



Angular 11

# 1.1 About Angular

- Angular is one of the most powerful and performance-efficient JavaScript frameworks to build **single-page applications** for both web and mobile. The powerful features of Angular allow us to create complex, customizable, modern, responsive, and user-friendly web applications.
- Angular follows a **component-oriented** application design pattern to develop completely reusable and modularized web applications.
- Popular web platforms like Google Adwords, Google Fiber, Adsense have built their user interfaces using Angular.

# 1.2 Why Angular

## **Cross-Browser Compliant**

Internet has evolved significantly from the time Angular 1.x was designed. Creating a web application that is cross-browser compliant was difficult with Angular 1.x framework. Developers had to come up with various workarounds to overcome the issues. Angular helps to create cross-browser compliant applications easily.

## **Typescript Support**

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

## **Web Components Support**

Component-based development is pretty much the future of web development. Angular is focused on component-based development. The use of components helps in creating loosely coupled units of application that can be developed, maintained, and tested easily.

# 1.2 Why Angular

## **Better support for Mobile App Development**

Desktop and mobile applications have separate concerns and addressing these concerns using a single framework becomes a challenge. Angular 1 had to address the concerns of a mobile application using additional plugins. However, the Angular framework, addresses the concerns of both mobile as well as desktop applications.

## **Better performance**

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

# 1.3 SPA

- 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.
- For example, have a look at the Amazon web application. When you click on the various links present in the navbar present in any of the web pages of this application, the whole page gets refreshed. This is because visibly, a new request is sent for the new page for almost each user click. You may hence observe that it is not a SPA.
- But, if you look at the Gmail web application, you will observe that all user interactions are being handled without completely refreshing the page.
- Modern web applications are generally SPAs. SPAs provide a good user experience by communicating asynchronously (a preferable way of communication) with a remote web server (generally using HTTP protocol) to dynamically check the user inputs or interactions and give constant feedback to the user in case of any errors, or wrongful/invalid user interaction. They are built block-by-block making all the functionalities independent of each other. All desktop apps are SPAs in the sense that only the required area gets changed based on user requests.
- Angular helps to create SPAs that will dynamically load contents in a single HTML file, giving the user an illusion that the application is just a single page.

# 1.4 TypeScript

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

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

TypeScript is Microsoft's extension for JavaScript which supports object-oriented features and has a strong typing system that enhances 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 the browser understand the code.

# 1.5 Features of Angular

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

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

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

**Cross-Platform:** Angular is a single platform that can be used to develop applications for multiple devices.

**Performance:** Angular performance has been improved a lot in the latest version. This has been done by automatically adding or removing reflect metadata from the polyfills.ts file which makes the application smaller in production.

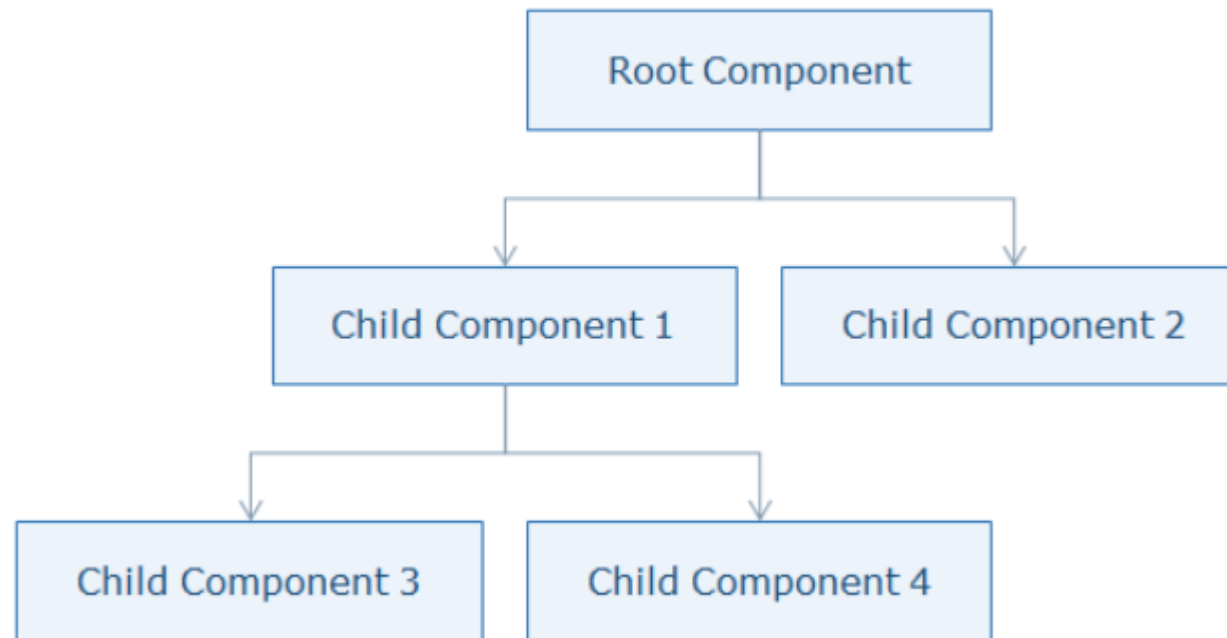
**Lean and Fast:** Angular application's production bundle size is reduced by 100s of kilobytes due to which it loads faster during execution.

**Bundle Budgets:** Angular will take advantage of the bundle budgets feature in CLI which will warn if the application size exceeds 2MB and will give errors if it exceeds 5MB. Developers can change this in angular.json.

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

# 1.6 Component based

- Angular follows component-based programming which is the future of web development. Each component created is isolated from every other part of our application. This kind of programming allows us to use components written using other frameworks.
- Inside a component, you can write both business logic and view.
- Every Angular application must have one top-level component referred to as 'Root Component' and several sub-components or child components.





## 2. Setup

To develop an application using Angular on a local system, you need to set up a development environment that includes the installation of:

Node.js (12.14.1 || >=14.0.0) and npm (min version required 6.13.4)

Angular CLI

Visual Studio Code

Install Node.js and Visual Studio Code from their respective official websites.

### Install Angular CLI

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

### Check Version

```
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 them, and perform development tasks such as testing, bundling, and deployment of applications.

## 2. Setup

### Command

`npm install -g @angular/cli`

`ng new <project name>`

`ng serve --open`

`ng generate <name>`

`ng build`

`ng update @angular/cli @angular/core`

### Purpose

Installs Angular CLI globally

Creates a new Angular application

Builds and runs the application on lite-server and launches a browser

Creates class, component, directive, interface, module, pipe, and service

Builds the application

Updates Angular to a newer version

## 2. Setup

### Demo steps:

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

```
ng new MyApp
```

2. The above command will display two questions. The first question is as shown below. Typing 'y' will create a routing module file (app-routing.module.ts).

```
? Would you like to add Angular routing? (y/N) y
```

3. The next question is to select the stylesheet to use in the application. Select CSS.

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

## 2. Setup

### 4. Run angular application

**ng serve** will build and run the application

```
ng serve --open
```

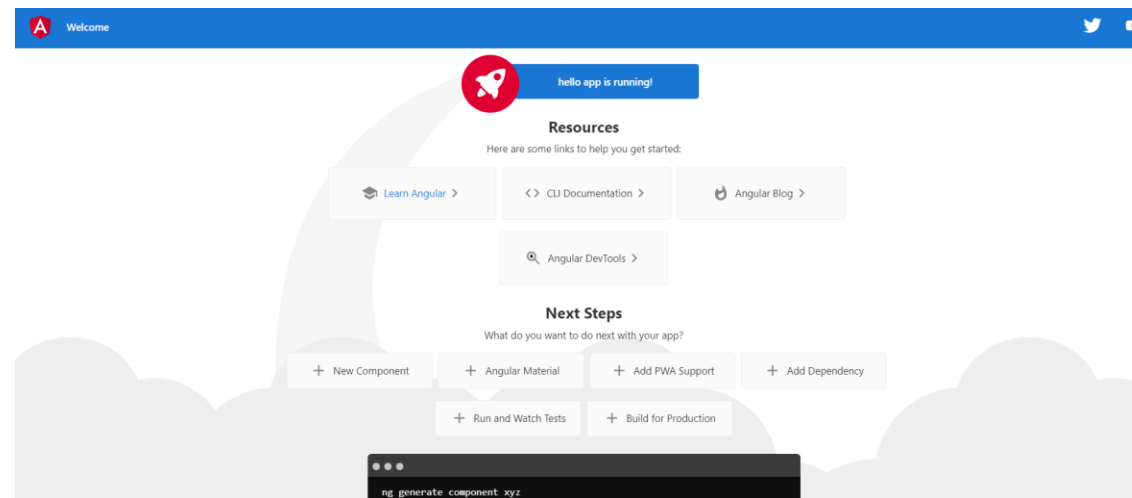
**--open** option will show the output by opening a browser automatically with the default port.

```
ng serve --open --port 3000
```

Use the following command to change the port number if another application is running on the default port(4200)

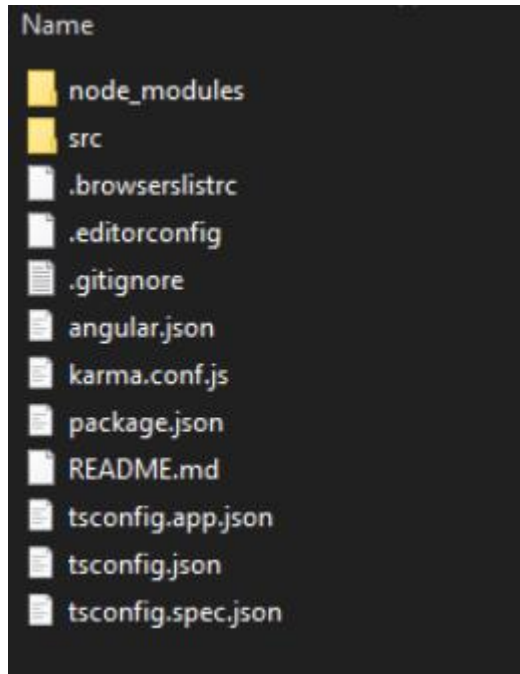
**Output:**

**http://localhost:4200/**



## 2. Setup

### Folder structure:



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

# 3. Create component and module

## **Why Components in Angular?**

A component is the basic building block of an Angular application

It emphasize the separation of concerns and each part of the Angular application can be written independently of one another

It is reusable

Observe for our AppComponent you have below files

1. app.component.ts
2. app.component.html
3. app.component.css
4. app.component.spec.ts

**@Component:** Adds component decorator to the class which makes the class a component

**selector:** Specifies the tag name to be used in the HTML page to load the component

**templateURL:** 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

**styleUrls:** Specifies the stylesheet file which contains CSS styles to be applied to the template.

**export:** Every component is a class (AppComponent, here) and export is used to make it accessible in other components

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

app.component.ts

# 3. Create component and module

## Best Practices - Coding Style Rules

Always **write one component per file**. This will make it easier to read, maintain, and avoid hidden bugs. It makes code reusable and less mistake-prone.

Always define **small functions** which makes it easier to read and maintain

The recommended pattern for file naming convention is **feature.type.ts**. For example, to create a component for Login, the recommended filename is login.component.ts. Use the upper camel case for class names. For example, LoginComponent.

Use a **dashed case** for the selectors of a component to keep the names consistent for all custom elements. Ex: app-root



# 3. Create component and module

## Best Practices - Coding Style Rules

Always **write one component per file**. This will make it easier to read, maintain, and avoid hidden bugs. It makes code reusable and less mistake-prone.

Always define **small functions** which makes it easier to read and maintain

The recommended pattern for file naming convention is **feature.type.ts**. For example, to create a component for Login, the recommended filename is login.component.ts. Use the upper camel case for class names. For example, LoginComponent.

Use a **dashed case** for the selectors of a component to keep the names consistent for all custom elements. Ex: app-root

## 3.2 Modules

Modules in Angular are used to organize the application. It sets the execution context of an Angular application.

A module in Angular is a class with the **@NgModule** decorator added to it. @NgModule metadata will contain the **declarations** of components, pipes, directives, services that are to be used across the application.

Every Angular application should have **one root module** which is loaded first to launch the application.

Submodules should be configured in the root module.

imports **BrowserModule** class  
which is needed to run the  
application inside the browser

imports **NgModule** class to define  
metadata of the module

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

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

**imports** property should contain all  
module classes to be used across  
the application

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

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

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

app.module.ts

**providers** property should contain all service classes. You will learn about the services later in this course

**bootstrap** declaration should contain the root component to load. In this example, AppComponent is the root component that will be loaded in the HTML page

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

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

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

app.module.ts

Line 1: imports **enableProdMode** from the core module

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

Line 4: import **AppModule** which is the root module to bootstrap

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

Line 7: checks if you are working in a production environment or not

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

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

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

import { AppModule } from '../app/app.module';
import { environment } from '../environments/environment';

if (environment.production) {
  enableProdMode();
}

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

Main.ts

**<app-root>** loads the root component in the HTML page.

app-root is the selector name given to the component. This will execute the component and renders the template inside the browser.

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

Index.html

## 3.2 Module

### Best Practices - Coding Style Rules

Always add a **Module** keyword to the end of the module class name which provides a consistent way to quickly identify the modules. Ex: AppModule.

Filename convention for modules should end with **module.ts**

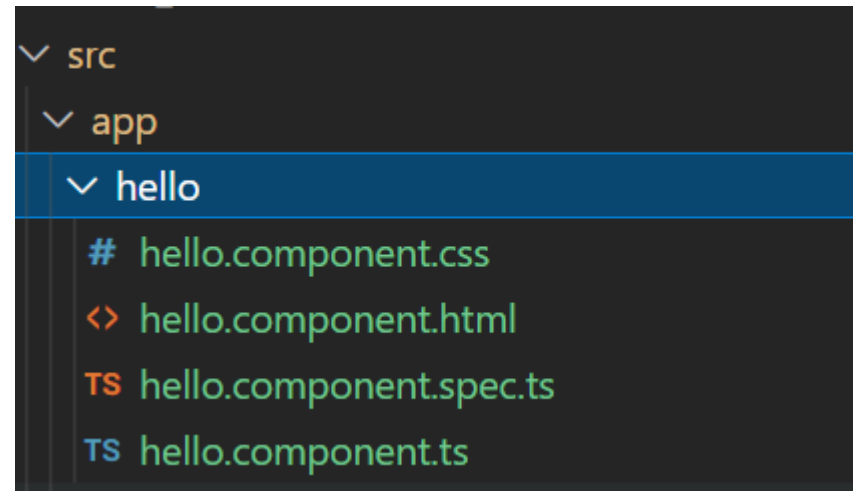
Always name the module with the feature name and the folder it resides in which provides a way to easily identify it.

## 3.3 Creating a component

Create a new component called hello using the following CLI command

```
ng generate component hello
```

This command will create a new folder with the name hello with the following files placed inside it



Check app.module.ts , component will be added in declarations

Change in index.html, add component selector instead of root component selector, change in bootstrap in app.module.ts



# 3.4 Templates

## Introduction to Templates

- Templates separate the **view layer from the rest of the framework**.
- You can change the view layer without breaking the application.
- Templates in Angular represents a view and its role is to display data and change the data whenever an event occurs
- The default language for templates is **HTML**

## Creating a template

Template can be defined in two ways:

- Inline Template
- External Template

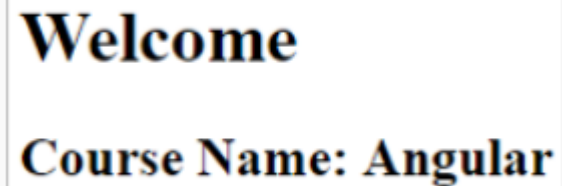
## 3.4 Templates

### Inline Template:

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

@Component({
  selector: 'app-root',
  template: `
    <h1> Welcome </h1>
    <h2> Course Name: {{ courseName }}</h2>
  `,
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  courseName = "Angular";
}
```

Use backtick character ` for multi-line strings



The screenshot shows a rectangular box with a thin black border. Inside the box, the text "Welcome" is displayed on the first line, and "Course Name: Angular" is displayed on the second line. Both lines of text are rendered in a bold, black, serif font.

**Welcome**

**Course Name: Angular**

## 3.4 Templates

### External Template:

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

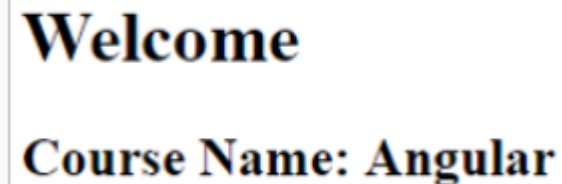
app.component.html

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

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

app.component.ts

**templateUrl** property is used to bind an external template file with the component



Welcome

Course Name: Angular

# 3.4 Elements of Templates

## Introduction to Templates

- Templates separate the **view layer from the rest of the framework**.
- You can change the view layer without breaking the application.
- Templates in Angular represents a view and its role is to display data and change the data whenever an event occurs
- The default language for templates is **HTML**

## Creating a template

Template can be defined in two ways:

- Inline Template
- External Template

## 3.4 Elements of Templates

basic elements of template syntax:

- HTML
- Interpolation
- Template Expressions
- Template Statements

### HTML

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

### Interpolation

Interpolation is one of the forms of data binding where component's data can be accessed in a template. For interpolation, double curly braces `{{ }}` is used.

### Template Expressions

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

```
{{ expression }}
```

## 3.4 Elements of Templates

### 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 you write `{{ courseName }}`, `courseName` should be the property of the component to which this template is bound.

### Template Statement

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

```
(event) = statement
```

## 3.4 Elements of Templates

### Change Detection

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

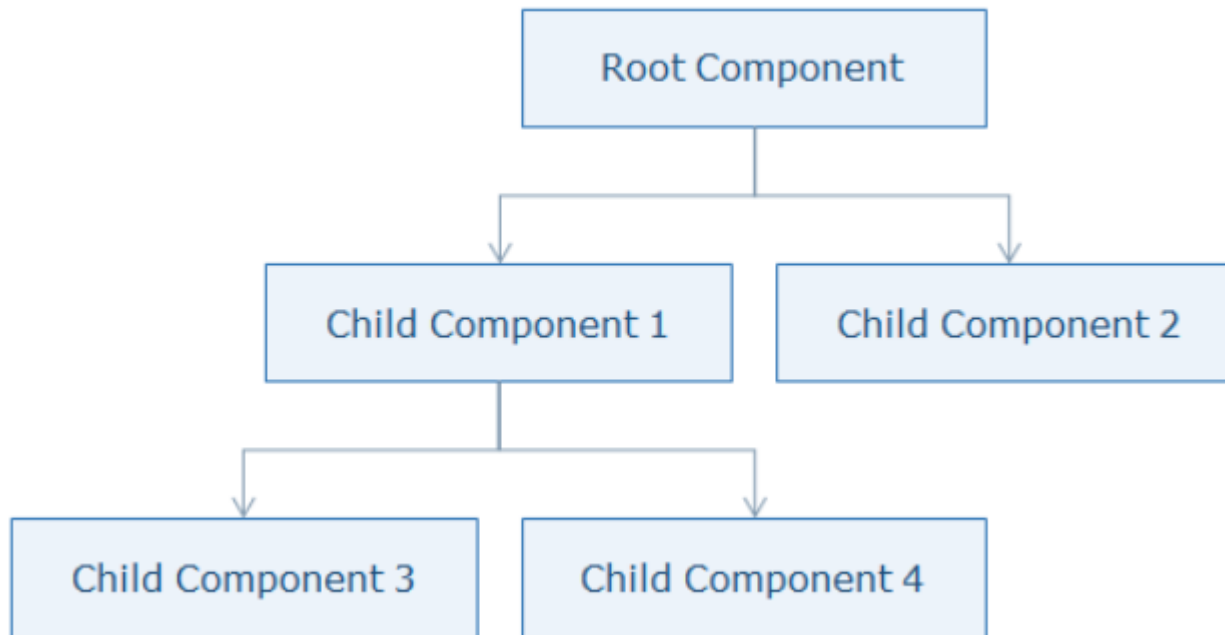
- Change Detection is a process in Angular that keeps views in sync with the models.
- In Angular, the flow is **unidirectional** from top to bottom in a component tree. A change in a web application can be caused by events, Ajax calls, and timers which are all asynchronous.

Who informs Angular about the changes?

- **Zones** inform Angular about the changes in the application. It automatically detects all asynchronous actions at run time in the application.

## 3.4 Elements of Templates

- Angular runs a **change detector algorithm** on each component from top to bottom in the component tree. This change detector algorithm is automatically generated at run time which will check and update the changes at appropriate places in the component tree.
- Angular is very fast though it **goes through all components from top to bottom for every single event** as it generates VM-friendly code. Due to this, Angular can perform hundreds of thousands of checks in a few milliseconds.





## 3.5 Directives

Now that you are familiar with the concept of templates, let us now understand directives in Angular.

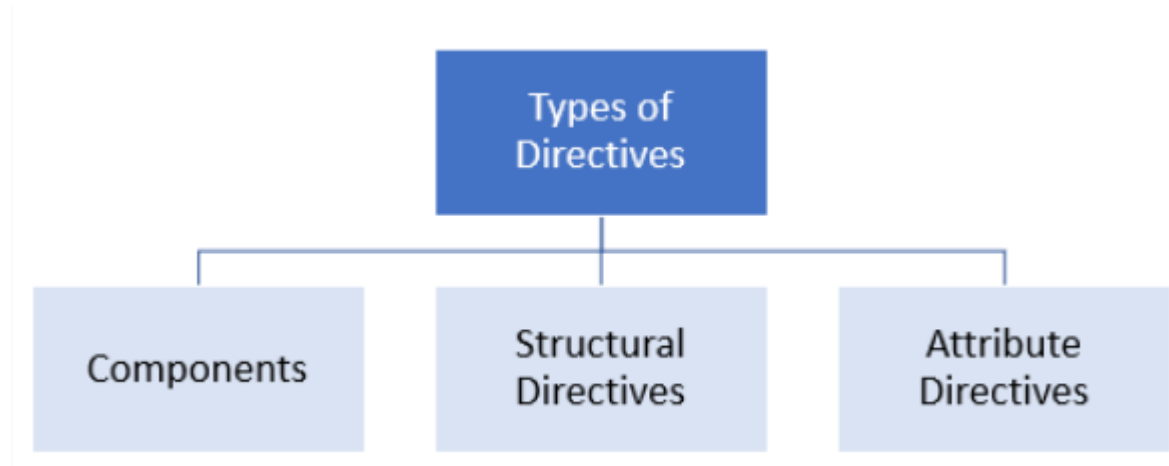
Directives are used **to change the behavior of components or elements**. It can be used in the form of HTML attributes.

You can create directives using classes attached with **@Directive** decorator which adds metadata to the class.

### Why Directives?

- It modify the DOM elements
- It creates reusable and independent code
- It is used to create custom elements to implement the required functionality

## 3.5 Directives



### Components

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

### Structural Directives

- A Structural directive changes the DOM layout by adding and removing DOM elements

```
*directive-name = expression
```

## 3.5 Directives

Angular has few built-in structural directives such as:

- `ngIf`
- `ngFor`
- `ngSwitch`

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

```
*ngIf = "expression"
```

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

```
*ngFor = "expression"
```

**ngSwitch** adds or removes DOM trees when their expressions match the switch expression. Its syntax is comprised of two directives, an attribute directive, and a structural directive.

It is very similar to a switch statement in JavaScript and other programming languages.

# 3.5 Directives

## Attribute Directives

### NgClass:

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

```
[style.<cssproperty>] = "value"
```

```
[ngStyle]="{  
    color:colorName,  
    'font-weight':fontWeight,  
    borderBottom: borderStyle  
}">
```

# 3.5 Directives

## Attribute Directives

Attribute directives change the appearance/behavior of a component/element.

Following are built-in attribute directives:

- ngStyle
- ngClass

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

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

### NgStyle

```
[style.<cssproperty>] = "value"
```

```
[ngStyle]="{  
    color:colorName,  
    'font-weight':fontWeight,  
    borderBottom: borderStyle  
}">
```

## 3.6 Data Binding

Data Binding is a mechanism where data in view and model are in sync. Users should be able to see the same data in a view which the model contains.

As a developer, you need to bind the model data in a template such that the actual data reflects in the view.

There are **two types of data bindings** based on the direction in which data flows.

- One-way Data Binding
- Two-way Data Binding

Data Direction	Syntax	Binding Type
One-way (Class -> Template)	<code>{{ expression }}</code> <code>[target] = "expression"</code>	<ul style="list-style-type: none"><li>•Interpolation</li><li>•Property</li><li>•Attribute</li><li>•Class</li><li>•Style</li></ul>
One-way (Template - > Class)	<code>(target) = "statement"</code>	<ul style="list-style-type: none"><li>•Event</li></ul>
Two-way	<code>[(target)] = "expression"</code>	<ul style="list-style-type: none"><li>•Two way</li></ul>

## 3.6 Data Binding

**Property binding** is used when its required to set the property of a class with the property of an element

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

Property binding will not work for a few elements/pure attributes like ARIA, SVG, and table span. In such cases, you need to go for attribute binding.

**Attribute binding** can be used to **bind a component property to the attribute directly**

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

```
[style.styleproperty]
```

## 3.6 Data Binding

### **Event Binding**

User actions such as entering text in input boxes, picking items from lists, button clicks may result in a flow of data in the opposite direction: from an element to the component.

Event binding syntax consists of a target event with ( ) on the left of an equal sign and a template statement on the right.

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

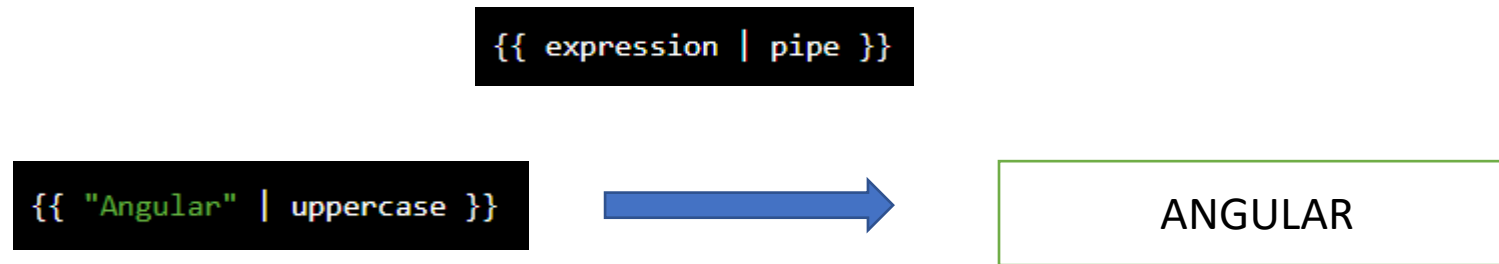
### **Two Binding**

Two-way data binding is a mechanism where if model property value changes, it updates the element to which the property is bound and vice versa. It uses [()] (banana in a box) syntax.



## 3.7 Pipes for Data Transformation

- Pipes are used to format the data before displaying it to the user. A pipe takes expression value as an input and transforms it into the desired output.



## 3.7.1 Built In Pipes

### **Why Pipes?**

Pipes is a beautiful way of transforming the data inside templates. It provides a clean and structured code.

Following is the list of built-in pipes available:

1. uppercase
2. lowercase
3. titlecase
4. currency
5. date
6. percent
7. slice
8. decimal
9. json
10. i18nplural
11. i18nselect

## 3.7.2 Passing Parameters to Pipes

A pipe can also have optional parameters to change the output. To pass parameters, after a pipe name add a colon(:) followed by the parameter value.

```
pipename : parametervalue
```

A pipe can also have multiple parameters as shown below

```
pipename : parametervalue1:parametervalue2
```

## 3.7.2 Passing Parameters to Pipes

### currency

This pipe displays a currency symbol before the expression. By default, it displays the currency symbol \$

```
{{ expression | currency:currencyCode:symbol:digitInfo:locale }}
```

**currencyCode** is the code to display such as INR for the rupee, EUR for the euro, etc.

**symbol** is a Boolean value that represents whether to display currency symbol or code.

**code**: displays code instead of a symbol such as USD, EUR, etc.

**symbol** (default): displays symbol such as \$ etc.

**symbol-narrow**: displays the narrow symbol of currency. Some countries have two symbols for their currency, regular and narrow. For example, the Canadian Dollar CAD has the symbol as CA\$ and symbol-narrow as \$.

**digitInfo** is a string in the following format

{minIntegerDigits}.{minFractionDigits} - {maxFractionDigits}

minIntegerDigits is the minimum integer digits to display. The default value is 1

minFractionDigits is the minimum number of digits to display after the fraction. The default value is 0

maxFractionDigits is the maximum number of digits to display after the fraction. The default value is 3

**locale** is used to set the format followed by a country/language. To use a locale, we need to register the locale in the root module

## 3.7.2 Passing Parameters to Pipes

**digitInfo** is a string in the following format

`{minIntegerDigits}.{minFractionDigits} - {maxFractionDigits}`

`minIntegerDigits` is the minimum integer digits to display. The default value is 1

`minFractionDigits` is the minimum number of digits to display after the fraction. The default value is 0

`maxFractionDigits` is the maximum number of digits to display after the fraction. The default value is 3

**locale** is used to set the format followed by a country/language. To use a locale, we need to register the locale in the root module.

•

```
{{ expression | currency:currencyCode:symbol:digitInfo:locale }}
```

## 3.7.2 Passing Parameters to Pipes

### DATE

This pipe can be used to display the date in the required format

```
{{ expression | date:format:timezone:locale }}
```

An **expression** is a date or number in milliseconds

The **format** indicates in which form the date/time should be displayed. Following are the pre-defined options for it.

'medium': equivalent to 'MMM d, y, h:mm:ss a' (e.g. Jan 31, 2018, 11:05:04 AM)

'short': equivalent to 'M/d/yy, h:mm a' (e.g. 1/31/2018, 11:05 AM)

'long': equivalent to 'MMMM d, y, h:mm:ss a z' (e.g. January 31, 2018 at 11:05:04 AM GMT+5)

'full': equivalent to 'EEEE, MMMM d, y, h:mm:ss a zzzz' (e.g. Wednesday, January 31, 2018 at 11:05:04 AM GMT+05:30)

'fullDate': equivalent to 'EEEE, MMMM d, y' (e.g. Wednesday, January 31, 2018)

'longDate': equivalent to 'MMMM d, y' (e.g. January 31, 2018)

'mediumDate': equivalent to 'MMM d, y' (e.g. Jan 31, 2018)

'shortDate': equivalent to 'M/d/yy' (e.g. 1/31/18)

'mediumTime': equivalent to 'h:mm:ss a' (e.g. 11:05:04 AM)

'shortTime': equivalent to 'h:mm a' (e.g. 11:05 AM)

'longTime': equivalent to 'h:mm a' (e.g. 11:05:04 AM GMT+5)

'fullTime': equivalent to 'h:mm:ss a zzzz' (e.g. 11:05:04 AM GMT+05:30)

**Timezone** to be used for formatting. For example, '+0430' (4 hours, 30 minutes east of the Greenwich meridian) If not specified, the local system timezone of the end-user's browser will be used.

**locale** is used to set the format followed by a country/language. To use a locale, we need to register the locale in the root module.

## 3.7.2 Passing Parameters to Pipes

### PERCENT:

This pipe can be used to display the number as a percentage

```
{{ expression | percent:digitInfo:locale }}
```

**digitInfo** is a string in the following format

**{minIntegerDigits}.{minFractionDigits} - {maxFractionDigits}**

- minIntegerDigits is the minimum integer digits to display. The default value is 1
- minFractionDigits is the minimum number of digits to display after the fraction. The default value is 0.
- maxFractionDigits is the maximum number of digits to display after the fraction. The default value is 3.

**locale** is used to set the format followed by a country/language. To use a locale, we need to register the locale in the root module.

## 3.7.2 Passing Parameters to Pipes

### SLICE:

This pipe can be used to extract a subset of elements or characters from an array or string respectively.

```
{{ expression | slice:start:end }}
```

The **expression** can be an array or string

**start** represents the starting position in an array or string to extract items. It can be a

- positive integer which will extract from the given position till the end
- negative integer which will extract the given number of items from the end

**end** represents the ending position in an array or string for extracting items. It can be

- positive number that returns all items before the end index
- negative number which returns all items before the end index from the end of the array or string



## 3.7.2 Passing Parameters to Pipes

### JSON:

This pipe can be used to display the given expression in the form of a JSON string. It is mostly for debugging.

```
{{ expression | json }}
```

### i18nplural:

This pipe can be used to map numeric values against an object containing different string values to be returned. It takes a numeric value as input and compares it with the values in an object and returns a string accordingly.

```
{{ expression | i18nplural:mappingObject }}
```

### i18nselect:

This pipe is similar to i18nplural but evaluates a string value instead.

```
{{ expression | i18nselect:mappingObject }}
```

**mappingObject** is an object containing strings to be displayed for different values provided by the expression

## 3.7.3 Custom Pipe

We have explored built-in pipes so far. Supposed if you want to implement functionalities such as sorting, filtering, etc., go for custom pipes as there are no built-in pipes available.

We can create our own custom pipe by inheriting the **PipeTransform** interface.

PipeTransform interface has a **transform** method where we need to write custom pipe functionality.

```
ng generate pipe <<pipename>>
```

```
@Pipe({
  name: 'pipename'
})
export class classname implements PipeTransform {

  transform(value: any, ...args: any[]): any {
  }
}
```

transform method has two arguments, the first one is the value of the expression passed to the pipe and the second is the 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.

## 3.8 Services

Dependency Injection (DI) is a mechanism where the required resources will be injected into the code automatically.

Angular comes with an in-built dependency injection subsystem.

### **Why Dependency Injection?**

It is because DI:

- allows developers to reuse the code across applications.
- makes the code loosely coupled.
- makes application development and testing much easier.
- allows the developer to ask for the dependencies from Angular. There is no need for the developer to explicitly create/instantiate them.

## 3.8 Services

A service in Angular is a class that contains some functionality that can be **reused** across the application. A service is a **singleton** object. Angular services are a mechanism of abstracting shared code and functionality throughout the application.

Angular Services come as objects which are wired together using dependency injection.

Angular provides a few inbuilt services also can create custom services.

### 1. Generate Service

```
ng generate service <<service name>>
```

This creates two files

- <<service name>>.service.ts -> Used for writing Service Logic
- <<service name>>.service.spec.ts -> Used for Testing Service

## 3.8 Services

In <<service name >>.service.ts, you can find Injectable decorator

```
@Injectable({  
  providedIn: 'root'  
})
```

- @Injectable() decorator makes the class injectable into application components
- providedIn property registers the service at the root level (app module).

In app.module.ts add the service name in providers

```
providers: [/* <<Service Name>> */],
```

- When the Service is provided at the root level, Angular creates a **singleton** instance of the service class and injects the same instance into any class that uses this service class.
- In addition, Angular also optimizes the application if registered through providedIn property by removing the service class if none of the components use it.

## 3.8 Services

There is also a way to limit the scope of the service class by registering it in the providers' property inside the **@Component** decorator.

Providers in component decorator and module decorator are independent.

Providing a service class inside a component creates a **separate instance** for that component and its nested components.

```
@Component({  
  selector: 'app-root',  
  templateUrl: './app.component.html',  
  styleUrls: ['./app.component.css'],  
  providers: [/*<< service name >>*/],  
})
```

### Injecting a Service

The only way to inject a service into a component/directive or any other class is through a constructor(constructor injection)

```
constructor(/*private service: <ServiceClassName>*/){  
}
```

## 3.8 Services

There is also a way to limit the scope of the service class by registering it in the providers' property inside the **@Component** decorator.

Providers in component decorator and module decorator are independent.

Providing a service class inside a component creates a **separate instance** for that component and its nested components.

```
@Component({  
  selector: 'app-root',  
  templateUrl: './app.component.html',  
  styleUrls: ['./app.component.css'],  
  providers: [/*<< service name >>*/],  
})
```

### Injecting a Service

The only way to inject a service into a component/directive or any other class is through a constructor(constructor injection)

```
constructor(/*private service: <ServiceClassName>*/){  
}
```

## 3.8 Services Summary

- Use Services to share the data and functionality only as they are ideal for sharing them across the app.
- Always create a service with single responsibility as if it has multiple responsibilities, it will become difficult to test.
- Always use `@Injectable()` decorator, as Angular injector is hierarchical and when it is provided to a root injector, it will be shared among all the classes that need a service.
- If `@Injectable()` decorator is used, optimization tools used by Angular CLI production builds will be able to perform tree shaking and remove the unused services from the app.
- When two different components need different instances of a service, provide the service at the component level.



## 3.8 RxJS Observables

### **RxJS**

Reactive Extensions for JavaScript (RxJS) is a third-party library used by the Angular team.

RxJS is a reactive streams library used to work with **asynchronous streams of data**.

Observables, in RxJS, are used to represent asynchronous streams of data. Observables are a more advanced version of Promises in JavaScript

### **Why RxJS Observables?**

Angular team has recommended Observables for asynchronous calls because of the following reasons:

- 1.Promises emit a single value whereas observables (streams) emit many values
- 2.Observables can be cancellable where Promises are not cancellable. If an HTTP response is not required, observables allow us to cancel the subscription whereas promises execute either success or failure callback even if the results are not required.
- 3.Observables support functional operators such as map, filter, reduce, etc.,

## 3.9 Server Communication

### Server Communication using HttpClient

- Most front-end applications communicate with backend services using HTTP Protocol
- While making calls to an external server, the users must continue to be able to interact with the page. That is, the page should not freeze until the HTTP request returns from the external server. So, all HTTP requests are asynchronous.
- **HttpClient** from `@angular/common/http` to communicate must be used with backend services.
- Additional benefits of HttpClient include testability features, typed request and response objects, request and response interception, Observable APIs, and streamlined error handling.
- **HttpClientModule** must be imported 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. HTTP methods like get, post, put, and delete are made used off.
- JSON is the default response type for HttpClient

## 3.9 Server Communication

1. Add HttpClientModule to the **app.module.ts** to make use of HttpClient class.

```
...  
import { HttpClientModule } from '@angular/common/http';  
...  
  
@NgModule({  
  imports: [BrowserModule, HttpClientModule],  
  ...  
})  
export class AppModule { }
```

2. Inject HttpClient using constructor in the required component

```
constructor(private http: HttpClient) { }
```

## 3.9 Server Communication

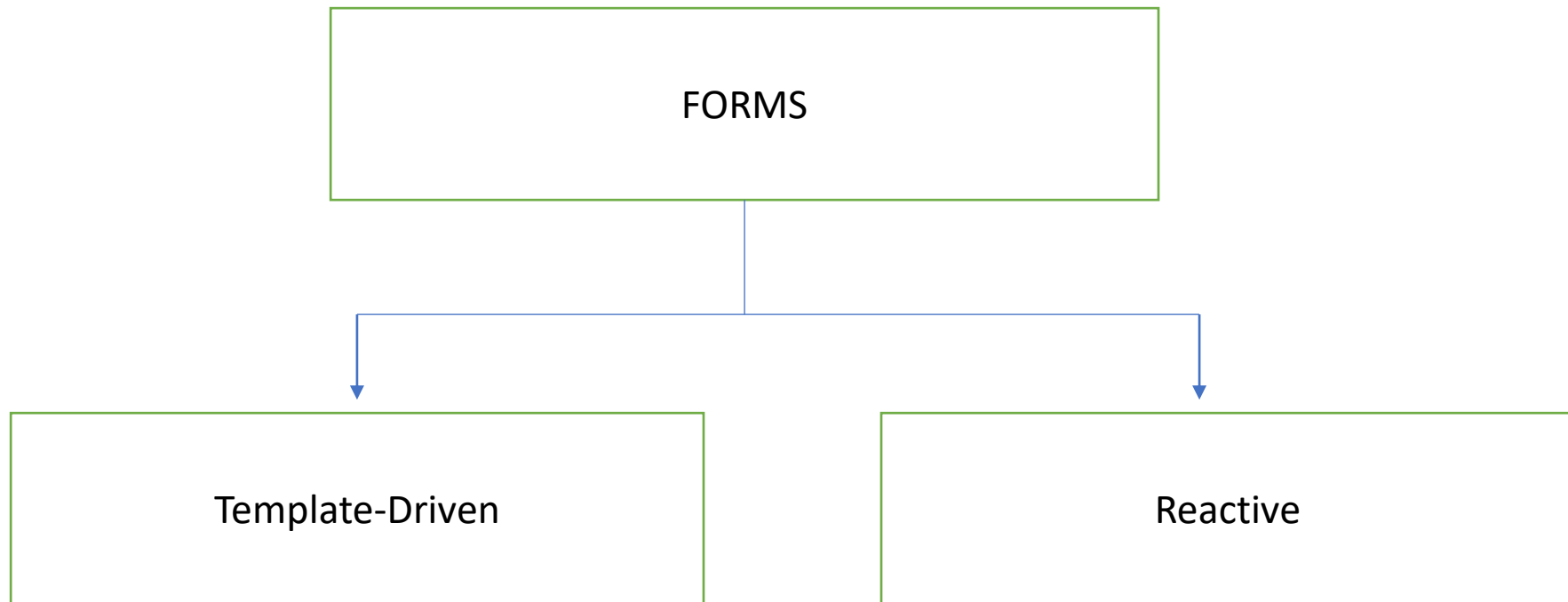
1. Add HttpClientModule to the **app.module.ts** to make use of HttpClient class.

```
...  
import { HttpClientModule } from '@angular/common/http';  
...  
  
@NgModule({  
  imports: [BrowserModule, HttpClientModule],  
  ...  
})  
export class AppModule { }
```

2. Inject HttpClient using constructor in the required component

```
constructor(private http: HttpClient) { }
```

## 3.10 Angular Forms



## 3.10 Template Driven

1. Import FormsModule in app.module.ts

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

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

## 3.10 Template Driven

ngForm:

- is a built-in directive that will have an instance of each form element created in the application
- has its own features which will get added to the form element on the page
- must have the name attribute as it is mandatory when [(ngModel)] is used on form elements
- input element is an instance of FormControl class which gets registered with the name attribute value
- ngModel is used to track the state and validity of an element using the following keywords

Keyword	Purpose
valid	true if element value is valid
invalid	true if element value is invalid
dirty	true if element value is changed
pristine	true if element value is unchanged
touched	true if the element gets focus
untouched	true if the element doesn't have a focus in it

## 3.10 Template Driven

Angular also has the following built-in CSS classes to change the appearance of the control based on its state

CSS Class	Purpose
ng-valid	Applied if control's value is valid
ng-invalid	Applied if control's value is invalid
ng-dirty	Applied if control's value is changed
ng-pristine	Applied if control's value is not changed
ng-touched	Applied if control is touched/gets focus
ng-untouched	Applied if control is not touched/doesn't get focus



## 3.10 Reactive Forms

Forms are crucial part of web applications through which majority of data input is received from users.

Forms can be created in a reactive style in Angular, for which form control objects must be created in a component class. They should also bind them with HTML form elements in the template.

A component can observe the form control state changes and react to them. Thus, to create reactive forms, **FormBuilder** class must be used, for which **ReactiveFormsModule** has to be imported.

Then, built-in validators using the Validators class can be used. For example, if 'required' validator is used, it can be accessed as Validators.required.

### **Following are the advantages of Reactive Forms:**

- Unit testing(using the Jasmine framework) on the validation logic can be performed, as it is written inside the component class.
- Form changes or events can be heard easily using reactive forms. Each FormGroup or FormControl has few events like valueChanges, statusChanges, etc., which can be subscribed to.
- Reactive forms are used in creating medium to large scale applications

## 3.11 Routing

Routing means navigation between multiple views on a single page.

Routing allows to express some aspects of the application's state in the URL. The full application can be built without changing the URL.

### Why Routing?

Routing allows to:

- Navigate between the views
- Create modular applications

Angular component router belongs to **@angular/router** module. To make use of routing, **Routes, RouterModule** classes must be imported.

Configuration should be done for the routes and the **router will look for a corresponding route when a browser URL is changed.**

**Routes is an array that contains all the route configurations.** Then, this array should be passed to the **RouterModule.forRoot()** function in the application bootstrapping function

## 3.11 Route Guards

In the Angular application, users can navigate to any URL directly. That's not the right thing to do always.

Consider the following scenarios

- Users must login first to access a component
- The user is not authorized to access a component
- User should fetch data before displaying a component
- Pending changes should be saved before leaving a component

These scenarios must be handled through route guards.

A guard's return value controls the behavior of the router

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

Angular has `canActivate` interface which can be used to check if a user is logged in to access a component

## 3.12 Nested Components

- Nested component is a component that is loaded into another component
- The component where we load another component is called a container component/parent component
- We are loading the root component in the index.html page using its selector name, similarly, if we want to load one component into another one, we will load it using its selector name in the template i.e., the HTML page of the container component

## 3.13 Passing data from Container Component to Child Component

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

## 3.14 Passing data from Child Component to Container Component

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

## 3.15 Component life cycle

A component has a life cycle that is managed by Angular. It includes creating a component, rendering it, creating and rendering its child components, checks when its data-bound properties change, and destroy it before removing it from the DOM.

Angular has some methods/hooks which provide visibility into these key life moments of a component and the ability to act when they occur.

Following are the lifecycle hooks of a component. The methods are invoked in the same order as mentioned in the table below:

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

## 3.15 Component life cycle

### **Lifecycle Hooks**

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