

HSBC 

Infosys
be more



About Angular 6

Angular is one of the most powerful and performance-efficient JavaScript framework to build single page applications for both web and mobile. The powerful features of Angular allows us to create complex, customizable, modern, responsive and user friendly web applications.

Angular follows component oriented application design pattern to develop completely reusable and modularized web applications. Popular web platforms like Google Adwords, Google Fiber, Adsense has built their user interfaces using Angular.

In this course, we will discuss about components, modules, directives, data binding, pipes, HttpClient , Routing and much more. We will be using Angular CLI to speed up the development process of Angular applications. Angular CLI is a command line interface tool to scaffold and build Angular applications. Angular CLI offers all best practices right from development till deployment stage.

Target Audience: Developers

Prerequisite

Concepts you must know before doing this course

Ability to create static webpage using HTML

Ability to style HTML pages using CSS and Bootstrap

Ability to write client side scripting using Typescript for making HTML pages dynamic

Ability to use node package manager (npm) for module download

Recommended resources to learn the prerequisite concepts

HTML

CSS

Twitter Bootstrap

TypeScript

Software Requirement

Web Browser: Google Chrome / Internet Explorer Version 9+ / Mozilla Firefox

Tools/IDE's: Visual Studio Code / Visual Studio 2015+ / Eclipse (Neon onwards) / Angular IDE by Genuitec

Learning Outcomes

Users face issues like slow responses, more waiting time in traditional web applications. These problems can be overcome by creating a web application as a single page application. Single page applications are good for developing responsive websites.

Angular framework helps in developing single page applications. It is a unified platform which is used to develop web applications for both mobile and desktop.

This course will help you to create component based web application using Angular concepts.

By the end of the course, you will be able to:

- Create a component based application using Angular components
- Enhance the functionality of components using Angular directives
- Create Angular forms and bind them with model data using data binding
- Validate forms using Angular built-in or custom validators
- Format the rendered data using Angular built-in or custom pipes
- Make components interact and share data using Angular Input and Output decorators
- Communicate with remote server using Angular HttpClient class with RxJS Observables
- Build a single page application by using synchronous or asynchronous Angular routing

As part of this course, you will be learning Angular 6 concepts by using a web app namely mCart to purchase mobile gadgets and you will also be developing an application progressively called PoolCarz which allows users to share rides with others

Why Angular?

Angular 1 is a JavaScript framework from Google which is used for the development of Web applications. Google has decided to move to Angular version 2 and above due to the following reasons.

Cross Browser Compliant:

From the time Angular 1.x was designed, the internet has evolved significantly. Creating a web application which is cross browser compliant was a difficult task in Angular 1.x framework. Developers had to come up with various workarounds to overcome the issues. Angular helps to create cross browser compliant applications easily.

Typescript Support:

Angular is written in Typescript and allows the user to build applications using Typescript. Typescript is a superset of JavaScript and is more powerful language. Use of Typescript in application development improves productivity significantly.

Web Components Support:

Component based development is pretty much future of web development. Angular is focused on component based development.

Use of component helps in creating loosely coupled units of application which can be developed , maintained and tested easily.

Better support for Mobile App Development:

Desktop applications and mobile applications has separate concerns and addressing these concerns using a single framework becomes a challenge. Angular 1 had to do address concerns of mobile application using additional plugins. Angular is a single framework which addresses concerns of both mobile and desktop application.

Better performance:

Angular framework is better in its performance in terms of browser rendering, animation and accessibility across all the components. This is due to the modern approach of handling issues compared to earlier Angular version 1.x

What is Angular?

Let us now understand what is Angular and what kind of applications can be built using Angular

Angular is an open source **JavaScript** framework for building both mobile and desktop web applications

Angular is exclusively used to build **Single Page Applications (SPA)**

Angular is completely rewritten and is not an upgrade to Angular 1

Developers prefer TypeScript to write Angular code. But other than TypeScript, we can also write code using JavaScript (ES5 or ECMAScript 5)

Why most developers prefer TypeScript for Angular?

TypeScript is Microsoft's extension for JavaScript which supports object oriented features and has strong typing system which enhances the productivity

TypeScript supports many features like annotations, decorators, generics etc. A very good number of IDE's like Sublime text, Visual Studio Code, Nodeclipse etc., are available with TypeScript support

TypeScript code is compiled to JavaScript code using build tools like npm, bower, gulp, webpack etc., to make browser understand the code

Features of Angular

Let's discuss the features of Angular

Easier to learn: Angular is more modern and easier for developers to learn. It is more streamlined framework where developers will be focusing on writing JavaScript classes

Good IDE support: Angular is written in TypeScript which is a superset of JavaScript and supports all ECMAScript 6 features. Many IDEs like Eclipse, Microsoft Visual Studio, Sublime Text etc., has good support for TypeScript.

Familiar: Angular has retained many of its core concepts from earlier version (Angular 1), though it is a complete re-write. This means developers who are already proficient in Angular 1 will find it easy to migrate to Angular

Cross Platform: Angular is a single platform which can be used to develop applications for multiple devices

Lean and Fast: Angular applications production bundle size is reduced by 100s of kilobytes due to which it loads faster during execution

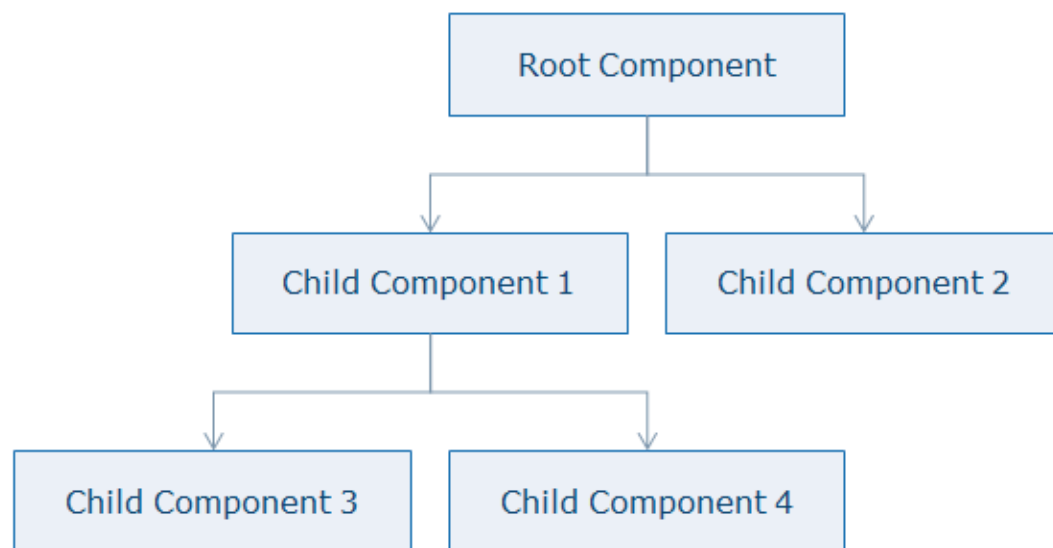
Simplicity: Angular 1 has 70+ directives like ng-if, ng-model etc., whereas Angular has very less number of directives as we use [] and () for bindings in HTML elements

Component-based

Angular follows component based programming which is the future of web development. Each component we create is isolated from every other part of our application. This kind of programming allows us to use components written using other frameworks.

Inside a component, we write both business logic and view.

Every Angular application will have one top-level component and several sub components.



Angular in Web Application Stack



mCart Application – User Stories

In this course, we will learn Angular framework by exploring the implementations of business requirements of an application called mCart.

mCart is an online shopping application that helps its users to purchase mobiles and tablet devices. This application allows users to login for purchasing mobiles and tablet devices. Users can select the products and add them to the cart. Once selection is done, users can go to the cart page for payment. Users can search for a product, sort the products list based on rating or price and can filter the products list based on manufacturer, operating system and price.

User Stories
Login to the application to buy tablets/mobiles
Search for a specific product
Filter products based on manufacturer, price and operating system
View the details of a specific product
Sort the products based on popularity and price
Add products to cart which I want to buy
Change the quantity of the products selected for purchase
Checkout for closing the purchase
Log out from the application

Journey Ahead

We will learn Angular course by building the mCart application. Below is the roadmap to achieve it.

1. Setup Angular development environment
2. Create a component and module
3. Use directives in Angular application
4. Implement data binding in Angular application
5. Apply pipes into Angular application
6. Create nested components and share data between them
7. Add validations to the forms in Angular application
8. Create services in Angular application
9. Implement routing in Angular application

Angular Development Environment Setup


To develop an application using Angular on a local system, we need to set up a development environment which includes installation of


- Node.js (min version required 8.x)
- Angular CLI
- Visual Studio Code


1. Install node.js from software house/software center as shown below or take help from Software house to get it installed

Download the Node.js source code or a pre-built installer for your platform, and start developing today.

LTS
Recommended For Most Users


Windows Installer
node-v8.11.3-x64.msi


macOS Installer
node-v8.11.3.pkg


Source Code
node-v8.11.3.tar.gz

Windows Installer (.msi)
Windows Binary (.zip)
macOS Installer (.pkg)
macOS Binary (.tar.gz)
Linux Binaries (x86/x64)
Linux Binaries (ARM)
Source Code

32-bit	64-bit	
32-bit	64-bit	
64-bit		
64-bit		
32-bit	64-bit	
ARMv6	ARMv7	ARMv8
node-v8.11.3.tar.gz		

To check whether node is installed or not in your machine, go to **node command prompt** and check the node version by typing the following command. It will display the version of node installed.

```
D:\> node -v  
v8.9.1
```

2. Install Angular CLI

Angular CLI can be installed using node package manager as shown below

Note: we have to set proxy if connecting to your organizations repository

```
npm config set registry <<npm repository name>>
```

```
npm login <<url>>
```

Username and Password should be provided if asked

Now run the following command to install CLI.

```
D:\> npm install -g @angular/cli
```

Test successful installation of Angular CLI using the following command

Note: Sometimes additional dependencies might throw an error during CLI installation but still check whether CLI is installed or not using the following command. If the version gets displayed, you can ignore the errors

```
D:\> ng -v
```

A terminal window with a black background and white text. The prompt is 'D:\>ng -v'. Below the prompt, the text 'Angular CLI: 6.0.7' is displayed in a large, stylized font. Below this, the following information is shown: 'Node: 8.11.2', 'OS: win32 x64', 'Angular:', and '...'.

```
D:\>ng -v
Angular CLI: 6.0.7
Node: 8.11.2
OS: win32 x64
Angular:
...
```

Angular CLI is a command line interface tool to build Angular applications. It makes application development faster and easier to maintain.

Using CLI, we can create projects, add files to it and can perform development tasks such as testing, bundling and deployment of applications.

Command	Purpose
<code>npm install -g @angular/cli</code>	Installs Angular CLI globally
<code>ng new <project name></code>	Creates a new Angular application
<code>ng serve --open</code>	Builds and runs the application on lite-server and launches a browser
<code>ng generate <name></code>	Creates class, component, directive, interface, module, pipe and service
<code>ng build</code>	Builds the application

3 .Install Visual Studio code from software

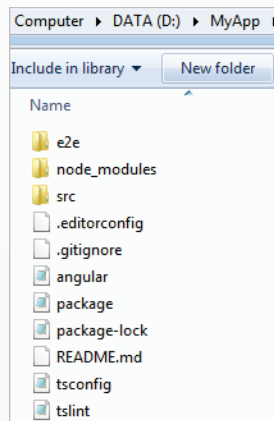
Demo 1 : Creating an Angular Application

Highlights:

- Creating an Angular application using Angular CLI
- Exploring the Angular folder structure

Demo Steps:

1. Create an application with name 'MyApp' using the following cli command
D:\>ng new MyApp
2. This will create the following folder structure

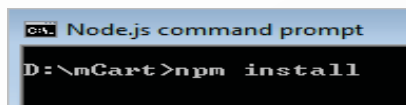


File	Purpose
e2e/	This folder contains all End-to-End (e2e) tests of the application written in Jasmine and run by the Protractor
node_modules/	Node.js creates this folder and puts all npm modules installed as listed in package.json
src/	All application related files will be stored inside it
angular.json	Configuration file for Angular CLI where we set several defaults and also configure what files to be included during project build
package.json	This is node configuration file which contains all dependencies required for Angular
tsconfig.json	This is Typescript configuration file where we can configure compiler options
tslint.json	This file contains linting rules preferred by Angular style guide

mCart Application Setup

After Angular development setup is done on your machine, let us now download mCart case study to your local machine. Download the application

- After downloading the project, open node.js command prompt and navigate to mCart folder. Run 'npm install' command to install the npm packages as shown below



```
C:\> Node.js command prompt
D:\mCart>npm install
```

This will create a folder called node_modules with all the dependencies installed inside it

Note: Sometimes Angular will throw errors during installation. This can be due to unavailability of some additional dependencies in your machine. After installation, always check if node_modules folder is created under project root folder. If it is created, you can ignore the errors occurred during installation.

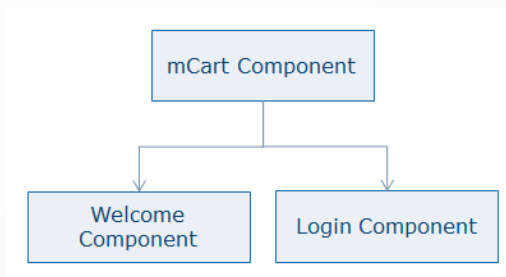
- Type the following command to run the application. This will open a browser with default port as 4200
D:\mCart>ng serve --open

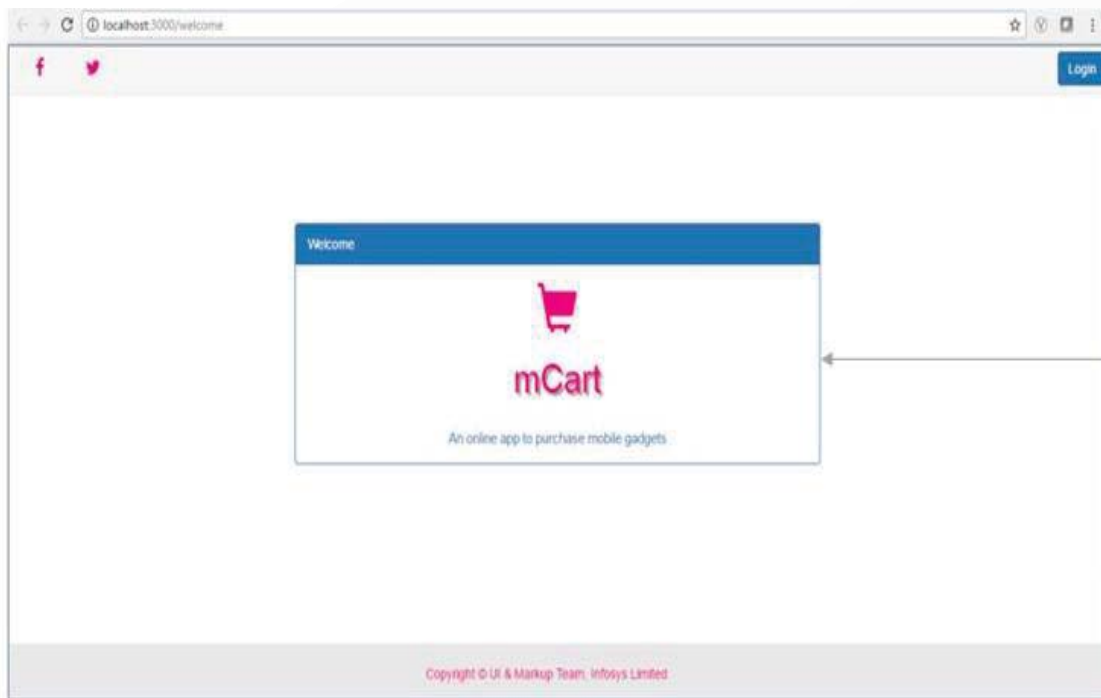
Components

Why Components in Angular?

- A component is the basic building block of an Angular application
- Components emphasizes the separation of concerns and each part of Angular application can be written independently of one another
- Components are reusable

For example, observe in mCart application the top most component is mcart component(app component) which consists of child components called welcome component, login component etc.,





Welcome Component

Root Component:
mCart Component

Creating A Component

- Open Visual studio Code IDE. Go to File menu and select Open Folder option. Select MyApp folder we have created earlier.
- Observe for our AppComponent we have below files
 - app.component.ts
 - app.component.html
 - app.component.css
- Let's explore each one of them
- Go to src folder-> app -> open app.component.ts file
- Observe the following code

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

Line 3: Adds component decorator to the class which makes the class a component

Line 4: Specifies the tag name to be used in the HTML page to load the component

Line 5: Specifies the template or HTML file to be rendered when component is loaded in HTML page. Template represents the view to be displayed

Line 6: Specifies the stylesheet file which contains CSS styles to be applied to the template.

Line 8: Every component is a class(AppComponent) and export is used to make it accessible in other components

Line 9: Creates a property with name title and initializes it to value 'app'

- Open `app.component.html` from `app` folder and observe the following code snippet in that file

```
1 <div style="text-align:center">_LF
2   -<h1>_LF
3   - - - Welcome to - {{title}}!_LF
4   -</h1>
```

Line 3: Accessing the class property by placing property called title inside {{ }}. This is called interpolation which is one of the data binding mechanism to access class properties inside template

Modules

As we have seen how to create component, now let us explore about modules.

- Modules in Angular are used to organize the application. Angular applications are collection of modules.
- A module in Angular is a class with `@NgModule` decorator added to it. `@NgModule` metadata will contain the declarations of components, pipes, directives, services which are to be used across the application.
- Every Angular application should have one root module which is loaded first to launch the application.
- We can have sub modules also which should be configured in the root module.

Root Module

In `app.module.ts` file placed under `app` folder, we have the following code

```

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

```

Line 1: imports BrowserModule class which is needed to run application inside browser

Line 2: imports NgModule class to define metadata of the module

Line 4: imports AppComponent class from app.component.ts file. No need to mention .ts extension as Angular by default considers the file as .ts file

Line 7: declarations property should contains all user defined components, directives, pipes classes to be used across the application. We have added our AppComponent class here

Line 10: imports property should contains all module classes to be used across the application

Line 13: providers property should contains all service classes. We will discuss about services later in this course

Line 14: bootstrap declaration should contains the root component to load. In this example, AppComponent is the root component which will be loaded in the html page

Bootstrapping Root Module

In main.ts file placed under src folder, observe the following code

```
1 import { enableProdMode } from '@angular/core';  
2 import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';  
3  
4 import { AppModule } from './app/app.module';  
5 import { environment } from './environments/environment';  
6  
7 if (environment.production) {  
8   enableProdMode();  
9 }  
10  
11 platformBrowserDynamic().bootstrapModule(AppModule);
```

Line 1 : Imports enableProdMode from the core module

Line 2: Import platformBrowserDynamic class which is used to compile the application based on the browser platform

Line 4: Import AppModule which is the root module to bootstrap

Line 5: imports environment which is used to check whether the type of environment is production or development

Line 7: Checks if we are working in production environment or not

Line 8: enableProdMode() will enable production mode which will run application faster

Line 11: bootstrapModule() method accepts root module name as parameter which will load the given module i.e., AppModule after compilation

Loading root component in HTML Page

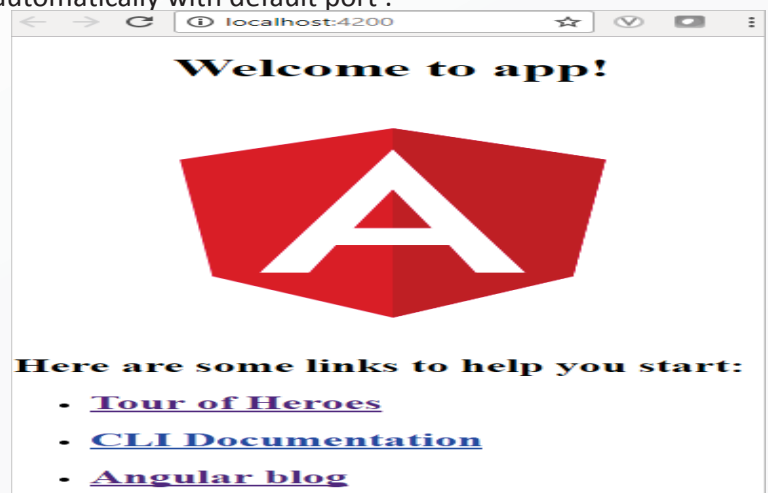
Open index.html under src folder

```
1      <!doctype html>
2      <html lang="en">
3      <head>
4          <meta charset="utf-8">
5          <title>MyApp</title>
6          <base href="/">
7          <meta name="viewport"
8              content="width=device-width, initial-scale=1">
9          <link rel="icon" type="image/x-icon"
10              href="favicon.ico">
11      </head>
12      <body>
13          <app-root></app-root>
14      </body>
15      </html>
```

Executing Angular Application

Now let us execute the application and check the output.

- Open terminal in visual studio code by selecting View Menu -> Integrated Terminal
- Type the following command to run the application
D:\MyApp>ng serve --open
- ng serve will build and run the application
- --open option will show the output by opening browser automatically with default port .
- Use the following command to change the port number if another application is running on the default port(4200)
D:\MyApp>ng serve --open --port 3000
- Following is the output of MyApp Application



Templates

- The default language for templates is HTML
- Templates in Angular represents a view whose role is to display data and change the data whenever an event occurs

Why Templates?

- Templates separates view layer from the rest of the framework. We can change the view layer without breaking the application

Creating a template

We can define a template in two ways

- Inline Template
- External Template

Inline Template

We can create template inline into component class itself using template property of @Component decorator.

app.component.ts

```
1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'app-root',
5   template: `
6     <h1> Welcome </h1>
7     <h2> Course Name: {{ courseName }}</h2>
8   `,
9   styleUrls: ['./app.component.css']
10 })
11 export class AppComponent {
12   courseName: string = "Angular";
13 }
```

Line 5-8: Default template used is external template. We can even write html code inside component using template property. Use back tick character (`) for multi line strings.

Output:

Welcome
Course Name: Angular

External Template

- By default, Angular CLI uses external template
- It binds template with a component using templateUrl option

Example

app.component.html

```
1 <h1>Welcome </h1>
2 <h2>Course Name: {{courseName}}</h2>
```

app.component.ts

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

Line 5: templateUrl property is used to bind external template file with the component

Output:

Welcome

Course Name: Angular

Elements of Template

Let us now understand the basic elements of template syntax:

- HTML
- Interpolation
- Template expressions
- Template statements

HTML

Angular uses HTML as a template language. In the below example, the template contains pure HTML code.

Interpolation

Interpolation is one of the form of data binding where we can access component's data in a template. For interpolation, we use double curly braces `{{ }}`.

Template Expressions

The text inside `{{ }}` is called as template expression.

`{{ expression }}`

- Angular first evaluates the expression and returns the result as a string. The scope of a template expression is component instance.
- That means, if we write `{{ courseName }}`, `courseName` should be the property of the component to which this template is bound to.

Template Statement

- Template Statements are the statements which responds to an user event.
(event) = statement
- For example (click) = "changeName()"
- This is called event binding. In Angular, all events should be placed in ().

Example:

app.component.ts

```
1  ... LF
2  export class AppComponent { LF
3    - - - - - courseName: string = "Angular"; LF
4    LF
5    - - - - - changeName() { LF
6    - - - - - - - - - - - - - - - this.courseName = "TypeScript"; LF
7    - - - - - } LF
8  }
```

Line 5-7: changeName is a method of AppComponent class where we are changing courseName property value to "TypeScript"

app.component.html

```
<h1> Welcome </h1> LF
<h2> Course Name: {{ courseName }} </h2> LF
<p (click)="changeName()">Click here to change</p>
```

Line 3: changeName() method is binded to click event which will be invoked on click of a paragraph at run time. This is called event binding.

Output:

Welcome

Course Name: Angular

[Click here to change](#)

When user clicks on the paragraph, course name will be changed to 'Typescript'

Welcome

Course Name: TypeScript

[Click here to change](#)

Change Detection

Now let us explore how Angular detects the changes and updates it in the application at respective places. Angular applications **runs five times faster** than Angular 1.x applications due to its improved **change detection mechanism**.

So, what is change detection mechanism and how it helps to run Angular applications so faster?

- Change Detection is a process in Angular which keeps views in sync with the models.
- In Angular, the flow is unidirectional from top to bottom in a component tree. Basically change can be caused by three things:
 - Events – click, submit etc.,
 - Ajax calls – Fetching data from a remote server
 - Timers –setTimeout(), setInterval()

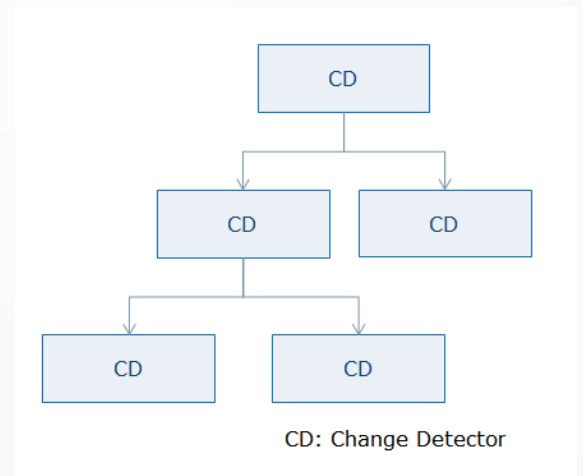
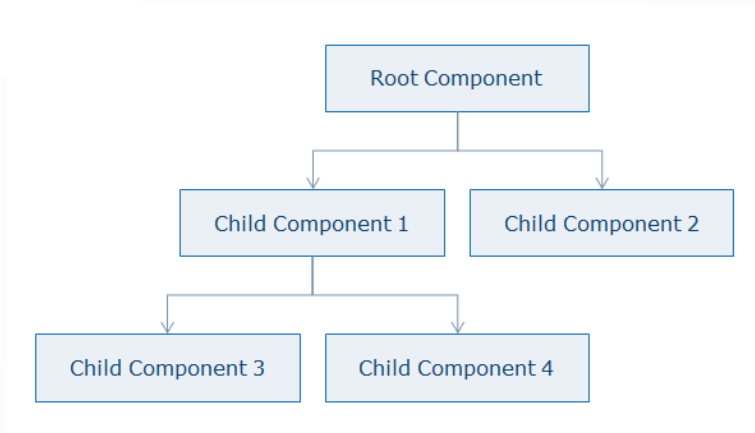
These are all asynchronous

But who informs Angular about the changes?

- **Zones** notifies Angular about the change detection. It monkey-patches (dynamically extending or modifying the existing classes or methods) all asynchronous operations in an application which is why Angular can easily find when to update the DOM.

Now let us see what will happen next when change detection is triggered.

- In Angular, each component has its own change detector which allows us to control how and when the change detection is performed.



- Let us assume that in an Angular application, an event is fired, may be a button is clicked. Then zones notifies this to Angular which performs change detection. This is wonderful as unidirectional data flow is predictable than cycles.
- Then how about the performance of an application when change detection is getting performed from top to bottom for every single event
- Angular is very fast and can perform hundreds of thousands of check of checks within few milliseconds as Angular generates **VM friendly code**.

What does that mean?

- Angular creates change detector classes at runtime for each component which are monomorphic as they know exactly what the shape of component's model is. VMs can perfectly optimize this code which makes it to execute very fast. We don't have to take care about this as Angular does it automatically.

Directives

As we understood the concept of templates, let us now understand directives in Angular.

- Directives are used to change the behavior of components or elements. We can use directives in the form of HTML attributes.
- We create directives using classes attached with `@Directive` decorator which adds metadata to the class.

Why Directives?

- Directives are used whenever we need to modify the DOM elements
- Directives are used to create reusable and independent code
- We can create custom elements to implement the required functionality

Types of Directives

There are three types of directives available in Angular

- Components
- Structural Directives
- Attribute Directives

Components

- Components are directives with a template or view.
- @Component decorator is actually @Directive with templates

Structural Directives

- A Structural directive changes the DOM structure by adding and removing DOM elements.
*directive-name = expression
- Angular has few built-in structural directives such as:
 - ngIf
 - ngFor
 - ngSwitch

ngIf

ngIf directive renders components or elements conditionally based on whether or not an expression is true or false.

Syntax: `*ngIf = "expression"`

ngIf directive doesn't hide the element instead it removes the element from the DOM tree.

Why elements are removed instead of hide?

- When we hide an element, the component's behavior changes. It remains in the DOM, continues to listen to events and Angular keeps on checking for changes being done on the elements.
- Even though it is invisible, the component tie up resources that might be more useful elsewhere.
- The performance and memory burden can be extensive. On the positive side, showing the element again is very fast as the component's previous state is preserved and ready to display.
- For ngIf in Angular, if it is set to false, Angular removes the element from DOM, stops change detection for that component, removes it from DOM events and destroys the component. The component will be garbage collected and frees up memory.
- Component's state might be expensive to re-construct. When ngIf is true, Angular recreates the component and its child components and runs every component's initialization logic again. This could be expensive as a component re-fetches data again that had been in memory just moments ago.
- Although there are pros and cons to each approach, it is best to use ngIf to remove unwanted components rather than hiding them.

Example:

app.component.ts

```
1  ... LF
2  export class AppComponent { LF
3    --isAuthenticated: boolean; LF
4    --submitted: boolean = false; LF
5    --userName: string; LF
6    LF
7    --onSubmit(name: string, password: string) { LF
8      ----this.submitted = true; LF
9      ----this.userName = name; LF
10     ----if (name === "admin" && password === "admin") LF
11       ----this.isAuthenticated = true; LF
12     ----else LF
13       ----this.isAuthenticated = false; LF
14     --} LF
15 }
```

Line 7 -14: onSubmit method is invoked when user clicks on submit button in the template. This method checks the username and password values entered are correct or not and accordingly assigns a Boolean value to isAuthenticated variable

app.component.html

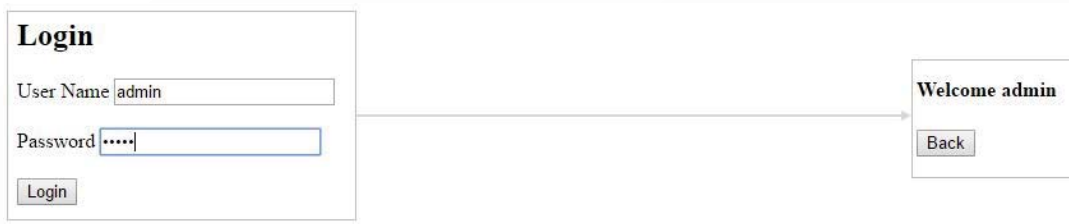
```
1  <div *ngIf="!submitted"> LF
2    --... LF
3    --<button (click)="onSubmit(username.value,password.value)">Login</button> LF
4  </div> LF
5  LF
6  <div *ngIf="submitted"> LF
7    --<div *ngIf="isAuthenticated; else failureMsg"> LF
8      ----<h4> Welcome -{{userName}} </h4> LF
9    --</div> LF
10   --<ng-template #failureMsg> LF
11     ----<h4> Invalid Login !!! Please try again...</h4> LF
12   --</ng-template> LF
13   --<button type="button" (click)="submitted=false">Back</button> LF
14 </div>
```

Line 7: Div tag will be displayed if 'isAuthenticated' property value is true otherwise it will go to else part and renders a template with id failureMsg

Line 10-12: ng-template is an angular element for rendering HTML. It is never displayed directly.

Output:

After entering correct credentials and clicking on Login button



After entering incorrect credentials and clicking on Login button



ngFor

ngFor directive is used to iterate over collection of data i.e., arrays

Syntax: *ngFor = "expression"

Example:

app.component.ts

```
1  ... LF
2  export class AppComponent { LF
3    courses: any[] = [ LF
4      { id: 1, name: "TypeScript" }, LF
5      { id: 2, name: "Angular" }, LF
6      { id: 3, name: "Node JS" }, LF
7      { id: 1, name: "TypeScript" } LF
8    ]; LF
9  }
```

Line 3-8 :Creating an array of objects

app.component.html

```
1  <ul> LF
2    <li *ngFor="let course of courses; let i = index"> LF
3      {{i}} - {{course.name}} LF
4    </li> LF
5  </ul>
```

Line 2: ngFor iterates over courses array and displays the value of name property of each course. It also stores index of each item in a variable called i

Line 3: {{ i }} displays the index of each course and course.name displays the name property value of each course

Output:

- 0 - TypeScript
- 1 - Angular
- 2 - Node JS
- 3 - TypeScript

ngSwitch

- ngSwitch adds or removes DOM trees when their expressions match the switch expression. Its syntax is comprised of two directives, an attribute directive and a structural directive.
- It is very similar to a switch statement in JavaScript and other programming languages.

Example:

app.component.ts

```
1 ... LF
2 export class AppComponent { LF
3   value: number = 0; LF
4   LF
5   nextChoice() { LF
6     this.value++; LF
7   } LF
```

Line 3 : Creates a value property and initializes it to zero

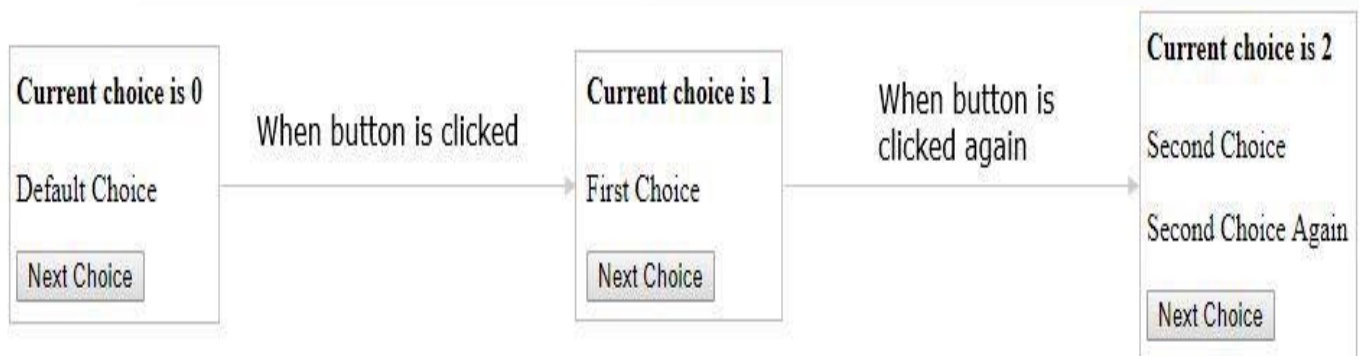
Line 5-7: nextChoice() increments the value when invoked

app.component.html

```
1 ... LF
2 <div [ngSwitch]="value"> LF
3   <p *ngSwitchCase="1">First Choice</p> LF
4   <p *ngSwitchCase="2">Second Choice</p> LF
5   <p *ngSwitchCase="3">Third Choice</p> LF
6   <p *ngSwitchCase="2">Second Choice Again</p> LF
7   <p *ngSwitchDefault>Default Choice</p> LF
8 </div> LF
... LF
```

Line 2: ngSwitch takes the value and based upon the value inside the value property, it executes *ngSwitchCase. Paragraph elements will be added/removed from the DOM based on the value passed to switch case.

Output:



Custom Structural Directive

- We can create custom structural directive when there is no built-in directive available for required functionality. For example, we can use custom structural directive for creating custom validators for form elements.
- **Example:**
 - Problem Statement:** Create a custom structural directive called 'repeat' which should repeat the element given number of times. As there is no built-in directive available to implement this, let us create it using a custom directive
- To create a custom structural directive, we need to create a class annotated with @Directive

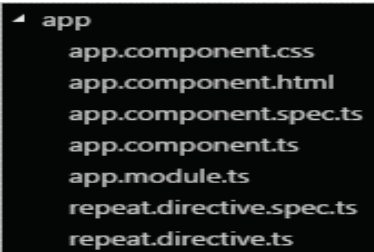
```
@Directive({  
  })  
class MyDirective{}
```

Example:

Generate a directive called 'repeat' using the following command

D:\MyApp>ng generate directive repeat

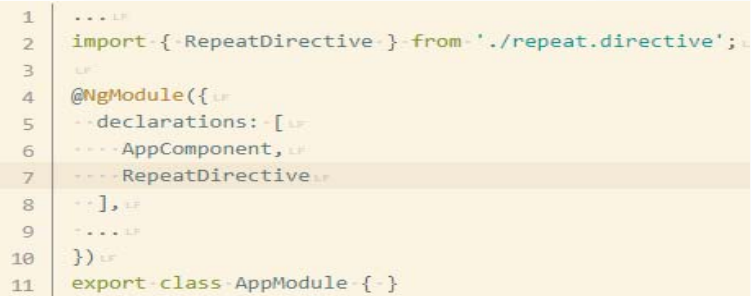
This will create two files under src\app folder with names repeat.directive.ts and repeat.directive.spec.ts (this is for testing). Now the app folder structure will look as shown below:



```
└─ app
   ├── app.component.css
   ├── app.component.html
   ├── app.component.spec.ts
   ├── app.component.ts
   ├── app.module.ts
   ├── repeat.directive.spec.ts
   └── repeat.directive.ts
```

It also adds repeat directive to the root module i.e., app.module.ts to make it available to the entire module as shown below in Line 7

app.module.ts



```
1  ... LP
2  import { RepeatDirective } from './repeat.directive';
3  LP
4  @NgModule({ LP
5    declarations: [ LP
6      AppComponent, LP
7      RepeatDirective LP
8    ], LP
9    ... LP
10 }) LP
11 export class AppModule { }
```

Open repeat.directive.ts file and add the following code

```
1 import { Directive, TemplateRef, ViewContainerRef, Input } from '@angular/core';
2
3 @Directive({
4   selector: '[appRepeat]'
5 })
6 export class RepeatDirective {
7
8   constructor(private _templateRef: TemplateRef<any>, private _viewContainer: ViewContainerRef) {}
9
10  @Input() set appRepeat(count: number) {
11    for (var i = 0; i < count; i++) {
12      this._viewContainer.createEmbeddedView(this._templateRef);
13    }
14  }
```

Line 3: Annotate the class with @Directive which represents the class as a directive and specify the selector name inside brackets

Line 8: Create a constructor and inject two classes called TemplateRef which acquires <ng-template> content and another class called ViewcontainerRef which access the html container to add or remove elements from it

Line 10: Create a setter method for appRepeat directive by attaching @Input() decorator which specifies that this directive will receive value from the component. This method takes the number passed to appRepeat directive as an argument.

Line 12: As we need to render the elements based on the number passed to the appRepeat directive, run a for loop in which pass the template reference to createEmbeddedView method which renders the elements into the DOM. This structural directive creates an embedded view from the Angular generated <ng-template> and inserts that view in a view container.

app.component.html

```
1 <h3>Structural Directive</h3>
2 <p *appRepeat="5">I am being repeated...</p>
```

Line 2: appRepeat directive is applied to the paragraph element. It will render five paragraph elements into the DOM

Output:

Structural Directive

I am being repeated...

I am being repeated...

I am being repeated...

I am being repeated...

I am being repeated...

Custom Structural Directive - exportAs Property

- exportAs property is used to define a name for a directive using which we can access directive class properties and methods in a component template.
- exportAs property can be applied to components and directives
- **Syntax:**

```
@Directive
class MyDirective{
  selector:'name',
  exportAs:'name1,name2,...'
}
```

We can give multiple names to export a directive.

Example:

Modify the repeat.directive.ts file used in custom structural directive as shown below

```

1 import { Directive, TemplateRef, ViewContainerRef, Input } from '@angular/core';
2
3 @Directive({
4   selector: '[appRepeat]',
5   exportAs: 'repeat,changeText'
6 })
7 export class RepeatDirective {
8
9   constructor(private _templateRef: TemplateRef<any>, private _viewContainer: ViewContainerRef) {}
10
11   repeatElement(count: number) {
12     for (var i = 0; i < count; i++) {
13       this._viewContainer.createEmbeddedView(this._templateRef);
14     }
15   }
16
17   changeElementText(count: number) {
18     for (var i = 0; i < 5; i++) {
19       document.getElementsByTagName("p").item(i).innerHTML = "Text is changed..."
20     }
21   }
22 }

```

Line 5: appRepeat directive is exported with two names here i.e., repeat and changeText. We can use multiple names to export a directive.

Line 11: repeatElement() method will take a count value as input and creates and renders the elements on the page

Line 17: changeElementText() method will take a count value as input and changes the rendered html elements text

Modify app.component.html file used in custom structural directive as shown below

```
1 <h3>Structural Directive with exportAs property</h3> LF
2 LF
3 <ng-template appRepeat #rd="repeat" #ct="changeText"> LF
4 ...<p>I am being repeated...</p> LF
5 </ng-template> LF
6 LF
7 <button (click)="rd.repeatElement(5)">Repeat Element</button> LF
8 <button (click)="ct.changeElementText(5)">Change Text</button>
```

Line 3: Here the template section is binded with template variables 'rd' and 'ct'. Each template variable has been assigned with the exported names of a directive class. Now in a template, we can access directive class methods using these template variables called 'rd' and 'ct' where each is a reference to 'repeat' and 'changeText' names of a directive.

Line 7: repeatElement() method of a directive class is accessed using the template variable rd. When this button is clicked, it invokes method which will in turn renders five paragraphs on the page.

Line 8: changeElementText() method of a directive class is accessed using the template variable ct. When this button is clicked, it invokes the method which will in turn changes the text of all the rendered paragraphs.

Output:

Structural Directive with exportAs property

Repeat Element Change Text

When 'Repeat Element' button is clicked, it renders the below output

Structural Directive with exportAs property

I am being repeated...
I am being repeated...
I am being repeated...
I am being repeated...
I am being repeated...

Repeat Element Change Text

When 'Change Text' button is clicked, it renders the below output

Structural Directive with exportAs property

Text is changed...
Text is changed...
Text is changed...
Text is changed...
Text is changed...

Repeat Element Change Text

Attribute Directives

- Attribute directives changes the appearance / behavior of a component / element
- Following are built-in attribute directives
 - ngStyle
 - ngClass

ngStyle

This directive is used to modify a component / element's style. We can use the following syntax to set single css style to the element which is also known as style binding

[style.<cssproperty>] = "value"

Example:

app.component.ts

```
1 | ... LF
2 | export class AppComponent { LF
3 |   colorName: string = 'yellow'; LF
4 |   color: string = 'red'; LF
5 | }
```

Line 3-4: colorName and color properties are initialized to default values

app.component.html

```
1 | <div [style.background-color]="colorName" [style.color]="color">
2 |   - Uses fixed yellow background LF
3 | </div>
```

Line 1: style.background-color will set the background-color of the text to yellow and style.color directive will set the color of the text to red.

Output:

Uses fixed yellow background

If there are more than one css styles to apply, we can use ngStyle attribute.

Example:

app.component.ts

```
1  ...  
2  export class AppComponent {  
3    colorName: string = 'red';  
4    fontWeight: string = 'bold';  
5    borderStyle: string = '1px solid black';  
6  }
```

app.component.html

```
1  <p [ngStyle]="{  
2      color:colorName,  
3      'font-weight':fontWeight,  
4      borderBottom: borderStyle  
5  }">  
6    Demo for attribute directive ngStyle  
7  </p>
```

Line3-5: Create three properties called colorName, fontWeight and borderStyle and initialize them with some default values

Line 1-5: NgStyle directive is used here to set multiple CSS styles for the given text

Output:

Demo for attribute directive ngStyle

ngClass

It allows us to dynamically set and change the CSS classes for a given DOM element. We can use the following syntax to set single css class to the element which is also known as class binding

[class.<css_class_name>] = "property/value"

Example:

app.component.ts

```
1 ... LF
2 export class AppComponent { LF
3   isBordered: boolean = true; LF
4 }
```

Line 3: Create a Boolean property called isBordered and initialize it to true

app.component.css

```
1 .bordered { LF
2   border: 1px dashed black; LF
3   background-color: #eee; LF
4 }
```

app.component.html

```
1 <div [class.bordered]="isBordered"> LF
2   Border {{ isBordered ? "ON" : "OFF" }} LF
3 </div>
```

Line 1: Bind the isBordered property with the CSS class bordered. Bordered CSS class will be applied only if isBordered property evaluates to true.

Output:



If we have more than one CSS classes to apply, then we will go for ngClass syntax

Syntax: [ngClass] = "{css_class_name1 : Boolean expression, css_class_name2: Boolean expression,}"

Example:

app.component.ts

```
1  ... LF
2  export class AppComponent { LF
3  ... isBordered: boolean = true; LF
4  ... isColor: boolean = true; LF
5  }
```

Line 3-4: Two Boolean properties called isBordered and isColor are initialized to true

app.component.html

```
1  <div [ngClass]="{bordered: isBordered, color: isColor}">
2  ... Border {{ isBordered ? "ON" : "OFF" }} LF
3  </div>
```

Line 1: Two CSS classes called bordered and color are applied to div tag. Both the classes are bound with isBordered and isColor properties where the CSS classes will be applied to div tag only if the properties return true

app.component.css

```
1  .bordered { LF
2  ... border: 1px dashed black; LF
3  ... background-color: #eee; LF
4  } LF
5  LF
6  .color { LF
7  ... color: blue; LF
8  }
```

Output:

Border ON

Custom Attribute Directives

- We can create custom attribute directive when there is no built-in directive available for the required functionality. For Example, consider the following problem statement
- **Problem Statement:** Create an attribute directive called 'showMessage' which should display the given message in a paragraph when user clicks on it and should change the text color to red. As there is no built-in directive available to implement this functionality, we need to go for custom directive
- To create a custom attribute directive, we need to create a class annotated with @Directive

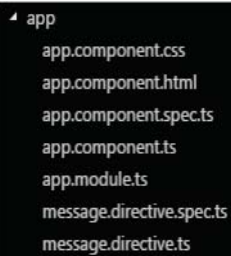
```
@Directive({  
  
  })  
class MyDirective { }
```

Example:

Generate a directive called 'message' using the following command

```
D:\MyApp>ng generate directive message
```

This will create two files under src\app folder with names message.directive.ts and message.directive.spec.ts (this is for testing). Now the app folder structure will look as shown below:



```
app
  app.component.css
  app.component.html
  app.component.spec.ts
  app.component.ts
  app.module.ts
  message.directive.spec.ts
  message.directive.ts
```

It also adds message directive to the root module i.e., app.module.ts to make it available to the entire module as shown below

```
1  ... LF
2  import { MessageDirective } from './message.directive'; LF
3  LF
4  @NgModule({ LF
5    declarations: [ LF
6      AppComponent, LF
7      MessageDirective LF
8    ], LF
9    ... LF
10 }) LF
11 export class AppModule { }
```

Open message.directive.ts file and add the following code

```
1 import { Directive, ElementRef, Renderer2, HostListener, Input } from '@angular/core';  
2  
3 @Directive({  
4   selector: '[appMessage]'  
5 })  
6 export class MessageDirective {  
7  
8   @Input('appMessage') message: string;  
9  
10  constructor(private el: ElementRef, private renderer: Renderer2) {  
11    renderer.setStyle(el.nativeElement, 'cursor', 'pointer');  
12  }  
13  
14  @HostListener('click') onClick() {  
15    this.el.nativeElement.innerHTML = this.message;  
16    this.renderer.setStyle(this.el.nativeElement, 'color', 'red');  
17  }  
18 }
```

Line 3: Create a directive class with the selector name as appMessage

Line 8: @Input('appMessage') will inject the value passed to 'appMessage' directive into the 'message' property

Line 10: Use constructor injection to inject ElementRef which holds the html element reference in which directive is used and Renderer2 which is used to set the CSS styles.

Line 11: Using Renderer2 reference, we are changing the cursor to pointer symbol

Line 14-16: onClick method is invoked when click event is triggered on the directive which will display the message received and changes the element color to red

app.component.html

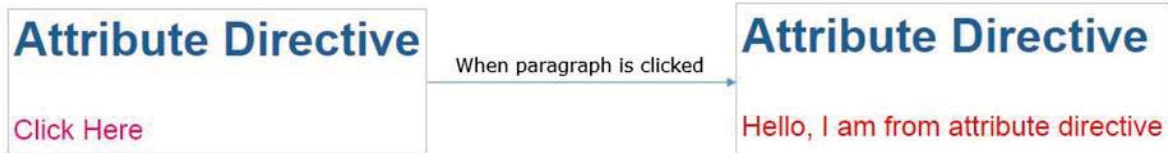
```
1 <h3>Attribute Directive</h3> LF
2 <p [appMessage]="myMessage">Click Here</p>
```

Line 2: Apply appMessage directive by assigning 'myMessage' property which will be sent to the directive on click of the paragraph

app.component.css

```
1 h3 { LF
2   --color: #369; LF
3   --font-family: Arial, Helvetica, sans-serif; LF
4   --font-size: 250%; LF
5 } LF
6 p { LF
7   --color: #ff0080; LF
8   --font-family: Arial, Helvetica, sans-serif; LF
9   --font-size: 150%; LF
10 }
```

Output:



Custom Attribute Directive - exportAs Property

- We have discussed exportAs property in custom structural directive concept. Similarly, let us apply this on a custom attribute directive as well.

Example:

In the custom attribute directive example, let us call onClick() method of a directive in a component template directly. For this, modify message.directive.ts file as shown below

```
1 import { Directive, ElementRef, Renderer2, HostListener, Input } from '@angular/core';
2
3 @Directive({
4   selector: '[appMessage]',
5   exportAs: 'changeMessage'
6 })
7 export class MessageDirective {
8
9   @Input('appMessage') message: string;
10
11   constructor(private el: ElementRef, private renderer: Renderer2) {
12     renderer.setStyle(el.nativeElement, 'cursor', 'pointer');
13   }
14
15   onClick() {
16     this.el.nativeElement.innerHTML = this.message;
17     this.renderer.setStyle(this.el.nativeElement, 'color', 'red');
18   }
19 }
```

Line 5: Directive class is exported with another name called 'changeMessage'.

Line 15: onClick() method will be directly called in a component template. We have removed click event binding here.

Modify app.component.html file as shown below

```
1 <h3>Attribute Directive with exportAs property</h3>
2
3 <div [appMessage]="myMessage" #msg="changeMessage">
4   <p (click)="msg.onClick()">Click Here</p>
5 </div>
```

Line 3: 'changeMessage' name is assigned to template variable called 'msg'. Now using 'msg' variable, we can access directive class methods.

Line 4: We are invoking onClick() method of a directive class using 'msg' template variable. When this paragraph is clicked, it will change the paragraph text and its color.

Output:

Attribute Directive with exportAs property
Click Here

When user clicks on 'Click Here' text, it renders the below output

Attribute Directive with exportAs property
Hello, I am from attribute directive

Data Binding

DataBinding is a mechanism used to coordinate between what users see on the screen and the data in the class.

As a developer, we need to declare bindings between binding sources and HTML target elements. The following are the three categories of bindings available in Angular based on the direction in which data flows.

Data Direction	Syntax	Binding type
One-way (class->view)	<code>{{ expression }}</code> <code>[target] = "expression"</code> <code>bind-target="expression"</code>	Interpolation Property Attribute Class Style
One-way (view->class)	<code>(target) = "statement"</code> <code>on-target = "statement"</code>	Event
Two-way	<code>[(target)] = "expression"</code> <code>bindon-target = "expression"</code>	Two way

Binding types other than interpolation have a target name to the left side of equal sign surrounded by [] or () or preceded by a prefix (bind-, on-, bindon-)

Now let us understand what is target here?

The target of data binding can be an (element or component or directive) property, an (element or component or directive) event, or rarely an attribute name.

One-way Data Binding

The following is the list of binding types for one-way data binding:

- Property Binding
- Attribute Binding
- Class Binding
- Style Binding
- Event Binding

Let us now understand each type of binding with examples.

Property Binding

- We use property binding when we want to set the property of a view element.
- Syntax:

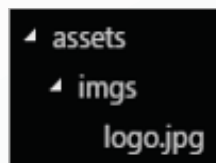
```
<img [src] = 'imageUrl' />
```

or

```
<img bind-src = 'imageUrl' />
```
- Here we are binding component property imageUrl property value to the image element property src.
- Interpolation is an alternative to property binding in many cases. But we must use property binding when we want to set a non-string value.

Example:

First, create a folder called 'imgs' under assets folder and copy the image into that folder



app.component.ts

```
1 | ... LF
2 | export class AppComponent { LF
3 |   imgUrl: string = 'assets/imgs/logo.jpg'; LF
4 | }
```

Line 3 : Create a property called imgUrl and initialize it to the image path

app.component.html

```
1 | <img [src]='imgUrl' width=200 height=100>
```

Line 1: Bind imgUrl property with src property. This is called property binding

Note: we can also write this statement as

```
1 | <img bind-src='imgUrl' width=200 height=100>
```

Output:



Attribute Binding

- Attribute binding can be used to set the value of an attribute directly.
- We must use attribute binding when there is no element property to bind. For example, consider ARIA, SVG and table span attributes which are pure attributes and do not correspond to set element properties. In these cases, we should use attribute binding to bind component property to the attribute directly. For example,
`<td colspan = "{{ 2+3 }}">Hello</td>`
- This gives an error as td element does not have a colspan property. It has colspan attribute, but interpolation and property binding can set only properties, not attributes.
- We use attribute bindings to create and bind such attributes.
- Attribute binding syntax starts with prefix attr. Followed by a dot sign and the name of the attribute. We then set the attribute value to an expression.
`<td [attr.colspan] = "2+3">Hello</td>`

Example:

app.component.ts

```
1 ... LF
2 export class AppComponent { LF
3   value: string = "2"; LF
4 }
```

Line 3: Create a property called value and initialize to 2

app.component.html

```
1 <table border=1> LF
2   <tr> LF
3     <td [attr.colspan]="value"> First </td> LF
4     <td>Second</td> LF
5   </tr> LF
6   <tr> LF
7     <td>... LF
8   </tr> LF
9 </table>
```

Line 3: attr.colspan will inform Angular that colspan is an attribute so that the given expression is evaluated and assigned to it. This is called attribute binding

Output:

First	Second	
Third	Fourth	Fifth

Style Binding

- Style binding is used to set inline styles. Syntax starts with prefix style, followed by a dot and the name of a CSS style property.

Syntax: [style.styleproperty]

Example:

```
1 | <button [style.color]="isValid?'blue':'red'">Hello</button>
```

Here button text color will be set to blue if the expression isValid is true, otherwise red.

- Some style bindings will have unit extension.

Example:

```
1 | <button [style.font-size.px]="isValid?3:6">Hello</button>
```

Here text font size will be set to 3 px if the expression isValid is true, otherwise it will be set to 6px.

- We prefer ngStyle directive to set multiple inline styles at the same time.

Event Binding

- User actions such as entering text in input boxes, picking items from lists, button clicks may result in a flow of data in opposite direction: from an element to the component.
- Event binding syntax consists of a target event with () on the left of an equal sign and a template statement on the right.

Example:

```
1 | <button (click)="onSubmit(username.value,password.value)">Login</button>
```

We can even write in the following way

```
1 | <button on-click ="onSubmit(username.value,password.value)">Login</button>
```

Two Way Data Binding

- It is a mechanism where if model property value changes, it updates the element to which the property is binded to and vice versa. It uses `[(ngModel)]` (banana in a box)

Syntax: `[(ngModel)]`

Example:

```
1 | <input [(ngModel)]="course.courseName">
```

Behind the scenes, this is equivalent to

```
1 | <input [ngModel]="course.courseName" (ngModelChange)="course.courseName=$event">
```

We can even write in the following form

```
1 | <input bindon-ngModel="course.courseName">
```

app.component.ts

```
1  ... LF
2  export class AppComponent { LF
3    name: string = "Angular"; LF
4  }
```

Line 3: Create a property called name and initialize it to value 'Angular'

app.component.html

```
1  <input type="text" [(ngModel)]="name"><br/> LF
2  <div>Hello, {{ name }}</div>
```

Line 1: Bind name property with text box using ngModel placed in [()] which is representation of two way data binding. Here whatever we type in the textbox at run time will be assigned to property name and when we change name property value, it will be reflected in the textbox

app.module.ts

To make two way data binding i.e., [(ngModel)] work, import FormsModule class to the root module as shown below

```
1  import { FormsModule } from '@angular/forms';
2  ... LF
3  LF
4  @NgModule({ LF
5    imports: [ LF
6      BrowserModule, LF
7      FormsModule LF
8    ], LF
9    ... LF
10  }) LF
11  export class AppModule { }
```

Output:

```
Angular  
Hello , Angular
```

Pipes

- Pipes is a beautiful way of transforming the data inside templates. It provides clean and structured code.
- Following is the list of built-in pipes available:
 - uppercase
 - lowercase
 - titlecase
 - currency
 - date
 - percent
 - slice
 - decimal
 - json
 - i18nplural
 - l18nselect
- Syntax: {{ expression | pipe }}

Built in pipes

- **Uppercase**

- This pipe converts the template expression into uppercase.
- **Syntax:** {{ expression | uppercase }}
- **Example:** {{ "Laptop" | uppercase }}
 - **Output:** LAPTOP

- **Lowercase**

- This pipe converts the template expression into lowercase.
- **Syntax:** {{ expression | lowercase }}
- **Example:** {{ "LAPTOP" | lowercase }}
 - **Output:** laptop

- **Titlecase**

- This pipe converts the first character in each word of the given expression into capital letter.
- **Syntax:** {{ expression | titlecase }}
- **Example:** {{ "product details " | titlecase }}
 - **Output:** Product Details

Passing Parameters to Pipes

- A pipe can also have optional parameters to change the output. To pass parameters, after a pipe name add a colon (:) followed by the parameter value.
 - **Syntax:** pipename : parametervalue
- Pipe can also have multiple parameters as shown below
 - **Syntax:** pipename : parametervalue1:parametervalue2

Built in pipes

- Currency
 - This pipe displays currency symbol before the expression. By default, it displays currency symbol \$
 - **Syntax:** {{ expression | currency:currencyCode:symbol:digitInfo:locale }}
 - **CurrencyCode** is the code to display such as INR for the rupee, EUR for the euro etc.,
 - **Symbol** is a Boolean value which represents whether to display currency symbol or code.
 - **code:** displays code instead of symbol such as USD, EUR etc.,
 - **symbol** (default): displays symbol such as \$ etc.,
 - **symbol-narrow:** displays the narrow symbol of currency. Some countries have two symbols for their currency, regular and narrow. For example, the Canadian Dollar CAD has the symbol as CA\$ and symbol-narrow as \$.
 - **digitInfo** is a string in the following format
{minIntegerDigits}.{minFractionDigits} - {maxFractionDigits}
 - minIntegerDigits is the minimum integer digits to display. Default value is 1
 - minFractionDigits is the minimum number of digits to display after fraction. Default value is 0
 - maxFractionDigits is the maximum number of digits to display after fraction. Default value is 3
 - **locale** is used to set the format followed by a country/language. To use locale, we need to register the locale in the root module.

Built in pipes

- Date
 - This pipe can be used to display date in required format
 - **Syntax:** {{ expression | date:format:timezone:locale }}
 - **Expression** is a date or number in milliseconds
 - **Format** indicates in which form date/time should be displayed. Following are the pre-defined options for it.
 - 'medium': equivalent to 'MMM d, y, h:mm:ss a' (e.g. Jan 31, 2018, 11:05:04 AM)
 - 'short': equivalent to 'M/d/yy, h:mm a' (e.g. 1/31/2018, 11:05 AM)
 - 'long': equivalent to 'MMMM d, y, h:mm:ss a z' (e.g. January 31, 2018 at 11:05:04 AM GMT+5)
 - 'full': equivalent to 'EEEE, MMMM d, y, h:mm:ss a zzzz' (e.g. Wednesday, January 31, 2018 at 11:05:04 AM GMT+05:30)
 - 'fullDate': equivalent to 'EEEE, MMMM d, y' (e.g. Wednesday, January 31, 2018)
 - 'longDate': equivalent to 'MMMM d, y' (e.g. January 31, 2018)
 - 'mediumDate': equivalent to 'MMM d, y' (e.g. Jan 31, 2018)
 - 'shortDate': equivalent to 'M/d/yy' (e.g. 1/31/18)
 - 'mediumTime': equivalent to 'h:mm:ss a' (e.g. 11:05:04 AM)
 - 'shortTime': equivalent to 'h:mm a' (e.g. 11:05 AM)
 - 'longTime': equivalent to 'h:mm a' (e.g. 11:05:04 AM GMT+5)
 - 'fullTime': equivalent to 'h:mm:ss a zzzz' (e.g. 11:05:04 AM GMT+05:30)

Built in pipes: Date conti.

- **Timezone** to be used for formatting. For example, '+0430' (4 hours, 30 minutes east of the Greenwich meridian) If not specified, the local system timezone of the end-user's browser will be used.
- **locale** is used to set the format followed by a country/language. To use locale, we need to register the locale in the root module.

Built in pipes

- Percent
 - This pipe can be used to display number as a percentage
 - **Syntax:** {{ expression | percent:digitInfo:locale }}
 - **digitInfo** is a string in the following format
{minIntegerDigits}.{minFractionDigits} - {maxFractionDigits}
 - minIntegerDigits is the minimum integer digits to display. Default value is 1
 - minFractionDigits is the minimum number of digits to display after fraction. Default value is 0
 - maxFractionDigits is the maximum number of digits to display after fraction. Default value is 3
 - **locale** is used to set the format followed by a country/language. To use locale, we need to register the locale in the root module.

Built in pipes

- Slice
 - This pipe can be used to extract subset of elements or characters from an array or string respectively.
 - **Syntax:** {{ expression | slice:start:end }}
 - **Expression** can be an array or string
 - **start** represents the starting position in an array or string to extract items. It can be a
 - positive integer which will extract from the given position till the end
 - negative integer which will extract the given number of items from the end
 - **end** represents the ending position in an array or string for extracting items. It can be
 - positive number which return all items before end index
 - negative number which returns all items before end index from the end of the array or string

Built in pipes

- Number
 - This pipe can be used to format a number.
 - **Syntax:** {{ expression | number:digitInfo }}
 - **Expression** should be numeric
 - **digitInfo** is a string in the following format
{minIntegerDigits}.{minFractionDigits} - {maxFractionDigits}
 - minIntegerDigits is the minimum integer digits to display. Default value is 1
 - minFractionDigits is the minimum number of digits to display after fraction. Default value is 0
 - maxFractionDigits is the maximum number of digits to display after fraction. Default value is 3

Built in pipes

- **json**
 - This pipe can be used to displays the given expression in the form of JSON string. It is mostly for debugging.
 - **Syntax:** {{ expression | json }}
 - **Example:** {{ {'productId':1234, 'productName':'Samsung Mobile'} | json }} will display {"productId":1234, "productName":"Samsung Mobile"}
- **i18nplural**
 - This pipe can be used to map numeric values against an object containing different string values to be returned. It takes a numeric value as input and compares it with the values in an object and returns a string accordingly.
 - **Syntax:** {{ expression | i18nplural:mappingObject }}
 - **mappingObject** is an object containing different strings to be returned for different numeric values
- **i18nselect**
 - This pipe is similar to i18nplural but evaluates a string value instead.
 - **Syntax:** {{ expression | i18nselect:mappingObject }}
 - **mappingObject** is an object containing strings to be displayed for different values provided by the expression

Custom Pipes

- We have explored built in pipes so far. But if we want to implement functionalities such as sorting, filtering etc., we should go for custom pipes as there are no built-in pipes available
- We can create our own custom pipe by inheriting PipeTransform interface
- PipeTransform interface has transform method where we need to write custom pipe functionality
- **Syntax:**
 - ```
@Pipe({
 name: 'pipename'
})
export class classname implements PipeTransform {
 transform(value: any, ...args:any[]): any {}
}
```
- transform method has two arguments, first one is the value of the expression passed to the pipe and second is the variable arguments. We can have multiple arguments based on the number of parameters passed to the pipe. Transform method should return the final value.

## Example - sort.pipe.ts

```
3 @Pipe({
4 name: 'sort'
5 })
6 export class SortPipe implements PipeTransform {
7
8 transform(value: string[], args?: string): string[] {
9 if (args === "prodName") {
10 return value.sort((a: any, b: any) => {
11 if (a.productName < b.productName) {
12 return -1;
13 } else if (a.productName > b.productName) {
14 return 1;
15 } else {
16 return 0;
17 }
18 });
19 }
20 else if (args === "price") {
21 return value.sort((a: any, b: any) => {
22 if (a.price < b.price) {
23 return -1;
24 } else if (a.price > b.price) {
25 return 1;
26 } else {
27 return 0;
28 }
29 });
30 }
31 return value;
32 }
33 }
```

- Line 3-5: @Pipe decorator creates a pipe with name called sort
- Line 6: Inherit PipeTransform interface for custom pipe
- Line 8: Override transform method of PipeTransform interface to write the sorting functionality. This method stores the value passed into the first argument called value and the parameters of the pipe into second argument called args.
- Line 9-30: Based on the second argument value, sorting functionality is implemented either by name or price.
- Line 31: Returns the sorted products array

## Example - app.component.ts

```
9 ··sortoption: string = ""; LF
10 ··productsList = [LF
11 ····{ productName: "Samsung J7", price: 18000 }, LF
12 ····{ productName: "Apple iPhone 6S", price: 60000 }, LF
13 ····{ productName: "Lenovo K5 Note", price: 10000 }, LF
14 ····{ productName: "Nokia 6", price: 15000 }, LF
15 ····{ productName: "Vivo V5 Plus", price: 26000 } LF
16 ··]; LF
```

- **Line 9:** sortoption property holds the value selected from the dropdown. It will be either product name or product price
- **Line 10-16:** productsList array holds list of products where each product has productName and price properties

## Example - app.component.html

```
4 <select [(ngModel)]="sortoption">_LF
5<option value="prodName">Product Name</option>_LF
6<option value="price">Price</option>....._LF
7</select>

_LF
8 _LF
9 <table border="1">_LF
10<thead>_LF
11<tr>_LF
12<th>Product Name</th>_LF
13<th>Price</th>_LF
14</tr>_LF
15</thead>_LF
16<tbody>_LF
17<tr *ngFor="let products of productList | sort:sortoption">_LF
18<td>{{products.productName}}</td>_LF
19<td>{{products.price}}</td>_LF
20</tr>_LF
```

- **Line 4-7:** Renders a dropdown with two options as ProductName and Price which is binded to sortoption property
- **Line 17-20:** Renders a row for each product in productList. Sort pipe is applied to the productList where the option selected from the dropdown will be passed as parameter to the sort pipe. ProductsList will be sorted based on product name or price.

## Example - app.module.ts

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

- **Line 10:** SortPipe is added to the declarations property to make it available to the entire module.

## Example - output

### Sorting Products list using custom pipe

Sort the list based on

| Product Name    | Price |
|-----------------|-------|
| Apple iPhone 6S | 60000 |
| Lenovo K5 Note  | 10000 |
| Nokia 6         | 15000 |
| Samsung J7      | 18000 |
| Vivo V5 Plus    | 26000 |

### Sorting Products list using custom pipe

Sort the list based on

| Product Name    | Price |
|-----------------|-------|
| Lenovo K5 Note  | 10000 |
| Nokia 6         | 15000 |
| Samsung J7      | 18000 |
| Vivo V5 Plus    | 26000 |
| Apple iPhone 6S | 60000 |

## Nested Components

- Nested component is a component which is loaded into another component
- The component where we load another component is called as container component/parent component
- We are loading root component in index.html page using its selector name, similarly if we want to load one component into another one, we will load it using its selector name in the template i.e., html page of container component
- Let us now explore how to create multiple components and load one into another

## Example

### Popular Courses

View Courses list

Renders courses list when button is clicked

### Popular Courses

View Courses list

| Course ID | Course Name |
|-----------|-------------|
| 1         | Node JS     |
| 2         | Typescript  |
| 3         | Angular     |
| 4         | React JS    |



## Example

```
1 | D:\MyApp>ng generate component coursesList
```

```
└─ app
 └─ courses-list
 ├── courses-list.component.css
 ├── courses-list.component.html
 ├── courses-list.component.spec.ts
 └── courses-list.component.ts
```

- Create a component called coursesList using the following CLI command
- This command will create four files called courses-list.component.ts, courses-list.component.html, courses-list.component.css, courses-list.component.spec.ts and places them inside a folder called courses-list under app folder as shown below
- This command will also add CoursesList component to the root module.

## Example - app.module.ts

```
1 ...
2 import { CoursesListComponent } from './courses-list/courses-list.component';
3
4 @NgModule({
5 declarations: [
6 AppComponent,
7 CoursesListComponent
8],
9 ...
10 })
11 export class AppModule { }
```

- **Line 7:** CoursesListComponent is added to the declarations property to make it available to all other components in the module.

## Example - courses-list.component.ts

- **Line 4-9:** courses is an array of objects where each object has properties called courseId and courseName

```
3 | LF
4 | courses = [LF
5 | { courseId: 1, courseName: "Node JS" }, LF
6 | { courseId: 2, courseName: "Typescript" }, LF
7 | { courseId: 3, courseName: "Angular" }, LF
8 | { courseId: 4, courseName: "React JS" } LF
9 |]; LF
10 | LF
```

## Example

```
3 <tbody>_LF
4 <tr *ngFor="let course of courses">_LF
5 <td>{{course.courseId}}</td>_LF
6 <td>{{course.courseName}}</td>_LF
7 </tr>_LF
8 </tbody>_LF
9 </table>
```

```
1 tr{ _LF
2 text-align:center; _LF
3 }
```

```
1 ..._LF
2 <button (click)="show=true">View Courses list</button>

_LF
3 <div *ngIf="show">_LF
4 <app-courses-list></app-courses-list>_LF
5 </div>
```

- **courses-list.component.html**
  - Line 4-7 : ngFor iterates over courses array and renders courseId and courseName values
- Add the following code in **courses-list.component.css**
  - Line 1-3: adds center alignment to table row in html page
- **app.component.html**
  - Line 2: click event is binded to button which will initialize show property to true when it is clicked
  - Line 4: It loads CoursesList component only if show property value is true.

## Nested Components - exportAs Property

- We can access child component properties and methods in a container component by exporting child component using exportAs property
- As discussed in directives concept, exportAs property can be applied to components and directives.

## Example

```
5 templateUrl: './courses-list.component.html', LF
6 styleUrls: ['./courses-list.component.css'], LF
7 exportAs: 'courselist' LF
8 }) LF

18 LF
19 changeCourse(name: string) { LF
20 this.course = []; LF
21 for (var i = 0; i < this.courses.length; i++) { LF

8 <tbody> LF
9 <tr *ngFor="let c of course"> LF
10 <td>{{c.courseId}}</td> LF
11 <td>{{c.courseName}}</td> LF
```

- In the nested components example, modify **courses-list.component.ts** file as shown below
  - Line 7: CoursesListComponent is exported with 'courselist' name
  - Line 19: changeCourse() method will take course name as an input, fetches the course details and assigns it to course array
- Modify **courses-list.component.html** as shown below
  - Line 9: course array is rendered on the page in a table

## Example

```
3 Select a course to view
4 <select #course (change)="cl.changeCourse(course.value)">
5 <option value="Node JS">Node JS</option>
6 <option value="Typescript">Typescript</option>
7 <option value="Angular">Angular</option>
8 <option value="React JS">React JS</option>
9 </select>
10

11

12
13 <app-courses-list #cl="courselist"></app-courses-list>
```

- Modify **app.component.html** as shown below
  - **Line 4-9:** A drop down will be displayed with course names. When a course is selected, it invokes `changeCourse()` method of child component using template variable 'cl'.
  - **Line 13:** It loads child component i.e., `CoursesListComponent` where we have binded exported name `courselist` to template variable called `cl`. Now using `cl`, we can access properties and methods of child component i.e., `CoursesListComponent` in component template.

## Example - output

### Popular Courses

Select a course to view

Node JS ▼  
Node JS  
Typescript  
Angular  
React JS

### Popular Courses

Select a course to view

Angular ▼

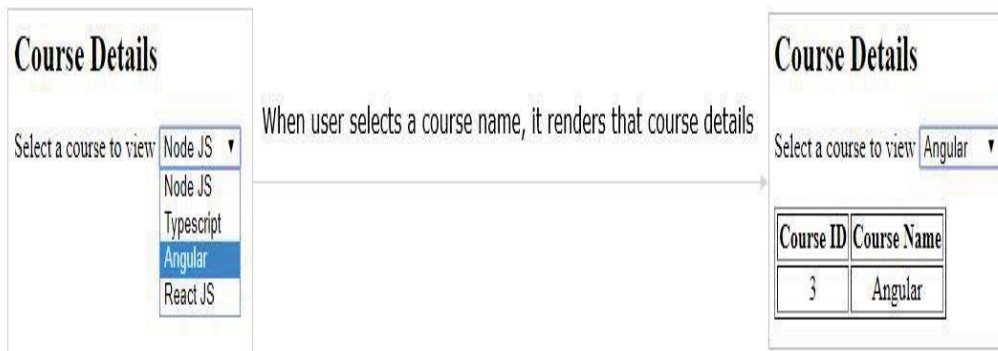
| Course ID | Course Name |
|-----------|-------------|
| 3         | Angular     |



## Passing data from Container Component to Child Component

- Component communication is needed if we want to share data between the components
- Let us explore how to pass data from container/parent component to child component
- We can use @Input decorator in the child component on any property type like arrays, objects etc.

## Example



- **Problem Statement:** Let us create an AppComponent which displays a dropdown with list of courses as values in it. Create another component called CoursesList component and load it in AppComponent which should display the course details. When user selects a course from dropdown, corresponding course details should be loaded

## Example

```
7 @Input().set cName(name: string) { LF
8 ...this.course = []; LF
9 ...for (var i = 0; i < this.courses.length; i++) { LF
10 ...if (this.courses[i].courseName === name) { LF
11 ...this.course.push(this.courses[i]); LF
12 ...} LF
13 ...} LF
```

```
4 ...<tr *ngFor="let c of course"> LF
5 ...<td>{{c.courseId}}</td> LF
6 ...<td>{{c.courseName}}</td> LF
7 ...</tr> LF
```

- Open **courses-list.component.ts** file used in the previous example
- Add input setter method for property cName in the component as shown below in Line 18
  - Line 7: @Input() specifies that cName property will receive value from its container component
  - Line 9-13: for loop will iterate over courses array and it checks for the courseName validity. If it matches, it adds that object to the course array
- **courses-list.component.html**
  - Line 4-7: ngFor iterates on course array and displays courseId and courseName properties in a table

## Example

```
2 Select a course to view <select #course (change)="name = course.value">
3 <<<<option value="Node JS">Node JS</option> LF
4 <<<<option value="Typescript">Typescript</option> LF
5 <<<<option value="Angular">Angular</option> LF
6 <<<<option value="React JS">React JS</option> LF
7 <<<</select>

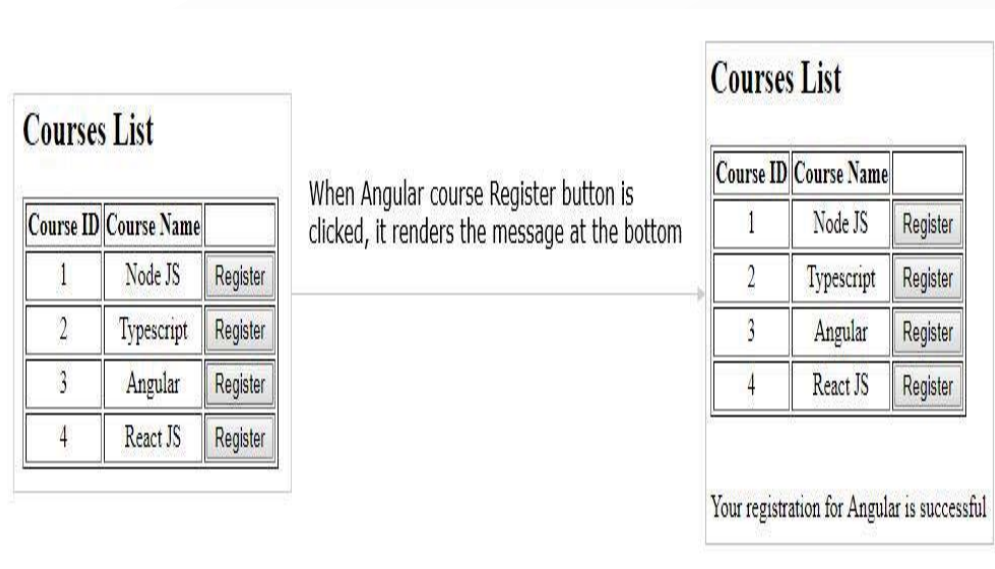
 LF
8 LF
9 <app-courses-list [cName]="name"></app-courses-list>
```

- Add the following code in **app.component.html**
  - Line 2-7 : It displays a drop down to select a course to display its details. When user selects a value, it assigns selected value to name property.
  - Line 9: This will load CoursesListComponent and passes the name property value to cName property of CoursesListComponent class.

## Passing data from Child to Container Component

- Now let us explore how to pass data from child to container component.
- If a child component wants to send data to its parent component, then it must create a property with @Output decorator.
- The only method for child component to pass data to its parent component is through events. The property must be of type EventEmitter

## Example



- **Problem Statement:** Let us create an AppComponent which loads another component called CoursesList component. Create another component called CoursesListComponent which should display the courses list in a table along with register button in each row. When user clicks on register button, it should send that courseName value back to AppComponent where it should display registration successful message along with courseName

## Example

```
3 @Output() onRegister = new EventEmitter<string>();
4 ...
5 ...
6 register(courseName: string) {
7 this.onRegister.emit(courseName);
```

```
6 <td>{{course.courseName}}</td>
7 <td><button (click)="register(course.courseName)">Register</button></td>
8 </tr>
```

- **courses-list.component.ts**

- Line 3: Create a property called onRegister of type EventEmitter and attach @Output decorator which makes the property to send the data from child to parent
- Line 7: this line emits the courseName value i.e, send the courseName value back to parent component.

- **courses-list.component.html**

- Line 7: When user clicks this button, it invokes register method by passing courseName value.

## Example

```
3 <app-courses-list (onRegister)="courseReg($event)"></app-courses-list> LF
4

 LF
5 LF
6 <div *ngIf="message">{{message}}</div>
```

```
5 LF
6 **courseReg(courseName: string)**{ LF
7 ***this.message = `Your registration for ${courseName} is successful`; LF
8 **} LF
```

- **app.component.html**

- Line 3: Binds onRegister event with courseReg method of parent component. When CoursesListComponent emits the value, onRegister event is triggered and it invokes courseReg method. \$event holds the value emitted by CoursesListComponent
- Line 6: This renders the message property value which holds the value emitted

- **app.component.ts**

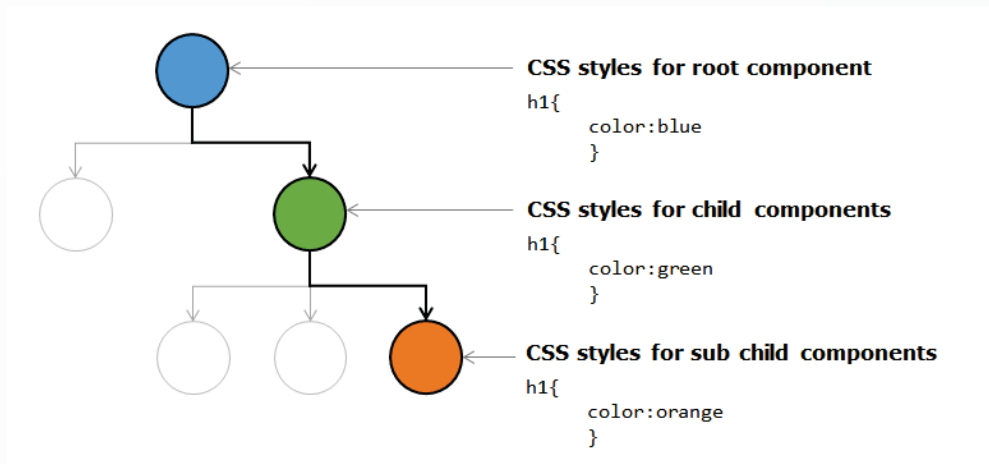
- Line 6: courseReg method is invoked when onRegister event emits
- Line 7: Assigns the string to message property which will be rendered in the template



## Shadow DOM

- Shadow DOM is a web components standard by W3C. It enables encapsulation for DOM tree and styles. It means that shadow DOM hides DOM logic behind other elements and also confines styles only for that component.
- For example, in an Angular application, we will create n number of components and each component will have its own set of data and CSS styles. When we integrate them, there is a chance that the data and styles may be applied to entire application. Shadow DOM encapsulates data and styles for each component so that it doesn't flow through the entire application
- In the below example shown, each component is having its own styles defined and they are confined to themselves

## Shadow DOM



## Component Styling

- Angular provides a mechanism for specifying component specific styles. We will provide styles for a component that won't leak out into the rest of the page.
- The following are the different ways to add styles to a component
  - By using styleUrls metadata
  - By using styles metadata
  - Inline into the template
- Using styleUrls property
  - We can load CSS styles declared in external files into a component by using styleUrls property.
  - styleUrls is an array property where we can load multiple CSS files into a component.
- Using styles property
  - We can add CSS styles to a component by adding styles property to the component metadata.
  - Styles is an array property where we can define multiple CSS classes for a component
- Template Inline Styles
  - We have another option to add css styles to component i.e., using inline style
  - We can directly embed styles in HTML template using <style> tag

## Examples - styleURL

### app.component.html

```
1 <div class="highlight">_LF
2 - - Container - Component _LF
3 </div>_LF
4 <app-child></app-child>
```

### app.component.css

```
1 .highlight {_LF
2 - - border: 2px solid coral; _LF
3 - - background-color: aliceblue; _LF
4 - - text-align: center; _LF
5 - - margin-bottom: 20px; _LF
6 }
```

```
1 | D:\MyApp>ng generate component Child
```

### app.component.ts

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

### child.component.ts

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

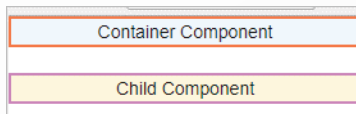
### child.component.css

```
1 .highlight {_LF
2 - - border: 2px solid violet; _LF
3 - - background-color: cornsilk; _LF
4 - - text-align: center; _LF
5 - - margin-bottom: 20px; _LF
6 }
```

### child.component.html

```
1 <div class="highlight">_LF
2 - - Child - Component _LF
3 </div>_LF
```

## Example – styleURL output



```
Elements Console Sources Network
▼<style>
 .highlight[_ngcontent-c0] {
 border: 2px solid coral;
 background-color: aliceblue;
 text-align: center;
 margin-bottom: 20px;
 }
 </style>
 ▼<style>
 .highlight[_ngcontent-c1] {
 border: 2px solid violet;
 background-color: cornsilk;
 text-align: center;
 margin-bottom: 20px;
 }
 </style>
 </head>
 ▼<body>
 ▼<app-root _ngghost-c0 ng-version="4.3.3">
 <div _ngcontent-c0 class="highlight">
 Container Component
 </div>
 ▼<app-child _ngcontent-c0 _ngghost-c1>
 <div _ngcontent-c1 class="highlight">
```

All the styles are moved to the head tag of html page and Angular will create a marker (\_ngcontent-\*) for each style for encapsulation.

The markers are added to the corresponding component selectors to encapsulate styles

## Example – style Property

```
5 --styles:[`
6 .highlight {
7 border: 2px solid coral;
8 background-color:AliceBlue;
9 text-align: center;
10 margin-bottom: 20px;
11 }
12 `], LF
```

```
1 <div class="highlight"> LF
2 - - - Container - Component LF
3 </div> LF
```

```
5 --styles:[`
6 .highlight {
7 border: 2px solid violet;
8 background-color:cornsilk;
9 text-align: center;
10 margin-bottom: 20px;
11 }
12 `], LF
```

- **app.component.ts**
  - Line 5-12: Add CSS styles specific to this component in styles property.
- **app.component.html**
  - Line 1: CSS class i.e., highlight is applied to the div tag
- **child.component.ts**
  - Line 5-12: Add CSS styles specific to this component in styles property.

## Example – style property

### child.component.html

```
1 <div class="highlight">_LP
2 - Child Component _LP
3 </div>
```

Container Component

Child Component

```
Elements Console Sources Network
▼<style>
 .highlight[_ngcontent-c0] {
 border: 2px solid coral;
 background-color: aliceblue;
 text-align: center;
 margin-bottom: 20px;
 }
 </style>
 <style>
 .highlight[_ngcontent-c1] {
 border: 2px solid violet;
 background-color: cornsilk;
 text-align: center;
 margin-bottom: 20px;
 }
 </style>
</head>
▼<body>
 <app-root _ngghost-c0 ng-version="4.3.3">
 <div _ngcontent-c0 class="highlight">
 Container Component
 </div>
 <app-child _ngcontent-c0 _ngghost-c1>
 <div _ngcontent-c1 class="highlight">
```

All the styles are moved to the head tag of html page and Angular will create a marker (\_ngcontent-\*) for each style for encapsulation.

The markers are added to the corresponding component selectors to encapsulate styles

## Example – inline styling

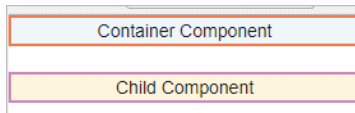
```
1 <style>
2 .highlight {
3 border: 2px solid coral;
4 background-color: aliceblue;
5 text-align: center;
6 margin-bottom: 20px;
```

```
1 <style>
2 .highlight { border: 2px solid violet;
3 background-color: cornsilk;
4 text-align: center;
5 margin-bottom: 20px;
6 }
```

- Remove the highlight css class from styles property in app.component.ts file and add it to **app.component.html** file as shown below
  - Line 1-6: Add CSS styles inside the template using style tag. These styles will be specific to this component
- Similarly, remove css highlight class from styles property of child.component.ts file and add it to **child.component.html** file as shown below
  - Line 1-6: Add CSS styles inside the template using style tag. These styles will be specific to this component



## Example – inline styling



```
Elements Console Sources Network
▼ <style>
 button[_ngcontent-c0] {
 background-color: cyan;
 border: 1px solid #777;
 }
 </style>
 ▼ <style>
 button[_ngcontent-c1] {
 background-color: blue;
 border: 1px solid #777;
 }
 </style>
 </head>
 ▼ <body>
 ▼ <app-root _ngghost-c0 ng-version="4.3.3">
 <button _ngcontent-c0>Click Me</button>
 ▼ <app-child _ngcontent-c0 _ngghost-c1>
 <button _ngcontent-c1>Click Me</button>
 </app-child>
 </app-root>
```

All the styles are moved to the head tag of html page and Angular will create a marker (\_ngcontent-\*) for each style for encapsulation.

The markers are added to the corresponding component selectors to encapsulate styles

## View Encapsulation

- View Encapsulation defines how we can encapsulate CSS styles into a component without flowing them to the rest of the page. Angular has built-in view encapsulation which enables us to use Shadow DOM.
- We can control how an encapsulation should work by using its three modes.
- The following are the three modes of encapsulation
  - ViewEncapsulation.Emulated (default)
  - ViewEncapsulation.Native
  - ViewEncapsulation.None
- **ViewEncapsulation.Emulated**
  - This is the default encapsulation type in Angular. It emulates style encapsulation even if shadow DOM is not available in the browsers.
  - This option is useful when we want to use a third party component which comes with its own styles and might affect our application
  - When this encapsulation type is used, it re-writes the styles to the document head with some attributes.
  - Since there is no shadow DOM, Angular has to write the styles to the document head. In order to enable scoped styles, it extends the CSS selectors so that they don't collide with other selectors defined in other components. That's why we see `_ngcontent-*` attributes

## View Encapsulation

- ViewEncapsulation.Native
  - It enables Angular to use native shadow DOM
- ViewEncapsulation.None
  - Angular doesn't use shadow DOM in this option.
  - All the styles defined in a component will be applied to the entire document. i.e., a component can overwrite another component's styles. This is an unscoped strategy.

## Example - ViewEncapsulation.Native

### first.component.css

```
1 | .cmp {
2 | padding: 6px;
3 | margin: 6px;
4 | border: blue 2px solid;
5 | }
```

### first.component.html

```
1 | <div class="cmp">First Component</div>
```

### second.component.css

```
1 | .cmp {
2 | border: green 2px solid;
3 | padding: 6px;
4 | margin: 6px;
5 | }
```

### second.component.html

```
1 | <div class="cmp">Second Component</div>
```

### app.component.css

```
1 | .cmp {
2 | padding: 8px;
3 | margin: 6px;
4 | border: 2px solid red;
5 | }
```

### app.component.html

```
1 | <h3>CSS Encapsulation with Angular</h3>
2 | <div class="cmp">
3 | App Component
4 | <app-first></app-first>
5 | <app-second></app-second>
6 | </div>
```

## Example - ViewEncapsulation.Native

### app.component.ts

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

### first.component.ts

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

### second.component.ts

```
1 import { Component, ViewEncapsulation } from '@angular/core';
2
3 @Component({
4 selector: 'app-second',
5 templateUrl: './second.component.html',
6 styleUrls: ['./second.component.css'],
7 encapsulation: ViewEncapsulation.Native
8 })
9 export class SecondComponent {
10
11 }
```

## Example - ViewEncapsulation.Native

### CSS Encapsulation with Angular

App Component

First Component

Second Component

```
Elements Console Sources Network Performance
▼ <style>
 .cmp[_ngcontent-c0] {
 padding: 8px;
 margin: 6px;
 border: 2px solid red;
 }
 </style>
 <style>
 .cmp[_ngcontent-c1] {
 padding: 6px;
 margin: 6px;
 border: blue 2px solid;
 }
 </style>
</head>
▼ <body>
 <app-root _ngcontent-c0 ng-version="4.3.3">
 <h3 _ngcontent-c0>CSS Encapsulation with Angular</h3>
 <div _ngcontent-c0 class="cmp">
 App Component
 <app-first _ngcontent-c0 _ngcontent-c1></app-first>
 <app-second _ngcontent-c0>
 <#shadow-root (open)>
 <style></style>
 <style></style>
 <style>.cmp {
 border: green 2px solid;
 padding: 6px;
 margin: 6px;
 }</style>
 <div class="cmp">Second Component</div>
 </shadow-root>
 </app-second>
 </div>
 </app-root>
</body>
```

Styles for the app component and first component are added to the head tag with the markers set

For second component, it has enabled the shadow root and the styles are encapsulated inside <app-second> selector. We can now see that there are no styles of second component are written to the document head. Styles now are placed in the component's template inside the shadow root.

## Example - ViewEncapsulation.None

Set ViewEncapsulation to none mode in **app.component.ts** file

```
1 import { Component, ViewEncapsulation } from '@angular/core';
2
3 @Component({
4 selector: 'app-root',
5 styleUrls: ['./app.component.css'],
6 templateUrl: './app.component.html',
7 encapsulation: ViewEncapsulation.None
8 })
9 export class AppComponent {
10 }
```

Set ViewEncapsulation to none mode in **second.component.ts** file

```
1 import { Component, ViewEncapsulation } from '@angular/core';
2
3 @Component({
4 selector: 'app-second',
5 templateUrl: './second.component.html',
6 styleUrls: ['./second.component.css'],
7 encapsulation: ViewEncapsulation.None
8 })
9 export class SecondComponent {
10
11 }
```

## Example - ViewEncapsulation.None

### CSS Encapsulation with Angular

App Component

First Component

Second Component

```
Elements Console Sources Network Performance
<style>.cmp {
 padding: 8px;
 margin: 6px;
 border: 2px solid red;
}
</style>
<style>
.cmp[_ngcontent-c1] {
 padding: 6px;
 margin: 6px;
 border: blue 2px solid;
}
</style>
<style>.cmp {
 border: green 2px solid;
 padding: 6px;
 margin: 6px;
}
</style>
</head>
<body>
<app-root ng-version="4.3.3">
 <h3>CSS Encapsulation with Angular</h3>
 <div class="cmp">
 App Component
 <app-first _ngghost-c1>
 <div _ngcontent-c1 class="cmp">First Component</div>
 </app-first>
 <app-second>
 <div class="cmp">Second Component</div>
 </app-second>
 </div>
</app-root>
```

Styles for the first component are added to the head tag with the markers set and styles of root and second components are moved to head tag without any marker tag i.e., second component styles have overwritten the root component styles

For second component, as the encapsulation is none, there are no markers set and its styles have overridden the root component styles



## Component Life Cycle

- A component has a life cycle managed by Angular which contains creating a component, rendering it, creating and rendering its child components, checks when its data bound properties change and destroys it before removing it from the DOM
- Angular has some methods/hooks which provides visibility into these key life moments of a component and the ability to act when they occur.
- Following are the lifecycle hooks of component. The methods will be invoked in the same order as mentioned in the below table

## Component Life Cycle

Interface	Hook	Support
OnChanges	ngOnChanges	Directive, Component
OnInit	ngOnInit	Directive, Component
DoCheck	ngDoCheck	Directive, Component
AfterContentInit	ngAfterContentInit	Component
AfterContentChecked	ngAfterContentChecked	Component
AfterViewInit	ngAfterViewInit	Component
AfterViewChecked	ngAfterViewChecked	Component
OnDestroy	ngOnDestroy	Directive, Component

## Example

```
1 import { Component, OnInit, DoCheck, AfterContentInit, AfterContentChecked, LP
2 ... AfterViewInit, AfterViewChecked, OnDestroy } from '@angular/core'; LP
3 LP
4 ... LP
5 export class AppComponent implements OnInit, DoCheck, AfterContentInit, AfterContentChecked, LP
6 ... AfterViewInit, AfterViewChecked, OnDestroy { LP
7 ... LP
8 ... ngOnInit() { ... } LP
9 ... LP
10 ... ngDoCheck() { ... } LP
11 ... LP
12 ... ngAfterContentInit() { ... } LP
13 ... LP
14 ... ngAfterContentChecked() { ... } LP
15 ... LP
16 ... ngAfterViewInit() { ... } LP
17 ... LP
18 ... ngAfterViewChecked() { ... } LP
19 ... LP
20 ... ngOnDestroy() { ... } LP
```

**Line 1 :** Import the interfaces of lifecycle hooks

**Line 5-6:** Inherit interfaces which has life cycle methods to override

**Line 8-20:** Override all lifecycle hooks

## Lifecycle Hooks

- `ngOnChanges` – It will be invoked when Angular sets data bound input property i.e., property attached with `@Input()`. This will be invoked whenever input property changes its value
- `ngOnInit` – It will be invoked when Angular initializes the directive or component
- `ngDoCheck` - It will be invoked for every change detection in the application
- `ngAfterContentInit` – It will be invoked after Angular projects content into its view
- `ngAfterContentChecked` – It will be invoked after Angular checks the bindings of the content it projected into its view
- `ngAfterViewInit` – It will be invoked after Angular creates component's views
- `ngAfterViewChecked` – It will be invoked after Angular checks the bindings of the component's views
- `ngOnDestroy` – It will be invoked before Angular destroys directive or component

## Example

```
6 export class AppComponent implements OnInit, OnDestroy,
7 AfterContentInit, AfterContentChecked,
8 AfterViewInit, AfterViewChecked,
9 OnDestroy {
13 ngOnInit() {
14 console.log("Init");
15 }
16 ngOnDestroy() {
17 console.log("Change detected");
18 }
19 ngDoCheck() {
20 console.log("Change detected");
21 }
22 ngAfterContentInit() {
23 console.log("After content init");
24 }
25 ngAfterContentChecked() {
26 console.log("After content checked");
27 }
28 ngAfterViewInit() {
29 console.log("After view init");
30 }
31 ngAfterViewChecked() {
32 console.log("After view checked");
33 }
34 ngOnDestroy() {
35 console.log("Destroy");
36 }
37 }
```

- **app.component.ts**
  - Line 6-9: Inherit all lifecycle interfaces
  - Line 13-39: Overriding all the lifecycle methods and logging a message

## Example

```
1 <div> * LF
2 --<h1>I'm a container component</h1> LF
3 --<input type="text" [(ngModel)]='data'> LF
4 --<app-child [title]='data'></app-child> LF
5 </div>
```

```
1 ... LF
2 export class ChildComponent implements OnChanges { LF
3 LF
4 @Input() title = 'I\'m a nested component'; LF
5 ... LF
6 ngOnChanges(changes) { LF
7 console.log("changes in child:"+JSON.stringify(changes)); LF
8 } LF
9 LF
10 }
```

```
1 <h2>Child-Component</h2> LF
2 <h2>{{title}}</h2> --
```

- **app.component.html**

- Line 3: textbox is binded with data property
- Line 4: Loads child component and data property is binded with title property of the child component

- **child.component.ts**

- Line 4: title is an input property which receives value from App component
- Line 6: Override ngOnChanges method which will be invoked whenever input property changes its value.

- **child.component.html**

## Example

**I'm a container component**

Angular

**Child Component**

**Angular**

top	Filter	Default levels
Init		
Change detected		
After content init		
After content checked		
changes in child:{"title":{"currentValue":"Angular","firstChange":true}}		
After view init		
After view checked		
Angular is running in the development mode. Call enableProdMode() to enable the production mode.		
Change detected		
After content checked		
After view checked		

As change happened, first it invoked `ngDoCheck()` followed by `ngAfterContentChecked`, `ngOnChanges` of child component class, `ngAfterViewChecked`

## Forms

- Forms are the crucial part of web applications through which we get majority of our data input from users. We can create two different types of forms in Angular
  - Template driven Forms** - Used to create small to medium sized forms
  - Model driven Forms or Reactive Forms** - Used to create large size forms

```
1 ... LP
2 import { FormsModule } from '@angular/forms'; LP
3
4 @NgModule({ LP
5 ... LP
6 imports: [LP
7 ... BrowserModule, LP
8 ... FormsModule LP
9], LP
10 ... LP
11 }) LP
12 export class AppModule { }
```

Line 2: Import FormsModule from @angular/forms module

Line 8: Add FormsModule in the imports property to make it available to the entire module



## Template Driven Forms

- Template driven forms are the forms which are created using Angular template syntax.
- In template driven form, data binding, validation etc., will be written in the template itself.

## Example

```
1 export class Course { LF
2 -- constructor(LF
3 -- -- public courseId: number, LF
4 -- -- public courseName: string, LF
5 -- -- public duration: string LF
6 --) - { - } LF
7 }
```

```
2 import { Course } from './course'; LF
3 LF
4 @Component({ LF
5 -- selector: 'app-course-form', LF
6 -- templateUrl: './course-form.component.html', LF
7 -- styleUrls: ['./course-form.component.css'] LF
8 }) LF
9 export class CourseFormComponent { LF
10 LF
11 -- course = new Course(1, 'Angular', '5 days'); LF
12 -- submitted = false; LF
13 LF
14 -- onSubmit() - { this.submitted = true; - } LF
15 }
```

- **course.ts**
  - **Line 1-6:** Create a model class called Course with properties courseId, courseName and duration
- **course-form.component.ts**
  - **Line 2:** Imports Course Model class from course.ts file
  - **Line 11:** Creates an instance of course model by passing some default values to its constructor
  - **Line 12:** Creates a submitted property and initializes it to false. This property is used to show and hide the form based on the submission
  - **Line 14:** onSubmit() method is invoked on submit button click which initializes submitted property to true

## Example

```
1 *... LF
2 "styles":-[LF
3 *... "styles.css", LF
4 *... "./node_modules/bootstrap/dist/css/bootstrap.min.css" LF
5 *...], LF
6 ...
```

- angular.json
- course-form.component.html

```
1 <div class="container"> LF
2 LF
3 --<div [hidden]="submitted"> LF
4 --<h1>Course-Form</h1> LF
5 --<form (ngSubmit)="onSubmit()" #courseForm="ngForm"> LF
6 LF
7 --<div class="form-group"> LF
8 --<label for="id">Course-Id</label> LF
9 --<input type="text" class="form-control" required [(ngModel)]="course.courseId" name="id" #id="ngModel"> LF
10 --<div [hidden]="id.valid || id.pristine" class="alert alert-danger"> LF
11 --Course-Id is required LF
12 --</div> LF
... LF
```

## ngForm

- ngForm is a directive which Angular automatically creates and attaches it to the <form> tag. It adds additional features to the form element. It holds the controls with an ngModel directive and name attribute and monitors their properties including their validity. It also has its own valid property which is true only if every control is valid
- Defining a name attribute is mandatory when [(ngModel)] is used on form controls.
- Internally Angular creates FormControl instances and registers them with NgForm directive. Each FormControl is registered with the name assigned to the name attribute.
- Track control state and validity with ngModel: We can also track the state of a control using ngModel in a form. Below are the available keywords to track control state.
- Angular also has built-in CSS classes to change the appearance of the control based on its state. Following are the CSS classes available

## ngForm

Keyword	Purpose
<b>valid</b>	Valid control value
<b>invalid</b>	Invalid control value
<b>dirty</b>	Changed control value
<b>pristine</b>	Unchanged control value
<b>touched</b>	True if control is touched
<b>untouched</b>	True if control is not touched

CSS Class	Purpose
ng-valid	Validates the control's value
ng-invalid	Applied if control's value is invalid
ng-dirty	Applied if control's value is changed
ng-pristine	Checks if control's value is not changed
ng-touched	Applied if control is touched
ng-untouched	Applied if control is not touched

## ngForm

- Advantages of Template Driven Forms
  - Simplicity: As we are putting entire code for data binding and validation in a form template, it is simple to code and very useful to build small to medium sized forms
  - We can prefer template driven forms to create small to medium sized forms
- Disadvantages of Template Driven Forms
  - As the tags or complex validations increase in the template, readability of the form decreases
  - We cannot perform unit testing on form validation logic. Only way to test is to run end to end test with a browser

## Template Driven Forms - updateOn Option

- Angular runs the control validation process whenever a form control value changes. For example, if you have an input bounded to a form control, Angular runs the control validation for every keystroke.
- A form with heavy validation requirements, updating on every keystroke can sometimes be too expensive.
- Angular provides a new option that improves performance by delaying form control updates until *blur* or *submit* event.
- The possible values for updateOn are:
  - change : Default. The value updates on every change.
  - blur : The value updates once the form lost its focus.
  - submit : The value updates once form is submitted .
- These validations can be used on both types of forms at input level or at form level using updateOn property.

```
8 <input type="text" class="form-control" required [(ngModel)]="course.courseId"
9 [ngModelOptions]="{ updateOn: 'blur' }" name="id" #id="ngModel">
10 </div [hidden]="id.valid || id.pristine" class="alert alert-danger">

17 <input type="text" class="form-control" required [(ngModel)]="course.courseName"
18 [ngModelOptions]="{ updateOn: 'submit' }" name="name" #name="ngModel">
19 </div [hidden]="name.valid || name.pristine" class="alert alert-danger">

3 <h1>Course Form</h1>
4 <form (ngSubmit)="onSubmit()" #courseForm="ngForm" [ngFormOptions]="{ updateOn: 'submit' }">
5 </form>
```

## Model Driven Forms or Reactive Forms

- Reactive forms are Angular way of creating forms in a reactive style. With Reactive forms, we create form control objects in a component class and bind them with HTML form elements in the template.
- As we create and manipulate form control objects directly in the component class, we can push model values into the form controls and fetch user-changed values back from the form. The component can observe changes in form control state and react to those changes.
- One advantage of working with form control objects directly is that value and validity updates are always synchronous and under our control.
- We will be using **FormBuilder** class to create reactive forms which has simplified syntax. We need to import **ReactiveFormsModule** to create reactive forms.
- We can use built-in validators using **Validators** class. For example, if we want to use required validator, it can be accessed as **Validators.required**



## Example

```
3 import { ReactiveFormsModule } from '@angular/forms';
4
5 import { AppComponent } from './app.component';
6 import { RegistrationFormComponent } from './registration-form/registration-form.component';
7
8 @NgModule({
9 declarations: [
10 AppComponent,
11 RegistrationFormComponent
12],
13 imports: [
14 ReactiveFormsModule,
15 RegistrationFormComponent
16],
17 })
export class AppModule {
}
```

### • app.module.ts

- Line 3: Import ReactiveFormsModule from @angular/forms module
- Line 13: Add ReactiveFormsModule in the imports declaration to create reactive forms

## Example

```
1 import { Component, OnInit } from '@angular/core';
2 import { FormBuilder, FormGroup, Validators } from '@angular/forms';
3
4 @Component({
5 selector: 'app-registration-form',
6 templateUrl: './registration-form.component.html',
7 styleUrls: ['./registration-form.component.css']
8 })
9 export class RegistrationFormComponent implements OnInit {
10
11 registerForm: FormGroup;
12
13 constructor(private formBuilder: FormBuilder) {}
14
15 ngOnInit() {
16 this.registerForm = this.formBuilder.group({
17 firstName: ['', Validators.required],
18 lastName: ['', Validators.required],
19 address: this.formBuilder.group({
20 street: [],
21 zip: [],
22 city: []
```

- **registration-form.component.ts**

- **Line 2:** Import FormBuilder class to create a reactive form. Also import FormGroup class to create a group of form controls and Validators for validation
- **Line 11:** Create a property registerForm of type FormGroup
- **Line 13:** Inject a FormBuilder instance using constructor
- **Line 16:** formBuilder.group() is a factory method that creates a FormGroup. It takes an object whose keys and values are FormControl names and their definitions
- **Line 17-21:** Create form controls such as firstName, lastName, and address as sub group with fields street, zip and city. These fields are form controls. Configure built-in validators for each control using ['', Validators.required] syntax where the first parameter is the default value for the control and second parameter is an array of validations. If multiple validators are to be applied, then we should give it as ['', [Validators.required, Validators.maxLength(10)]]

## Example

```
3 <form [formGroup]="registerForm">
4 <div class="form-group">
5 <label>First Name</label>
6 <input type="text" class="form-control" formControlName="firstName">
7 <p *ngIf="registerForm.controls.firstName.errors" class="alert alert-danger">This field is required!</p>
8 </div>
9 <div class="form-group">
10 <label>Last Name</label>
11 <input type="text" class="form-control" formControlName="lastName">
12 <p *ngIf="registerForm.controls.lastName.errors" class="alert alert-danger">This field is required!</p>
13 </div>
14 <div class="form-group">
15 <fieldset formGroupName="address">
16 <label>Street</label>
17 <input type="text" class="form-control" formControlName="street">
18 <label>Zip</label>
19 <input type="text" class="form-control" formControlName="zip">
20 <label>City</label>
21 <input type="text" class="form-control" formControlName="city">
22 </fieldset>
23 </div>
24 <button type="submit" (click)="submitted=true">Submit</button>
25 </form>
26

27 <div [hidden]="!submitted">
28 <h3>Employee Details</h3>
29 <p>First Name: {{ registerForm.get('firstName').value }}</p>
30 <p>Last Name: {{ registerForm.get('lastName').value }}</p>
31 <p>Street: {{ registerForm.get('address.street').value }}</p>
32 <p>Zip: {{ registerForm.get('address.zip').value }}</p>
33 <p>City: {{ registerForm.get('address.city').value }}</p>
34 </div>
35 </div>
```

### • registration-form.component.html

- Line 3: formGroup is a directive which binds HTML form with the FormGroup property created inside a component class. We have created a FormGroup in component with name registerForm. Here form tag is binded with FormGroup name called registerForm
- Line 6 , 11: Two textboxes for first name and last name are binded with the form controls created in the component using formControlName directive
- Line 24: When submit button is clicked, it initializes submitted property value to true
- Line 27 : div tag will be hidden if form is not submitted
- Line 29-33: Using get() method of FormGroup, we are fetching values of each FormControl and rendering it

## Example

```
1 | .ng-valid[required] { LF
2 | border-left: 5px solid #42A948; /* green */ LF
3 | } LF
4 | LF
5 | .ng-invalid:not(form) { LF
6 | border-left: 5px solid #a94442; /* red */ LF
7 | }
```

```
1 | <app-registration-form></app-registration-form>
```

- **registration-form.component.css**

- Line 1-3: ng-valid css class changes left border of textbox to green if form control has valid input
- Line 5-7: ng-invalid class changes left border of textbox to red if form control has invalid data

- **app.component.html**

- Line 1: Loads RegistrationFormComponent in the root component

## Reactive Forms - updateOn Option

- Similar to template driven forms, updateOn option can also be applied on Reactive forms
- In reactive form demo, modify **registration-form.component.ts** as shown below
  - Line 17: firstname will be updated with new value only on blur event.

```
15 | ngOnInit() {
16 | this.registerForm = this.formBuilder.group({
17 | firstName: ['', { updateOn: 'blur', validators: [Validators.required] }],
18 | lastName: ['', Validators.required]
19 | });
20 | }
21 | }
```

### Registration Form

First Name

Last Name

### Employee Details

First Name: Harry

Last Name: Potter

## Custom Validation

- We can create custom validators when there are no built-in validators are available to implement the required functionality.
- We can create custom validators for both template driven and reactive forms. Let us see how to create custom validator for template driven forms.
- **Custom Validation – Template Driven Forms**
  - **Problem Statement:** To create custom validator for checking email pattern. Though we have built in validator available for email in Angular, we will be creating custom validator as the built-in email validator will not check for complete pattern needed.
  - For template driven forms, we need to add validation keyword as an attribute to the control. So we need to create a directive for custom validation logic.
  - We will use the same example used for template driven forms.

## Example

```
3 *** public courseId: number, LF
4 *** public courseName: string, LF
5 *** public duration: string, LF
6 *** public email: string LF
7 ** } LF
```

```
10 LF
11 ** course = new Course(1, 'Angular', '5 days', 'james@gmail.com'); LF
12 ** submitted = false; LF
13 LF
```

```
2 import { NG_VALIDATORS, FormControl, Validator } from '@angular/forms'; LF
3 LF
4 @Directive({ LF
5 *** selector: '[validateEmail]', LF
6 *** providers: [LF
7 ***** { provide: NG_VALIDATORS, useExisting: EmailValidator, multi: true } LF
8 *****] LF
9 }) LF
10 export class EmailValidator implements Validator { LF
11 LF
12 *** validate(control: FormControl): { [key: string]: any } { LF
13 ***** const emailRegexp = /^[a-zA-Z0-9_\-\.]+@[a-zA-Z0-9_\-\.]+\.[a-zA-Z]{2,5}$/; LF
14 ***** if (!emailRegexp.test(control.value)) { LF
15 ***** return { "emailValid": true }; LF
16 ***** } LF
17 ***** return null; LF
```

- In **course.ts**, add another field for email as shown below at Line 6.
- In **course-form.component.ts** file, pass default value to email field as shown below at Line 11
- Create a file with name **email.validator.ts** under course-form folder to implement custom validation functionality for email field
  - **Line 2** : Import NG\_VALIDATORS which is provider with extensive collection of validators
  - **Line 7**: Register EmailValidator directive with NG\_VALIDATORS so that Angular recognizes the role of the directive in the validation process. Multi:true adds EmailValidator class to the existing list of validators available in NG\_VALIDATORS
  - **Line 10**: Inherit Validator interface
  - **Line 12**: Override validate() method which takes FormControl as a parameter and returns an object
  - **Line 13-17**: Defines email pattern and test whether value entered matches with the given pattern or not. If it doesn't match it returns an object with key as 'emailValid' and value as true, otherwise returns null

## Example

```
7 import { EmailValidator } from './course-form/email.validator';
8
9 @NgModule({
10 declarations: [
11 AppComponent,
12 CourseFormComponent,
13 EmailValidator
```

- Add EmailValidator class in the root module i.e., **app.module.ts** as shown below at Line 7 and 13
- Add the following code in **course-form.component.html** file for email field as shown below
  - **Line 35:** Add validateEmail directive as an attribute in the email text field
  - **Line 39:** Displays error message for email field if it is not valid. We are using the key 'emailValid' to check for the email errors.

```
35 name="email" #email="ngModel" validateEmail>
36
37 <div *ngIf="email.errors && (email.dirty || email.touched)" class="alert alert-danger">
38 <div [hidden]="!email.errors.required">Email is required</div>
39 <div [hidden]="!email.errors.emailValid">Email is invalid</div>
40 </div>
```



## Custom Validation

- **Custom Validation – Reactive Forms**

- Let us see how to create custom validator for reactive forms.
- We will use the example created for Reactive Forms:
- In that example, let us add one more field called email for which we will implement custom validation logic.

## Example

```
13email:['',validateEmail] LF
14}); LF
15} LF
16} LF
17 LF
18 function validateEmail(c: FormControl) { LF
19let EMAIL_REGEXP = /^[a-zA-Z0-9_\-\.]+@([a-zA-Z0-9_\-\.]+)\.([a-zA-Z]{2,5})$/; LF
20 LF
21return EMAIL_REGEXP.test(c.value) ? null : { LF
22emailValid: { LF
23valid: false LF
24} LF
25}; LF
26}
```

```
<label>Email</label><input type= text class= form-control formcontrolname= email > LF
<p *ngIf="registerForm.controls.email.errors?.emailValid" class="alert alert-danger">Invalid Format!</p>
</div>
```

- We will write a separate function in **registration-form.component.ts** for custom validation as shown below.
  - Line 18-26: In this function, we have taken regular expression pattern for email and testing with the input value. If they are same, it returns null otherwise returns an object with name emailValid with one property called valid set to false
  - Line 13: Binds custom validator called validateEmail to the email field.
- Now add html controls for email field in **registration-form.component.html** file as shown below
  - Line 7: Displays error message if email validation fails. errors object holds the error messages of all form controls

## Dependency Injection

- Dependency Injection (DI) is a mechanism where the required resources will be injected into the code automatically.
- Angular comes with an in-built dependency injection subsystem.
- Why Dependency Injection?
  - DI allows developers to reuse the code across application.
  - DI makes the code loosely coupled.
  - DI makes the application development and testing much easier.
  - DI allows the developer to ask for the dependencies from Angular. There is no need for the developer to explicitly create/instantiate them.

## Services

- A service in Angular is a class which contains some functionality that can be reused across the application. A service is a singleton object. Angular services are a mechanism of abstracting shared code and functionality throughout the application.
- Angular Services come as objects which are wired together using dependency injection.
- Angular provides few inbuilt services. We can also create custom services.
- Why Services?
  - Services can be used to share the code across components of an application.
  - Services can be used to make http requests.
- Creating a Service
  - Create a class with `@Injectable()` decorator.
  - `@Injectable()` decorator makes the class injectable into application components.

## Example - services

```
1 export class Book { LF
2 --- id: number; LF
3 --- name: string; LF
4 }
```

```
2 LF
3 export var BOOKS: Book[] = [LF
4 --- { "id": 1, "name": "HTML 5" }, LF
5 --- { "id": 2, "name": "CSS 3" }, LF
6 --- { "id": 3, "name": "Java Script" }, LF
7 --- { "id": 4, "name": "Ajax Programming" }, LF
8 --- { "id": 5, "name": "jQuery" }, LF
9 --- { "id": 6, "name": "Mastering Node.js" }, LF
10 --- { "id": 7, "name": "Angular JS 1.x" }, LF
11 --- { "id": 8, "name": "ng-book 2" }, LF
12 --- { "id": 9, "name": "Backbone JS" }, LF
13 --- { "id": 10, "name": "Yeoman" } LF
14];
```

- Create a file with name **book.ts** under book folder and add the following code.
  - Line 1-4: Create a Book class with two properties id and name to store book id and book name
- Create a file with name **books-data.ts** under book folder and add the following code.
  - Line 3-14: Books is an array of type Book class which holds books objects where each object has id and name properties

## Example - services

```
1 | D:\MyApp\src\app\book>ng generate service book --module=app
```

```
book
book.component.css
book.component.html
book.component.spec.ts
book.component.ts
book.service.spec.ts
book.service.ts
book.ts
```

```
2 | import { BookService } from './book/book.service';
3 |
4 | @NgModule({
5 | ...
6 | providers: [BookService],
7 | bootstrap: [AppComponent]
8 | })
9 | export class AppModule { }
```

- Create a service called **BookService** under book folder using the following CLI command
- This will create two files called book.service.ts and book.service.spec.ts as shown below
- --module=app will add BookService class to providers property in **app.module.ts** file automatically as shown below
  - Line 2: Imports BookService class into root module class
  - Line 6: Adds BookService to the providers property to make it injectable into the components of the root module

## Example - services

```
5 | @Injectable() LF
6 | export class BookService { LF
7 | LF
8 | getBooks() { LF
9 | return Promise.resolve(BOOKS); LF
10 | } LF
11 | }
```

```
1 | ... LF
2 | import { BookService } from './book.service'; LF
3 | LF
4 | ... LF
5 | export class BookComponent implements OnInit { LF
6 | LF
7 | books: Book[]; LF
8 | LF
9 | constructor(private bookService: BookService) {} LF
10 | LF
11 | getBooks() { LF
12 | this.bookService.getBooks().then(books => this.books = books); LF
```

- Add the following code in **book.service.ts**
  - Line 6: @Injectable() decorator makes the class as service which can be injected into components of an application
  - Line 9-11: getBooks() method returns Books data as a promise object. Promise is a javascript object which returns data asynchronously.
- Add the following code in **book.component.ts** file
  - Line 2: Imports BookService class into a component
  - Line 9: Inject BookService class using a constructor
  - Line 11: Invokes getBooks() method from BookService class which in turn returns a promise type of response. Promise has then() method which will be invoked when promise response is received from service class. then() has success callback function where the response received will be passed to the parameter books which in turn assigns it to the local variable books

## Example - services

```
1 <h2>My Books</h2> LF
2 <ul class="books"> LF
3 <li *ngFor="let book of books"> LF
4 {{book.id}} {{book.name}} LF
5 LF
6
```

### book.component.html

Line 4 - 6: ngFor iterates on books array and displays book id and name on the page



## Injecting a Service

- There are two steps to inject a service.
  - Add providers property in the module class so that service class is available to the entire application.
    - **Syntax:** providers: [MyService]
    - If service is needed only for specific component(s), it can be added in those component classes instead of module.
  - Add a constructor in a component class with Service class as argument.
    - **Syntax:** constructor(private service: MyService){ }
    - MyService will be injected into the component through constructor injection by the framework

## RxJS

- RxJS
  - Reactive Extensions for JavaScript (RxJS) is a third party library used by Angular team.
  - RxJS is a reactive streams library used to work with asynchronous streams of data.
  - Observables, in RxJS, are used to represent asynchronous streams of data. Observables are more advanced version of Promises in JavaScript
- Why RxJS Observables?
  - Angular team has recommended Observables for asynchronous calls because of the following reasons:
  - Promises emit a single value whereas observables (streams) emit many values
  - Observables can be cancellable where Promises are not cancellable. If any http response is not required, observables allows us to cancel the subscription whereas promises execute either success or failure callback even if we don't need the result
  - Observables support functional operators such as map, filter, reduce etc.,

## Server Communication using HttpClient

- Most front-end applications communicate with backend services using HTTP Protocol
- When we make calls to an external server, we want our user to continue to be able to interact with the page. That is, we don't want our page to freeze until the HTTP request returns from the external server. So, all HTTP requests are asynchronous.
- HttpClient from @angular/common/http is used to communicate with backend services.
- Additional benefits of HttpClient include testability features, typed request and response objects, request and response interception, Observable apis, and streamlined error handling.
- We need to import **HttpClientModule** from @angular/common/http in the module class to make http service available to the entire module. Import HttpClient service class into a component's constructor. We can make use of http methods like get, post, put and delete.
- JSON is the default response type for HttpClient

## Example

```
2 import { HttpClientModule } from '@angular/common/http';
3 ...
4
5 @NgModule({
6 imports: [BrowserModule, HttpClientModule],
7 ...
8 })
```

```
1 import { Injectable } from '@angular/core';
2 import { HttpClient } from '@angular/common/http';
3 import { Observable } from 'rxjs';
4 import { catchError, tap } from 'rxjs/operators';
5
6 import { Book } from '../book';
7
8 @Injectable()
9 export class BookService {
10
11 private booksUrl = './assets/books.json';
12
13 constructor(private http: HttpClient) {}
14
15 getBooks(): Observable<Book[]> {
16 return this.http.get<Book[]>(this.booksUrl).pipe(
17 tap(data => console.log('Data fetched: '+JSON.stringify(data))),
18 catchError(this.handleError));
19 }
```

- Add HttpClientModule to the **app.module.ts** to make use of HttpClient class.
  - Line 2: imports HttpClientModule from @angular/common/http module
  - Line 6: Includes HttpClientModule class to make use of Http calls
- Add getBooks() method to BookService class in **book.service.ts** file as shown below
  - Line 2: Imports HttpClient class from @angular/common/http module.
  - Line 3: Imports Observable class from rxjs module
  - Line 4: Imports rxjs operators
  - Line 11: Stores the json file path in a variable called booksUrl
  - Line 13: Injects HttpClient class into service class
  - Line 16-18: Makes an asynchronous call (ajax call) by using get() method of HttpClient class.
  - Line 18: handleError is error handling method which throws the error message back to the component

## Server Communication using HttpClient

- Error handling
- What happens if the request fails on the server, or if a poor network connection prevents it from even reaching the server?
- There are two types of errors that can occur. The server might reject the request, returning an HTTP response with a status code such as 404 or 500. These are error responses.
- Or something could go wrong on the client-side such as network error that prevents the request from completing successfully or an exception thrown in an RxJS operator. These errors produce JavaScript `ErrorEvent` objects.
- `HttpClient` captures both kinds of errors in its `HttpErrorResponse` and we can inspect that response to find out what really happened.
- We need to do error inspection, interpretation and resolution in service not in the component.

## Example – error handling

```
4 import { Observable } from 'rxjs';
5 import { HttpResponse } from '@angular/common/http';
6
7 import { Book } from './book';
8
9 @Injectable()
10 export class BookService {
11
12 private booksUrl = './assets/books.json';
13
14 // ...
15
16 private handleError(err: HttpResponse) {
17 let errMsg: string = '';
18 if (err.error instanceof Error) {
19 // A client-side or network error occurred. Handle it accordingly.
20 console.log('An error occurred:', err.error.message);
21 errMsg = err.error.message;
22 } else {
23 // The backend returned an unsuccessful response code.
24 // The response body may contain clues as to what went wrong,
25 console.log('Backend returned code ${err.status}');
26 errMsg = err.error.status;
27 }
28 }
29 }
```

- Add the following error handling code in **book.service.ts** file
  - Line 5 : HttpResponse module class should be imported to understand the nature of error thrown
  - Line 18-21 : An instance of Error object will be thrown if any network or client side error is thrown
  - Line 25-26 : Handling of errors due to unsuccessful response codes from backend

## Example – error handling

```
7 constructor(private bookService: BookService) {} LF
8 LF
9 getBooks() { LF
10 this.bookService.getBooks().subscribe(LF
11 books => this.books = books, LF
12 error => this.errorMessage = <any>error); LF
13 } LF
14 LF
```

```
1 ... LF
2 <ul class="books"> LF
3 <li *ngFor="let book of books"> LF
4 {{book.id}} - {{book.name}} LF
5 LF
6 LF
7 LF
8 <div class="error" *ngIf="errorMessage">{{errorMessage}}</div>
```

- Modify the code in **book.component.ts** file as shown below
  - Line 7: Inject the BookService class into the component class through constructor
  - Line 9-13: Invokes the service class method getBooks() which makes an http call to books.json file and the response is returned.
- **book.component.html**
  - Line 2-6 : Displays books details
  - Line 8: Displays error message when http get operation fails

## Server Communication using HttpClient

- Retrying Http Requests
  - Few errors can be momentary and are most unlikely to repeat . Such errors could be cleared up on making the same call few seconds later.
  - Most of these errors might occur when dealing with an external source like a database or web service which can have network or other temporary issues .
  - Such requests can be mitigated by using retry function.
- Http Client Full Response
  - Http response body may not return all the data we may need. Sometimes servers return special headers or status codes to indicate certain conditions that are important to the application workflow.
  - We can fetch the full response using observe option in HttpClient.
- Interceptors
  - Interceptors is one of the major feature of @angular/common/http module.They are placed in between client and backend .
  - They are used for transforming the http requests that are supposed to be sent to backend and vice versa. In addition, they can also be used for sending headers.
  - Interceptors use clone() property to duplicate the requests thereby making it mutable because requests are immutable in nature.
  - They are used mostly for making minor changes to requests/responses for authentication, caching behavior and XSRF protection .



## Example - Retrying Http Requests

- Modify **book.service.ts** file used in HttpClient demo as shown below

```
3 import { catchError, tap, retry } from 'rxjs/operators';
4 import { Observable } from 'rxjs';
5 import { HttpResponse } from '@angular/common/http';
6
7 import { Book } from './book';
8
9 @Injectable()
10 export class BookService {
11
12 private booksUrl = './assets/nothing.json';
13
14 constructor(private http: HttpClient) {}
15
16 getBooks(): Observable<Book[]> {
17 return this.http.get<Book[]>(this.booksUrl).pipe(
18 retry(3),
19 tap(data => console.log('fetched Data from json')),
20);
21 }
22 }
```

```
Angular is running in the development mode. Call
enableProdMode() to enable the production mode.
core.js:3562
✖ ▶ GET http://localhost:4200/assets/nothing.json 404 (Not Found) zone.js:2933
✖ ▶ GET http://localhost:4200/assets/nothing.json 404 (Not Found) zone.js:2933
✖ ▶ GET http://localhost:4200/assets/nothing.json 404 (Not Found) zone.js:2933
✖ ▶ GET http://localhost:4200/assets/nothing.json 404 (Not Found) zone.js:2933
Backend returned code 404
book.service.ts:28
>
```

## Example - Http Client Full Response

```
10 ...getBooks(): Observable<HttpResponse<Book[]>> {
11 ...return this.http.get<Book[]>(this.booksUrl, { observe: 'response' }).pipe(
12 ...tap(books => console.log(books.headers.get('Date'))),
13 ...catchError(this.handleError))
```

```
11 ...return this.http.get<Book[]>(this.booksUrl, { observe: 'response' }).pipe(
12 ...tap(books => console.log(books.headers.get('Date'))),
13 ...catchError(this.handleError))
```



- Modify getBooks() method in **book.service.ts** file as shown below
  - Line 11-12 : Headers are accessed by requesting response from Observe option during HttpClient get() call. Here we are accessing an header named 'Date'. We should specify the type parameter as HttpResponse to get the full http response.
- Modify getBooks() method in **book.component.ts** file as shown below
  - Line 18 : Accessing books data with response.body property

## Example - Interceptors

- For this, add the following code in **app.module.ts** file
  - Line 10-13: Wires-up **Interceptor1** into the application by importing **HTTP\_INTERCEPTORS** token in the app module .
- Create a new file called **book.interceptor.ts** and add the following code
  - Line 5-6 : The interceptor class should be injectable and define an intercept method to implement **HttpInterceptor** . The method takes two arguments , **req** is the request object and should be of type **HttpRequest** and **next** should be the handler that is of type **HttpHandler** , that uses the **handle** method to return **HttpEvent** observable type.
  - Line 10-12 :As **Http Requests** are immutable , we need to clone them to modify and then return the modified response . This interceptor changes every occurrence of **HTTP** in the **httpRequest** to **HTTPS** and sends a header token along with the request .

```
7 @NgModule({
8 imports: [BrowserModule, HttpClientModule],
9 declarations: [AppComponent, BookComponent],
10 providers: [BookService, {
11 provide: HTTP_INTERCEPTORS,
12 useClass: Interceptor1,
13 multi: true,
14 }],
15 bootstrap: [AppComponent]
```

```
5 @Injectable()
6 export class Interceptor1 implements HttpInterceptor {
7
8 intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
9
10 const authReq = req.clone({
11 headers: req.headers.set('Authorization23', 'Password')
12 });
13 return next.handle(authReq);
14 }
15 }
```

## Example - Interceptors

The screenshot shows a web application titled "My Books" with a list of 10 items. The network inspector is open, showing a list of requests. The selected request is for "books.json /assets". The headers tab is active, displaying various request headers.

**My Books**

- 1 HTML 5
- 2 CSS 3
- 3 Java Script
- 4 Ajax Programming
- 5 jQuery
- 6 Mastering Node.js
- 7 Angular JS 1.x
- 8 ng-book 2
- 9 Backbone JS
- 10 Yeoman

**Network Inspector**

**Name**

- books.json /assets
- info?t=1517923580052 /sockjs-node

**Headers**

**Accept-Encoding:** gzip, deflate, br  
**Accept-Language:** en,en-IN;q=0.9,en-US;q=0.8,fr;q=0.7  
**Authorization23:** Password  
**Connection:** keep-alive  
**Host:** localhost:4200  
**If-None-Match:** W/"1b2-4FTs9zb8oAtUBzh3SqrHeIRm6g"  
**Referer:** http://localhost:4200/  
**User-Agent:** Mozilla/5.0 (Windows NT 6.3; Win64; x64) AppleWebKit/537.36 (KH

## Server Communication using HttpClient

- Handling Non-JSON data in HttpClient
  - If the server is returning an non-json response , we need to tell the HTTPClient to expect a textual response using response object as the default response type of http client is of JSON type
  - The observable returned will be of type string if the response is of type text

## Example - Handling Non-JSON data in HttpClient

```
10 ...getBooks(): Observable<string>={ LF
11 ...return this.http.get(this.txtUrl, { responseType: 'text' }); LF
12 ...pipe(tap(data => console.log(data.length)), LF
13 ...catchError(this.handleError)); LF
```

```
1 | Welcome to demo on fetching textual response. Book Id is 1 and book name is Angular.
```

```
19 ...this.bookService.getBooks().subscribe(LF
20 ...data=>this.msg = data, LF
21 ...error=>this.errorMessage = <any>error); LF
```

```
1 <h2>Data from text file:</h2> LF
2 LF
3 {{msg}} LF
4 LF
5 <div class="error" *ngIf="errorMessage">{{errorMessage}}</div>
```

- Modify the code in **book.service.ts** file as shown below
  - Line 11 : responseType token changes the default json response to Textual response which returns an observable of type string.
- Create **sample.txt** file under assets folder and add the following content in it.
- Modify the code in **book.component.ts** file as shown below
  - Line 20: Response returned to the component is assigned to local property called 'msg'
- Modify **book.component.html** file as shown below
  - Line 3: Displays the text response returned

## Example - Handling Non-JSON data in HttpClient

**Data from Text File :**

Welcome to demo on fetching textual response. Book Id is 1 and book name is Angular

Angular is running in the development mode. Call enableProdMode() to enable the production mode.

91 core.js:3675

book.service.ts:16

## Routing

- Routing is navigation between multiple views in a single page
- Routing allows us to express some aspects of the application's state in the URL. We can build the full application without ever changing the URL.
- Why Routing?
  - Routing allows us to
  - Maintain the state of the application
  - Implement modular applications
  - Implement the application based on the roles



## Routing

- Configuring Router
  - Angular uses Component Router to implement routing
  - We should add a <base> tag to the head tag in our HTML page to tell router how to compose navigation URLs
  - **Syntax:** <base href="/">
  - Angular component router belongs to @angular/router module. To make use of routing, we should import Routes, RouterModule classes
  - We need to configure the routes and the router will look for a corresponding route when a browser url is changed
  - Routes is an array which contains all the route configurations. Then we need to pass this array to the RouterModule.forRoot() function in the application bootstrapping function

## Example

```
2 import { RouterModule, Routes } from '@angular/router';
3
4 import { BookComponent } from './book/book.component';
5 import { DashboardComponent } from './dashboard/dashboard.component';
6 import { BookDetailComponent } from './book-detail/book-detail.component';
7
8 const appRoutes: Routes = [
9 { path: 'dashboard', component: DashboardComponent },
10 { path: '', redirectTo: '/dashboard', pathMatch: 'full' },
11 { path: 'books', component: BookComponent },
12 { path: 'detail/:id', component: BookDetailComponent }
13];
14
15 @NgModule({
16 imports: [
17 RouterModule.forRoot(appRoutes)
```

- Add the following in **app-routing.module.ts** which is created under app folder
  - Line 2: Imports Routes and RouterModule classes
  - Line 8-13: Configure the routes where each route should contain the path to navigate and the component class which has to be invoked for a specific path. We need to provide route configuration for default path i.e., path:" and redirect it to the a specific route using redirectTo option. pathMatch is required if redirectTo option is used which specifies how the given path should match. Here pathMatch:full tells Router to match the given path completely. pathMatch has another value called 'prefix' where it checks if the path begins with the given prefix. The path 'detail/:id' where we specified the route parameter id which will receive different values based on the book selected
  - Line 17: Pass the appRoutes array to forRoot method of RouterModule class to configure with the Router and add it to the imports property

## Example

```
12 |
13 @NgModule({
14 imports: [BrowserModule, HttpClientModule, FormsModule, AppRoutingModule],
15 declarations: [AppComponent, BookComponent, DashboardComponent, BookDetailComponent],
16 providers: [BookService],
17 bootstrap: [AppComponent]
18 })
19 export class AppModule { }
```

- Now configure Router module to our NgModule imports in **app.module.ts** to make it available to the entire application
  - Line 13: Imports AppRoutingModule from routing module file
  - Line 16: Add AppRoutingModule class to the imports property

## Routing

- After configuring the routes, the next step is to decide how to navigate. Navigation will happen based on some user action
- To navigate programmatically, we can use `navigate()` method of Router class. Inject the router class into the component and invoke navigate method as shown below on like clicking a hyperlink etc.,  
`this.router.navigate([url, parameters])`
  - url is the route path to which we want to navigate
  - Parameters are the route values passed along with the url
- Route Parameters
  - Parameters passed along with URL are called route parameters.
- Accessing Route Parameters
  - To access route parameters, use `ActivatedRoute` class

## Example – router link

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

```
1 <h1>{{title}}</h1>
2 <nav>
3 <a [routerLink]='["/dashboard"]' routerLinkActive="active">Dashboard
4 <a [routerLink]='["/books"]' routerLinkActive="active">Books
5 </nav>
6 <router-outlet></router-outlet>
```

- **app.component.ts**
- **app.component.html**
  - Line 3-4: Create hyperlinks and a routerLink directive and specify the paths to navigate. Here, if user clicks on Dashboard, it will navigate to /dashboard. routerLinkActive applies the given css class to the link when it is clicked to make it look as active link(active is a css class defined in app.component.css which changes the link color to blue in this case)
  - Line 6 : <router-outlet> is the place where the output of the component associated with the given path will be displayed. For example, if user click on Books, it will navigate to /books which will execute BooksComponent class as mentioned in the configuration details and the output will be displayed in the router-outlet class

## Example – route parameters

```
1 import { Component, OnInit } from '@angular/core';
2 import { Router } from '@angular/router';
3
4 import { Book } from '../book/book';
5 import { BookService } from '../book/book.service';
6
7 @Component({
8 selector: 'app-dashboard',
9 templateUrl: './dashboard.component.html',
10 styleUrls: ['./dashboard.component.css']
11 })
12 export class DashboardComponent implements OnInit {
13
14 books: Book[] = [];
15 constructor(
16 private router: Router,
17 private bookService: BookService) {}
18
19 ngOnInit() {
20 this.bookService.getBooks().
21 .subscribe(books => this.books = books.slice(1, 5));
22 }
23 gotoDetail(book: Book) {
24 this.router.navigate(['/detail', book.id]);
25 }
26 }
```

- Add the following code in **dashboard.component.ts** file
  - Line 2: Import Router class from @angular/router module
  - Line 16: Inject into the component class through a constructor
  - Line 24: this.router.navigate() method is used to navigate to a specific URL programmatically. Navigate() method takes two arguments- first one is the path to navigate and the second one is the route parameter value to pass. Here the path will be detail/<book\_id>

## Example – route parameters

```
1 import { ActivatedRoute, ParamMap } from '@angular/router';
2 import { switchMap } from 'rxjs/operators';
3
4
5 import { Book } from '../book/book';
6 import { BookService } from '../book/book.service';
7
8 @Component({
9 selector: 'app-book-detail',
10 templateUrl: './book-detail.component.html',
11 styleUrls: ['./book-detail.component.css']
12 })
13 export class BookDetailComponent implements OnInit {
14
15 book: Book;
16 error: any;
17 sub: any;
18
19 constructor(private bookService: BookService, private route: ActivatedRoute) {}
20
21 ngOnInit() {
22 this.sub = this.route.paramMap.switchMap((params: ParamMap) => {
23 this.bookService.getBook(+params.get('id'))
24 }, subscribe((book: Book) => this.book = book);
```

- Add the following code in **book-detail.component.ts** file
  - Line 2: Imports ActivatedRoute class to access route parameters
  - Line 19: Injects ActivatedRoute class into the component class through a constructor
  - Line 22-24: ActivatedRoute class has a paramMap observable method which holds the route parameters. It has switchMap() method to process the route parameters. ParamMap has get() method to fetch a specific parameter value

## Route Guards

- In Angular application, user can navigate to any url directly. That's not the right thing to do always.
- Consider the following scenarios
  - User must login first to access a component
  - User is not authorized to access a component
  - User should fetch data before displaying a component
  - Pending changes should be saved before leaving a component
- We should go for **route guards** to handle these scenarios
  - A guard's return value controls the behavior of router
    - If it returns true, the navigation process continues
    - If it returns false, the navigation process stops
- Angular has canActivate interface which can be used to check if a user is logged in to access a component
- We need to override canActivate() method in the guard class as shown below



## Example

- Add the following code to **login.component.html** file
  - Line 2: div tag will render error message for incorrect credentials
  - Line 4-8: A reactive form with two fields username and password is displayed

```
1 ... LF
2 <div *ngIf="invalidCredentialMsg" style="color:red">{{invalidCredentialMsg}}</div>
 LF
3 <div style="position:relative;left:20px"> LF
4 -> <form [formGroup]="loginForm" (ngSubmit)="onFormSubmit()"> LF
5 -> -> <p>User Name <input formControlName="username"></p> LF
6 -> -> <p>Password <input type="password" formControlName="password" style="position:relative;left:10px"></p>
7 -> -> <p><button type="submit">Submit</button></p> LF
8 -> </form> LF
9 </div>
```

## Example

```
6 onFormSubmit() { LF
7 let uname = this.loginForm.get('username').value; LF
8 let pwd = this.loginForm.get('password').value; LF
9 this.loginService.isUserAuthenticated(uname, pwd).subscribe(LF
10 authenticated => { LF
11 if (authenticated) { LF
12 this.router.navigate(['/books']); LF
13 } else { LF
14 this.invalidCredentialMsg = 'Invalid Credentials. Try again.';
15 } LF
16 } LF
```

```
1 export class User { LF
2 .. constructor(public userId:number, public username:string, public password:string) { }
3 }
```

- Add the following code to **login.component.ts** file
  - Line 6: onFormSubmit() method is invoked when submit button is clicked in Login Form
  - Line 7-8: Fetching username and password values
  - Line 9: Invoking isUserAuthenticated method of LoginService class which will check for the validity of username and password values and returns a Boolean value
  - Line 11-15: If the response is true, it will navigate to BooksComponent else assigns error message to invalidCredentialMsg property
- Add the following code to **user.ts** file
  - Line 1-3: A User model class with three properties userId, username and password is created

## Example

```
1 ...
2 import { User } from './user';
3
4 const USERS = [
5 ...new User(1, 'user1', 'user1'),
6 ...new User(2, 'user2', 'user2')
7];
8 let usersObservable = Observable.of(USERS);
9
10 @Injectable()
11 export class LoginService {
12 ...
13 ...private isloggedIn: boolean = false;
14 ...
15 ...getAllUsers(): Observable<User[]> {
16 ...return usersObservable;
17 }
18 ...
19 ...isUserAuthenticated(username: string, password: string): Observable<boolean> {
20 ...return this.getAllUsers().
21 ...map(users => {
22 ...let user = users.find(user => (user.username === username) && (user.password === password));
23 ...if (user) {
24 ...this.isloggedIn = true;
25 } else {
26 ...this.isloggedIn = false;
27 }
28 ...return this.isloggedIn;
29 });
30 }
31 ...
32 ...isUserLoggedIn(): boolean {
33 ...return this.isloggedIn;
34 }
35 }
```

- Add the following code to **login.service.ts** file
  - Line 2: Imports User model class
  - Line 4-7: Creates an array called USERS of type User
  - Line 8: Converts USERS array as an observable type
  - Line 15-17: getAllUsers() method returns usersObservable array
  - Line 18: isUserAuthenticated method takes username and password values as inputs and return a Boolean value of type Observable
  - Line 19-28: Invokes getAllUsers() methods which returns an observable array. After receiving it, it will find the entered credentials exist in the array or not. If user exists, assigns true value to isloggedIn property otherwise false value to it
  - Line 32-34: isUserLoggedIn() method returns the value of isloggedIn which we will use in LoginGuardService class

## Example

```
3 export class LoginGuardService implements CanActivate {
4
5 constructor(private loginService: LoginService, private router: Router) {}
6
7 canActivate(): boolean {
8 if (this.loginService.isUserLoggedIn()) {
9 return true;
10 }
11 this.router.navigate(['/login']);
12 return false;
13 }
```

- Create another service class called **login-guard.service** and add the following code
  - Line 3: Inherits CanActivate interface to LoginGuardService class
  - Line 7: Overrides canActivate() method
  - Line 8-12: Invokes isUserLoggedIn method from LoginService class which returns a Boolean value representing whether a user is logged in or not. If user logs in, canActivate returns true otherwise navigates to login component asking user to login first to access BooksComponent

## Example

```
1 ... LF
2 const loginRoutes: Routes = [LF
3 - - - - { LF
4 - - - - path: '', LF
5 - - - - component: LoginComponent LF
6 - - - - } LF
7]; LF
8 LF
9 @NgModule({ LF
10 - - imports: [RouterModule.forChild(loginRoutes)], LF
11 - - exports: [RouterModule] LF
12 - - }) LF
```

```
1 ... LF
2 import {LoginGuardService} from './login/login-guard.service'; LF
3 import { LoginComponent } from './login/login.component'; LF
4 LF
5 @NgModule({ LF
6 - - ... LF
7 - - providers: [BookService, LoginService, LoginGuardService], LF
8 - - ... LF
9 }) LF
10 export class AppModule { }
```

- Add the following code in **login-routing.module.ts**
  - Line 2-7: Creates a routes array and maps empty path to LoginComponent
  - Line 10: Binds it with RouterModule using forChild method
- Add the following code in **app.module.ts**
  - Line 2-3: Imports LoginGuardService class and LoginComponent
  - Line 7: Binds LoginGuardService class to providers property

## Asynchronous Routing

- When an Angular application has lot of components, it will increase the size of the application. In such scenario, at some point, application takes lot of time to load
- To overcome this problem, we can go for asynchronous routing i.e, we need to load modules lazily only when they are required instead of loading them at the beginning of the execution
- Lazy Loading has following benefits:
  - Modules are loaded only when requested by the user
  - We can speed up load time for users that only visit certain areas of the application
- Lazy Loading Route Configuration:
  - For modules to be lazily loaded, create a separate routing configuration file for that module and map empty path to the component of that module

## Example - Lazy Loading Route Configuration

```
6 | LP
7 | const bookRoutes: Routes = [LP
8 | { LP
9 | path: '', LP
10 | component: BookComponent, LP
11 | canActivate: [LoginGuardService] LP
12 | } LP
13 |]; LP
14 |
```

```
8 | const appRoutes: Routes = [LP
9 | { path: 'dashboard', component: DashboardComponent }, LP
10 | { path: '', redirectTo: '/login', pathMatch: 'full' }, LP
11 | { path: 'books', loadChildren: './book/book.module#BookModule' }, LP
12 | { path: 'detail/:id', component: BookDetailComponent }, LP
13 | { path: 'login', component: LoginComponent } LP
14 |]; LP
15 |
```

- In the example in the previous concept, consider BookComponent. If we want to load it lazily, create **book-routing.module.ts** file and map empty path to BookComponent(Line 9-10)
- In the root routing configuration file **app-routing.module**, bind 'book' path to the BookModule using loadChildren property as shown below
  - Line 11: Binds books path to BookModule using loadChildren property

## Example - Lazy Loading Route Configuration

```
0
7 @NgModule({
8 imports: [CommonModule, BookRoutingModule],
9 declarations: [BookComponent]
10 })
11 export class BookModule { }
```

- Create **book.module.ts** file and add the following code
  - Line 7: Adds BookRoutingModule class to imports array
  - Line 8: Adds BookComponent to the declarations property



**Thank You**