a performance standpoint, Angular 2 is 5 times faster compared to AngularJS.

AngularJS was not built for mobile devices, where as Angular 2 on the other hand is designed from the ground up with mobile support in mind.

With Angular 2 we have more language choices. In addition to native JavaScript we can use TypeScript, Dart, PureScript, Elm, etc. Among all these languages, TypeScript is the most popular language for developing Angular 2 applications as it provides the following benefits.

1. Intellisense
2. Autocompletion
3. Code navigation
4. Advanced refactoring
5. Strong Typing
6. Supports ES 2015 (also called ES 6) features like classes, interfaces and inheritance.

Angular 4 is released in 2017. So, **What is the difference between Angular 2 and Angular 4**

If you have worked with both Angular 1 and Angular 2, then you already know that the API's and patterns that we use to build applications are very different between these 2 versions. From a developer stand point, it is like learning 2 different frameworks. Since Angular 2 is a complete rewrite from Angular 1, moving from Angular 1 to Angular 2 is a total breaking change.

However, changing from Angular 2 to Angular 4 and even future versions of Angular, won't be like changing from Angular 1. It won't be a complete rewrite, it will simply be a change in some core libraries. From a developer standpoint, *building* an application using Angular 2 and Angular 4 is not very different. We still use the same concepts and patterns. Angular 4 is simply, the next version of Angular 2. The underlying concepts are still the same and if you have already learnt Angular 2 then you're well prepared to switch to Angular 4.

The most important point to keep in mind is, Angular 4 is backwards compatible with Angular 2 for most applications.

What has changed and what is new in Angular 4

Some under the hood changes to reduce the size of the AOT (Ahead-of-Time) compiler generated code. Migrating to Angular 4 may reduce the production bundles by hundreds of kilobytes. As a developer this change will not affect the way we write angular applications in any way.

TypeScript 2.1 and 2.2 compatibility. Angular is updated with a more recent version of TypeScript, for better type checking throughout our application. Up until Angular 4, only TypeScript 1.8 was supported. With Angular 4, we can use typescript 2.1 or 2.2 which means we can use all the new features of TypeScript with Angular 4.

Animation features are pulled out of @angular/core and are moved into their own package. This means that if you don't use animations, this extra code will not end up in your production bundles. On the other hand, if you do have animations in your application, you may have to change your existing code to pull the animation features from the animations package.

We can now use an if/else style syntax with *ngIf structural directive. In Angular 2, to implement if/else logic, we use 2 *ngIf structural directives. With Angular 4, we can use it's new if/else style syntax with *ngIf structural directive. We will discuss an example of this in our upcoming assignments.

What happened to Angular 3. Why did we move straight from Angular 2 to Angular 4. What is the reason for skipping Angular 3.

Except the Router library, all the other Angular core libraries are versioned the same way and are shipped as NPM packages as you can see below. While all the other core angular packages are at Version 2, the router library is already at Version 3.

Library	Version
@angular/core	v2.3.0
@angular/compiler	v2.3.0
@angular/compiler-cli	v2.3.0
@angular/http	v2.3.0
@angular/router	v3.3.0

Due to this misalignment of the router package's version, the angular team decided to go straight for Angular v4. This way, all the core packages are aligned which will be easier to maintain and help avoid confusion in the future.

Some naming guidelines are also provided by the Angular team

1. Use "AngularJS" to describe versions 1.x or earlier
2. Use "Angular" for versions 2.0.0 and later.
3. Use the version numbers "Angular 4.0", "Angular 2.4" when you need to talk about a specific release. Otherwise it is just enough if you use the word Angular. For example, you are an Angular developer and not Angular 2.0 developer or Angular 4.0 developer.
4. Angular 4 is simply, the next version of Angular 2. The underlying concepts are still the same and if you have already learnt Angular 2 then you're well prepared to switch to Angular 4.

Do I have to learn AngularJS 1 before learning Angular 2

No. You can think of AngularJS 1 and Angular 2 as 2 different frameworks. The concepts, the API's and patterns that we use to build applications are very different between these 2 versions. So there is no need to learn AngularJS 1 before you learn Angular 2.

Do I have to learn Angular 2 before learning Angular 4

From a developer standpoint, building an application using Angular 2 and Angular 4 is not very different. We still use the same concepts, APIs and patterns. Angular 4 is simply, the next version of Angular 2 and contains a few changes and enhancements as discussed above. So you really have 2 options here. Learn Angular 2 first and then the changes and enhancements introduced in Angular 4, OR learn all the angular concepts using version 4. Either ways you are an Angular developer.

Assignment 43: CRUD Operations Sample Project

<https://www.youtube.com/watch?v=DAUIZO2RALw&t=856s>

Download the source code from :

Repo for finished project: https://github.com/jakblak/ng4_CRUD

And follow the instructions and explanations as per the Video.