

INTRODUCTION TO ANGULAR

Angular is an open source JavaScript framework to build web applications using HTML, CSS, and JavaScript / TypeScript



INTRODUCTION TO ANGULAR

- Angular is a open-source JavaScript-based framework for building client-side web applications.
- The Angular is a development platform for building a Single Page Application(SPA) for mobile and desktop.
- It uses Typescript & HTML to build Apps.
- The Angular itself is written using the Typescript.
- Angular is a client-side JavaScript framework that was specifically designed to help developers build SPAs in accordance with best practices for web development.
- AngularJs is developed by Google in 2010.

INTRODUCTION TO ANGULAR

Problems with AngularJs:

- There is no doubt that AngularJs was very popular in its time. But, AngularJS lacked many essential features offered by other JavaScript frameworks in the market.
- The problems with AngularJS are:
 - Large bundle size
 - Performance issues
 - Did not support mobile devices

INTRODUCTION TO ANGULAR

- To resolve the issues encountered in AngularJs, Google launched Angular, an open-source JavaScript framework.
- This was a complete rewrite of AngularJs and worked on a different paradigm.
- The early version of Angular was named Angular 2. Then later, it was renamed just Angular.
- Then Angular Team releases new versions of the Angular versions regularly; the last stable version available is Angular 16(May 2023).

FEATURES OF ANGULAR

Two-Way Data Binding

- Data binding is automatic and fast. Changes made in the View are automatically updated in the component class and vice versa.

Powerful Routing Support

- Angular's Powerful **routing engine** loads the page asynchronously on the same page, enabling us to create a Single Page Application.

Expressive HTML

- Angular enables us to **use programming constructs** like if conditions, for loops, etc., to render and control the HTML pages.

FEATURES OF ANGULAR

Modular Design

- Angular follows the **modular design**. You can create Angular modules to organize better and manage our codebase.

Built-in Back End Support:

- Angular has built-in support to communicate with the back-end servers and execute any business logic or retrieve data

Active Community

- Angular is Supported by google and has a very good active community of supporters.

SETTING UP DEVELOPMENT ENVIRONMENT

- Before getting started with Angular, we must set up our development environment and install the required tools to work with the Angular application
 - Choosing and Installing Editor
 - Install NPM
 - Choosing Language
 - Module Loader

Choosing an Editor:

- Can choose any editor of your choice, including Visual Studio code, Sublime text, Atom, Brackets, WebStorm, etc.

INTRODUCTION TO ANGULAR 2

Installing Package Manager

- To install Angular and dependencies, we need Node Package Manager or NPM.
- NPM is used to install libraries, Packages & applications from Public repositories.
- The Angular Applications must be written in Javascript. This gives us a few options, including the current version of Javascript (i.e., ES6), Typescript, etc.
- The Typescript is a popular choice here. The Angular code is also written using Typescript

INTRODUCTION TO ANGULAR 2

Choosing a Module Loader

- Module loader takes a group of modules with their dependencies and **merges them into a single file** in the correct order. This process is called Module bundling.
- There are many module loaders available. The most popular Module loader is **Webpack**
- Webpack is a powerful module loader. It takes modules with dependencies and generates static assets representing those modules. Webpack can bundle any file: JavaScript, TypeScript, CSS, SASS, LESS, images, HTML, fonts, etc.
- The Webpack also comes with a development server. The Development server uses the Webpack's watch mode.

ANGULAR REQUIREMENTS: INSTALL NODEJS AND NPM

Installing Angular

- To install Angular, follow the below steps:

Step 1: Node.js Installation

- Install Node.Js from <https://nodejs.org/en/download/>
- Reason for installing NodeJS: A web browser (Chrome, Firefox, etc.) understands only JavaScript; we have to transpile our Typescript to JavaScript.
- Therefore, the Typescript **transpiler requires Node.Js to generate the Typescript** code for JavaScript

ANGULAR REQUIREMENTS: INSTALL NODEJS AND NPM

Step 2: Angular CLI installation

- The Angular CLI helps us to **quickly create an Angular application** with all the configuration files and packages in one single command.
- It also helps us to add feature (components,directives,services, etc.) to existing Angular applications.
- The Angular CLI creates the Angular Application and uses Typescript, Webpack (for Module bundling), and Karma (for unit testing).

INSTALLING ANGULAR CLI

- The first step is to install the Angular CLI. We use the following command.
 - `npm install -g @angular/cli`
- We can find out the Angular CLI Version Installed by Using the Command
 - `ng version`
- To create your project, you have to run the following command:
 - `ng new <project-name>`
- Running your new Angular Project
 - To run your application all you need to do is type the following command
 - `ng serve --open`

DIRECTORY STRUCTURE

- The root application folder contains subfolders **node_modules** and **src**. It also contains a few configuration files

browserslistrc:

- Ensures the **compatibility** of the Angular app with different browsers.

editorconfig:

- This is the configuration file for the Visual Studio code editor.

gitignore:

- Git configuration to ensure autogenerated files are not committed to source control.

angular.json:

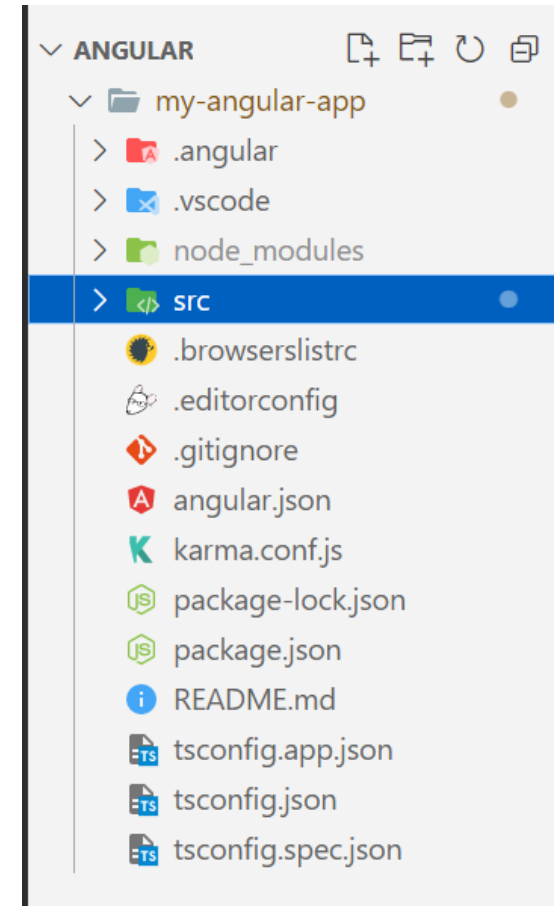
- This is the configuration file for Angular CLI. The older versions of Angular used the file angular-cli.json

karma.conf.js:

- The Configuration file for **the karma test runner**.

package.json:

- The package.json is an npm configuration file that lists the third-party packages that your project depends on. We also have package-lock.json



README.md:

- The Read me file

tsconfig.json:

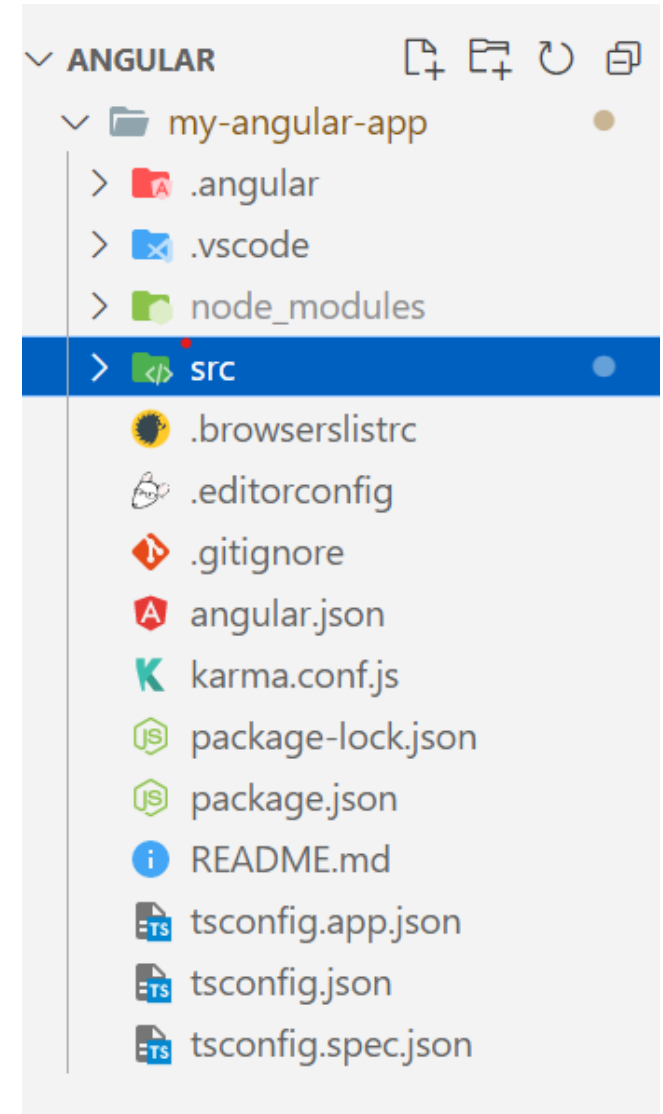
- tsconfig.app.json & tsconfig.spec.json are Typescript configuration files. The tsconfig.json is the Typescript compiler configuration file.
- The tsconfig.json file specifies the compiler options required for the Typescript to compile (transpile) the project.
- The tsconfig.app.json is used for compiling the code, while tsconfig.spec.json is for compiling the tests

node_modules:

- All our external dependencies are downloaded and copied here by NPM.

Src: This is where our application lives.

- The src folder consists of three subfolders -> app folder, assets folder, environment folder, along with many other files.



app folder:

- The Angular CLI creates the root component, a root module, and a unit test class to test the component.

app.component.ts:

- This Typescript file defines the logic for the app's root component named AppComponent

app.component.html & app.component.css:

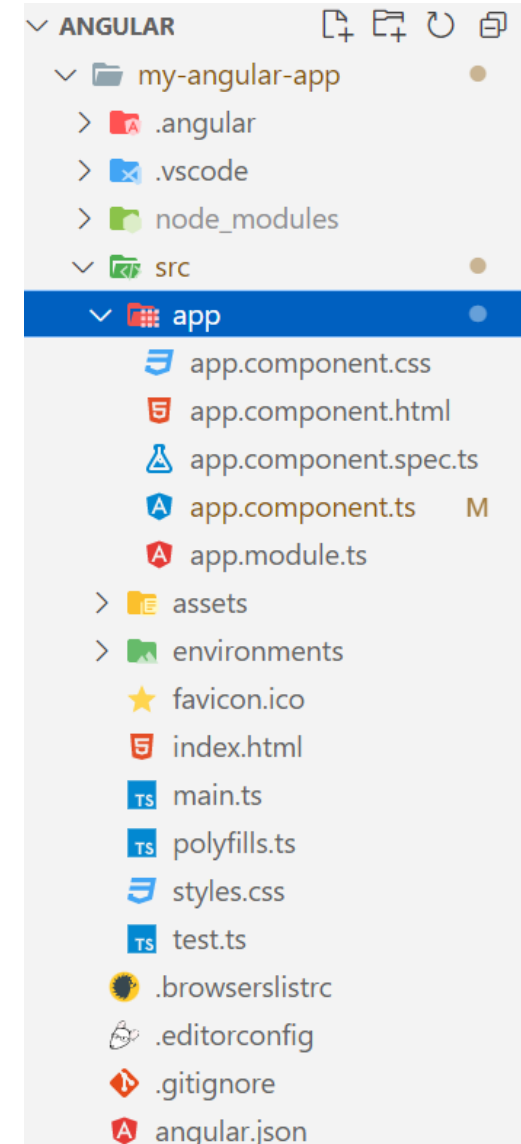
- The .html file defines the HTML templates associated with the root AppComponent
- the .CSS file defines the base CSS stylesheet for the root AppComponent.

app.component.spec.ts:

- This .spec.ts file defines a unit test for the root AppComponent.

app.module.ts:

- The app.module.ts defines the root module (AppModule), that is responsible for **assembling the application**.



DIRECTORY STRUCTURE

assets folder:

- This folder consists of image files & other assets files to be copied whenever we are building our own application.

environment folder:

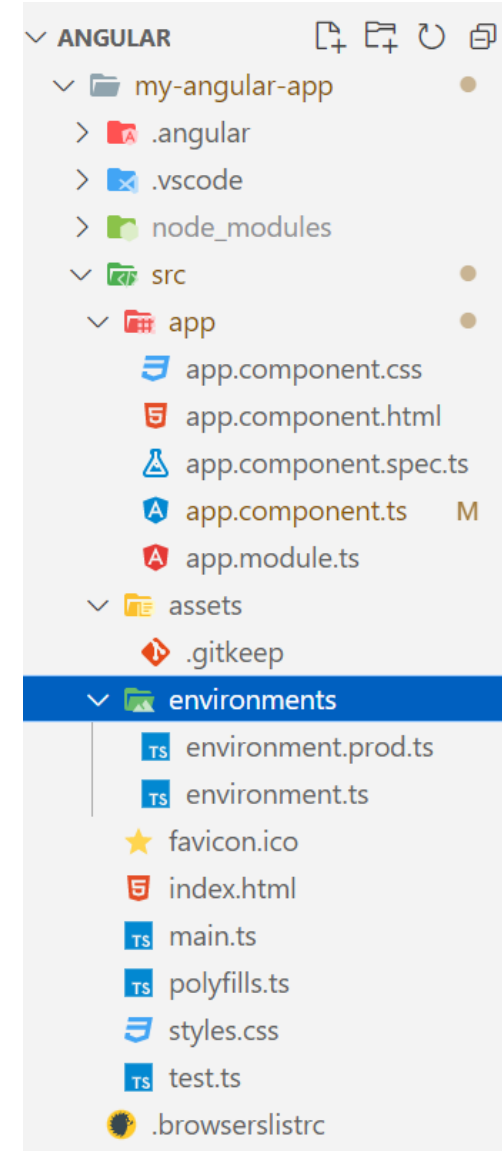
- This folder contains two environment files i.e. Development environment & Production environment.

favicon.ico:

- This file consists of an icon that is displayed in the Browser Tab.

index.html:

- This is the main HTML page served when your web-page is displayed in the browser. We can see the page source of a web-page by inspection (Ctrl + shift + I) in the web browser.



DIRECTORY STRUCTURE

main.ts:

- This main.ts file is the entry point of our web-app. It compiles the web-app & bootstraps the AppModule to run in the browser.

polyfills.ts:

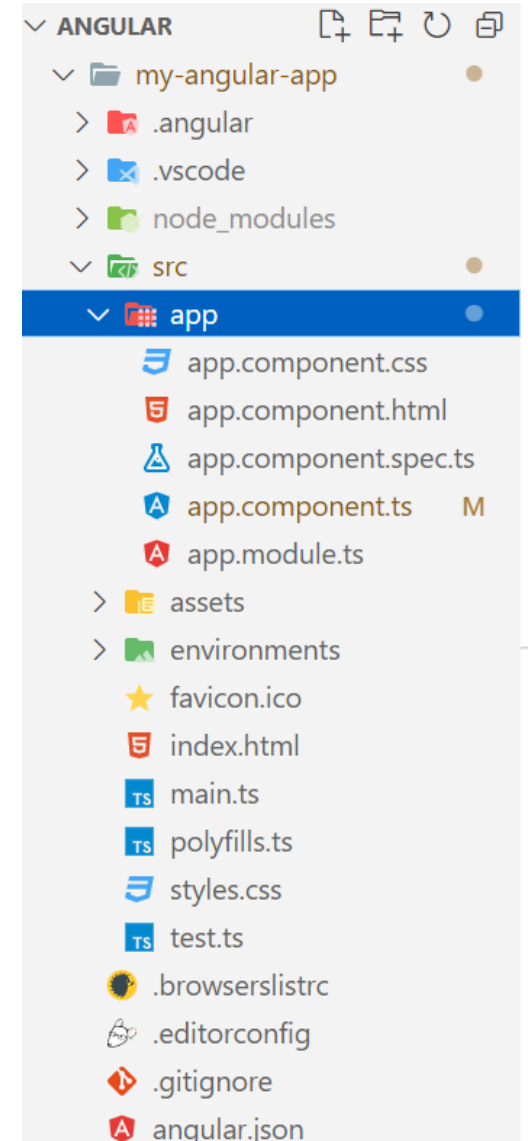
- This file consists of a few lines of codes which makes our application compatible for different browsers.

test.ts:

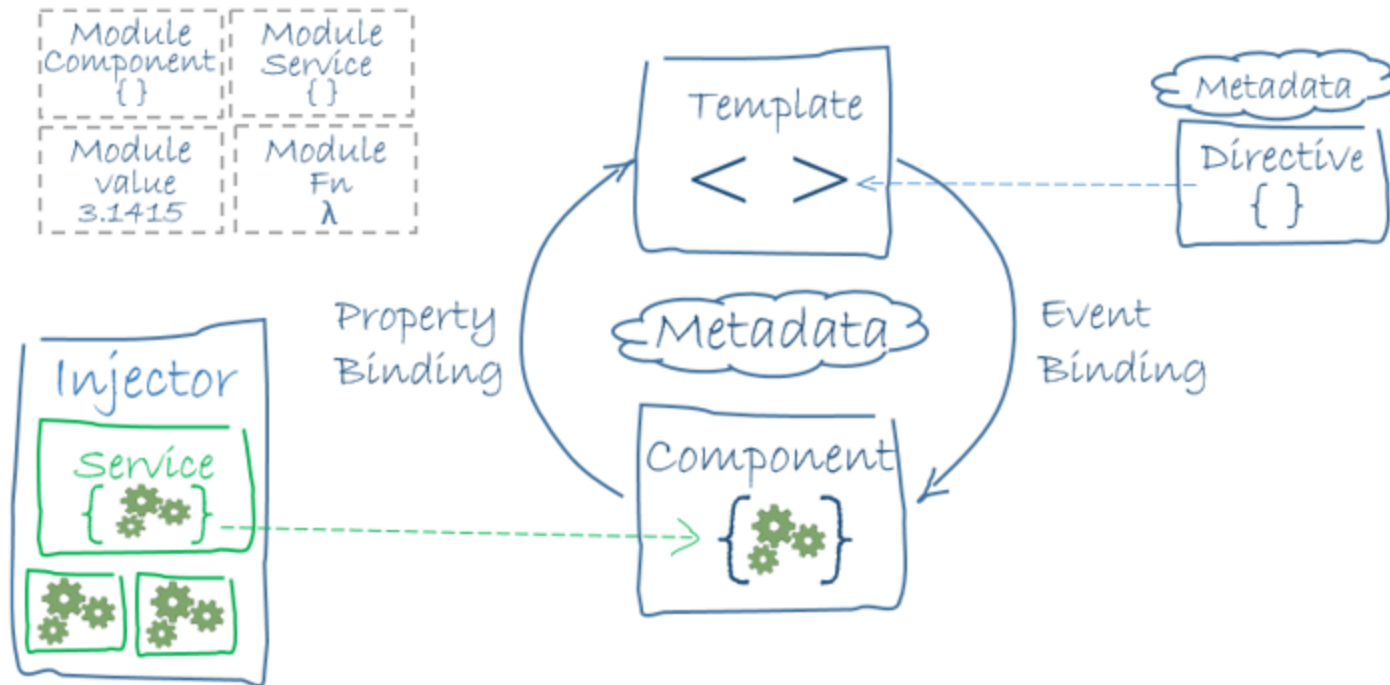
- This file is the main entry point for the Unit Testing of the web application.

styles.css:

- This file supplies style to the project with CSS.



BUILDING BLOCKS OF ANGULAR APPLICATION



The main building blocks of Angular are:

- Modules
- Components
- Templates
- Metadata
- Data binding
- Directives
- Services
- Dependency injection

<https://angular.io/guide/architecture>

MODULES

What is a Module in Angular?

- A module is a container. In other words, we can say that a module is a mechanism to group related components, services, directives, and pipes.

Why we need Modules in Angular?

- Modular development is one of the most important aspects of software development. Good software will always have self-contained modules. So, in Angular, you would like to create separate physical typescript or JavaScript files that will have self-contained code.
- Then definitely you would have some kind of reference mechanism by using which the modules can be referred from the places where you want to use these modules. In TypeScript, we can do this by using the “import” and “export” keywords.
- That means the modules which need to be exposed should be declared using the “export” keyword while the modules which want to import the exported modules should have the import keyword.

MODULES

- When we create a new angular project using the Angular CLI command, one module i.e. AppModule (within the app.module.ts file) is created by default within the app folder which you can find inside the src folder of your project.
- For any angular application, there should be at least one module.
- The AppModule (app.module.ts file) is the default module or root module of our angular application. Let us first understand this default module and then we will see how to create and use our own modules.

MODULES

- AppModule is the module name and it is decorated with the @NgModule decorator.
- The @NgModule is imported from the angular core library i.e. @angular/core
- Here, we call NgModule a decorator because it is prefixed with @ symbol.
- whenever you find something in angular, that is prefixed with @symbol, then you need to consider it as a decorator.
- The @NgModule decorator accepts one object and that object contains some properties in the form of arrays. By default, it has included 4 arrays (declarations, imports, providers, and bootstrap)

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

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

Angular core library

```
import { AppRoutingModule } from './app-routing.module';
```

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

```
@NgModule({
```

```
  declarations: [  
    AppComponent
```

```
  ],
```

```
  imports: [  
    BrowserModule,
```

```
    AppRoutingModule
```

```
  ],
```

```
  providers: [],
```

```
  bootstrap: [AppComponent]
```

```
})
```

Decorator

```
export class AppModule { }
```

Module

DECLARATIONS ARRAY (DECLARATIONS):

- The declarations array is an array of components, directives, and pipes. Whenever you add a new component, first the component needs to be imported and then a reference of that component must be included in the declarations array.
- By default AppComponent (app.component.ts file) is created when you create a new angular project and you can find this file within the app folder which you can find inside the src folder.
- If you open the app.component.ts file, then you will find the following code.

```
MyAngularApp > src > app > TS 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    title = 'MyAngularApp';
10 }
```

MODULES

- The component name is AppComponent and in the root module, first, this component (AppComponent) is imported, and then its reference is included in the declarations array, as shown in the below image.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppRoutingModuleModule } from './app-routing.module';
```

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

**Component
Import**

```
@NgModule({
```

```
  declarations: [
```

```
    AppComponent
```

Component Declare

```
  ],
```

```
  imports: [
```

```
    BrowserModule,
```

```
    AppRoutingModuleModule
```

```
  ],
```

```
  providers: [],
```

```
  bootstrap: [AppComponent]
```

```
})
```

```
export class AppModule { }
```

IMPORTS ARRAY (IMPORTS):

- In the imports section, we need to include other modules (Except @NgModule).
- By default, it includes two modules:
 - BrowserModule
 - AppRoutingModule.
- Like the components, here also, first you need to import the modules, and then you need to include a reference of that module in the imports section

```
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 { }
```


Providers Array (providers):

- Whenever you create any service for your application, first you need to include a reference of that service in the provider's section and then only you can use that service in your application.

Bootstrap Array (bootstrap):

- This section is basically used to bootstrap your angular application i.e. which component will execute first.
- At the moment we have only one component i.e. AppComponent and this is the component that is going to be executed when our application starts. So, this AppComponent is included in the bootstrap array.

```
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 { }
```

MODULES

How to create a Module in Angular?

- To create a module in angular using Angular CLI.

```
ng generate module modulename  
ng g module modulename
```

- Here, g means generate, and you use either g or generate to create the module.

How do you create multiple modules?

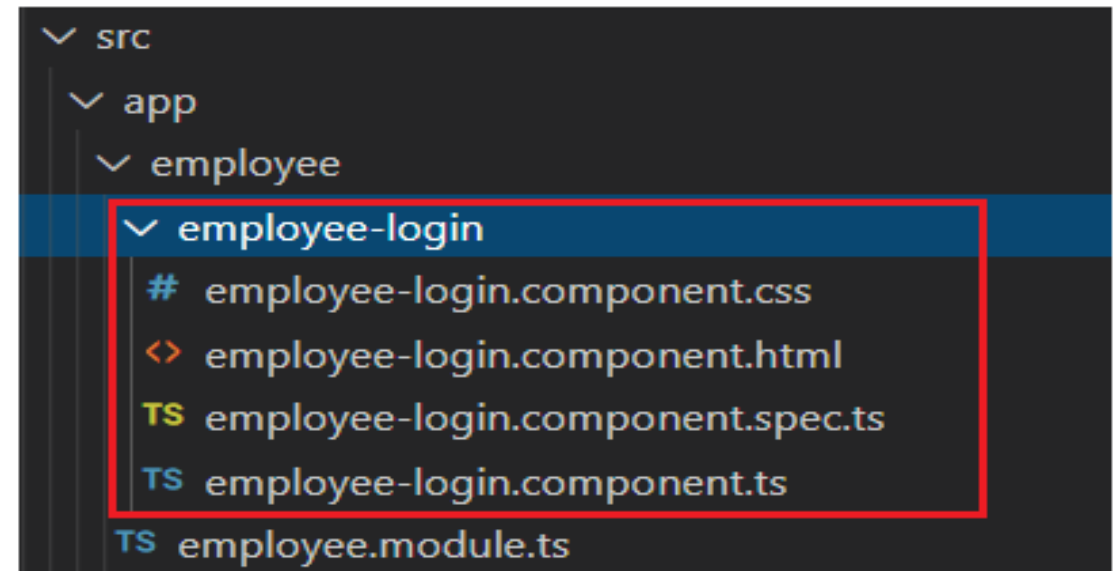
Step1: Create an Employee module

```
ng g module Employee
```

Step2: Creating EmployeeLogin Component

within the Employee module folder

```
ng g c Employee/EmployeeLogin
```



Step3: Adding EmployeeLogin component reference in Employee module

- you need to add the reference of EmployeeLoginComponent in the Employee module. But, this work is automatically done by the angular framework for us.
- If you open the employee.module.ts file, then you will see that the import and declarations sections are automatically done as shown in the below image.

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

import { EmployeeLoginComponent } from '../employee-login/employee-login.component';

@NgModule({
  declarations: [EmployeeLoginComponent],
  imports: [
    CommonModule
  ]
})
export class EmployeeModule { }
```

- Note: If you declare a component in one module, then you can't declare the same component in another module.
- If you try then you will get an error when you run your application. At the moment you already declared the EmployeeLoginComponent within the Employee module and if you try to declare it again on the AppModule module then you will get an error when you run your application.

Step4: Adding Employee module reference in AppModule import section

- In order to use the EmployeeLoginComponent, you need to include the reference of the Employee module into the import section of your root module i.e. the AppModule as shown in the below image.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { EmployeeModule } from './employee/employee.module';
```

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

export class AppModule { }
```

ANGULAR DECORATORS

- Decorators are the features of Typescript and are implemented as functions.
- The name of the decorator starts with @ symbol followed by brackets and arguments. That means in angular whenever you find something which is prefixed by @ symbol, then you need to consider it as a decorator.
- The decorator provides metadata to angular classes, property, value, method, etc. and decorators are going to be invoked at runtime.

Commonly used Decorators:

- There are many built-in decorators are available in angular. Some of them are as follows:
 - @NgModule to define a module.
 - @Component to define components.
 - @Injectable to define services.
 - @Input and @Output to define properties

TYPES OF DECORATORS IN ANGULAR:

- In Angular, the Decorators are classified into 4 types.

Class Decorators:

@Component and @NgModule

Property Decorators:

@Input and @Output (These two decorators are used inside a class)

Method Decorators:

@HostListener (This decorator is used for methods inside a class like a click, mouse hover, etc.)

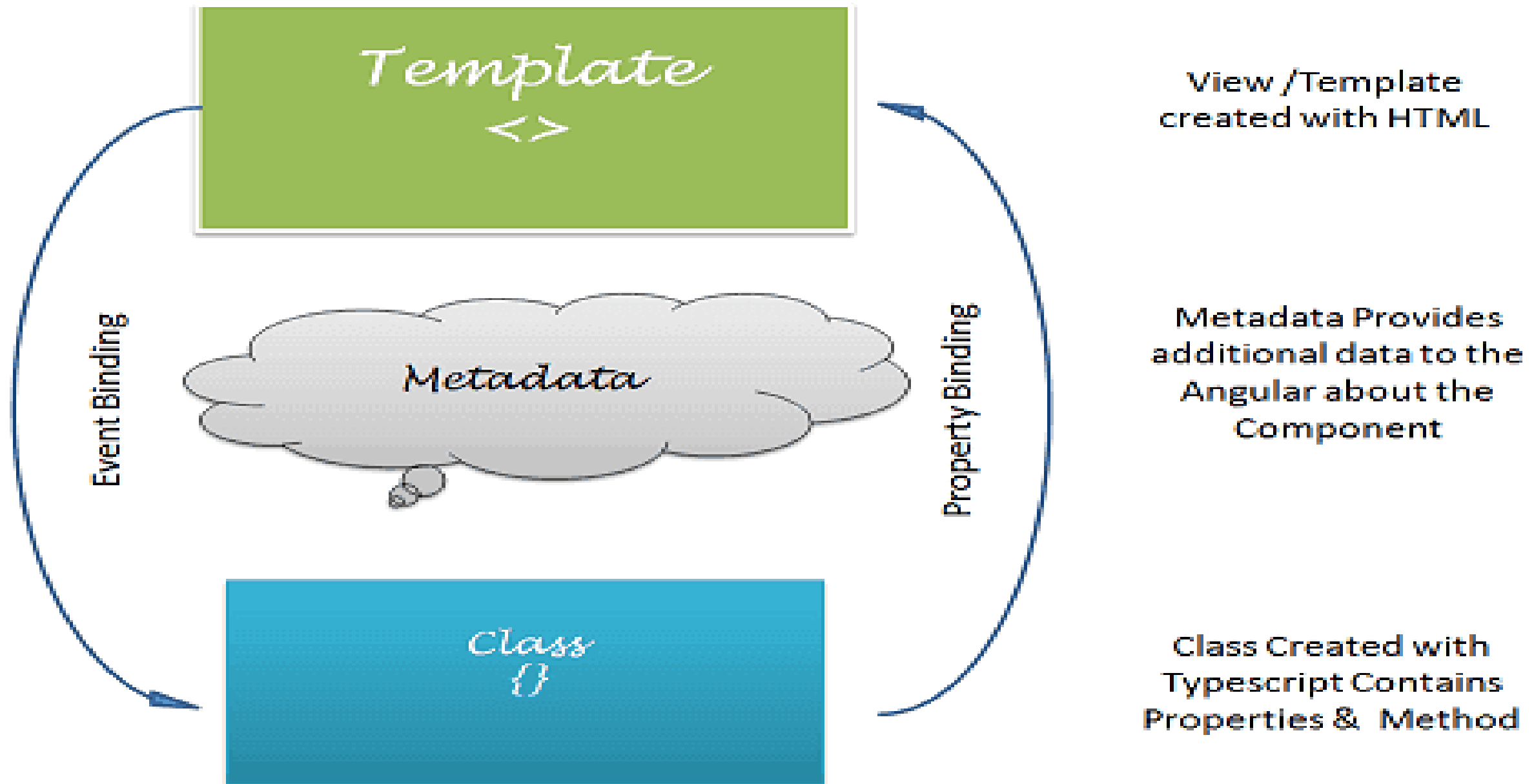
Parameter Decorators:

@Inject (This decorator is used inside class constructor).

COMPONENT

- The component contains the data & user interaction logic that defines how the View looks and behaves.
- A view in Angular refers to a template (HTML).
- The Angular Components are plain javascript classes defined using @component Decorator.
- The Decorator provides the component with the View to display & Metadata about the class
- The Component is responsible for providing the data to the view.
- The Angular does this by using data binding to get the data from the Component to the View.
- The Angular applications will have lots of components.
- Each component handles a small part of the UI.
- These components work together to produce the complete user interface of the application
- The Components consist of three main building blocks
 - Template
 - Class
 - MetaData

Angular Component



Angular Component-Template:

- The template defines the layout of the View and defines what is rendered on the page.
- The Templates are created with HTML.
- Two ways of specifying the Template in Angular.
 - Defining the Template Inline
 - Provide an external Template

Angular Component- class:

- The class is the code associated with Template (View).
- The Class is created with Typescript.
- Class Contains the Properties & Methods

```
export class AppComponent
{
    title : string ="app"
}
```

Angular Component-Metadata:

- Metadata Provides additional information about the component to the Angular.
- Angular uses this information to process the class. The Metadata is defined with a decorator.
- The Components are defined with a `@component` class decorator.
 - It is the `@component` decorator, which defines the class as a Component of the Angular
- Component metadata properties
 - `selector`
 - `template/templateUrl`
 - `styles/styleUrls`

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

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

```
export class AppComponent {  
  title = 'MyAngularApp';  
}
```

Importing the component decorator
from angular core library

Decorating the class with @Component
decorator and providing the metadata

Creating class to define data and logic for
the view

Import Component Decorator:

- First, you need to import the Component decorator as shown in the below image.

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

Create a class

- If you want to create a class using Typescript language, then you need to use the keyword class followed by the class name as shown in the below image.

```
export class AppComponent {  
  title = 'MyAngularApp';  
}
```

- The export keyword in typescript language is very much similar to the public keyword in java applications which allows this class (i.e. AppComponent) to be used by the other components within the application.
- The title here is a property and the data type is a string by default. This property is initialized with a default value of “MyAngularApp”.

Applying the component decorator to the class:

- The next step is to decorate the class with the `@Component` decorator. A class will only become a component when it is decorated with the `@Component` decorator as shown in the below image.

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

What is a Selector in Angular?

- A selector is one of the properties of the object that we use along with the component configuration.
- A selector is used to identify each component uniquely into the component tree, and it also defines how the current component is represented in the HTML DOM.
- When we create a new component using Angular CLI, the newly created component looks like this.

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

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

- Here in `app.component.ts`, notice that we have one property called `selector` along with the unique name used to identify the app component in the HTML DOM tree once it is rendered into the browser.
- Once we run our Angular app in the browser, we can open the browser developer tools and go to the elements tab, where we can see the component rendered.
- As you can see, the `<app-root>` is rendered initially because the app component is the root component for our application.
- If we have any child components, then they all are rendered inside the parent selector.
- Basically, the selector property of the component is just a string that is used to identify the component or an element in the DOM.
- By default, the selector name may have an `app` as a prefix at the time of component creation, but we can update it.

What is a template/templateUrl

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

Why Template

- Templates separate view layer from the rest of the framework so we can change the view layer without breaking the application.

How to create template

- There are two ways of defining templates,
 - Inline Template
 - External Template

Inline Template:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `<h1>Hello {{title}}</h1>`,
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Welcome to Angular first application';
}
```

External Template:

- By default, Angular CLI uses the external template. It binds the template with a component using the templateUrl option.
- templateUrl is used in external format, whereas, in the inline Template, we use a template instead of templateUrl.

Example:

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

Elements of Templates

HTML

Interpolation

Template Expressions

Template Statements

HTML

- Angular uses HTML as a template language.

Interpolation

- Interpolation is one of the forms of data binding where we can access a 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 a component instance.
- That means, if we write `{{ Name }}`, Name should be the property of the component to which this template is bound to.

Template Statements

- Template Statements are the statements which respond to a user event.

`(event) = {{Statement}}`

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

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

With an inline template

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

With an external view template

- We have Visual Studio editor IntelliSense, code-completion, and formatting features and
- Not only the code in “`app.component.ts`” is clean, but it is also easier to read and understand

What Happens when we build the angular application?

- When we build the angular application, the TypeScript files, i.e., the `.ts` files, are compiled to the individual JavaScript files, i.e., the `.js` file. These javascript files are going to be rendered by the browser. We have a typescript file with the name `app.component.ts`, so when we build the project, it will create a javascript file with the name `app.component.js`.

How to create a Component in Angular?

- To create a component using Angular CLI, you need to use the following command.

```
ng g c componentname
```

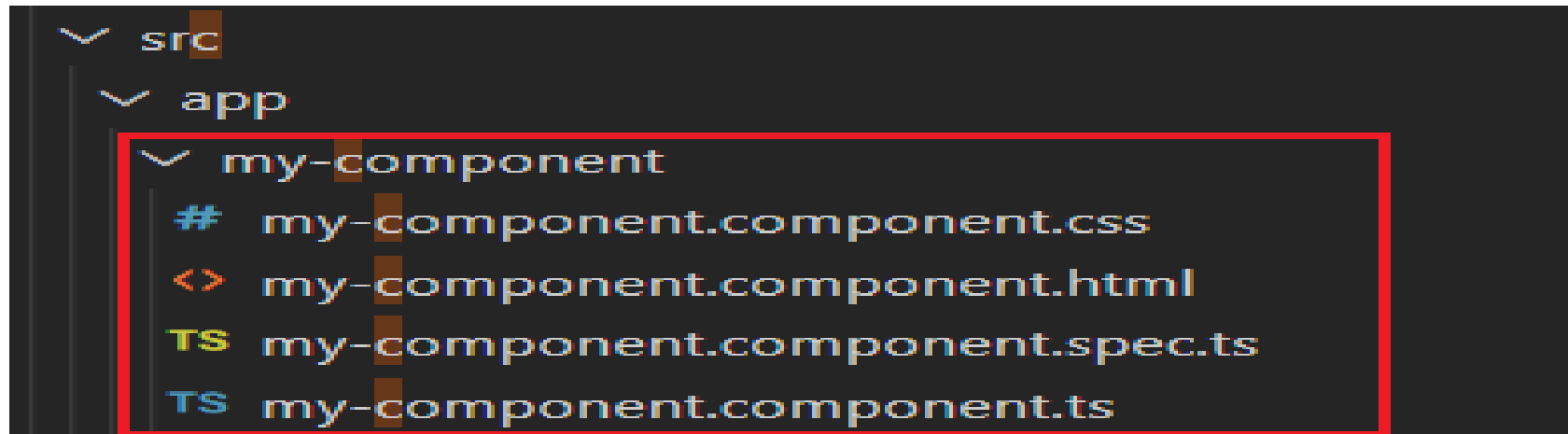
- Here, ng stands for angular, which will call the angular CLI.
- Here, g is the abbreviation for generating
- Here, c is the abbreviation for component
- The component is the type of element that will be generated.
- Here, the component name is the name of the component.
- Suppose you want to create a component with the name MyComponent,

```
PS D:\AngularProjects\MyAngularApp> ng g c MyComponent
```

- Once you type the required command and press the enter button, it will take some time to create the component. Once the component is created successfully, you should get the following output.

```
CREATE src/app/my-component/my-component.component.html (27 bytes)
CREATE src/app/my-component/my-component.component.spec.ts (664 bytes)
CREATE src/app/my-component/my-component.component.ts (298 bytes)
CREATE src/app/my-component/my-component.component.css (0 bytes)
UPDATE src/app/app.module.ts (590 bytes)
```

- it will create four files i.e. along with the ts file, it also creates HTML, spec, and CSS files. In your project, it will create a folder with the name MyComponent inside your app folder and within that MyComponent folder, it will place the above four files.



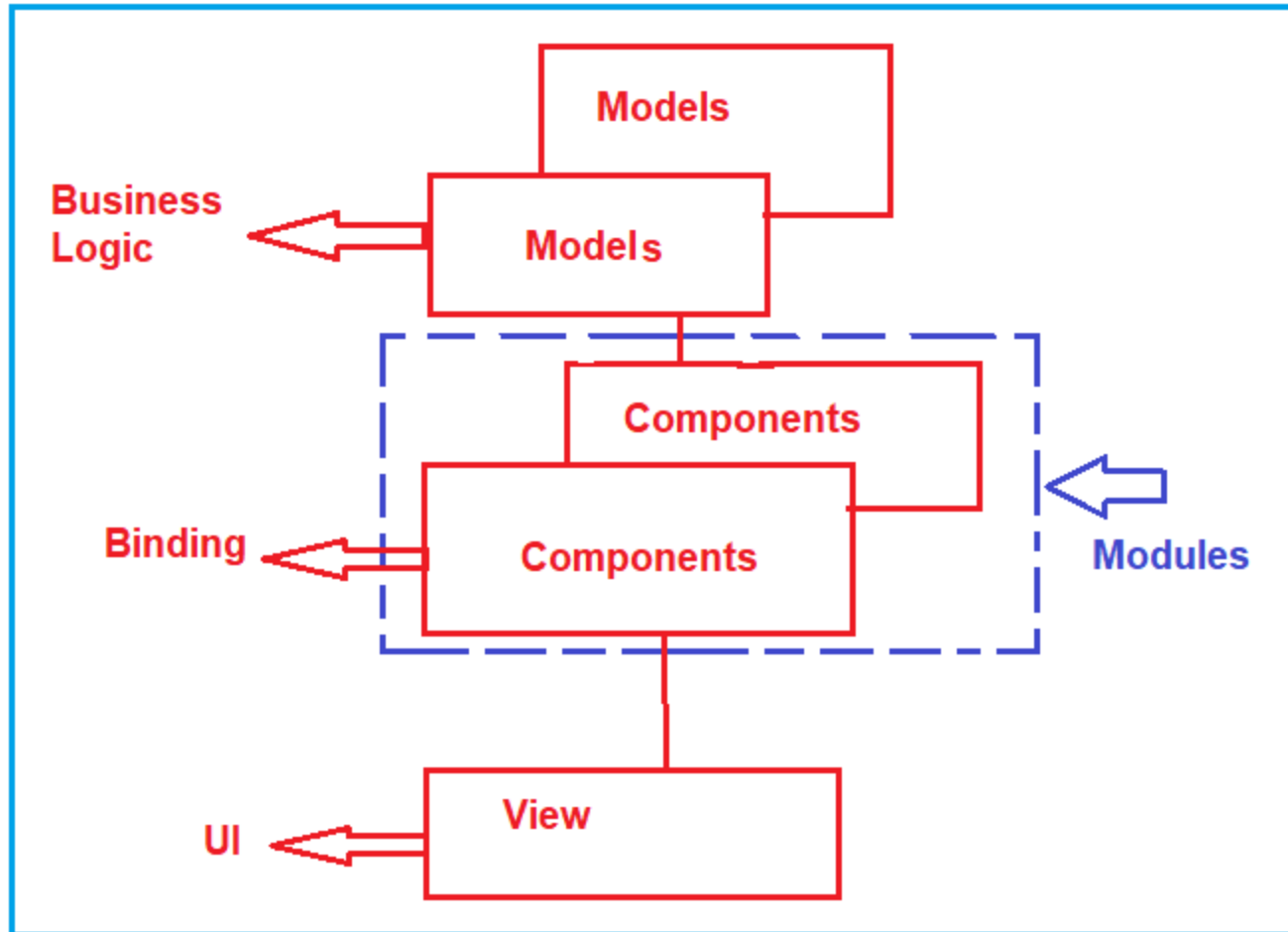
- we must include its reference in the root module i.e. AppModule.
- You can find this root module within the app.module.ts file.
- Now, if you open the app.module.ts file, then automatically the angular framework includes the reference of our newly created MyComponent component in the declarations array as shown in the below image.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { MyComponentComponent } from './my-component/my-component.component';

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

export class AppModule { }
```

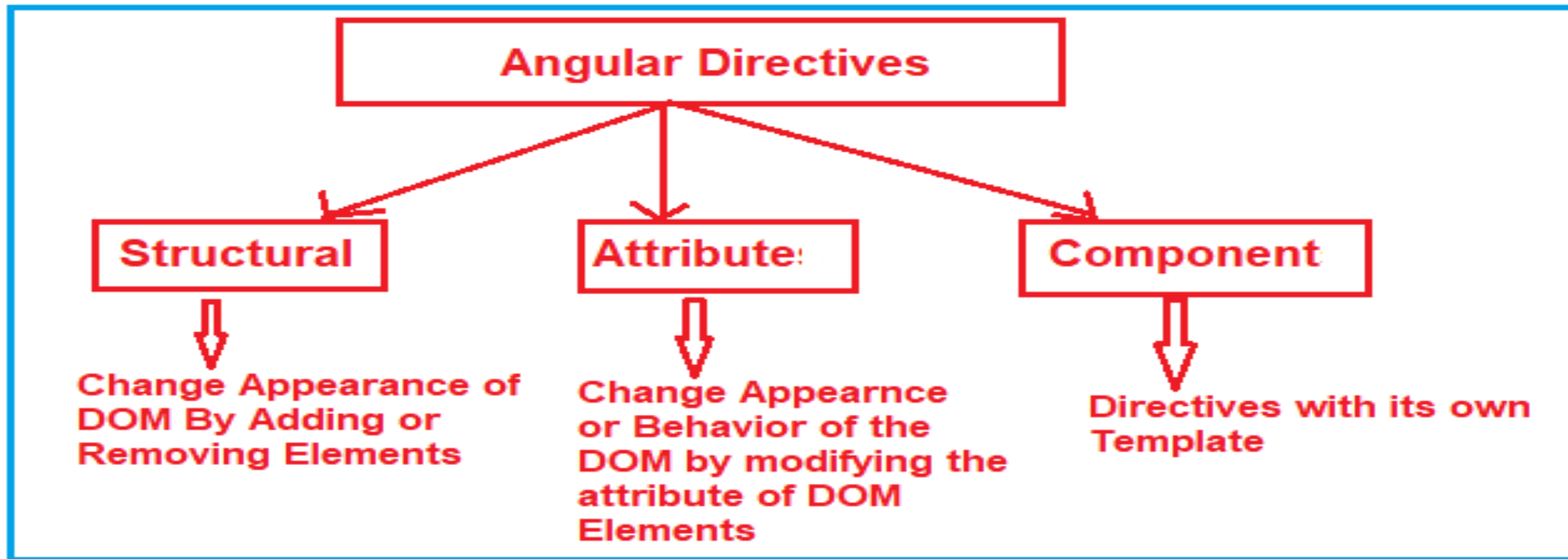

- In Angular, the binding code is technically called a Component.
- In enterprise projects, we can have many components. With many components, it is very difficult for us to handle the project. So, what we can do is we will group the components logically into modules.



DIRECTIVES

- The Angular Directives are the elements which are basically used to change the behavior or appearance or layout of the DOM (Document Object Model) element.

Types of Directives in Angular:



DIRECTIVES

Structural Directives:

- The Structural Directives are responsible for the HTML layout.
- They will shape or reshape the HTML view by simply adding or removing the elements from the DOM.
- These directives are basically used to handle how the component or the element should render in a template.
- There are three structural directives are available.
- They are as follows:
 - NgFor (*ngFor)
 - NgIf (*ngIf)
 - NgSwitch (*ngSwitch)

ANGULAR NGFOR DIRECTIVE

- The ngFor directive is very much similar to the “for loop” used in most of the programming languages. So, the NgFor directive is used to iterate over a collection of data.

Syntax:

*ngFor=“let <value> of <collection>”

ngFor – Local Variables:

Index: This variable is used to provide the index position of the current element while iteration.

First: It returns boolean true if the current element is the first element in the iteration else it will return false.

Last: It returns boolean true if the current element is the last element in the iteration else it will return false.

Even: It returns boolean true if the current element is even element based on the index position in the iteration else it will return false.

Odd: It returns boolean true if the current element is an odd element based on the index position in the iteration else it will return false.

Angular ngIf Directive

- The ngIf is a structural directive and it is used to add or removes the HTML element and its descendant elements from the DOM layout at runtime conditionally. That means it conditionally shows the inline template.
- The ngIf directive works on the basis of a boolean true and false results of a given expression. If the condition is true, the elements will be added into the DOM layout otherwise they simply removed from the DOM layout.

Syntax:

*ngIf = “expression”

Example:

```
<div *ngIf="isValid">  
  <b>The Data is valid.</b>  
</div>
```

When isValid value is true, then
this Div is going to be add into DOM

```
<div *ngIf="!isValid">  
  <b>The Data is invalid.</b>  
</div>
```

When isValid value is false, then
this Div is going to be add into DOM

Angular NgIf directive with else block:

Template Variable

```
<div *ngIf="isValid else elseblock">
  <b>The Data is valid.</b>
</div>

<ng-template #elseblock>

  <div >
    <b>The Data is invalid.</b>
  </div>

</ng-template>
```

When isValid is false, then this else block is going to be added into the DOM

NGSWITCH DIRECTIVE

The Angular ngSwitch directive is actually a combination of two directives i.e. an attribute directive and a structural directive. It is very similar to the switch statement of other programming languages like Java and C# but within a template.

Example:

Select Country

Select ▼

No Country code is selected

You Have Selected

You have not selected any country

Select Country

In ▼

You Have Selected

India

```
src > app > app.component.ts > AppComponent
1 import { Component } from '@angular/core';
2 @Component({
3   selector: 'app-root',
4   templateUrl: './app.component.html',
5   styleUrls: ['./app.component.css'],
6 })
7 export class AppComponent {
8   public dropDownValue = '';
9   setDropDownValue(drpValue: any) {
10     this.dropDownValue = drpValue.target.value;
11   }
12 }
13
```

Go to component

```
1 <h2>Select Country</h2>
2 <select (change)="setDropDownValue($event)">
3   <option value="">Select</option>
4   <option value="In">In</option>
5   <option value="US">US</option>
6   <option value="UK">UK</option>
7 </select>
8
9 <h2>You have selected</h2>
10 <div [ngSwitch]="dropDownValue">
11   <h3 *ngSwitchCase="'In'">India</h3>
12   <h3 *ngSwitchCase="'US'">United States</h3>
13   <h3 *ngSwitchCase="'UK'">United Kingdom</h3>
14   <h3 *ngSwitchDefault="">You have not selected any country</h3>
15 </div>
```


Attribute Directives:

- Attribute Directives are basically used to modify the behavior or appearance of the DOM element or the Component.
- In Angular, there are two in-built attribute directives available. They are as follows:

NgStyle:

This NgStyle Attribute Directive is basically used to modify the element appearance or behavior.

NgClass:

This NgClass Attribute Directive is basically used to change the class attribute of the element in the DOM or in the Component to which it has been attached.

Angular ngStyle Directive

- The ngStyle directive is used to set the DOM element style properties.

Example:

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

[Raw](#)[Copy](#)

```
<div>  
  <button [ngStyle]="{'color':'red', 'font-weight': 'bold', 'font-size.px':20}">Click  
Me </button>  
</div>
```

Angular ngClass Directive

- The Angular ngClass directive is used to add or remove CSS classes dynamically (run-time) from a DOM Element.
- Different mechanisms or methods to use the ngClass directive. They are as follows:

ngClass with string

ngClass with array

The ngClass with object

ngClass with string:

```
[ngClass]=" 'one' "
```

ngClass with array:

```
[ngClass]=" ['three', 'four', 'five'] "
```

ngClass Directive

- The ngClass directive adds and removes CSS classes on an HTML element.
- The syntax of the ngClass is as shown below.

`<element [ngClass]="expression">...</element>`

Where,

- element is the DOM element to which class is being applied
- expression is evaluated and the resulting classes are added/removed from the element. The expression can be in various formats like string, array or an object.

NgClass with a String

- You can use the String as expression and bind it to directly to the ngClass attribute. If you want to assign multiple classes, then separate each class with space as shown below.

```
<element [ngClass]="cssClass1 cssClass2">...</element>
```

Example

Add the following classes to the app.component.css

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

```
.size20 { font-size: 20px; }
```

Add the following to the app.component.html

```
<div [ngClass]="red size20"> Red Text with Size 20px </div>
```

NgClass with Array

- You can achieve the same result by using an array instead of a string as shown below. The syntax for ngClass array syntax is as shown below

```
<element [ngClass]="['cssClass1', 'cssClass2']">...</element>
```

Example

All you need to change the template as shown below

```
<div [ngClass]="['red','size20']">Red Text with Size 20px </div>
```

NgClass with Object

- You can also bind the ngClass to an object. Each property name of the object acts as a class name and is applied to the element if it is true. The syntax is as shown below

```
<element [ngClass]='{'cssClass1': true, 'cssClass2': true}'>...</element>
```

Example

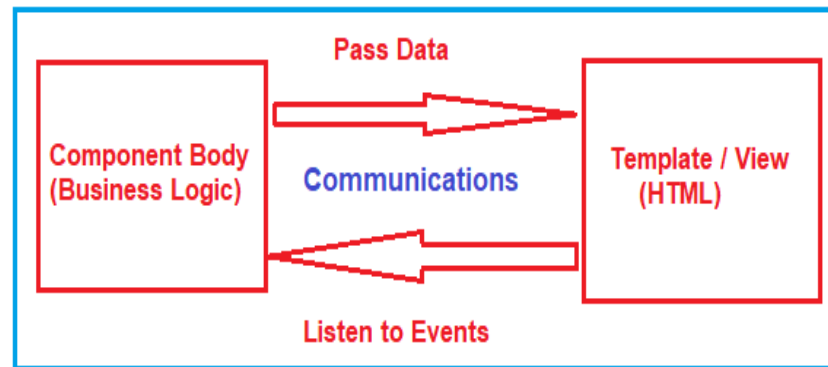
```
<div class="row">
```

```
  <div [ngClass]='{'red':true,'size20':true}'>Red Text with Size 20px</div>
```

```
</div>
```

DATA BINDING

- Data binding is a technique, where the data stays in sync between the component and the view.
- Data Binding means to bind the data (Component's file) with the View (HTML Content). That is whenever you want to display dynamic data on a view (HTML) from the component then you need to use the concept of Data binding.



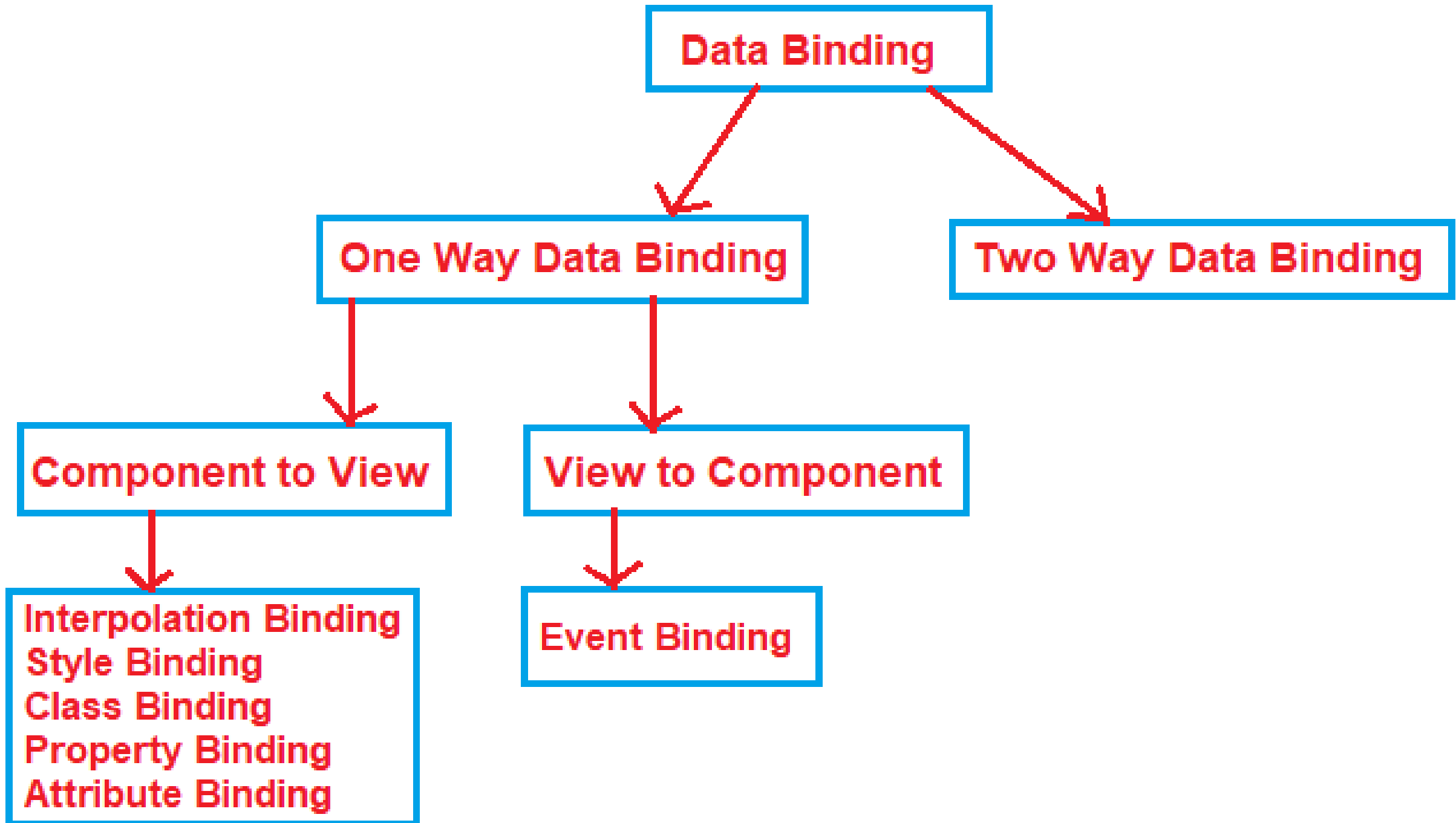
Types of Data Binding in Angular:

One-way Data Binding-

- where a change in the state affects the view (i.e. From Component to View Template) or change in the view affects the state (From View Template to Component).

Two-way Data Binding-

- where a change from the view can also change the model and similarly change in the model can also change in the view (From Component to View Template and also From View template to Component).



Angular Interpolation:

- If you want to display the read-only data on a view template (i.e. From Component to the View Template), then you can use the one-way data binding technique i.e. the Angular interpolation.
- The Interpolation in Angular allows you to place the component property name in the view template, enclosed in double curly braces

Syntax:

`{{propertyName}}.`

- The Angular Interpolation is a technique that allows the user to bind a value to a UI element.

Example:

Step1: Modify the index.html

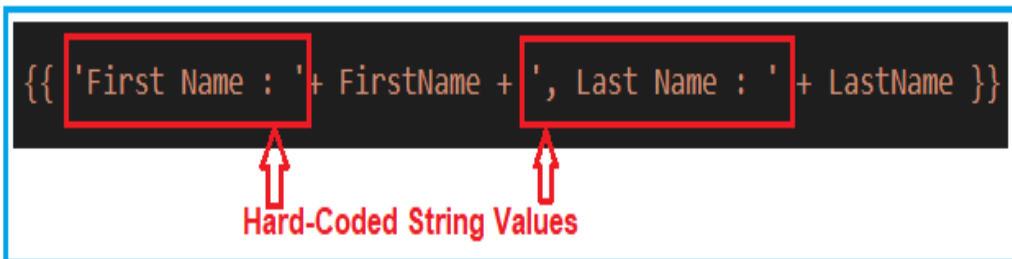
Please modify the index.html file as shown below.

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

Step2: modify app.component.ts file.

```
app.module.ts M  courses.component.ts U  app.component.ts M X  app.component.html M  index.html
src > app > app.component.ts > AppComponent
1 import { Component } from '@angular/core';
2 @Component({
3   selector: 'app-root',
4   template: `<div>
5     <h1>{{ 'First Name:' + firstName + ' , Last Name: ' + lastName }}</h1>
6   </div>`,
7   styleUrls: ['./app.component.css'],
8 })
9 export class AppComponent {
10   firstName: string = 'VDS';
11   lastName: string = 'Krishna';
12 }
13
```

Angular Interpolation with hardcoded string:



The diagram shows a code snippet for Angular interpolation: `{{ 'First Name : ' + firstName + ', Last Name : ' + lastName }}`. Two red boxes highlight the hardcoded string parts: `'First Name : '` and `, Last Name : '`. Red arrows point from these boxes to the text **Hard-Coded String Values** below.

```
{{ 'First Name : ' + firstName + ', Last Name : ' + lastName }}
```

Hard-Coded String Values

Angular Interpolation with Expression:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `<div>
                <h1> 10 + 5 * 7 - 6 = {{ 10 + 5 * 7 6 }} </h1>
            </div>`
})

export class AppComponent {
}
```

Method Interpolation in Angular Application:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `<div><h1> Full Name : {{ GetFullName() }} </h1></div>`})
export class AppComponent {
  FirstName : string = "VDS";
  LastName : string = "Krishna";
  GetFullName() : string{
    return this.FirstName + ' ' + this.LastName;
  }
}
```

Style Binding:

- The Angular Style Binding is basically used to set the style in HTML elements.
- You can use both inline as well as Style Binding to set the style for the element in Angular Applications.

Example:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `<div>
                <button style="color:red;font-weight:bold;
                font-size:20px">Click Me</button>
            </div>`
})
export class AppComponent {
}
```

Class Binding:

- The Angular Class Binding is basically used to add or remove classes to and from the HTML elements.
- It is also possible in Angular to add CSS Classes conditionally to an element, which will create the dynamically styled elements and this is possible because of Angular Class Binding.

Example:

- open the styles.css file and then copy and paste the following code in it. You can find styles.css file within the src folder of your project.

```
.boldClass{  
    font-weight:bold;  
}  
.italicClass{  
    font-style:italic;  
}  
.colorClass{  
    color:red;  
}
```

Index.html

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

Modifying app.component.ts file:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `<div>
    <button class='colorClass boldClass
              italicClass'>Click Me</button>
  </div>`
})
export class AppComponent {}
```

Property Binding:

- The Property Binding in Angular Application is used to bind the values of component or model properties to the HTML element. Depending on the values, it will change the existing behavior of the HTML element.

Syntax:

[property] = 'expression'

Example:

<span[innerHTML] = 'FirstName'>

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `<div>
    <span [innerHTML] = 'Title' ></span>
  </div>`
})

export class AppComponent {
  Title: string = "Welcome to Angular Tutorials";
}
```

The Spam elements innerHTML property is in a pair of square brackets []

The Component class Title property in in a pair of single quote

Angular Event Binding:

- When a user interacts with an application in the form of a keyboard movement, button click, mouse over, selecting from a drop-down list, typing in a textbox, etc. it generates an event.
- These events need to be handled to perform some kind of action. This is where event binding comes into the picture and in Angular Application, we can use event binding to get notified when these events occur.

Syntax:

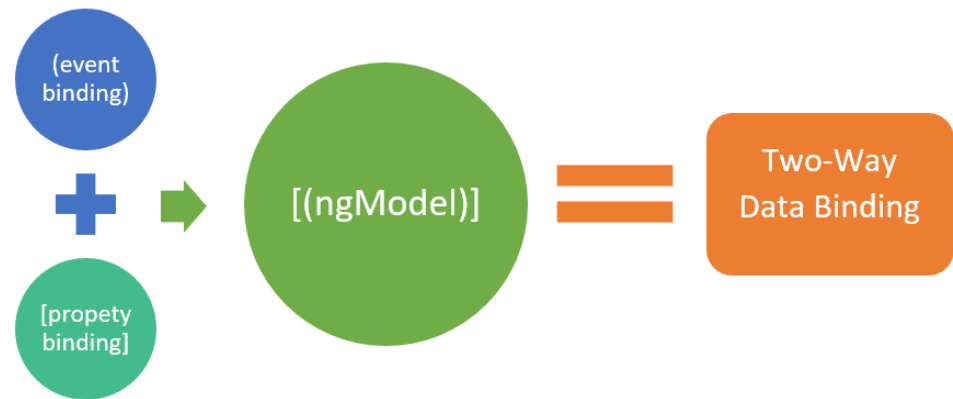
```
<button (click)="onClick()">Click Me </button>
```

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

```
1 import { Component } from '@angular/core';
2 @Component({
3   selector: 'app-root',
4   template: `<div>
5     <button (click)="onClick()">Click Me</button>
6   </div>`,
7   styleUrls: ['./app.component.css'],
8 })
9 export class AppComponent {
10   onClick(): void {
11     alert('Button is clicked');
12   }
13 }
```

Angular Two Way Binding

- The two-way data binding is used in the input type field or any form element where the user types or provides any value or changes any control value on one side. The same is automatically updated into the component variables. On the other side, vice-versa is also true.



ngModel Directive

- The ngModel directive with `[()]` syntax (also known as banana box syntax) syncs values from the UI to property and vice-versa. So, whenever the user changes the value on UI, the corresponding property value will get automatically updated.

`[()]` = `[]` + `()` where `[]` binds property, and `()` binds an event.

ngModel Directive

Two Way Data binding using ngModel Directive:

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

Note:

Open `app.module.ts` file

1. Include the following import statement in it

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

2. Also, include FormsModule in the 'imports' array of @NgModule

```
imports: [BrowserModule, FormsModule]
```

ngModel Directive

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

```
1 import { Component } from '@angular/core';
2 @Component({
3   selector: 'app-root',
4   template: `User Name : <input type="text" [(ngModel)]="userName" /><br />
5     {{ userName }}`,
6   styleUrls: ['./app.component.css'],
7 })
8 export class AppComponent {
9   userName: string = 'James Bond';
10 }
```

ANGULAR SERVICES

- The Angular Services are the piece of code or logic that are used to perform some specific task.
- A service can contain a value or function or combination of both. The Services in angular are injected into the application using the dependency injection mechanism.

Why do we need a service in Angular?

- Whenever you need to reuse the same data and logic across multiple components of your application, then you need to go for angular service.
- That means whenever you see the same logic or data-access code duplicated across multiple components, then you need to think about refactoring the same logic or data access code into a service.
- Without Angular Services, you would have to repeat the same logic or code in multiple components wherever you want this code or logic.

Step1: Creating Angular Service

- The angular service is composed of three things. You need to create an export class and you need to decorate that class with @Injectable decorator and you need to import the Injectable decorator from the angular core library.

syntax to create angular service:

```
import { Injectable } from '@angular/core';  
  
@Injectable()  
  
export class ServiceName {  
    //Method and Properties  
}
```

- Let say you want to create an angular service for fetching the student details and this student details is going to be used by many components.

```
PS D:\AngularProjects\MyAngularApp> ng generate service Student
```

- Once you press the enter button it will create two files within the app folder as shown below.

```
TS student.service.spec.ts  
TS student.service.ts
```

Modifying student.service.ts file:

```
import { Injectable } from '@angular/core';
@Injectable()
export class StudentService {
  getStudents(): any[] {
    return [
      {
        ID: 'std101', FirstName: 'xyz', LastName: 'abc',
        Branch: 'CSE', DOB: '29/02/2001', Gender: 'Female'
      },
      {
        ID: 'std101', FirstName: 'zzz', LastName: 'dbc',
        Branch: 'CSE', DOB: '12/07/2003', Gender: 'Male'
      },
      {
        ID: 'std101', FirstName: 'yyy', LastName: 'abc',
        Branch: 'CSE', DOB: '15/08/2002', Gender: 'Female'
      },
    ];
  }
}
```


Step2: Using the Service in Angular:

- Once you created the service, the next and the important step is to inject and use the service within the components.
- It is a three step process.

Import the service,
register the service and
use the service

```
-- Other Import statements  
import {StudentService} from './student.service';
```

Import the service

```
@Component({  
  -- Other Properties  
  
  providers:[StudentService]  
  
})
```

Register the service

```
export class AppComponent {  
  students: any[];  
  
  constructor(private _studentService: StudentService) {  
    this.students = this._studentService.getStudents();  
  }  
  
}
```

Use the service

Modifying app.component.ts file:

```
import { Component } from '@angular/core';
import { StudentService } from '../student.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  providers: [StudentService]
})
export class AppComponent {
  students: any[];

  constructor(private _studentService: StudentService) {
    this.students = this._studentService.getStudents();
  }
}
```

Modifying app.component.html file:

```
<table>
  <thead>
    <tr>
      <th>ID</th>
      <th>First Name</th>
      <th>Last Name</th>
      <th>Branch</th>
      <th>DOB</th>
      <th>Gender</th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor='let student of students'>
      <td>{{student.ID}}</td>
      <td>{{student.FirstName}}</td>
      <td>{{student.LastName}}</td>
      <td>{{student.Branch}}</td>
      <td>{{student.DOB}}</td>
      <td>{{student.Gender}}</td>
    </tr>
  </tbody>
</table>
```

ANGULAR PIPES

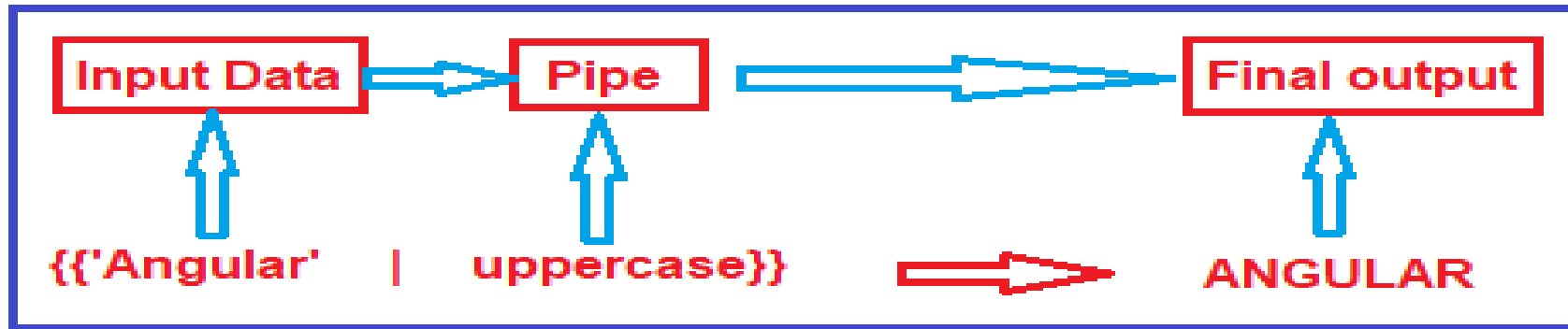
Why we need Angular Pipes?

- As we already know every web application starts with a simple task: first get the data, then transform the data into some format, and finally, show the formatted data to the users. Getting the data is very simple, you can create a local variable or a complex type to hold the data or even you may get the data from APIs.
- Once you get the data, then you could show the raw data as it is to the end-user, but that will not make a good user experience. To get a good user experience we need to modify the raw data into some specific format and in such cases, Angular Pipes plays an important role.

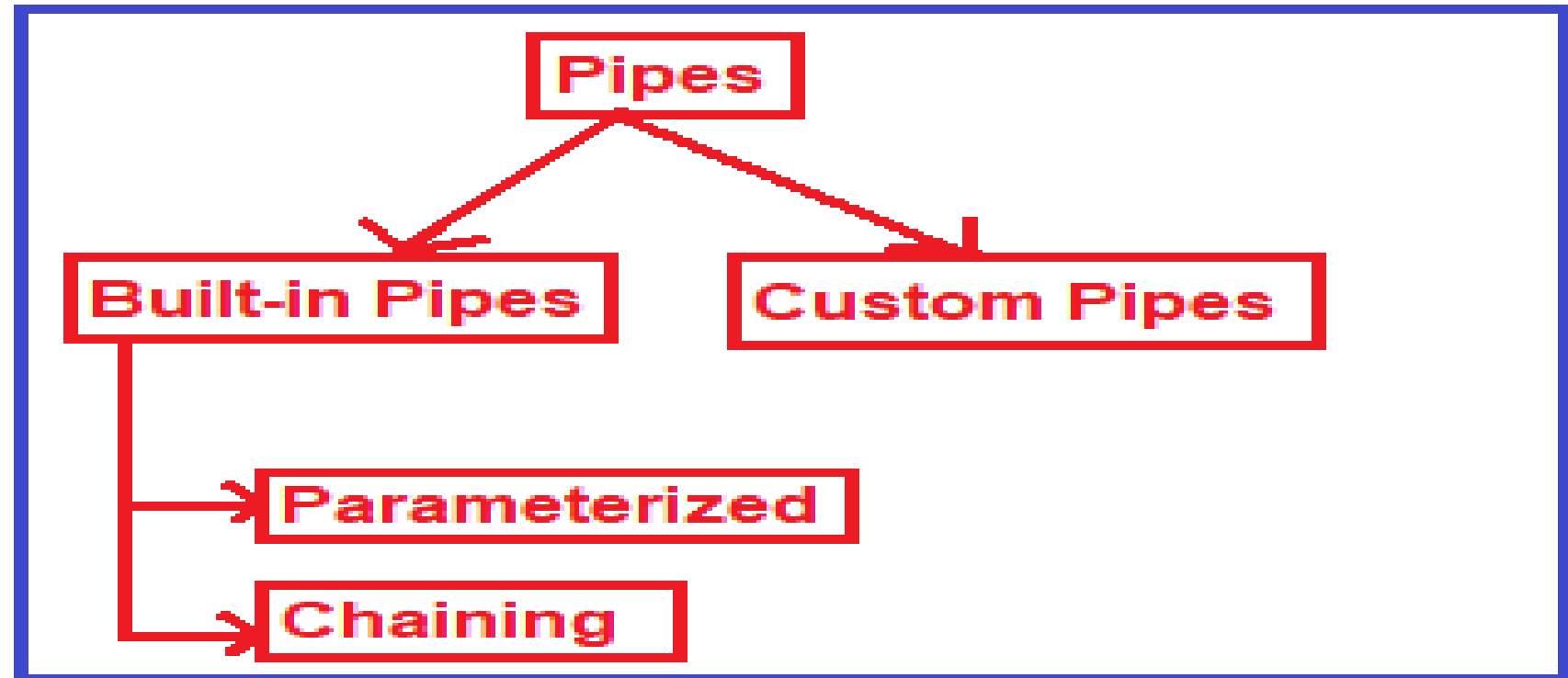
What are Pipes in Angular Application?

- The Angular Pipe takes the raw data as input and then transforms the raw data into some desired format.
- Using the Pipe (|) operator, we can apply the pipes features to any of the property in angular application.
- There are so many built-in pipes provides by Angular Framework such as lowercase, uppercase, titlecase, decimal, date, percent, currency etc. It is also possible to create custom pipes.

Syntax:



Types of Pipes in Angular:



built-in pipes:

lowercase:

This is used to convert the characters into lower case.

uppercase:

This is used to convert the characters into upper case.

titlecase:

This built-in pipe is used to convert the first character in each word to upper case.

date:

This pipe is used to convert a date to some specific format.

currency:

this pipe is used to convert number to currency with currency symbol.

Angular Parameterized Pipes:

- we can pass any number of parameters to the pipe using a colon (:) and when we do so, it is called Angular Parameterized Pipes.

Syntax:

`data | pipeName : parameter 1 : parameter 2 : parameter 3 ... parameter n`

Examples:

Date:- `{{DOB | date : "short"}}`

Currency:- `{{courseFee | currency : 'USD' : true : '1.3-3'}}`

Date Pipe:

```
export class AppComponent {  
    today: number = Date.now();  
}
```

Example:

`<p>Date Pipe : {{today | date}}</p>`

`<p>Full Date : {{today | date:'fullDate'}}</p>`

Currency Pipe:

- The Angular Currency Pipe is used to transform a number to a currency string, formatted according to locale rules that determine group sizing and separator, decimal-point character, and other locale-specific configurations.

Example:

```
export class AppComponent {  
    salary: number = 456723.50;  
}
```

<p>Currency USD in Symbol : {{salary | currency:'USD':true}}</p>

<p>Currency INR in Symbol : {{salary | currency:'INR':true}}</p>

<p>Currency USD in Code : {{salary | currency:'USD':false:'4.2-2'}}</p>

<p>Currency INR in Code : {{salary | currency:'INR':false:'1.3-3'}}</p>

where

The first parameter is the currency Code (i.e. USD or INR)

The second parameter is boolean – True to display the currency symbol where as false to display the currency code.

The third parameter ('1.3-3' or '4.2-2') specifies the number of integer and fractional digits.

Angular Custom Pipe:

- In order to create a custom pipe in angular, you have to apply the @Pipe decorator to a class which you can import from the Angular Core Library. The @Pipe decorator allows you to define the pipe name that you will use within the template expressions.

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

@Pipe({
  name: 'pipeName'
})

export class pipeClass implements PipeTransform {
  transform(parameters): returntype {}
}
```

Creating Angular Custom Pipe using Angular CLI:

- Let say we want to create a custom pipe with the name MyTitle. In order to create a custom MyTitle pipe open a new terminal and type **ng g pipe MyTitle -flat**

```
PS D:\AngularProjects\MyAngularApp> ng g pipe MyTitle --flat
CREATE src/app/my-title.pipe.spec.ts (192 bytes)
CREATE src/app/my-title.pipe.ts (219 bytes)
UPDATE src/app/app.module.ts (523 bytes)
```

```
Example:
Modify my-title.pipe.ts file:
import { Pipe, PipeTransform } from
 '@angular/core';
@Pipe({
    name: 'myTitle'
})
export class MyTitlePipe implements
 PipeTransform {
    transform(name: string, gender: string)
 string {
    if (gender.toLowerCase() == "male")
        return "Mr. " + name;
    else
        return "Miss. " + name;
    }
}

<table border="1">
    <thead>
        <tr>
            <th>Student ID</th>
            <th>Name</th>
            <th>DOB</th>
            <th>Gender</th>
            <th>CourseFee</th>
        </tr>
    </thead>
    <tbody>
        <tr *ngFor='let student of students'>
            <td>{{student.ID | uppercase}}</td>
            <td>{{student.Name | myTitle:student.Gender}}</td>
            <td>{{student.DOB | date:'dd/MM/yyyy'}}</td>
            <td>{{student.Gender | lowercase}}</td>
            <td>{{student.CourseFee
                                |currency:'USD':true}}</td>
        </tr>
    </tbody>
</table>
```

ANGULAR FORMS

Why we need Forms?

- Forms are the main building blocks of any type of application. When we use forms for login, registration, submission. Help request, etc.
- it is necessary that whatever forms we are developing, they should be user friendly. And it should have the indication of what went wrong by display user friendly message, etc.
- Forms are really very important to collect the data from the users. Often, each website contains forms to collect the user data.
- You can use forms to login, submit a help request, place an order, book a flight, schedule a meeting, and perform other countless data entry tasks.

What are Angular Forms?

- Developing forms requires design skill as well as framework support for two-way data binding, change tracking, validation, error handling, etc.
- The Angular Framework, provides two different ways to collect and validate the data from a user. They are as follows:

Template-Driven Forms

Model-Driven Forms (Reactive Forms)

Template-Driven Forms:

- Template Driven Forms are simple forms which can be used to develop forms. These are called Template Driven as everything that we are going to use in an application is defined into the template that we are defining along with the component.
- In order to use Template Driven Forms, we need to import `FormsModule` into the application root module i.e. `app.module.ts` file.
- Features of Template Driven Forms:
 - Easy to use.
 - Suitable for simple scenarios and fail for complex scenarios.
 - Similar to Angular 1.0 (Angular JS)
 - Two way data binding using `NgModule` syntax.
 - Minimal Component code
 - Automatic track of the form and its data.
 - Unit testing is another challenge

Model-Driven Forms (Reactive Forms) in Angular:

- In a model driven approach, the model which is created in the .ts file is responsible for handling all the user interactions and validations.
- For this, first, we need to create the model using Angular's inbuilt classes like `FormGroup` and `FormControl` and then we need to bind that model to the HTML form.
- This approach uses the Reactive forms for developing the forms which favor the explicit management of data between the UI (User Interface) and the Model. With this approach, we create the tree of Angular Form Controls and bind them in the Native Form Controls. As we create the form controls directly in the component, it makes it a bit easier to push the data between the data models and the UI elements.
- In order to use Reactive Forms, you need to import `ReactiveFormsModule` into the applications root module i.e. `app.module.ts` file.
- Features of Reactive Forms:
 - More flexible, but need a lot of practice
 - Handles any complex scenarios.
 - No data binding is done (Immutable data model preferred by most developers).
 - More component code and less HTML Markup.
 - Easier unit testing.
 - Reactive transformations can be made possible such as
 - Handling a event based on a denounce time.
 - Handling events when the components are distinct until changed.
 - Adding elements dynamically.

- Importing Forms module
- Create a Registration Form

NgForm:

- It is the directive which helps to create the control groups inside form directive. It is attached to the <form> element in HTML and supplements form tag with some additional features.

NgModel:

- When we add ngModel directive to the control, all the input elements are registered with the NgForm.
- It created the instance of the FormControl class from Domain model and assign it to the form control elements.
- The control keeps track of the user information and the state and the validation status of the form control.
- Next important thing is to consider is that when we use ngModel with form tag, then we should have to use the name property of the HTML control.
- Two main functionalities are provided by NgForm and NgModel are the permission to retrieving the values of the control associated with the form and then retrieving the overall state of the controls in the form.

```
<form #studentForm="ngForm" (ngSubmit)="RegisterStudent(studentForm)">
```

- studentForm is called the template reference variable and if you notice we have assigned “ngForm” as the value for the template reference variable studentForm. So the studentForm reference variable holds a reference to the form.
- Now the questions arises, whether or not we need to use this local variable. Well the answer is no. We are exporting ngForm in the local variable just to use some of the properties of the form and these properties are as follows:

studentForm.value :

It gives the object containing all the values of the field contain in the form.

studentForm.valid :

This gives the value indicating if the form is valid or not. If it is valid then the value is true else the value is false.

studentForm.touched :

It returns true or false when one of the field in the form is touched or entered.

As you can see, The form tag is not associated with any action method, then the question is how we post the form data to the component. The answer is using ngSubmit directive.

Understanding ngSubmit directive:

- Please have a look at the following ngSubmit directive. Here, we are using the Event Binding concept and we binding to the RegisterStudent method of the component. Instead of the submit event of the form, we are using ngSubmit which will send the actual HTTP request instead of just submitting the form.

```
(ngSubmit)="RegisterStudent(studentForm)"
```

- The ngSubmit directive will submits the form when we either hit the enter key or when we click the Submit button.
- When the form is submitted, RegisterStudent() method of the AppComponent class is called and we are passing it the studentForm.
- We do not have this method at the moment in the AppComponent class. We will create this method in just a bit.


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

@Component({
  selector: 'app-forms',
  templateUrl: './forms.component.html',
  styleUrls: ['./forms.component.css']
})
export class FormsComponent implements OnInit {
  constructor() { }
  ngOnInit(): void {
  }
  RegisterStudent(studentForm: NgForm): void {
    console.log(studentForm.value);
  }
}
```

How to fetch the form values and store in local variable:

- If you want to fetch the form control values and store in local variables then you need to modify the RegisterStudent method as shown below.

```
export class AppComponent {  
  RegisterStudent(studentForm: NgForm): void {  
  
    var firstName = studentForm.controls.firstName.value;  
    var lastName = studentForm.controls.lastName.value;  
    var email = studentForm.controls.email.value;  
  
    console.log(studentForm.value);  
  }  
}
```

SHARING DATA BETWEEN THE COMPONENTS

- Small components are always good to manage code in Angular. When you start writing small components then you have to share data between the components.
- These Components are useless if they are not able to communicate with each other. They need to communicate with each other if they want to serve any useful purpose.

Passing data from parent to child:

- The Parent Component can communicate with the child component by setting its Property.
- To do that the Child component must expose its properties to the parent component.
- The Child Component does this by using the **@Input decorator**

In the Child Component

- Import the @Input module from @angular/Core Library
- Mark those property, which you need data from parent as input property using @Input decorator

In the Parent Component

- Bind the Child component property in the Parent Component when instantiating the Child

Passing data to Parent Component:

- There are three ways in which parent component can interact with the child component
- Parent Listens to Child Event
- Parent uses Local Variable to access the child
- Parent uses a `@ViewChild` to get reference to the child component

Parent listens for child event

- The Child Component exposes an `EventEmitter` Property. This Property is adorned with the `@Output` decorator.
- When Child Component needs to communicate with the parent it raises the event. The Parent Component listens to that event and reacts to it.

How to Pass data to parent component using `@Output`

In the child component

Declare a property of type `EventEmitter` and instantiate it

Mark it with a `@Output` annotation

Raise the event passing it with the desired data

In the Parent Component

Bind to the Child Component using Event Binding and listen to the child events

Define the event handler function

Parent uses local variable to access the Child in Template

- Parent Template can access the child component properties and methods by creating the template reference variable
- The Template Reference variable is created, when you use `#<varibaleName>` and attach it to a DOM element. You can then, use the variable to reference the DOM element in your Template
- `<child-component #child></child-component>`
- The local variable can be used elsewhere in the template to refer to the child component methods and properties
- The parent-child binding must be done entirely within the parent template
- The parent component itself has no access to the child properties or methods using local variable.

Parent uses a @ViewChild() to get reference to the Child Component

- Injecting an instance of the child component into the parent as a @ViewChild is the another technique used by the parent to access the property and method of the child component.
- The @ViewChild decorator takes the name of the component/directive as its input. It is then used to decorate a property. The Angular then injects the reference of the component to the Property.
- In the Parent component, declare a property child which is of type ChildComponent

```
child: ChildComponent
```

- Next, decorate it with @ViewChild decorator passing it the name of the component to inject.

```
@ViewChild(ChildComponent) child : ChildComponent
```

- Now, the parent can access the properties and methods of child component