Angular 9

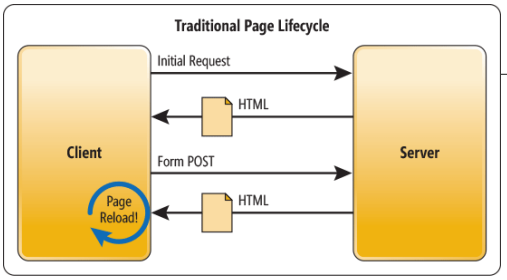
Chapter 1

Introduction to Angular

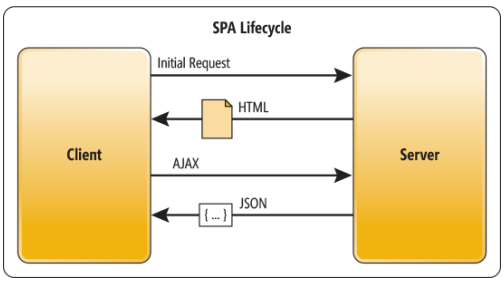
Introduction to Angular

- Angular is a JavaScript framework which makes you able to create reactive **Single Page Applications** (SPAs).
- Angular provides built-in features for animation, http service, and materials which in turn has features such as auto-complete, navigation, toolbar, menus, etc.
- The code is written in TypeScript, which compiles to JavaScript and displays the same in the browser.
- This is a leading front-end development framework which is regularly updated by Angular team of Google.
- Angular is completely based on components. It consists of several components forming a tree structure with parent and child components.

What is Single Page Application (SPA



- Multiple-page applications work in a “traditional” way. Every change eg. display the data or submit data back to server requests rendering a new page from the server in the browser.



- A single-page application is an app that works inside a browser and does not require page reloading during use. You are using this type of applications every day. These are, for instance: Gmail, Google Maps, Facebook or GitHub.

Pros of the Single-Page Application :

- SPA is fast, as most resources (HTML+CSS+Scripts) are only loaded once throughout the lifespan of application. Only data is transmitted back and forth.
- The development is simplified and streamlined. There is no need to write code to render pages on the server. It is much easier to get started because you can usually kick off development from a file file://URI, without using any server at all.
- SPAs are easy to debug with Chrome, as you can monitor network operations, investigate page elements and data associated with it.
- It's easier to make a mobile application because the developer can reuse the same backend code for web application and native mobile application.
- SPA can cache any local storage effectively. An application sends only one request, store all data, then it can use this data and works even offline.

Cons of the Single-Page Application :

- It is slow to download because heavy client frameworks are required to be loaded to the client.
- It requires JavaScript to be present and enabled. If any user disables JavaScript in his or her browser, it won't be possible to present application and its actions in a correct way.
- Compared to the "traditional" application, SPA is less secure. Due to Cross-Site Scripting (XSS), it enables attackers to inject client-side scripts into web application by other users.
- Memory leak in JavaScript can even cause powerful system to slow down
- In this model, back and forward buttons become dysfunctional.

Difference between AngularJS and Angular

AngularJS	Angular
AngularJS is common and popular name of the first version of Angular1.0.	Angular is common and popular name of the Angular's version beyond 2+
AngularJS is a JavaScript-based open-source front-end web framework.	Angular is a TypeScript-based open-source full-stack web application framework.
AngularJS uses the concept of scope or controller.	Instead of scope and controller, Angular uses hierarchy of components as its primary architectural characteristic.
AngularJS has a simple syntax and used on HTML pages along with the source location.	Angular uses the different expression syntax. It uses "[ ]" for property binding, and "( )" for event binding.
AngularJS is a simple JavaScript file which is used with HTML pages and doesn't support the features of a server-side programming language.	Angular uses of Microsoft's TypeScript language, which provides Class-based Object Oriented Programming, Static Typing, Generics etc. which are the features of a server-side programming language.
AngularJS doesn't support dynamic loading of the page.	Angular supports dynamic loading of the page.

Angular enhances HTML

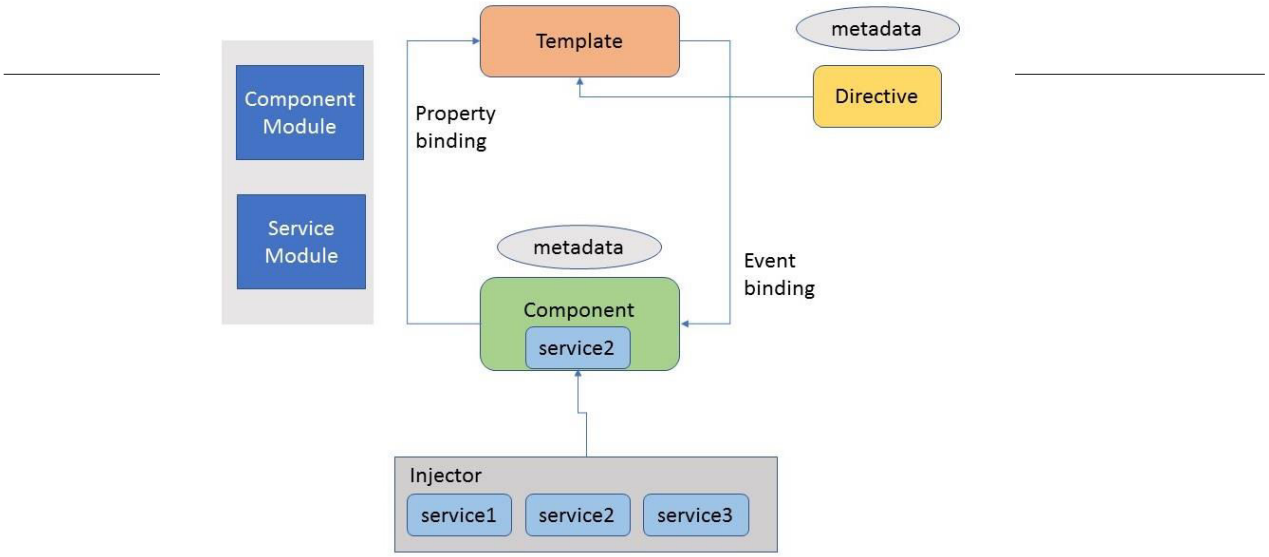
- Angular has set of directives to display dynamic contents at HTML pages. Angularextends HTML node capabilities for a web application.

- Angular provides data binding and dependency injection that reduces line of code.
- Angular extends HTML attribues with **Directives** and binds data to HTML with **Expressions**.
- Angular follows MVC Architecture.
- Angular communicates with **RESTFull** web services in modern applications.
- RESTFul web services are accessed by HTTP call
- RESTFul web services exchange JSON data

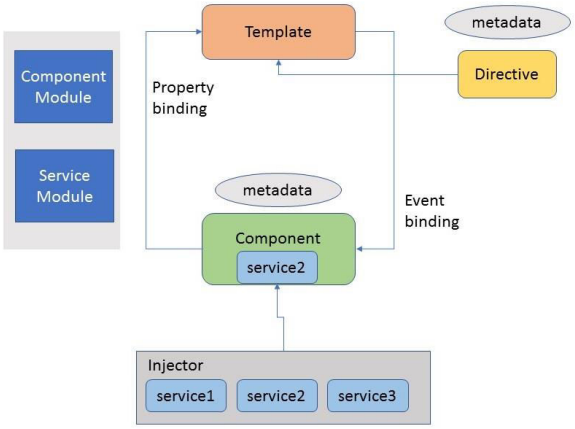
How to install and run Angular

- Install Visual code IDE from https://code.visualstudio.com
- Install NodeJS from https://nodejs.org/en/
- Use npm to install Angular CLI
  - npm install -g @angular/cli
- Create new Angular project
  - ng new my-dream-app
- Run angular project
  - cd my-dream-app
  - ng serve -o
- It will start angular server at default port number #4200 and make application accessible using http://localhost:4200

Architecture of Angular App

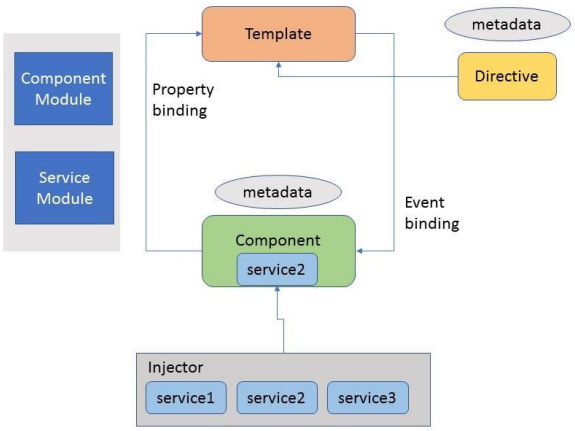


Template



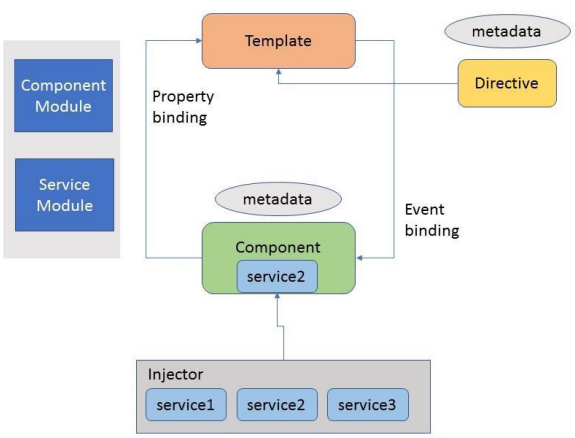
- Just like plain HTML, we can create angular templates which understand how to communicate with backing components.
- We can use plain html tags and component tags.
- These templates ultimately create DOM structure on web browser.
- Data in templates is held by backing components.

Component



- Components are the ones which hold data from view which is created because of templates.
- A Component and a Template collectively create a view.
- Components and corresponding templates communicate through property bindings and event bindings.

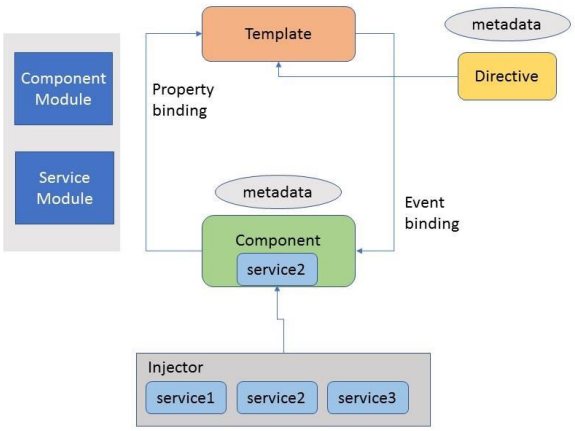
Data binding



- Data binding happens between template and component. It can be either one way or two way binding.

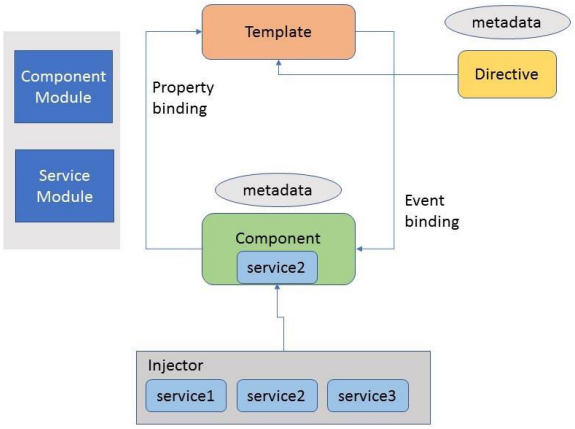
- Value binding is unidirectional and it is bound from component to template.
- Property binding is unidirectional and it is bound from component to template.
- Event binding is unidirectional and it is bound from template to component. When any event happens on DOM, then Component it notified.
- Two way data binding is bidirectional and combines property binding and event binding.

Metadata



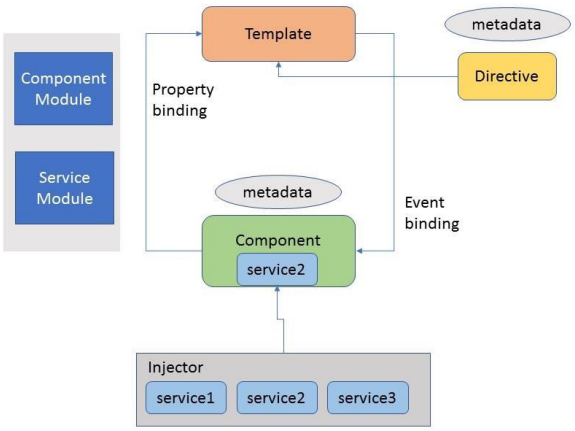
- Metadata provides all angular functionality to typescript classes. Without metadata, there is nothing angular about classes.
- Templates and components can communicate with the help of metadata.

Services



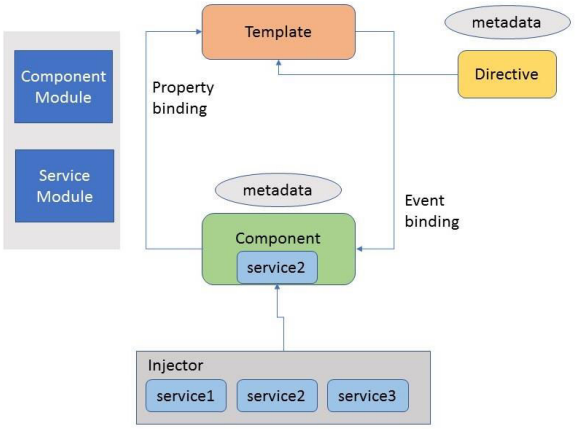
- Services are not a separate angular component. Anything can be service.
- Usually services include all the business logic and component merely consume it.
- It is good practice to have service layer in any application which is not dependent on view part.
- Usually services are injected in component where there are required using dependency injection mechanism.

Dependency injection



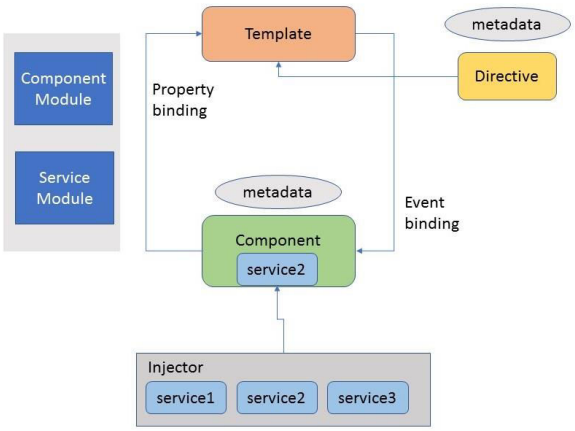
- Angular manages a pool of all service instances and stores them in a location.
- Whenever a component asks for a specific services, angular looks for that service in the pool.
- If service is found then it is given to component so that it can use it.
- If service is not found, a new instance is given to component and placed in pool so that other component can also be given.

Directives



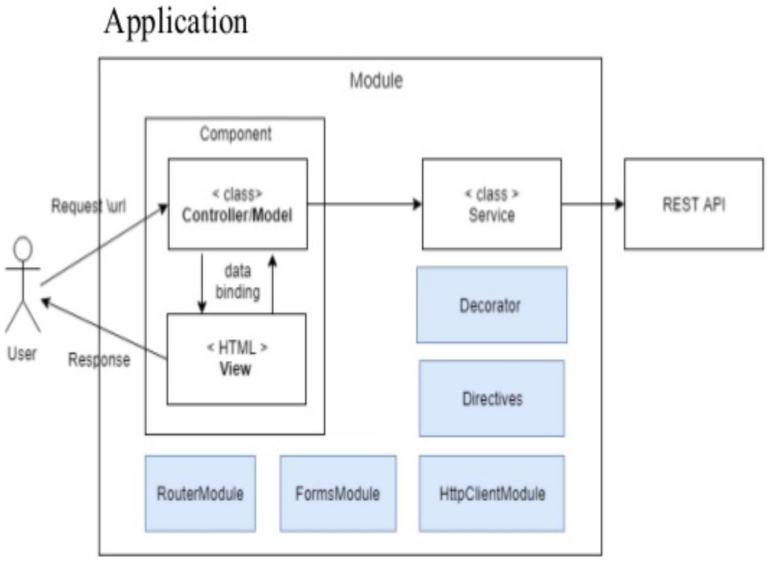
- Directives are classes marked with **@Directive** decoration.
- They can make structural or behavioural changes in angular apps.

Modules



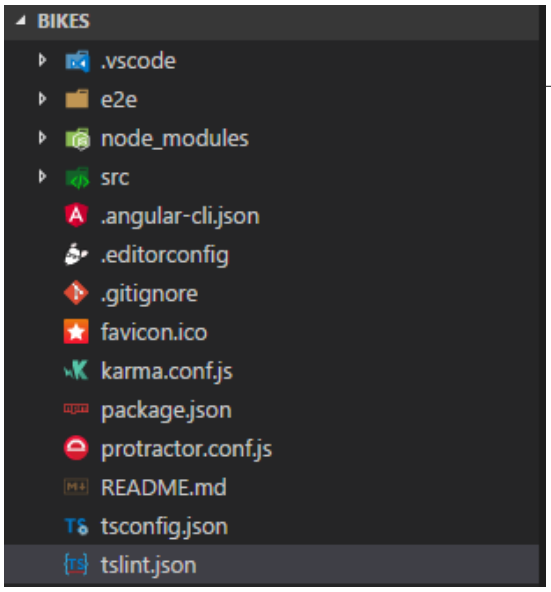
- Angular apps are modular. They are called NgModules.
- Angular apps have one angular class rootModule.
- **@NgModule** is a decorator function which takes a metadata object.
- We can define providers, imports, exports and declarations in **@NgModule**

MVC Architecture



- View : 
  Contains display logics, developed using HTML and Angular Directives
- Controller : 
  Contains navigation logic
  Decides data and view to be displayed
- Model : 
  Carry data between View and Controller

Angular project structure



- . **e2e** folder keeps files used for end to end testing
- **karma.conf.js** , **protractor.conf.js tsconfig.json, tslint.json** are respective configuration files.
- **package.json :** Defines dependencies required for our app.
- **node_modules:** Library of all dependencies.
- **src:** Actual place where we are going to write our code. It consists of components, pipes, directives, whatever we are going to develop.

Creating Component without Angular CLI

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

@Component({
    selector : 'app-title',
    template : `<div class="t3">
                    <h1 class="t1">SavitaSoft
                        <span class="t2">computer education</span>
                    </h1>
                </div>
                `,
    styles : ['.t1 {font-size: 40px; font-weight: bolder;color : white;padding : 2px}',
              '.t2 {font-size: 20px; font-weight: bolder;color : white;}',
              '.t3 {background-color:blue}'
            ]
})
export class AppTitleComponent{

}
```

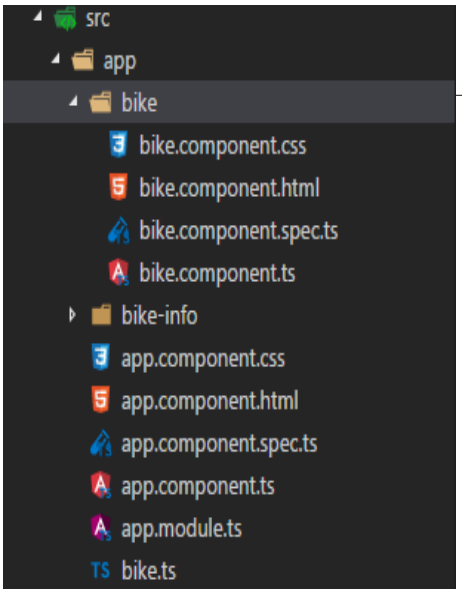
Tag

View

CSS settings

Component class

When Component is created using Angular CLI



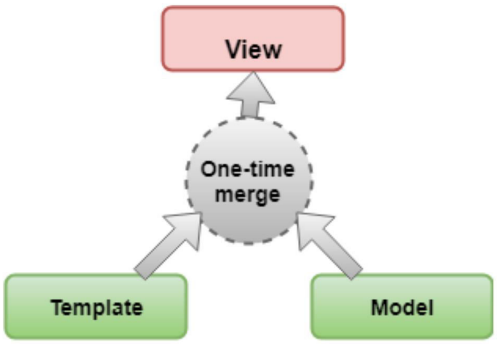
- When we create bike and bike-info components, this is how the structure looks like in src folder.

ng generate component bike
ng g c bike-info

- Angular CLI created respective html, ts, spec.ts and css file for each component.
- **html** is used to write template.
- **ts** file is actual typescript class in which we write component logic
- **css** is where we write styles for component.
- **spec.ts** is test case file

Lets look briefly about Data binding, Directives and Services

Data binding :



- Data Binding is the mechanism that binds the applications User Interface to the models.
- Using Data Binding, the user will be able to manipulate the elements present on the website using the browser.
- Therefore, whenever some variable has been changed, that particular change must be reflected in the Document Object Model or the DOM.
- It is a simple communication where HTML template is changed when we make changes in typescript code.

String Interpolation data binding

{{data | expression | method call}}

- String Interpolation is a one-way data-binding technique which is used to output the data from a TypeScript code to HTML template (view).
- It uses the template impression in double curly braces to show the data from the component to the view.

Template

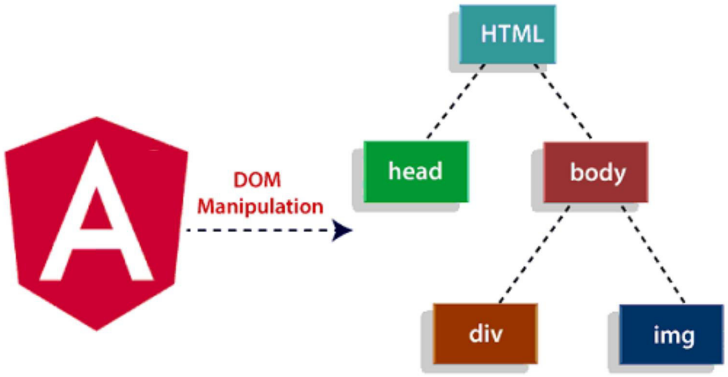
```
<h1>{{title}}</h1>

 Learn <b> {{course}}</b> with SavitaSoft.

2 * 2 = {{2 * 2}}

<div><img src="{{image}}"></div>
```

Component

```
export class AppComponent {
  title = 'Databinding';
  course ='Angular';
  image = './assets/image1.jpg'
}
```

Directives

- The Angular directives are used to manipulate the DOM.
- By using Angular directives, you can change the appearance, behavior or a layout of a DOM element.
- It also helps you to extend HTML.



ngFor directive

- The *ngFor directive is used to repeat a portion of HTML template once per each item from an iterable list (Collection).
- The ngFor is an Angular structural directive
- <li *ngFor="let item of items;"> ....
- Example

Template

```
<ul>
   <li *ngFor="let course of courses; ">
        {{course}}
   </li>
</ul>
```

Component

```
export class AppComponent{
   courses : string[] = ["C", "Java", "Python"];
}
```

Services

- Angular services are singleton objects that get instantiated only once during the lifetime of an application.

- They contain methods that maintain data throughout the life of an application, i.e. data does not get refreshed and is available all the time.

- The main objective of a service is to organize and share business logic, models, or data and functions with different components of an Angular application.

- A service is typically a class with a narrow, well-defined purpose who object can be injected into the component.

Services & Dependency injection

- Example

Service class

```
export class MyServices {
   courses : string[] = ["C", "Java",
"Python"];

   getCourses() {
         return courses;
   }
}
```

Component

```
export class AppComponent {

   courses : string[];

   constructor(private ser : MyService){
      this.courses = ser.getCourses();
   }

}
```

- Register Service into module as injectable

```
@NgModule({
      providers: [ MyServices ],
 ...
})
```

Routing

- Routing helps in directing users to different pages based on the option they choose on the main page.
- Based on the option they choose, the required Angular Component will be rendered to the user.

Routing

- Step 1 : Create multiple components

```
ng generate component home
ng generate component about
ng generate component contact
```

- Step 2 : Create a view by mapping a component to a URL path using the routes array in src/app/app-routing.module.ts file

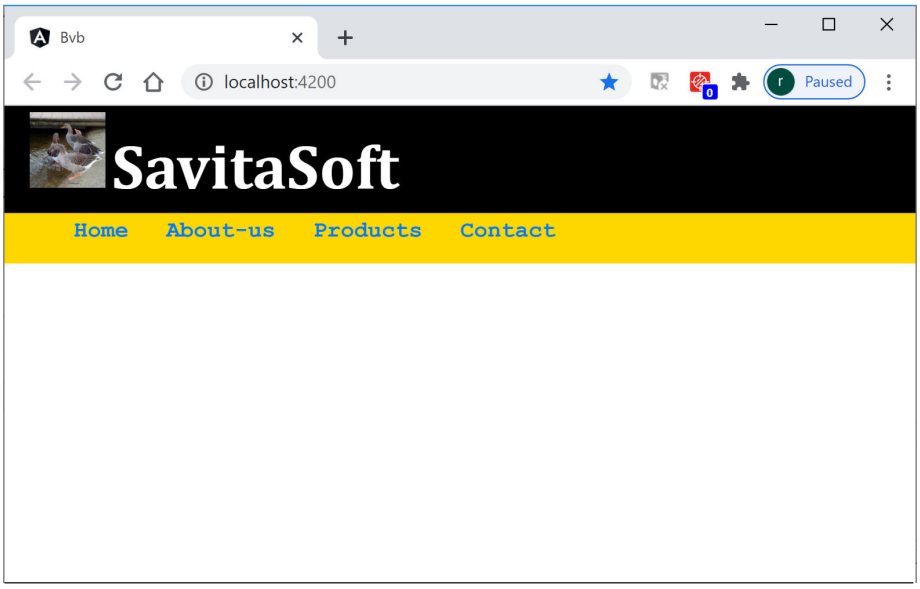
```
const routes: Routes = [
  { path: 'home', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  { path: 'contact', component: ContactComponent },
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Routing

- Step 3 : Add navigation and render the components

```
<ul>
    <li  id="u1"><a [routerLink]="['/home']">Home</a></li>
    <li  id="u1"><a [routerLink]="['/aboutus']">About-us</a></li>
    <li  id="u1"><a [routerLink]="['/contact']">Contact</a></li>
</ul>
<div class="c1">
    <router-outlet></router-outlet>
</div>
```

Assignment



Chapter 2

Data Binding

Data binding

- Data binding is one of the most important concepts of Angular.
- Data binding is the reflection of logic or variable in a model to the view of an app.
- Whenever the variable changes the view must update the DOM to reflect the new changes.

What is DOM?

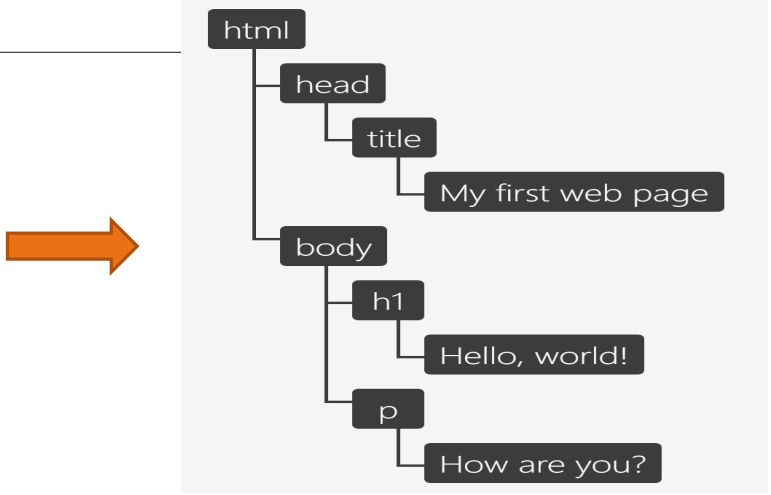
- The DOM is an object-based representation of the source HTML document.
- it is essentially an attempt to convert the structure and content of the HTML document into an object model that can be used by various programs like JavaScript.
- The object structure of the DOM is represented by what is called a “node tree”.
- It is so called because it can be thought of as a tree with a single parent stem that branches out into several child branches, each which may have leaves.

DOM example

HTML

```
<!doctype html>
<html lang="en">
 <head>
   <title>My first web page</title>
  </head>
 <body>
    <h1>Hello, world!</h1>
    <p>How are you?</p>
  </body>
</html>
```

DOM



HTML Attribute and DOM Property

- Attributes are defined by HTML whereas Properties are defined by the DOM (Document Object Model).
- A few HTML attributes have 1:1 mapping to properties. id is one example.
- Some HTML attributes don't have corresponding properties. colspan is one example.
- Some DOM properties don't have corresponding attributes. textContent is one example.
- Many HTML attributes appear to map to properties.

HTML Attribute and DOM Property

- Attributes initialize DOM properties and then they are done.
- Property values can change; attribute values can't.
- For example
  - <input type="text" value="Bob"> creates a corresponding DOM node with a value property initialized to "Bob".
  - When the user enters "Sally" into the input box, the DOM element value property becomes "Sally".
  - But the HTML value attribute remains unchanged as "Bob".
- The HTML attribute value specifies the initial value; the DOM value property is the current value.

HTML Attribute and DOM Property

Template :

```
<div style="background-color : black; color : white; margin-top : 50px;">
    Enter name : <input type="text" id="txt" value="BOB"  #itxt>
    <br>
    <br>
    <button (click)="display(itxt)">Display</button>
</div>
```

Component :

```
export class HomeComponent  {

    public display(inp) : void {
      console.log(inp.id);
      console.log(inp.getAttribute("id"));
      console.log(inp.value);
      console.log(inp.getAttribute("value"))
    }

}
```

Types of Data binding

- Data binding can be either **one-way data binding** or **two-way data binding**.
- **One-way data binding** will bind the data from the component to the view (DOM) or from view to the component.
- **One-way data binding** is unidirectional. You can only bind the data from component to the view or from view to the component.
- One-way data bindings - String Interpolation, Property Binding, Attribute Binding, Class Binding and Style Binding
- **Two-way data binding** exchanges data from the component to view and from view to the component.
- **Two-way data binding** will help users to establish communication bi-directionally.

String Interpolation

- String Interpolation is a **one-way data binding technique** which is used to output the data from a TypeScript code to HTML template (view).

- It uses the template expression in **double curly braces** to display the data from the component to the view.

  {{ data }}

- Example

```
<li>Name: {{ user.name }}</li>

<h1>{{title}}</h1>

 Learn <b> {{course}}</b> with Savitasoft.

 2 * 2 = {{2 * 2}}

 <div><img src="{{image}}"></div>
```

Property binding

- Property Binding is a **one-way data binding** technique.

- In property binding, we bind a property of a DOM element to a field in our component TypeScript code.

- Example

```
<img [src]="imgUrl"/>

<input type="email" [value]="user.email">

<h1 [textContent] = "title"></h1>
```

Attribute binding

- Attribute binding is used to bind an attribute property of a view element to a field of a component.
- Attribute binding is mainly useful where we don't have any property view with respect to an HTML element attribute example

  <td [colspan]="field"></td> raises *Template parse errors: Can't bind to 'colspan' since it isn't a known native property*

- Syntax [attr.attribute] = "field"
- Example

```
<table>
  <tr>
    <td [attr.colspan]="'3'">three</td>
  </tr>
  <tr>
    <td>1</td><td>2</td><td>3</td>
  </tr>
</table>
```

Style binding

- We can set the inline styles of a HTML element using the style binding.
- You can add styles conditionally to an element, hence creating a dynamically styled element.
- Syntax [style.style-property] = "style-value"
- Example

```
<p [style.color]="'red'">Give me red</p>

<button [style.border]="'5px solid yellow'">Save</button>

<button [style.color]="status=='error' ? 'red': 'blue'">Button 1</button>

<p [style.color]="getColor()"
   [style.font-size.px]="'20'"
   [style.background-color]="status=='error' ? 'red': 'blue'">
   paragraph with multiple styles
</p>
```

Style binding

- `<p [style.background-color]="'darkorchid'"> Quite something! </p>`
- You can also specify the unit, here for example we set the unit in em, but px, % or rem could also be used:

```
<p [style.font-size.px]="'14'">
  A paragraph at 14px!
</p>
```

- Conditionally set a style value depending on a property of the component:

```
<p [style.font-size.px]="isImportant ? '30' : '16'">
      Some text that may be important.
</p>
```

Multiple Style binding using NgStyle

- Simple style binding is great for single values, but for applying multiple styles the easiest way is to use NgStyle
- Syntax : [ngStyle]="value | property | function call"
- Example

Template :

```
<p [ngStyle]="myStyles">
        SavitaSoft
</p>
```

Component :

```
myStyles = {   'background-color': 'lime',
               'font-size': '20px',
               'font-weight': 'bold'
           }
```

Using Bootstrap in Angular project

- Download and install bootstrap into Angular project using Angular/Cli

  - npm install bootstrap --save
  - npm install jquery --save
  - npm install popper.js –save

  - The --save option instructed NPM to include the package inside of the dependencies section of your package.json automatically, thus saving you an additional stop

- Import bootstrap into angular.json file

```
"styles": [
    "node_modules/bootstrap/dist/css/bootstrap.min.css",
    "src/styles.css"
  ],
  "scripts": [
     "node_modules/jquery/dist/jquery.min.js",
     "node_modules/bootstrap/dist/js/bootstrap.min.js"
  ]
```

Using Font Awesome icons in Angular project

- Download and install Font awesome into Angular project using Angular/Cli

  - npm install font-awesome --save

- Import font awesome into angular.json file

```
"styles": [
  "node_modules/bootstrap/dist/css/bootstrap.min.css",
  "node_modules/font-awesome/css/font-awesome.css",
  "src/styles.css"
],
```

- Example of font awesome icon

```
<i class="fa fa-star" aria-hidden="true"></i>
```

Class binding

- Class binding is used to add or remove classes to and from the HTML elements.
- You can add CSS Classes conditionally to an element, hence creating a dynamically styled element.
- Syntax <div [**class**.<className>]="field/condition/function call">
- Example

```
<button class="btn btn-primary"  [class.active]="isActive">Save</button>
<div [class.red]="hasError" [class.size20]="hasError">Test</div>

export class AppComponent {
    isAcitve : boolean = true;
    hasError : boolean = true;
}
```

Multiple Class binding using ngClass

- You can bind multiple classes using ngClass directives
- Syntax : [ngClass]="'someClass'">
- Conditional [ngClass]="{'someClass': property1.isValid}">
- Multiple Condition [ngClass]="{'someClass': property1.isValid && property2.isValid}">
- Method expression [ngClass]="getSomeClass()"

  Method will inside of your component

```
getSomeClass(){
    const isValid=this.property1 && this.property2;
    return {someClass1:isValid , someClass2:isValid};
}
```

Multiple Class binding using ngClass

- Example

```
<ul *ngFor="let person of people">
    <li [ngClass]="{
        'text-success':person.country === 'UK',
        'text-primary':person.country === 'USA',
        'text-danger':person.country === 'HK'
    }">{{ person.name }} ({{ person.country }})
    </li>
</ul>
```

Event binding

- Event binding is used to handle the events raised from the DOM like button click, mouse move, Keypress etc.

- When the DOM event is triggered (eg. click, change, keyup), it calls the specified method in the component.

- For example:

```
<button (click)=“display()">Display</button>
```

DOM events

- **MouseEvent**
  - **click** — The event occurs when the user clicks on an element
  - **dblclick** — The event occurs when the user double-clicks on an element
  - **mousedown** — The event occurs when the user presses a mouse button over an element
  - **mouseenter** — The event occurs when the pointer is moved onto an element
  - **mouseleave** — The event occurs when the pointer is moved out of an element
  - **mousemove** — The event occurs when the pointer is moving while it is over an element
  - **mouseover** — The event occurs when the pointer is moved onto an element, or onto one of its children
  - **mouseout** — The event occurs when a user moves the mouse pointer out of an element, or out of one of its children
  - **mouseup** — The event occurs when a user releases a mouse button over an element

DOM events

- **FocusEvent**
  - **focus** — The event occurs when an element gets focus
  - **blur** — The event occurs when an element loses focus
  - **focusin** — The event occurs when an element is about to get focus
  - **focusout** — The event occurs when an element is about to lose focus
  - **change** — The event occurs when the content of a form element, the selection, or the checked state have changed (for <input>, <select>, and <textarea>)
- **KeyboardEvent**
  - **keydown** — The event occurs when the user is pressing a key
  - **keypress** — The event occurs when the user presses a key
  - **keyup** — The event occurs when the user releases a key

DOM events

- **InputEvent**
  - **input** The event occurs when an element gets user input

---

- **Other Events**
  - **offline** The event occurs when the browser starts to work offline
  - **online** The event occurs when the browser starts to work online
  - **resize** The event occurs when the document view is resized
  - **reset** The event occurs when a form is reset
  - **submit** The event occurs when a form is submitted

Event bubbling

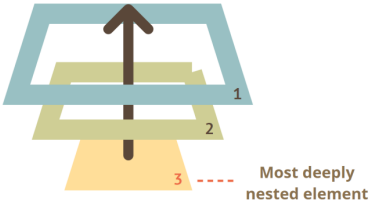
- When an event happens on an element, it first runs the handlers on it, then on its parent, then all the way up on other ancestors.

- This move is known as **Event bubbling or Event Propagation or Event Delegation**.
- Example

```
<form onclick="console.log('form')">FORM
  <div onclick="console.log('div')">DIV
    <p onclick="console.log('p')">P</p>
  </div>
</form>
```

A click on the inner <p> first runs onclick:

1. On that <p>.
2. Then on the outer .
3. Then on the outer <form>.
4. And so on upwards till the document object.



- Stop bubbling by $event.stopPropagation() method

Event object

- When an event occurs in HTML, the event belongs to a certain event object, like a mouse click event belongs to the MouseEvent object.

- All event objects are based on the Event Object, and inherit all of their properties and methods

- Example event objects

Event Object	Description
Event	The parent of all event objects
FocusEvent	For focus-related events
InputEvent	For user input
KeyboardEvent	For keyboard interaction
MouseEvent	For mouse interaction

Handling events

- **Template**

```
<div>
  Login ID : <input (keypress)="onKeyPress()" type="text" id="loginid"><br><br>
  <button (click)="onClick()">Login</button>
</div>
```

- **Component**

```
export class AboutusComponent  {
        public onKeyPress(){
          console.log("On keypress");
        }

        public onClick(){
          console.log("On click");
        }
}
```

Handling events

- **Template**

```
<div>
  Login ID : <input (keypress)="onKeyPress($event)" type="text" id="loginid"><br><br>
  <button (click)="onClick($event)">Login</button>
</div>
```

- **Component**

```
export class AboutusComponent  {
    public onKeyPress($event){
        console.log("On keypress",$event);
        console.log("Key : ",$event.code,$event.charCode);
    }

    public onClick($event){
      console.log("On click“,$event);
    }
}
```

Event filtering

- If you want to detect a particular key press and perform some action when that key is pressed

---

```
<input (keyup) = handleEvent($event) />

private handleEvent(event): void {
      // compare key code with code of enter key
      if(event.keyCode === 13) {
         console.log(‘Enter key pressed’)
      }
}
```

Event filtering

- Angular provides a more **advanced** method of Event Filtering

```
<input (keyup.enter) = handleEvent() />

private handleEvent(): void {
      console.log('Enter key pressed')
}
```

Template variable

- A template reference variable is often a reference to a DOM element within a template.
- Use the hash symbol (#) to declare a reference variable.
- The following reference variable, #phone, declares a phone variable on an <input> element.

```
<input #phone placeholder="phone number" />
```

- You can refer to a template reference variable anywhere in the component's template.
- Here, a <button> further refers to the phone variable.

```
<!-- phone refers to the input element; pass its `value` to an event handler -->
<button (click)="callPhone(phone.value)">Call</button>
```

Two-way binding

- Two-way data binding in Angular allows the exchange data from the component to view and from view to the component.

- It will help users to establish bi-bidirectionally communication.
- Two-way data binding can be achieved using a ngModel directive in Angular.
- The ngModel which is basically the combination of both the square brackets of property binding and parentheses of the event binding.

  [(Data Binding Target)] = “Property”

  For the input box, the data binding target is ngModel, which corresponds to the text in the input box.

Two-way binding

- Example

```
<input [(ngModel)]="user.name" name="name"/>
<input [(ngModel)]="user.age" name="age" type="number"/>
```

```
export class AppComponent {
   user: any = { name: '', age: ‘’
};
```

- In app.module.ts file you need to import ngModel directive from FormsModule that is present in @angular/forms file

Pipes

- Pipes are a useful feature in Angular.
- They are a simple way to transform values in an Angular template.
- Pipes are very useful when it comes to managing data within interpolation **"{{ | }}"**.
- In order to transform data, we use the | character.
- Example :

  {{ i will become uppercase one day | uppercase }}

- Pipes accept date, arrays, strings, and integers as inputs.
- The inputs will be converted in the desired format before displaying them in the browser.

Built-in Pipes

1. UpperCase - Transforms text to all upper case.

{{ value_expression | uppercase }}

2. LowerCase - Transforms text to all lower case.

{{ value_expression | lowercase }}

3. TitleCase - Capitalizes the first letter of each word and transforms the rest of the word to lower case. Words are delimited by any whitespace character, such as a space, tab, or line-feed character.

{{ value_expression | titlecase }}

Built-in Pipes

4. Decimal - Transforms numbers into text strings, according to various format specifications,

---

{{ value_expression | number [: digitsInfo [: locale]] }}

Examples :

```
e: number = 2.718281828459045;

<!--output '2.718'-->
<p>e (no formatting): {{e | number}}</p>

<!--output '002.71828'-->
<p>e (3.1-5): {{e | number:'3.1-5'}}</p>

<!--output '0,002.71828'-->
<p>e (4.5-5): {{e | number:'4.5-5'}}</p>
```

Built-in Pipes

5. DatePipe - Formats a date value according to locale rules.

{{ value_expression | date [: format] }}

- 'short': equivalent to 'M/d/yy, h:mm a' (6/15/15, 9:03 AM).
- 'medium': equivalent to 'MMM d, y, h:mm:ss a' (Jun 15, 2015, 9:03:01 AM).
- 'long': equivalent to 'MMMM d, y, h:mm:ss a z' (June 15, 2015 at 9:03:01 AM GMT+1).
- 'shortDate': equivalent to 'M/d/yy' (6/15/15).
- 'mediumDate': equivalent to 'MMM d, y' (Jun 15, 2015).
- 'longDate': equivalent to 'MMMM d, y' (June 15, 2015).
- 'fullDate': equivalent to 'EEEE, MMMM d, y' (Monday, June 15, 2015).
- 'shortTime': equivalent to 'h:mm a' (9:03 AM).
- 'mediumTime': equivalent to 'h:mm:ss a' (9:03:01 AM).
- 'longTime': equivalent to 'h:mm:ss a z' (9:03:01 AM GMT+1).
- 'fullTime': equivalent to 'h:mm:ss a zzzz' (9:03:01 AM GMT+01:00).

Exmple :

```
{{ dateObj | date:'medium' }}          // output is 'Jun 15, 2015, 9:43:11 PM'
{{ dateObj | date:'shortTime' }}        // output is '9:43 PM'
```

Built-in Pipes

6. JsonPipe - Converts a value into its JSON-format representation.
Useful for debugging.

{{ value_expression | json }}

Example

```
object: Object = { foo: 'bar',
                   baz: 'qux',
                   nested: { xyz: 3,
                             numbers: [1, 2, 3, 4, 5]
                           }
                   };


<p>{{object | json}}</p>
```

Custom pipes

- To create a Pipe definition, we need to first create a class that implements PipeTransform interface.

- We need to name the pipe “filesize” with Pipe decorator

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

@Pipe({ name: 'filesize' })
export class FileSizePipe implements PipeTransform {
        transform(size: number, extension: string = 'MB') {
          return (size / (1024 * 1024)).toFixed(2) + extension;
        }
}
```

- Register pipe class in your @NgModule under declaration

```
declarations: [
   //...
   FileSizePipe,
 ],
```

Assignments

1. Preposition title case -> using two way binding & custom pipes

1. Like and dislike component

1. 5 star component

Chapter 3

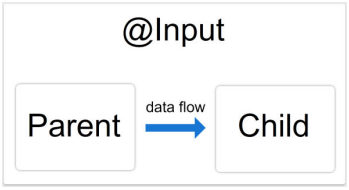
Building Re-usable Components

Building re-usable components :

- In this chapter you will be learning
  - How to interact one component to another component.
  - How to pass data from parent component to child component.
  - How to pass data from child component to parent component
  - Raise custom events
  - Apply styles
  - Shadow DOM
  - View encapsulation

@Input

- @Input() decorator in a child component allows child component to receive data from its parent component.



- In the child component

  @Input() item: string; // decorate the property with @Input() by importing from '@angular/core' file

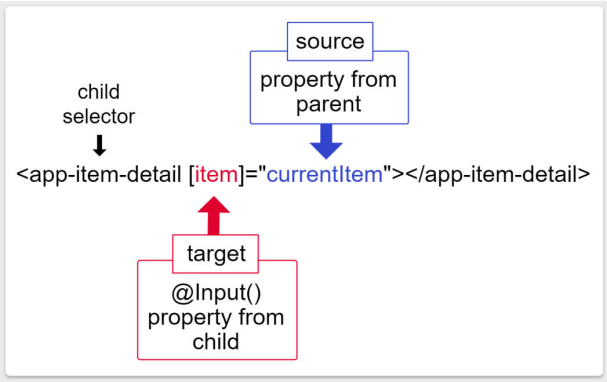
- Component file

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

export class ItemDetailComponent {
  @Input() item: string; // decorate the property with @Input()
}
```

@Input

- In the parent component

```
<app-item-detail [item]="currentItem"></app-item-detail>

export class AppComponent {
        currentItem = 'Television';
}
```



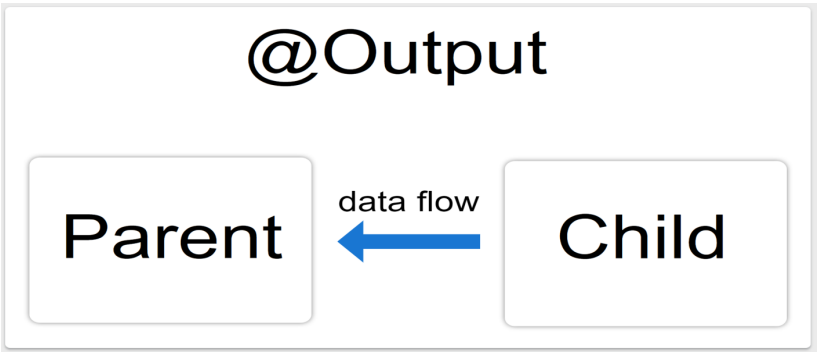
Alias name for Input property

@Input('itemdetails') item: string;

- Input property can have alias name.
- In above example itemdetails is an alias name assigned to item property.
- Parent component will be known to item property by itemdetails name.

Output property

- Use the @Output() decorator in the child component or directive to allow data to flow from the child out to the parent.
- An @Output() property should normally be initialized to an Angular EventEmitter with values flowing out of the component as events.



Output property

- In child component :

```
export class ItemOutputComponent {

  @Output() newItemEvent = new EventEmitter<string>();

  addNewItem(value: string) {
    this.newItemEvent.emit(value);
  }
}
```

- In child template :

```
<label>Add an item: <input #newItem></label>
<button (click)="addNewItem(newItem.value)">Add to parent's list</button>
```

Output property

- In the parent component :

```
export class AppComponent {
  items = ['item1', 'item2', 'item3', 'item4'];

  addItem(newItem: string) {
    this.items.push(newItem);
  }
}
```

- In the parent’s template :

```
<app-item-output (newItemEvent)="addItem($event)"></app-item-output>
```

<ng-content>

- <ng-content> is used to transfer data from parent component to child component
- In the child component

```
<div class="card">
    <div class="card-header"><ng-content select=".header"></ng-content></div>
    <div class="card-body"><ng-content select=".body"></ng-content></div>
    <div class="card-footer"><ng-content select=".footer"></ng-content></div>
</div>
```

- In the parents component

```
<app-bootstrap-panel>
    <div class="header">SavitaSoft</div>
    <div class="body">What ever I do I do it well</div>
    <div class="footer">ssce, vidyanagar, hubli</div>
</app-bootstrap-panel>
```

- There is no need of select if there is only one <ng-content>

Chapter 4

Directives

What are Directives?

- A directive modifies the DOM by changing the appearance, behavior, or layout of DOM elements.
- There are three main types of directives:
  - **Component** – Directives with templates.
  - **Attribute Directives** – Directives that change the behavior of a component or element but don't affect the template.
  - **Structural Directives** – Directives that change the behavior of a component or element by affecting the template or the DOM decoration of the UI.

Attribute Directives

- Attribute directives are mainly used for changing the appearance or behavior of a component or a native DOM element.
- There are some inbuild attribute directive available like ngClass, ngStyle and ngModel.
  - ngClass - ngClass directive changes the class attribute.
  - ngStyle - ngStyle is mainly used to modify or change the element's style attribute.
- We can also create any type of custom attribute-based directive as per our requirement.

Structural Directives

- Structural directives start with a * sign.
- These directives are used to manipulate and change the structure of the DOM elements.
- For example, *ngIf directive, *ngSwitch directive, and *ngFor directive.
  - ***ngIf Directive:** The ngIf allows us to Add/Remove DOM Element.
  - ***ngSwitch Directive:** The *ngSwitch allows us to Add/Remove DOM Element. It is similar to switch statement of C.
  - ***ngFor Directive:** The *ngFor directive is used to repeat a portion of HTML template once per each item from an iterable list (Collection).

*ngIf directive

- The ngIf Directives is used to add or remove HTML Elements according to the expression.
- The expression must return a Boolean value. If the expression is false then the element is removed, otherwise element is inserted.
- Syntax

```
<p *ngIf="condition">
   condition is true and ngIf is true.
</p>
<p *ngIf="!condition">
   condition is false and ngIf is false.
</p>
```

- The ngIf directive does not hide the DOM element. It removes the entire element along with its subtree from the DOM.
- It also removes the corresponding state freeing up the resources attached to the element.

*ngIf else block

- The ngIf Directives can also have else block
- Syntax

```
<div *ngIf="condition; else elseBlock">
    Content to render when condition is true.
</div>
<ng-template #elseBlock>
    Content to render when condition is false.
</ng-template>
```

*ngIf with if and else block

- The ngIf Directives can have both if and else block
- Syntax

```
<div *ngIf="condition; then ifblock else elseblock">
</div>

<ng-template #ifblock>
    Content to render when condition is true
</ng-template>

<ng-template #elseblock>
    Content to render when condition is false
</ng-template>
```

ngSwitch directive

- ngSwitch is a structural directive which is used to Add/Remove DOM Element which is similar to switch statement of C language.
- This is used when you have multiple conditions to be checked and render the contents.

Syntax

```
<container_element [ngSwitch]="switch_expression">
   <inner_element *ngSwitchCase="match_expresson_1">...</inner_element>
   <inner_element *ngSwitchCase="match_expresson_2">...</inner_element>
   <inner_element *ngSwitchCase="match_expresson_3">...</inner_element>
   <inner_element *ngSwitchDefault>...</element>
</container_element>
```

- Each ngSwitch expression value is checked with individual ngSwitchCase expression.
- Whenever the value of the match expression matches the value of the switch expression, the corresponding inner element is added to the DOM. All other inner elements are removed from the DOM

ngSwitch directive

- Example

```
<ul class="nav nav-pills">
   <li class="nav-item">
     <a class="nav-link" [class.active]="view=='student'"  (click)="onChangeView('student')">Student</a>
   </li>
   <li class="nav-item">
     <a class="nav-link" [class.active]="view=='employee'" (click)="onChangeView('employee')">Employee</a>
   </li>
   <li class="nav-item">
     <a class="nav-link" [class.active]="view=='others'" (click)="onChangeView('others')">Others</a>
   </li>
</ul>
<div [ngSwitch]="view">
   <div *ngSwitchCase="'student'"><h1>Student list</h1></div>
   <div *ngSwitchCase="'employee'"><h1>Employee list</h1></div>
   <div *ngSwitchDefault><h1>Others list</h1></div>
</div>
```

*ngFor directive

- The *ngFor directive is used to repeat a portion of HTML template once per each item from an iterable list (Collection).
- The ngFor is an Angular structural directive.
- Some local variables like Index, First, Last, odd and even are exported by *ngFor directive.
- Syntax

```
<li *ngFor="let item of items;"> {{item}} </li>
```

- Example

```
<tbody>
   <tr *ngFor="let movie of movies;">
      <td>{{movie.title}}</td>
      <td>{{movie.director}}</td>
      <td>{{movie.cast}}</td>
      <td>{{movie.releaseDate}}</td>
   </tr>
</tbody>
```

*ngFor directive

- The *ngFor directive is used to repeat a portion of HTML template once per each item from an iterable list (Collection).
- The ngFor is an Angular structural directive.
- Some local variables like Index, First, Last, odd and even are exported by *ngFor directive.
- Syntax

```
<li *ngFor="let item of items;"> {{item}} </li>
```

- Example

```
<tbody>
   <tr *ngFor="let movie of movies;">
      <td>{{movie.title}}</td>
      <td>{{movie.director}}</td>
      <td>{{movie.cast}}</td>
      <td>{{movie.releaseDate}}</td>
   </tr>
</tbody>
```

*ngFor exported variables

- index: number: The index of the current item in the iterable.
- count: number: The length of the iterable.
- first: boolean: True when the item is the first item in the iterable.
- last: boolean: True when the item is the last item in the iterable.
- even: boolean: True when the item has an even index in the iterable.
- odd: boolean: True when the item has an odd index in the iterable.

```
<ul>
   <li style="list-style-type:none"
      *ngFor="let emp of employees; index as i; even as e; odd as o; first as f; last as l">
               {{i}} {{emp}}   <span *ngIf="e">Even</span>
                               <span *ngIf="o">Odd</span>
                               <span *ngIf="f">First</span>
                               <span *ngIf="l">Last</span>
   </li>
</ul>
```

ngFor trackBy

- Angular provides the trackBy feature which allows you to track elements when they are added or removed from the array for performance reasons.
- If angular is creating DOM object for collection or array and if any changes in that collection or array ie you add, delete or modify the elements. In this case instead of updating all DOM related to this collection, it should update only required DOM objects.
- We need to define a trackBy method, which needs to identify each element uniquely, in the associated component and assign it to the ngFor directive as follows:
- Syntax

```
<tr *ngFor="let element of collection; trackBy: trackByMethod">
```

ngFor trackBy example

- **Template** :

```
<table class="table table-light">
  <tbody>
    <tr *ngFor="let course of courses; trackBy: trackByMethod">
      <td>{{course.id}}</td>
      <td>{{course.name}}</td>
      <td><button (click)="onRemove(course)">Remove</button></td>
    </tr>
  </tbody>
</table>
```

- **Component**

```
public trackByMethod(index,course) : number{
    return course.id;
}
```

Custom directive

- Creating a custom directive is just like creating an Angular component.
- To create a custom directive we have to replace @Component decorator with @Directive decorator.

```
@Directive({
  selector: '[caseconverter]'
})
export class CaseconverterDirective {

    constructor(private eleRef: ElementRef) {
    }
}
```

- The selector is wrapped with `[]`.
- Inject ElementRef class object for accessing the DOM element.

Handling events in Custom directive

- **Handling events in directives**

```
import { Directive, Host, HostListener } from '@angular/core';

@Directive({
  selector: '[caseconverter]'
})
export class CaseconverterDirective {

        @HostListener('focus') onFocus() {
          console.log("On focus");
        }

        @HostListener('blur') onBlur() {
          console.log("On blus");
        }

        @HostListener('keydown') onKeyDown() {
          console.log("On keydown");
        }
        constructor() { }
}
```

Custom directive example 1

- **Changing background color**

```
@Directive({
  selector: '[caseconverter]'
})
export class CaseconverterDirective {

  constructor(private el : ElementRef) {

  }

  @HostListener('focus') onFocus() {
        this.el.nativeElement.style.background = 'blue';

  }

  @HostListener('blur') onBlur() {
        this.el.nativeElement.style.background = 'green';
  }
}
```

Custom directive example 2

- **Directive to convert text to upper case**

```
@Directive({
  selector: '[caseconverter]'
})
export class CaseconverterDirective {

  constructor(private el : ElementRef) {
  }

  @HostListener('blur') onBlur() {
        let value : string = this.el.nativeElement.value;
        this.el.nativeElement.value = value.toUpperCase();
  }
}
```

Custom directive with @Input property

- Directives can have input property. In below example **format** is the input parameter of caseconverter directive.
- Template: `<input type="text" caseconverter [format]="'upper'">`
- Directive class

```
export class CaseconverterDirective {

    @Input() format : string;

    constructor(private el : ElementRef) { }

    @HostListener('blur') onBlur() {
        let value : string = this.el.nativeElement.value;
        if(this.format.toLowerCase() == "upper"){
            this.el.nativeElement.value = value.toUpperCase();
        } else {
            this.el.nativeElement.value = value.toLowerCase();
        }
    }
}
```

Directive selector as @Input property

- Directives can have input property. In below example **format** is the input parameter of caseconverter directive.
- Template: <input type="text" **[caseconverter]="'upper'">**
- Directive class

```
export class CaseconverterDirective {

    @Input('caseconverter') format : string;

    constructor(private el : ElementRef) {  }

    @HostListener('blur') onBlur() {
        let value : string = this.el.nativeElement.value;
        if(this.format.toLowerCase() == "upper"){
            this.el.nativeElement.value = value.toUpperCase();
        } else {
            this.el.nativeElement.value = value.toLowerCase();
        }
    }
}
```

Assignment

- Assignment to fold and unfold card