Angular Tutorial

1. Introduction

Angular is an open-source framework written in typescript, to build dynamic and single page client side applications using HTML and Typescript. The very first version of angular i.e **AngularJS** was built in 2010 and it was known as the **"Golden Child"** among such JavaScript frameworks. It was later completely re written and was introduced as *Angular 2* Written in TypeScript.

- **Why most developers prefer TypeScript for Angular?**

  TypeScript is Microsoft's extension for JavaScript which supports object-oriented features and has a 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

- **What is Single Page Applications?**

  A Single Page Application (SPA) is a web application that interacts with the user by dynamically redrawing any part of the UI without requesting an entire new page from the server. Angular helps to create SPAs which will dynamically load contents in a single HTML file, giving the user an illusion that the application is just a single page. All desktop apps are SPAs in the sense that only the required area gets changed based on user requests.

2. Installation

- Make sure that "node" is installed in your system or not. If not please do it first, because without node we can't able to install Angular.
- After that, the latest angular version can be installed globally using the below command:
  - npm install -g @angular/cli
- Once installed, the Angular CLI version can be checked using the below command:
  - ng v

Angular CLI is a command-line interface tool to build Angular applications. It makes application development faster and easier to maintain. Using CLI, you can create projects, add files to it and can perform development tasks such as testing, bundling and deployment of applications.

Commands and purposes used in Angular CLI

Command	Purpose
npm install -g @angular/cli	Installs Angular CLI globally
ng new <project name>	Creates a new Angular application

ng serve --open (or) ng s -o	Builds and runs the application on lite-server and launches a browser
ng generate <type_of_file_name> <name_of_the_file>	Creates class, component, directive, interface, module, pipe and service
ng build	Builds the application

Creating an Angular application - Demo

Create an application with the name '**MyApp**' using the following CLI command.

Example:- ng new MyApp

Important files in an Angular application

File	Purpose
node_modules/	Node.js creates this folder and puts all modules listed in package.json inside it.
src/	All the 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.

Steps to run an Angular application

- To run the Angular application, first, go inside the project folder, for example, **MyApp**
  - Cd MyApp
- Then, execute the following command to run the Angular application.
  - ng serve –open

  Note: You can change the port by typing the port number as shown below:
    - ng serve --port 8183 –open

3. Components

An Angular app is built component-by-component. In Angular, Components will help us break the entire application into smaller chunks, keep all the pages and their corresponding business logic separately and load each one of them whenever user request the specific page.

To create a new component called hello using the following CLI command:

- ng generate component hello

Angular Components are based on MVVM architecture.

MVVM stands for Model, View and ViewModel. It is a software design pattern that increases the readability and maintainability of the code by dividing the code into three sections namely Model, view and ViewModel.

- **Model: -** A Model can be used to define the structure of an entity. If a model for login is to be defined, it should be as follows:

```
class Login {
  userName: string;
  password: string;
}
```

In Angular, Typescript classes or interfaces are used to define the model.

- **View:** - A View is the visual representation of an application. In Angular, View is written in HTML templates. Templates are a part of the component in Angular.

- **ViewModel:** - A ViewModel contains the business logic of the page. View and ViewModel are connected through data binding. That means any change in the ViewModel property will be reflected on View by default. In Angular, ViewModel is written as a Typescript class.

Component Structure

- ***app.component.ts*** - contains the business logic of app component
- ***app.component.html*** - contains the view (HTML) of app component that gets rendered when the component loads in browser
- ***app.component.css*** - contains all the custom styling used for various HTML elements in app component
- ***app.component.spec.ts*** - test cases for the app component should be written in this file

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

Line 3: Adding the *@Component* decorator to the class makes the class a component

Line 4: It specifies the selector name i.e. app-root, which must be used to render the attached view for app component

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

Line 6: It specifies the array of stylesheets that should be applied on the template i.e. *app.component.html*

Line 8: Every component is a class (AppComponent) and ***export*** keyword is used to access the class in another file

Line 9: A class property title is defined with value **'MyApp'**

4. Modules

Modules are used as a mechanism to group a set of components, directives, pipes and services that are related to a particular domain of our angular application.

Types of Modules:

1. **Root Module:**
   - The main module of the application.
   - Usually named AppModule.
   - Contains the root component and bootstraps the application.
2. **Feature Modules:**
   - Smaller, focused modules that encapsulate specific features.
   - Can be lazy-loaded to improve initial load performance.

Root Module

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

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

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

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

Line 3: Imports AppRoutingModule class to define the routes in the app

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 contain all user-defined components, directives, pipes classes to be used across the application. We have added our AppComponent class here

Line 8: Imports property should contain all module classes to be used across the application

Line 9: Providers property should contain all service classes. We will discuss services later in this course

Line 10: Bootstrap declaration should contain the root component to load. In this example, AppComponent is the root component which will be loaded in the HTML page

Note: - ***1.*** Every Angular application has at least one module: the root module.

2. Modules can be nested to create hierarchical structures.

3. Lazy loading feature modules can improve performance.

4. Effective module design is crucial for building scalable and maintainable Angular applications.

5. Templates

Template in Angular represents a view whose role is to display data and change the data whenever an event occurs. It separates view layer from the rest of the framework. We can change the view layer without breaking the application. The default language for templates is HTML.

We can define Templates(View) for the components of an angular application in two different ways i.e.

1. ***Inline Template*** - Inline Templates are defined within the *@Component* decorator using the property template. Inline Templates should be used when we have very few lines code as part of the template.

```
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-hello',
  template:`
          <p>
            Hello {{ courseName }}
          </p>
        `,
  styles: [`p{ color: blue; }`]
})
export class HelloComponent implements OnInit {
  courseName: string = "Angular";
  constructor() { }
  ngOnInit() {
  }
}
```

2. ***External Template -*** For Using External templates, we can create the view in a separate file with .html extension and then link it to the corresponding component by attaching it to *@Component* metadata with templateUrl property. External Templates should be used when we have large templates for the component.

app.component.html

```
<div class="customStyle">
        <h1> Welcome </h1>
        <h2> Course Name: {{ courseName }}</h2>
```

```
    </div>
```

app.component.css

```
    .customStyle {
        text-align: center;
        font-family: Arial, Helvetica, sans-serif;
        color: blueviolet;
    }
```

app.component.ts

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

6. Data Binding

In simple words, we can understand data binding as a communication. A communication that happens between the Typescript Code of the component and the template. It ensures that changes made in one are automatically reflected in the other, creating a dynamic and interactive user experience.

Data Direction according to Binding Type with Syntax

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

1. Property Binding

Property binding is a fundamental technique in Angular that allows you to set the properties of DOM elements from your component's TypeScript code.

Note: Interpolation is an alternative to property binding in many cases.

The syntax for property binding is as given below:

[propertyName] = "value"

or

bind-propertyName = "src"

Examples:

```
<button class="btn btn-outline-primary"
[disabled]="buttonStatus">Click Me!!</button>
```

** buttonStatus is a property initialised to true in component class*

```
<img [src] = 'imageUrl' />
        or
<img bind-src = 'imageUrl' />
```

* imageUrl is a property of component class that holds the source path of the image.

2. Property Binding

Attribute binding is a powerful technique in Angular that allows you to dynamically set attributes of DOM elements based on values from your component's TypeScript code.

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.

Syntax: <td [attr.colspan] = "2+3">Hello</td>

One thing to remember here is that, both *Property Binding* and *Interpolation* deal with the *DOM properties* and not *HTML attributes*. But there are certain HTML attributes such as colspan, area etc... which do not have the corresponding DOM properties.

Now here comes the problem. If these HTML attributes do not have their DOM properties so we cannot use either *Property Binding* or *Interpolation*, then how are we going to deal with such elements ?

In all such cases we need to use *Attribute Binding*. ***Properties Vs Attributes:***

1. Attributes are defined by HTML where as properties are defined by the Document Object Model i.e. **DOM**
2. **DOM** is initialised by the HTML attributes and after that the job of attributes is done.
3. We can change the values of properties but we cannot change the value of attributes

Example:

```
<input id="userName" type="text" value="john doe" />
-----------------------------------------------------------------
userName.getAttribute('value') //attribute value gives john doe
userName.value  // property value also gives john doe
```

3. Class Binding

Consider we have a button and we want to use some bootstrap classes such as **"btn btn-success"** for styling the button. How can we do this?
For Example: -

```
<button class="btn btn-success">Click Me!!</button>
```

In this case we are hard coding the bootstrap class used on the button. What if at a later stage of time we wanted to remove or change the bootstrap class used. We won't be able to do so in either of the above cases. There comes ***Class Binding*** for the rescue.

Using class binding, we can add or remove CSS class names from an element based on some condition which might change at a later instance.
Syntax: [class.classname]

Class binding syntax starts with prefix class, followed by a dot (optional) and the name of a CSS class

Example 1: Applying the CSS class based on Component class property

```
<div [class.mycss] = "isValid"> Class Binding </div>
```

Example 2: Changing the CSS class based on Component class property

```
/* app.component.ts */
applyClass = true;

<!-- app.component.html -->
<button [class]="applyClass ? 'btn btn-success' :'btn btn-primary'">Click Me!!</button>
```

4. Style Binding

Consider we have a button and we want to use some inline styling classes such as "background-color:'orange'" for styling the button. How can we do this ?
we can add an HTML attribute into the button element directly as shown:
<button style="background-color: orange;">Click Me!!</button>

In this case we are hard coding the background color used on the button. What if at a later stage of time we wanted to remove or change the background-color. We won't be able to do so in either of the above cases. There comes Style Binding for the rescue.

Using Style Binding, we can add or remove CSS properties from an element based on some condition which might change at a later instance.
Syntax: [style.css-property]
Style binding syntax starts with prefix style, followed by a dot (optional) and the name of a CSS property.

Example 1: Applying and Changing the CSS style property based on Component class property:

```
<button [style.background-color]="applyStyle ? 'orange' : 'greenyellow'" [style.color]="applyStyle ? 'black' : 'white'">
    Click Me!!
</button>
```

Example 2: We can apply and change multiple css style properties using t he below syntax:

```
<button [style]="applyStyle ? styleObj1 : styleObj2">
    Click Me!!
</button>
```

5. Event Binding

So far we have very effectively used Property Binding, Attribute Binding, Interpolation, Class Binding and Style Binding, but we only passed data to the template. What if we want to react to (User) Events?

User actions such as entering text in input boxes, picking items from lists, button clicks may result in a flow of data in the opposite direction: from an element to the component.
Syntax: (eventName) = "template expression/function call"
The syntax of angular event binding consists of two parts:

- A target event name within parentheses on the left of an equal sign such as click, input, mouseenter, etc...
- A quoted template statement on the right hand side of equal sign

Example 1: function call with parameter

```
<button class="btn btn-success"
(click)="onSubmit(userName, password)">
        Login
</button>
```

Example 2: Using Template expression

```
<button class="btn btn-success" (click)="name='john doe'">
        Click Me!!
</button>
```

6. Two Way Binding

Two-way data binding is a powerful technique in Angular that allows for automatic synchronization of data between a component's property and an input element in its template. This bidirectional flow of data ensures that changes made in either the component or the view are reflected in the other.
Syntax:

- The [(ngModel)] directive is used to establish two-way binding.
- It's a combination of property binding ([]) and event binding (()).
- It uses ***ngModel*** directive enclosed within [()] (banana in a box) as shown below:

```
[(ngModel)]
```

Example:

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

Data Flow:

- **Component to View:** When the component's property value changes, it's automatically updated in the input element's value.
- **View to Component:** When the user interacts with the input element and changes its value, the component's property is updated accordingly.

7. Directives

Directives are classes that add new behavior to DOM elements in Angular applications. They extend the capabilities of HTML by allowing you to create custom elements and attributes with specific functionalities.

In fact, component is also a directive which is helping us create a custom DOM element! Apart from this, we can also use directives in the form of HTML attributes.

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(an HTML view). These are the most common type of directives. They have a template and can encapsulate both the view and the logic.

Structural Directives

These directives change the DOM structure by adding, removing, or manipulating elements.
Syntax: *directive-name = expression
Angular has few built-in structural directives such as:

- ngIf: - conditionally renders elements.
  In our angular apps, many times we might want to display or remove an HTML element based on a condition. *ngIf directive is a strucutual directive that renders.

components or elements conditionally based on whether an expression is *true* or *false*.
Syntax: *ngIf = "expression"

- ngFor : - iterates over a list and creates elements for each item.
  *ngFor directive is used to iterate over a collection of data i.e., arrays
  Syntax: *ngFor = "expression"

- ngSwitch: - conditionally renders different content based on a value.
  *ngSwitch adds or removes elements into the DOM tree when their expressions match the switch expression. Its syntax contains two directives i.e. 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

```
export class AppComponent {
    myChoice: number = 0;
    nextChoice() {
        this.myChoice++;
    }
}
```

  app.component.html

```
<div [ngSwitch]="myChoice">
    <p *ngSwitchCase="1">First Choice</p>
    <p *ngSwitchCase="2">Second Choice</p>
    <p *ngSwitchCase="3">Third Choice</p>
    <p *ngSwitchCase="2">Second Choice
    Again</p>
    <p *ngSwitchDefault>Default Choice</p>
</div>
```

  *ngSwitch takes the value of the property "myChoice" and based on its value, it executes *ngSwitchCase. Paragraph elements will be added/removed from the DOM based on the value passed to the switch case.

Attribute Directives

Angular's attribute directives are used to modify the behavior or the appearance of some existing element, component, or some other directive. As the name suggests, these directives are utilized as attributes of elements and within templates, they resemble the HTML attributes. One of the most common examples is the ***ngModel*** directive.

Some other in built Attribute Directives are:

- **ngClass:** dynamically adds or removes CSS classes. ngClass allows us to dynamically set and change the CSS classes for a given DOM element. If we have *more than one* CSS class to apply, then we will go for ngClass.

  Syntax: [ngClass] = "expression"

  Example:

  app.component.ts

```
export class AppComponent {
    isBordered: boolean = true;
    isColor: boolean = true;
}
```

  app.component.html

```
<div [ngClass]="{bordered: isBordered, myColor:
isColor}">
    Border {{ isBordered ? "ON" : "OFF" }}
</div>
.bordered {
    border: 1px dashed black;
    background-color: #eee;
}
.myColor {
    color: blue;
}
```

- **ngStyle:** dynamically applies inline styles. If there are more than one css styles to apply, we can use **ngStyle** attribute.

  Syntax: [ngStyle] = "expression"

  Example:

  app.component.ts

```
export class AppComponent {
```

```
        colorName: string = 'red';
        fontWeight: string = 'bold';
        borderStyle: string = '1px solid black';
    }
```

app.component.html

```
    <p [ngStyle]="{
            color:colorName,
            'font-weight':fontWeight,
            borderBottom: borderStyle
    }">
        Demo for attribute directive ngStyle
    </p>
```

8. Custom Directives

We have already seen how we can use built in directives such as ***ngFor, *ngIf, ngStyle** and **ngClass** to change the appearance or behavior of a component/element in angular. But the way they behave is also predefined, like ****ngIf*** can be reused at multiple places to render some content when condition holds ***true*** and remove the same from DOM if the condition becomes ***false***.

Similar to the **built in** directives, we can also create our own such directives, also known as ***custom directives.*** Such directives can be used to change the appearance of the components as we need.

In simple words, Custom directives in Angular allow you to extend the HTML vocabulary with your own custom elements and attributes. This provides a powerful way to encapsulate reusable UI components and behaviors, making your Angular applications more modular and maintainable.

When can you use Custom Directives?

- Custom directives can be used for accessing and manipulating the DOM directly
- It can also be used for some reusable functionality which needs to be applied on other parts of the application as well
- We can use custom directives to define custom validations for the template driven forms

Types of Custom Directives

Angular supports three main types of custom directives:

A. Component Directives:

Create a new custom element with its own template and logic. Use the @Component decorator to define them.
Example: <app-my-component></app-my-component>

While Angular doesn't have a specific "Component Directive" type, you can achieve similar functionality by combining components and directives. Here are two primary approaches:

1. Using a Component as a Directive:

- **Create a Component:** Define a component with the desired template and logic.
- **Use the Component as a Directive:** Use the *ngComponentOutlet directive to dynamically insert the component's template into another template.

```
// my-component.component.ts
@Component({
  selector: 'app-my-component',
  template: `
    <p>This is my component!</p>
  `
})
export class MyComponent {}

// app.component.ts
@Component({
  selector: 'app-root',
  template: `
    <div *ngComponentOutlet="myComponent"></div>
  `
})
export class AppComponent {
  myComponent = MyComponent;
}
```

2. Using a Directive to Control Component Behavior:

- **Create a Directive:** Define a directive that interacts with a component's properties or methods.
- **Apply the Directive to a Component:** Use the directive's selector to apply it to the component's template.

```
// highlight.directive.ts
@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  @Input('appHighlight') highlightColor = 'yellow';

  constructor(private el: ElementRef) {}

  ngOnInit() {
    this.el.nativeElement.style.backgroundColor =
this.highlightColor;
  }
}

// my-component.component.ts
@Component({
  selector: 'app-my-component',
  template: `
    <p appHighlight>This text is highlighted.</p>
  `
})
export class MyComponent {}
```

B. Custom Attribute Directives:

Attribute directives are used to modify the behavior or appearance of an existing DOM element. Use the @Directive decorator to define them.
Example: <input [appHighlight]="color">

1. Import Necessary Modules:

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

2.Define the Directive:

```
@Directive({
  selector: '[appHighlight]' // Selector to identify the elements to apply
  the directive to
})
export class HighlightDirective {
  @Input('appHighlight') highlightColor = 'yellow'; // Input property to
  customize the highlight color

  constructor(private el: ElementRef) {}

  ngOnInit() {
    this.el.nativeElement.style.backgroundColor = this.highlightColor;
  }
}
```

3. Use the Directive in a Template:

```
<p appHighlight>This text will be highlighted.</p>
```

C. Custom Attribute Directives:

Structural directives are used to modify the DOM layout by adding, removing, or manipulating elements.

1. Import Necessary Modules:

```
import { Directive, TemplateRef, ViewContainerRef } from
'@angular/core';
```

2. Define the Directive:

```
@Directive({
  selector: '[appIf]'
})
export class IfDirective {
  constructor(private templateRef: TemplateRef<any>, private
viewContainer: ViewContainerRef) {}

  @Input('appIf') set condition(condition: boolean) {
    if (condition) {
      this.viewContainer.createEmbeddedView(this.templateRef);
    } else {
      this.viewContainer.clear();
```

```
    }
  }
}
```

4. Use the Directive in a Template:

```
<ng-template #myTemplate>
  <p>This content will be conditionally displayed.</p>
</ng-template>
<div *appIf="condition" [ngTemplateOutlet]="myTemplate"></div>
```

How to create Custom Directives?

Step-1: We can use the angular cli command as given below to generate a custom directive:

Syntax: ng generate directive custom-directive-name

Let's create a custom directive with name **button:**

```
ng generate directive button
```

This creates **two** files in the application folder **button.directive.ts** and **button.directive.spec.ts** for directive implementation and directive test cases respectively. Once the directive is created, it would also be added to the corresponding module **declarations array.**

The created **button.directive.ts** file will have default code as below:

```
1. import { Directive } from '@angular/core'
2.
3. @Directive({
4.   selector: '[appButton]',
5. })
6. export class ButtonDirective {
7.   constructor() {}
8. }
```

Line 1: Directive is imported from @angular/core library

Line 3: @Directive is the decorator that signifies the ButtonDirective class as a Directive

Line 4: The @Directive decorator has a single property i.e. selector, which should be used to use this custom directive with the HTML elements in angular components. Interestingly, the selector is enclosed within **square brackets ([]),** and hence it is called as Attribute selector which helps angular to identify all the elements having the attribute named **appButton.**

9. Pipes

Pipes are a powerful feature in Angular that allow you to transform/format the data before displaying it in the template. They provide a clean and reusable way to format numbers, dates, currencies, and other data types.

Using Pipes in Templates:

To use a pipe in a template, you add it after the expression you want to transform, using the pipe symbol (|).
Syntax: {{ expression | pipe }}
Example:

```
<p>Today's date is: {{ today | date }}</p>
<p>The price is: {{ price | currency }}</p>
```

Buit in pipes		
uppercase	lowercase	titlecase
currency	date	percent
slice	decimal	json
i18nplural	i18nselect	

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: {{ value | pipename [: parametervalue] }}

A pipe can also have multiple parameters as shown below:

```
{{ value | pipename [: parametervalue1 [: parametervalue2] ] }}
```

Note: In the above syntax [ ] brackets are optional and can be removed.

Example:

Number: This pipe can be used to format a number.

Syntax:

{{ expression | number [:digitInfo [:locale]] }}

Example:

{{ 25000 | number }} will display 25,000

{{ 25000 | number:'.3-5' }} will display 25,000.000

10. Custom Pipes

If we want to implement functionalities such as sorting, filtering, etc., we should go for custom pipes as there are no corresponding built-in pipes available.

We can create our own custom pipe by inheriting PipeTransform interface

PipeTransform interface has a 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 the second is a variable "arguments". We can have multiple arguments based on the number of parameters passed to the pipe. The transform method should return the final value.

Example:

Let us create a custom pipe called **salutation** which should add **Mr.** or **Ms.** based on the parameter passed.

- Create a pipe using the following CLI command

  ng generate pipe salutation

  This will create two files called *salutation.pipe.ts* to write custom pipe functionality and *salutation.pipe.spec.ts*. This command will also add the pipe to the root module to make it available to the entire module.

salutation.pipe.ts

```
@Pipe({
  name: 'salutation'
})
export class SalutationPipe implements PipeTransform {
  transform(value: any, args?: any): any {
   if(args == 'M'){
    return "Mr."+value;
   }
   else if(args == 'F'){
     return "Ms."+value;
   }
   else{
     return value
   }
  }
}
```

Line 1-3: @Pipe decorator creates a pipe with the name **salutation**

Line 4: Inherit PipeTransform interface for custom pipe

Line 6: Overrides the transform method of the PipeTransform interface to write the functionality. This method stores the value passed into the first argument called value and the parameters of the pipe into the second argument called args.

Line 7-15: Based on the second argument value, functionality is implemented.

app.component.html

```
{{ username | salutation : 'M' }}
```

11. Nested Components

A nested component is a component which is loaded into another component. The component where we load another component is called the container component or parent component. In Angular, components are self-contained units of UI that encapsulate their own HTML, CSS, and TypeScript logic. Imagine a component as a container. Nested components allow you to place one component (the "child" component) inside another component (the "parent" component).

Example:

Child Component (app-product-card.component.ts):

```
import { Component, Input } from '@angular/core';
@Component({
  selector: 'app-product-card',
  template: `
    <div class="card">
      <h3>{{ product.name }}</h3>
      <p>{{ product.price }}</p>
      <img src="{{ product.imageUrl }}">
    </div>
  `
})
export class ProductCardComponent {
  @Input() product: any; // Receive product data from parent
}
```

Parent Component (app-product-list.component.ts):

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-product-list',
  template: `
    <div>
      <h2>Product List</h2>
      <app-product-card *ngFor="let product of products"
[product]="product"></app-product-card>
    </div>
  `
})
export class ProductListComponent {
```

```
  products = [
    { name: 'Product A', price: 10, imageUrl: '...' },
    { name: 'Product B', price: 20, imageUrl: '...' },
    // ...
  ];
}
```

Passing data from container component to child component

The primary method for passing data from a parent component to a child component in Angular is using the ***@Input()*** decorator. We can use ***@Input()*** decorator in the child component on any property type like arrays, objects, etc.

Example:

1. Define Input Property in Child Component:

   In the child component's class, use the @Input() decorator to define a property that will receive the data from the parent.

   ```
   import { Component, Input } from '@angular/core';
   @Component({
     selector: 'app-child',
     template: `
       <p>Data from parent: {{ dataFromParent }}</p>
     `
   })
   export class ChildComponent {
     @Input() dataFromParent: string;
   }
   ```

2. Pass Data from Parent Component:

   In the parent component's template, use property binding (square brackets []) to pass the data to the child component's input property.

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

```
    <app-child [dataFromParent]="parentData"></app-child>`
})
export class ParentComponent {
  parentData = 'Hello from parent!';
}
```

- **@Input():** This decorator tells Angular that the dataFromParent property in the child component is an input property and can receive data from the parent.
- **Property Binding:** The [dataFromParent]="parentData" in the parent's template binds the value of the parentData property in the parent component to the dataFromParent input property of the child component.

Passing data from child component to container component

To pass data from a child component to its parent component in Angular, you use the **@Output()** decorator and the **EventEmitter** class.

Exampe:

1. Define Output Property in Child Component:
    - In the child component's class, use the @Output() decorator to define an output property.
    - Create an instance of the EventEmitter class for this output property.

```
import { Component, Output, EventEmitter } from '@angular/core';
@Component({
  selector: 'app-child',
  template: `
    <button (click)="sendDataToParent()">Send Data</button>
  `
})
export class ChildComponent {
  @Output() dataFromChild = new EventEmitter<string>();
  sendDataToParent() {
    this.dataFromChild.emit('Hello from child!');
  }
}
```

2. Subscribe to Output in Parent Component:
    - In the parent component's template, use property binding to bind the child component's output property to an event handler in the parent component.

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-parent',
  template: `
    <app-child (dataFromChild)="receiveData($event)"></app-
child>
    <p>Data from child: {{ receivedData }}</p>
  `
})
export class ParentComponent {
  receivedData: string;
  receiveData(data: string) {
    this.receivedData = data;
  }
}
```

    - **@Output():** This decorator indicates that the dataFromChild property is an output property that can emit data.
    - **EventEmitter:** This class allows the child component to emit events and data.
    - **emit():** The emit() method is used to send data to the parent component.
    - **Event Binding:** In the parent's template, (dataFromChild)="receiveData($event)" binds the dataFromChild output event to the receiveData() method in the parent component.
    - **$event:** The $event variable in the receiveData() method receives the data emitted by the child component.

12.Component Life Cycle

A component has a life cycle managed by Angular which consists of 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. It refers to the series of events that occur from its creation to its destruction. These

events provide opportunities to execute specific actions at different stages of a component's existence.

Key Lifecycle Hooks:

1. **Constructor:**
    - The first method called when a component instance is created.
    - Primarily used for dependency injection and initializing class properties.
    - **Note:** Avoid complex logic or data fetching here, as input properties are not yet available.
2. **ngOnChanges:**
    - Called whenever input properties are bound to the component and their values change.
    - Useful for reacting to changes in data passed from the parent component.
3. **ngOnInit:**
    - Called once after the first ngOnChanges() call.
    - A good place to perform one-time initialization tasks, such as fetching data from a service.
4. **ngDoCheck:**
    - Called repeatedly during change detection to check for changes that Angular's default change detection might miss.
    - Use with caution as it can impact performance.
5. **ngAfterContentInit:**
    - Called after the component's content has been initialized.
    - Useful for accessing and manipulating projected content.
6. **ngAfterContentChecked:**
    - Called after every check of the component's content.
7. **ngAfterViewInit:**
    - Called after the component's view and its children's views have been initialized.
    - Useful for accessing and manipulating the DOM.
8. **ngAfterViewChecked:**
    - Called after every check of the component's view and its children's views.
9. **ngOnDestroy:**
    - Called before the component is destroyed.

- A good place to clean up subscriptions, detach event listeners, or release resources.

Example:

```
import {
  Component,
  OnInit,
  OnChanges,
  SimpleChanges,
  DoCheck,
  AfterContentInit,
  AfterContentChecked,
  AfterViewInit,
  AfterViewChecked,
  OnDestroy
} from '@angular/core';

@Component({
  selector: 'app-lifecycle-demo',
  template: `
    <p>Name: {{ name }}</p>
  `
})
export class LifecycleDemoComponent implements
  OnInit,
  OnChanges,
  DoCheck,
  AfterContentInit,
  AfterContentChecked,
  AfterViewInit,
  AfterViewChecked,
  OnDestroy {

  @Input() name: string;
  private previousName: string;

  constructor() {
```

```
  console.log('Constructor');
}

ngOnChanges(changes: SimpleChanges) {
  console.log('ngOnChanges', changes);
  this.previousName = changes['name'].previousValue;
}

ngOnInit() {
  console.log('ngOnInit');
  // Perform one-time initialization tasks here
}

ngDoCheck() {
  console.log('ngDoCheck');
  // Perform custom change detection here
}

ngAfterContentInit() {
  console.log('ngAfterContentInit');
  // Access and manipulate projected content here
}

ngAfterContentChecked() {
  console.log('ngAfterContentChecked');
}

ngAfterViewInit() {
  console.log('ngAfterViewInit');
  // Access and manipulate the DOM here
}

ngAfterViewChecked() {
  console.log('ngAfterViewChecked');
}

ngOnDestroy() {
```

```
    console.log('ngOnDestroy');
    // Clean up subscriptions, detach event listeners, etc.
  }
}
```

13. Services

Services are a crucial part of Angular applications. They are designed to provide a way to share data and logic across different components in a reusable and maintainable manner. Services in Angular are implemented with the help of **Dependency Injection** (DI).

What is Dependency injection?

Dependency Injection (DI) is a powerful design pattern and a core feature of the Angular framework. It's a mechanism that allows components and services to receive their dependencies (other services, data, or configuration) without having to create them themselves.

Code without DI

```
class Engine
{

}
class Car
{
    engine: Engine;
    constructor()
    {
        this.engine = new Engine();
    }
}
```

Engine class instantiated inside car class

Code with DI

```
class Engine
{

}
class Car
{
    engine: Engine;
    constructor(engineObj: Engine)
    {
        this.engine = engineObj;
    }
}
```

Engine class instance is injected from external source

Key Characteristics of Angular Services:

- **Singletons:** By default, services are singletons, meaning only one instance of a service exists throughout the application. This ensures consistent behavior and efficient resource management.
- **Injectability:** Services can be injected into components, other services, or directives using Angular's dependency injection system. This allows for loose coupling between different parts of your application.

- **Testability:** Services are easily testable as they are independent units of functionality. You can write unit tests for services to ensure they work as expected in isolation.
- **Reusability:** Services encapsulate specific logic or data access, making them reusable across multiple components within your application. This avoids code duplication and improves maintainability.

Introduction to HttpClientModule

While building Angular applications, we might need the application to communicate with back-end services to fetch or persist data. This is done using **HttpClientModule.** All HTTP requests are asynchronous. 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.

The HTTP get, post, put and delete methods will automatically convert the received JSON data from the back-end server, to any desired type. JSON is the default response type for HttpClient.
The following statement is used to fetch data from a server.

this.http.get(url)

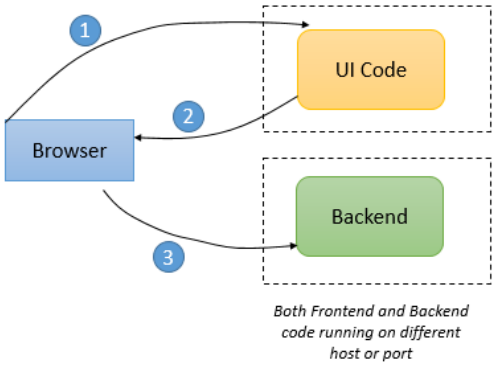
Introduction to RxJS Observables

What is an Observable?

In Angular, Observables are a powerful mechanism for handling asynchronous operations and data streams. They are a core part of the RxJS (Reactive Extensions for JavaScript) library, which is extensively used within the Angular framework.

```
export class DataService {
        constructor(private http: HttpClient) {}
        getData(): Observable<any> {
                return this.http.get('/api/data');
        }
}
```

Cross Origin Resource Sharing (CORS)



Both Frontend and Backend code running on different host or port

1. The browser sends a request to the UI code
2. The UI code is now loaded in the browser.
3. The UI code now asks the browser to send a request to the backend, which is running in a different host/port. This is a **cross origin** request. If the server does not allow such CORS (**Cross Origin Resource Sharing)** this will result in an error. The server has to indicate which Origins are allowed to request resource from it. Such a backend service is called as a CORS enabled service.

Retrying Http Requests using retry() of RxJS library

Some errors can be momentary and unlikely to repeat. Such errors may be resolved on simply making the same call a few seconds later. Most of these errors occur when dealing with an external source like a database or web service which can have a network or other temporary issues. Such errors can be mitigated by using the **retry** function of **RxJS library**.

This approach allows your application to handle transient network issues or server errors more gracefully by automatically retrying the HTTP request a specified number of times before giving up.

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';
import { Book } from './book';
import { retry } from 'rxjs/operators';
@Injectable()
export class BookService {
  private booksUrl = './assets/books.json';
  constructor(private http: HttpClient) { }
```

```
  getBooks(): Observable<Book[]> {
    return this.http.get<Book[]>(
      this.booksUrl
    ).pipe(
        retry(3));
  }
}
```

If the initial request to the server is not successful, *retry(3)* will try to connect to the URL three times.

Example: For entire service

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { catchError, tap } from 'rxjs/operators';
import { Observable } from 'rxjs/Observable';
import { HttpErrorResponse } from '@angular/common/http';
import { Book } from './book';
@Injectable()
export class BookService {
  private booksUrl = 'http://localhost:3000/books';
  constructor(private http: HttpClient) { }
  getBooks(): Observable<Book[]> {
   return  this.http.get<Book[]>(this.booksUrl);
  }
}
```

Data sharing between component to component using services ?

1.Create a Shared Service

- Create a new service using the Angular CLI: ng generate service shared/data
- This will create a file named `data.service.ts` in the `shared` folder.
- Define a property or a method in the service to hold or provide the data:
    - import { Injectable } from '@angular/core';
    - import { BehaviorSubject } from 'rxjs';
    - @Injectable({ providedIn: 'root' })
    - export class DataService {
    - private messageSource = new BehaviorSubject<string>('');
    - currentMessage = this.messageSource.asObservable();
    -

```
    constructor() { }

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

2. Component A (component-a.component.ts)

```
import { Component } from '@angular/core';
import { DataService } from './data.service';
@Component({
  selector: 'app-component-a',
  template: `
    <button (click)="sendMessage()">Send Message</button>
  `
})
export class ComponentA {
  constructor(private dataService: DataService) {}
  sendMessage() {
    this.dataService.changeMessage("Hello from Component A!");
  }
}
```

3. Component B (component-b.component.ts)

```
import { Component } from '@angular/core';
import { DataService } from './data.service';
@Component({
  selector: 'app-component-b',
  template: `
    <p>Message: {{ message }}</p>`
})
export class ComponentB {
  message: string = '';
  constructor(private dataService: DataService) {}
  ngOnInit() {
    this.dataService.currentMessage.subscribe(message => {
      this.message = message;
    });
  }
}
```

14. Template Driven Forms

Forms are a crucial part of web applications through which we get a majority of 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 forms

Template-driven forms use **ngForm** and **ngModel** directives to get information about the form and its controls and **ngSubmit** event to submit the forms.

- **ngForm:** Provides information about the current state of the form including a JSON representation of the form value and the validity state of the entire form
- **ngModel:** Provides two-way data binding between the view and component. It is also used to track the state and validity of the input field.
- ngSubmit: Fires an event specified by ngSubmit when the form is submitted.

Syntax:

```
<form #formRef = "ngForm" (ngSubmit) = "onSubmit()">
        <input class="form-control" name="controlname"
        [(ngModel)]="classVariable" #variable ="ngModel"/>
</form>
```

Here,

#formRef="ngForm" is a template variable that has an instance of *ngForm* or Angular from.

(ngSubmit) = "onSubmit()" : On submitting the form, onSubmit() method gets invoked.

[(ngModel)] = "classVariable": Binds the data entered in the input field of the form to the class variable specified in the corresponding **.ts** file

#variable = "ngModel": Another template variable which is bound to an instance of *ngModel.* This helps to track the state of the input field by checking if the user touched the control, if the value changed, or if the value became invalid.

Angular form validations

The below table lists the available keywords to track control state.

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

Angular also has built-in CSS classes to change the appearance of the control based on its state. Following are the CSS classes available
Built-In CSS classes to change the appearance based on State

State	Class if true	Class if false
Control has been visited	ng-touched	ng-untouched
Control's value is changed	ng-dirty	ng-pristine
Control's value is valid	ng-valid	ng-invalid

Example:

```
<div class="row">
  <div class="col-md-4 offset-md-4 text-center">
    <div class="card">
      <div class="card-header bg bg-info">
        <h5>User Login</h5>
      </div>
      <div class="card-body">
        <form #loginForm="ngForm" (ngSubmit)="verify()">
          <div class="form-group">
            <label for="uName">User Name:</label>
            <input type="text" id="uName" class="form-control" required
          [(ngModel)]="userName" name="userName"
              #uName="ngModel" />
            <div [hidden]="uName.valid || uName.pristine" class="alert alert-danger">
              User Name is required
            </div>
          </div>
          <div class="form-group">
```

```
          <label for="pwd">Password:</label>
          <input type="password" id="pwd" class="form-control" required
[(ngModel)]="password" name="password"
            #pwd="ngModel" />
          <div *ngIf="pwd.valid && pwd.untouched" class="alert alert-danger">
            Password is required
          </div>
        </div>
        <div class="form-group">
          <button class="btn btn-info" type="submit">Login</button>
        </div>
      </form>
      <div *ngIf="errorMessage!=null">
        <span class="error-message">{{errorMessage}}</span>
      </div>
      <div *ngIf="successMessage!=null">
        <span class="success-message">{{successMessage}}</span>
      </div>
    </div>
  </div>
 </div>
</div>
```

15. Reactive Forms

Reactive Forms in Angular provide a model-driven approach to managing form controls, their values, and validation states within your component class. This approach offers greater flexibility and control compared to template-driven forms, especially for complex forms. One advantage of working with form control objects directly is that value and validity updates are always synchronous and under our control.

Key Concepts:

- **FormControl:** Represents a single input field (e.g., text input, checkbox, select). It manages the value, validation status, and user interactions of that input.

- **FormGroup:** Represents a collection of FormControls. It aggregates the values of its child controls into a single object.
- **FormArray:** Represents an array of FormControls. Used when you need to handle an array of inputs dynamically.

How to Use Reactive Forms:

- **Import necessary modules:**

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

- **Create a FormGroup:**

```
constructor(private fb: FormBuilder) {}
myForm: FormGroup = this.fb.group({
  name: ['', Validators.required],
  email: ['', [Validators.required, Validators.email]],
  // ... other form controls
});
```

- **Bind form controls to template:**

```
<form [formGroup]="myForm" (ngSubmit)="onSubmit()">
  <div>
    <label for="name">Name:</label>
    <input type="text" id="name" formControlName="name">
    <div *ngIf="myForm.get('name')?.hasError('required') &&
myForm.get('name')?.touched">
      Name is required.
    </div>
  </div>
  <div>
    <label for="email">Email:</label>
    <input type="email" id="email" formControlName="email">
    <div *ngIf="myForm.get('email')?.hasError('required') &&
myForm.get('email')?.touched">
      Email is required.
    </div>
    <div *ngIf="myForm.get('email')?.hasError('email') &&
myForm.get('email')?.touched">
      Please enter a valid email address.
    </div>
  </div>
```

```
    <button type="submit">Submit</button>
  </form>
```

- **Handle form submission:**

```
  onSubmit() {
    if (this.myForm.valid) {
      // Handle form submission
      console.log(this.myForm.value);
    }
  }
```

Custom Validations

Custom validators in Angular Reactive Forms allow you to define specific validation rules beyond the built-in ones like required, email, minLength, etc. These custom rules can be tailored to your application's unique requirements.

Example:

```
...
export class RegistrationFormComponent implements OnInit {
 ...
  ngOnInit() {
    this.registerForm = this.formBuilder.group({
      firstName: ['', Validators.required],
      lastName: ['', Validators.required],
      address: this.formBuilder.group({
        street: [],
        zip: [],
        city: []
      }),
      email:['',validateEmail]
    });
  }
}
function validateEmail(c: FormControl) {
    let EMAIL_REGEXP = /^([a-zA-Z0-9_\-\.]+)@([a-zA-Z0-9_\-\.]+)\.([a-
zA-Z]{2,5})$/;
    return EMAIL_REGEXP.test(c.value) ? null : {
      emailError: {
        message: "Email is invalid"
```

```
    }
  };
}
```

16. Routings

Routing in Angular enables navigation between different views within a single-page application (SPA). It allows users to switch between components without reloading the entire page, creating a smoother and more interactive user experience.

How to Set Up Routing in Angular:

1. Create a Routing Module:

- Use the Angular CLI to generate a routing module:

```
ng generate module app-routing --flat --module=app
```

2. Define Routes in the Routing Module:

- Import RouterModule and Routes from @angular/router.
- Create an array of Routes objects, each defining a path and the corresponding component:

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { AboutComponent } from './about/about.component';
const routes: Routes = [
  { path: '', component: HomeComponent }, // Default route
  { path: 'about', component: AboutComponent },
  // ... other routes
];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

3. Import the Routing Module in app.module.ts:

```
import { AppRoutingModule } from './app-routing.module';
@NgModule({
  imports: [
    // ...
```

```
    AppRoutingModule
  ],
  // ...
})
export class AppModule { }
```

4. **Add a router-outlet in the app.component.html:**

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

5. **Create Navigation Links:**

```
<a routerLink="/">Home</a>
<a routerLink="/about">About</a>
```

Route parameters:

You can also pass parameters to the URL while routing. Parameters passed along with URL are called route parameters. To navigate programmatically, you can use navigate() method of Router class. Route parameters allow you to pass dynamic values within the URL to the target component. This is useful for scenarios like:

- Displaying details of a specific item (e.g., product details, user profile).
- Filtering data based on a parameter (e.g., category, search query).

Inject the router class into the component and invoke navigate method as shown below:

```
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

Example:

1. **Define the Route with Parameter:**

```
const routes: Routes = [
  { path: 'product/:id', component: ProductDetailComponent }
];
```

Here→ :id is the placeholder for the product ID parameter.

2. **Access the Parameter in the Component:**

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

```
@Component({ // ... })
export class ProductDetailComponent implements OnInit {
        constructor(private route: ActivatedRoute) { }
        ngOnInit() {
                this.route.paramMap.subscribe(params => {
                        this.productId = params.get('id');
                });
        }
}
```

3. **Navigate to Component Class through html**

```
<a routerLink="/product/123">View Product</a>
```

Route Guards

In Angular application, user can navigate to any url directly. Ideally, it should be checked whether a particular user is allowed to navigate to a particular url or not. This is achieved using **Route guards**. They act as gatekeepers, determining whether a user can navigate to or leave a particular route based on certain conditions.

Route Guards are the interfaces using which we can communicate the Router module of angular, whether it should allow the user the navigate to the requested route or not. The route guard interfaces decide this based on the truthy or falsy values returned by the class which implements the guard interfaces.

- If it returns **true**, the navigation process continues
- If it returns **false**, the navigation process stops

Types of Route Guards:

- **CanActivate:** Determines if a route can be activated and allows navigation based on certain conditions. This is the most common type of guard.
- **CanDeactivate:** Checks if a route can be deactivated, often used to prevent users from accidentally leaving a component with unsaved changes.
- **CanLoad:** Determines if a feature module can be loaded.
- **CanActivateChild:** Determines if a child route can be activated.
- **CanMatch:** Determines if a route matches the URL.
- **Resolve:** This Interface can be implemented by the classes in angular app to be a data provider. We can use the data provider with the router for resolving data while navigation.

How to Create Route Guards in Angular ?

We can create a class implementing required route Guard Interfaces using the below command:

ng generate guard auth

Once the guard is created successfully, Two new files i.e. **auth.guard.ts** and **auth.guard.spec.ts** are added to the project folder with the code as shown below:

```
Class AuthGuard implements CanActivate {
  canActivate( ): boolean { }
}
```

Example:

```
import { Injectable } from '@angular/core';
import { CanActivate, Router } from '@angular/router';
import { UserService } from './user-service'
@Injectable()
export class AccessGuardService implements CanActivate {
   constructor(private serv:UserService, private router: Router ) {};
   canActivate() {
      let data = this.serv.canLoginToday()
      console.log("here",this.serv.canLoginToday())
      if(this.serv.canLoginToday()) {
         this.router.navigate(['/login'])
         return true;

      } else {
         return false;
      }

   }
}
```

app-routing.module.ts.

```
const routes:Routes = [
    {path: '',redirectTo:"home", pathMatch: 'full',},
    {path:"home",component:HomeComponent
,canActivate:[AccessGuardService]},
```

```
    {path:"login",component:SuccessComponent}
];
```

17. Lazy Loading

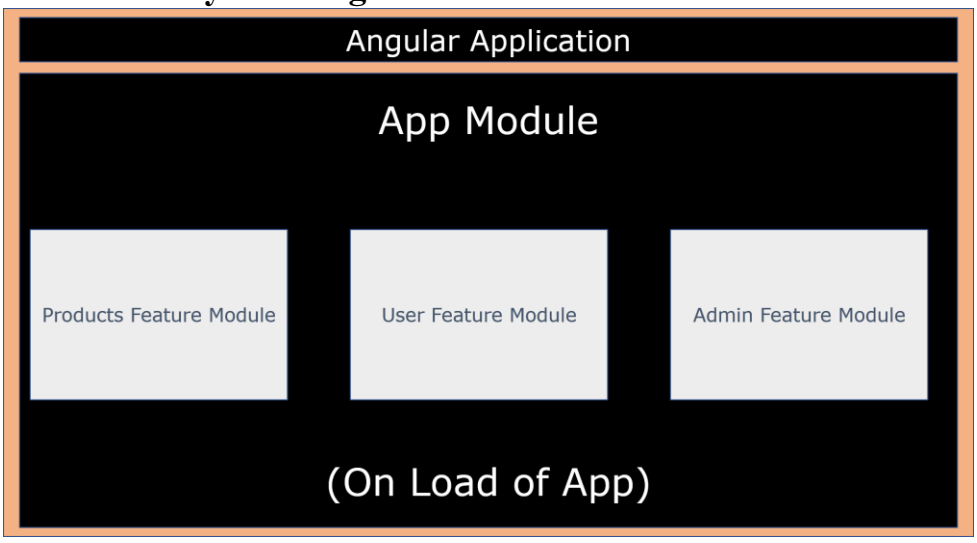
Lazy loading is a technique used in Angular to optimize the initial load time of your application. Instead of loading all modules and their associated components, services, and assets at once when the application starts, lazy loading allows you to load these modules only when they are actually needed by the user.

How Angular handles Lazy Loading ?

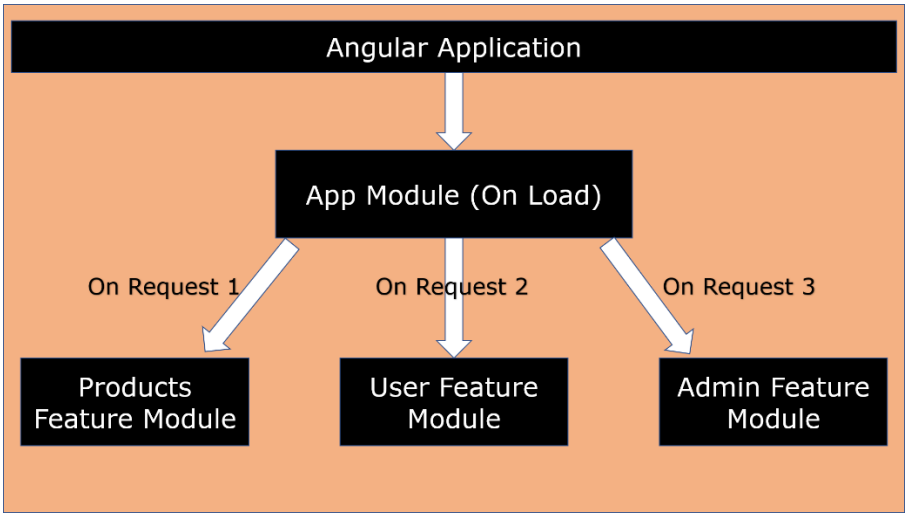
In Angular, we perform Lazy Loading mainly through modules.

Usually, every angular application will have a root module known as the AppModule where we mention all the imports and component declarations for that application. This module is then bundled during compilation and pushed to the browser. Hence, to implement Lazy Loading in angular, we split the root module into smaller modules where each module will contain the code for a particular feature of the application, and hence the application would load as below, with and without Lazy Loading.

Without Lazy Loading:-



With Lazy Loading:-



Lazy Loading in Angular - Demo:

Angular Implements **Lazy Loading** with the help of **asynchronous routes.** Here, we will learn how to implement Lazy Loading in angular using **async routes** step by step. **Problem Statement:-** Create two feature modules i.e. **products** and **orders** and a **home** component (in app module) to load the modules and home component only on click of the corresponding button as shown below:

To Load a particular feature on demand let us first create some feature modules in the angular application using the below commands:

ng generate module products --route products --module app.module

ng generate module orders --route orders --module app.module

Once these modules are created, routes to load the **products** and **orders** module will be added automatically in the **app-routing.module.ts** file Also routes to load **home** component on load of the application are added. The final code of **app-routing.module.ts** will be:

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { HomeComponent } from './home/home.component';
const routes: Routes = [
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  {
    path: 'products',
    loadChildren: () => import('./products/products.module').then(m =>
m.ProductsModule)
  },
```

```
  {
    path: 'orders',
    loadChildren: () => import('./orders/orders.module').then(m =>
m.OrdersModule)
  },
  {
    path: 'home',
    component: HomeComponent
  }
];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Also both products and orders module will have the following code generated by angular cli:

products-routing.module.ts

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { ProductsComponent } from './products.component';
const routes: Routes = [{ path: '', component: ProductsComponent }];
@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class ProductsRoutingModule { }
```

orders-routing.module.ts

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { OrdersComponent } from './orders.component';
const routes: Routes = [{ path: '', component: OrdersComponent }];
@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class OrdersRoutingModule { }
```

18. RxJS – Observables and Operators

RxJS stands for Reactive Extension for Javascript. RxJS is a JavaScript library that helps you work with asynchronous events and data streams. It uses the concept of "observables," which are like streams that continuously emit data. This allows you to write code that reacts smoothly to events like user clicks, data updates from a server, or changes in the application's state. RxJS provides a set of operators that let you manipulate these data streams, making it easier to handle complex scenarios, filter data, and combine multiple streams. By using RxJS, you can create more responsive and efficient applications with cleaner and more maintainable code.

RxJS also provides many utility functions for different purposes such as:

- To convert existing code for asynchronous operations into Observables
- To Iterate over the values in a data stream
- Mapping Values of different types
- Filtering data from a stream
- Composing two or more data streams together

RxJS Features:

- **Observables:** Imagine a flowing river. An Observable is like a pipe carrying that river's water. It's a source of data that can change over time.
- **Observer:** Think of someone standing by the riverbank with a cup. The Observer is the one who receives the water (data) flowing through the pipe (Observable).
- **Subscription:** The act of the Observer placing their cup under the pipe to receive the water is the Subscription. They can also remove their cup (unsubscribe) to stop receiving water.
- **Operators:** Like filters and valves, Operators modify the flow of water in the pipe. You can use them to clean the water, control the flow rate, or even split the water into different streams.
- **Subject:** A special type of pipe that can split the water and deliver it to multiple Observers simultaneously.
- **Schedulers:** Like a traffic controller, Schedulers control the timing and order of how the water flows through the pipe.

Advantages Of RxJS:

- It is supported by Javascript as well as Typescript and can be used with any javascript based library or framework such as angular, react, vuejs.etc...
- It very nicely handles the async operations by the help of observables and reactive programming together.

- It provides a huge collection of operators for mathematical, transformation, filtering, error handling and many more such tasks.

Observables, Observers and Subscription

We already saw the various features that RxJS provides such as Observables, Observers and Subscription.

- **Observable** is a wrapper around some asynchronous data source (a stream of data sources that keeps on emitting data one after other continuously). Observables are used to represent asynchronous data streams, such as:
  - HTTP responses (data fetched from a server)
  - User events (clicks, key presses)
  - Timer events (intervals, timeouts)

- **Observers** are there to execute some block of code whenever some data is received from the observable or when some error happens or when Observable reports that the data stream is over or completed. The **Observer** will execute all these blocks of code only when we **subscribe** to the Observable.

  The observers are a set of callback functions that help us handle the different types of values that an Observable emits. It includes:
  - **next:-** It sends any value such as numbers, arrays or objects when subscribed
  - **error:-** It sends a javascript error object when subscribed
  - **complete:-** It does not send any value, rather it marks the completion of the data stream. No further values can be sent after complete function has been called.

- **Subscription** A Subscription represents the execution of an Observable. It establishes a connection between the Observer and the Observable. It allows you to start receiving data from the Observable.

- **Unsubscribe:** An Observable is capable to run infinite times and emits values, but we may not want that to happen always. At certain point of time, we might need to stop emitting values otherwise it would lead to unnecessary waste of memory and computing power. We can prevent such issues by unsubscribing to an observable using the unsubscribe() method.
  Note: - We have lot to cover in RxJS……..