Angular

Angular

- Angular is typescript based framework to design client side Single Page Applications (SPA)
- TypeScript is a superset of ECMAScript 6 (ES6)
- The main building blocks for Angular are modules, components, templates, metadata, data binding, directives, services and dependency injection
- Angular 2 is complete rewrite of Angular JS

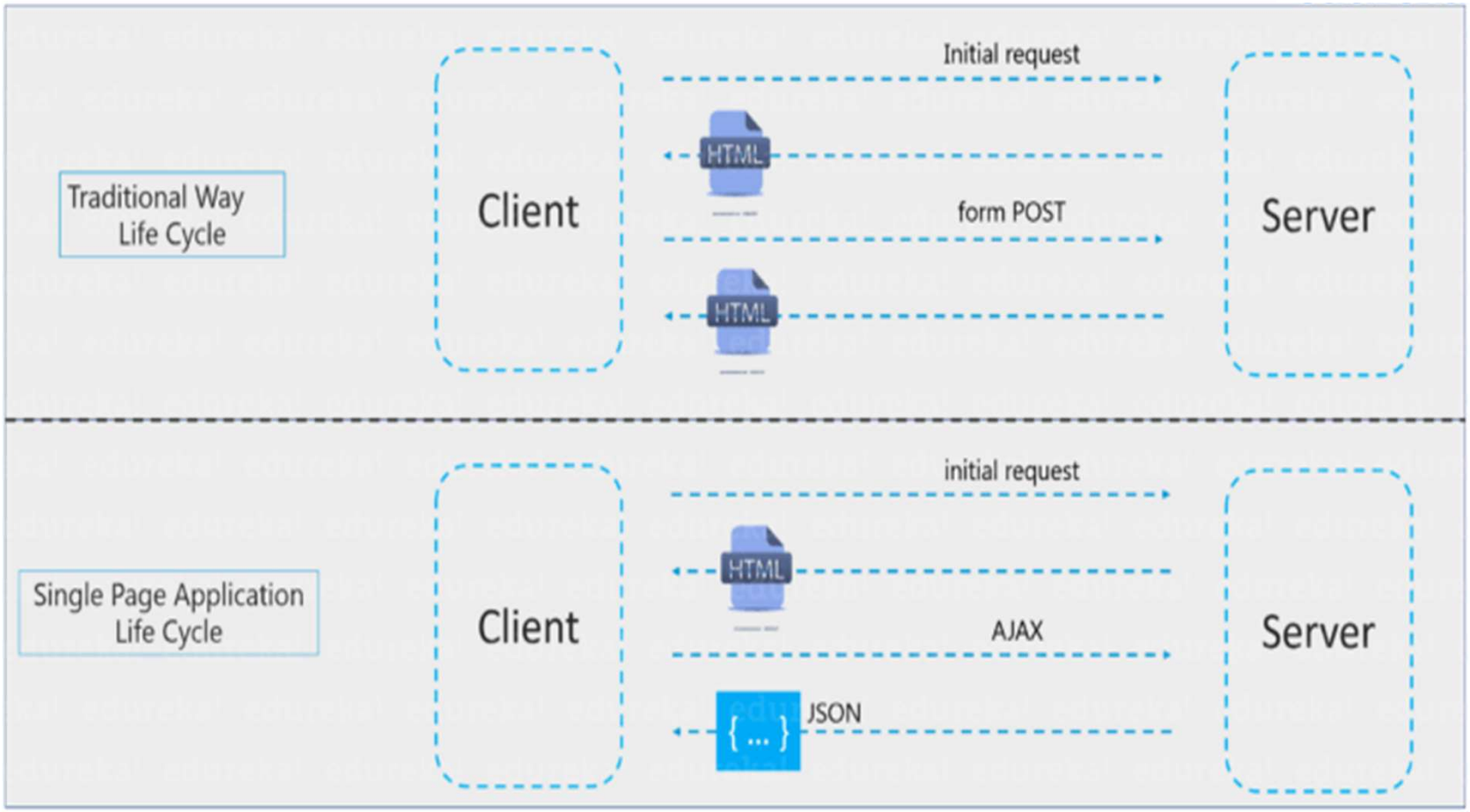
Angular Building Blocks

- The main building blocks of Angular are:
  - Modules
  - Components
  - Templates
  - Data binding
  - Directives
  - Services
  - Dependency injection

Single page Applications (SPA)

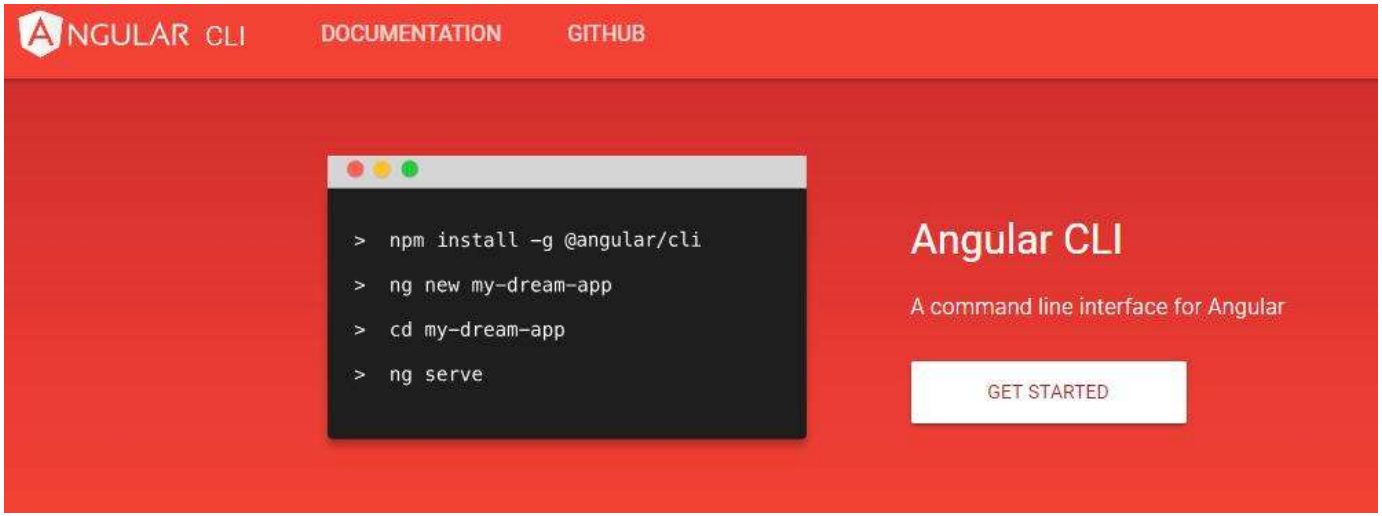
- Applications that are accessed via a web browser like other websites but offer more dynamic interactions
- The difference between a regular website and SPA is the reduced amount of page refreshes
- SPAs have a way to communicate with back-end servers without doing a full page refresh to get data loaded into our application
- As a result, the process of rendering pages happens mostly on the client-side

Single Page Applications (SPA)



Angular Environment setup

- To install Angular, we require the following −
  - Nodejs
  - NPM
  - Angular CLI
  - IDE for writing your code (atom, Microsoft VS Code etc)
- Commands help available at cli.angular.io



Angular Project setup

- To install Angular

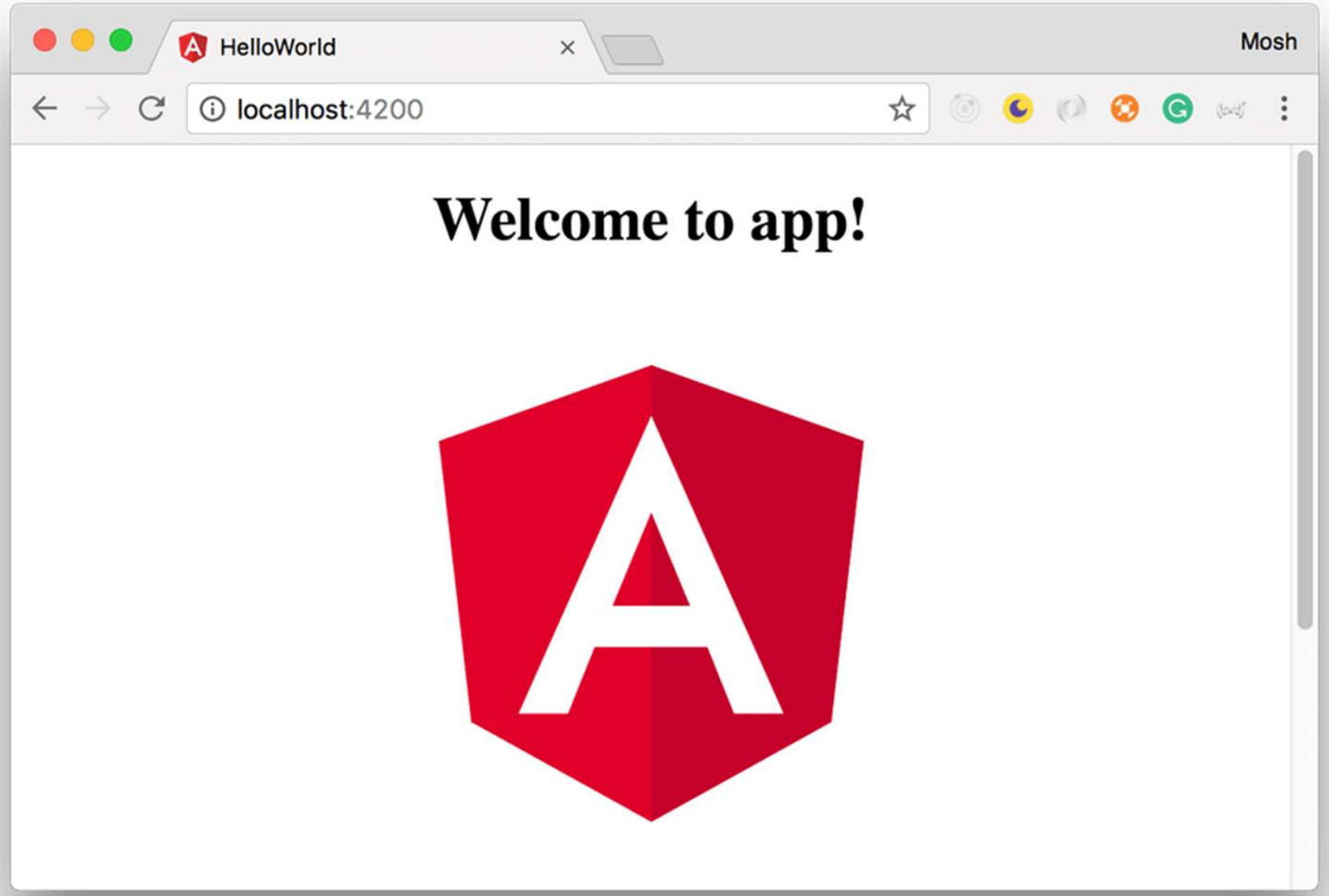
  npm install –g @angular/cli
- To start a workspace and new project

  ng new angular-test-project
- Creates a folder angular-test-project and creates a starter project
- Make the folder as current (Use CD command)

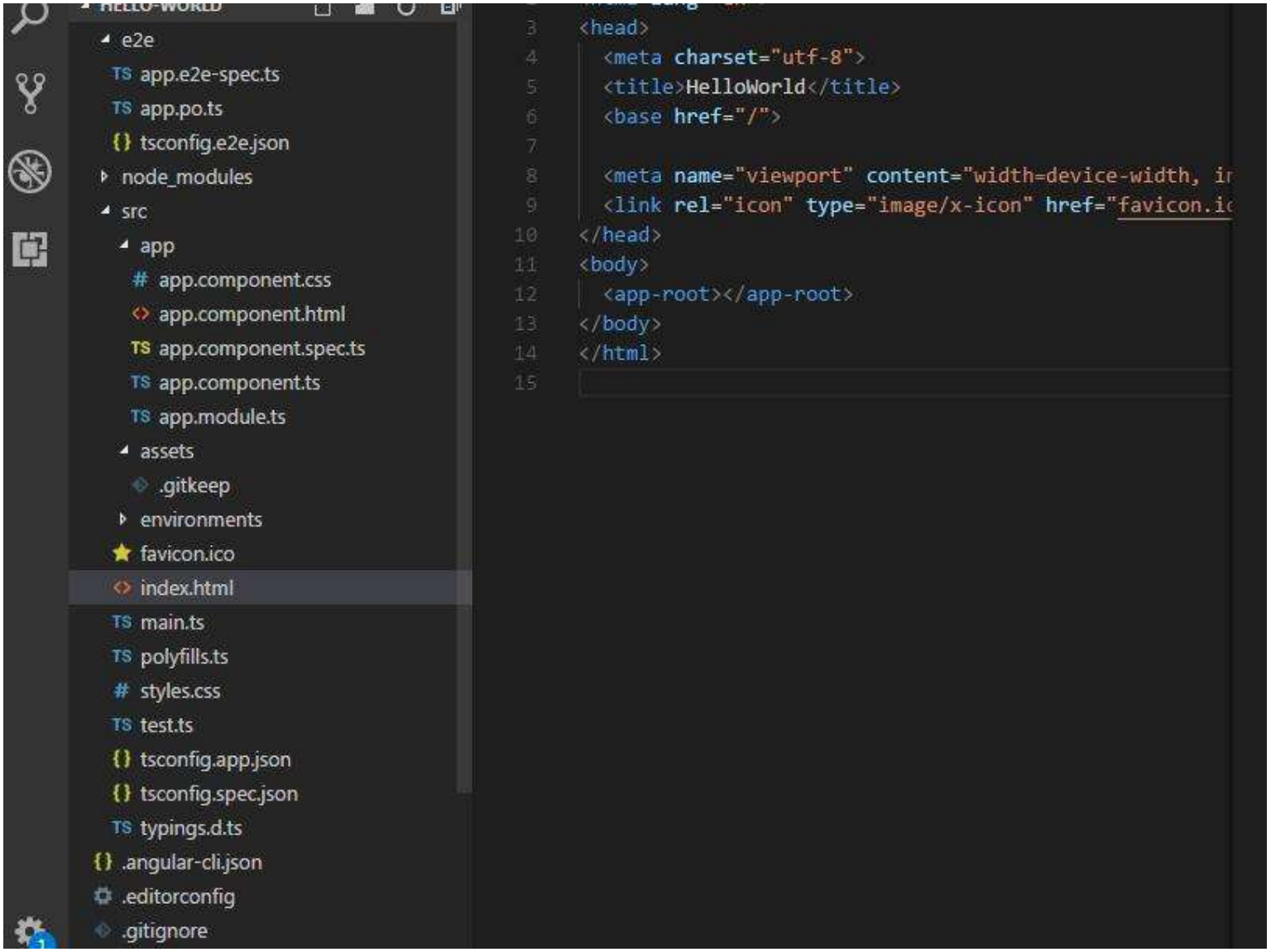
  cd angular-test-project
- Run the sample project

  ng serve
- Starts a server at port 4200

Angular startup web page



Angular Project Structure



Angular Project Structure

- **e2e** − end to end test folder. Mainly e2e is used for integration testing
- **node_modules** − all third party libraries required
- **src** − This folder is where we will work on the project using Angular
- **.angular.json** − It basically holds the project name, version of cli, etc.
- **.editorconfig** − This is the config file for the editor.
- **karma.conf.js** − used for unit testing via the protractor. All the information required for the project is provided in karma.conf.js file.
- **package.json** − The package.json file tells which libraries will be installed into node_modules when you run npm install.
- **protractor.conf.json** − tool for running end-to-end tests for Angular projects
- **tsconfig.json** − includes setting for the TypeScript compiler.
- **tslint.json** − includes the settings for TSLint which is a popular tool for linting TypeScript code. That means it checks the quality of our TypeScript code based on a few configurable parameters

Angular Project Structure

- **assets** -- folder to store assets like image files etc
- **evironment** -- Details of environment - production and development
- **index.html** - main file
- **main.ts** -- starting point - used in bootstrapping app module
- **polyfills.ts** – used to check browser compatibility
- **styles.css** -- Used to store global style sheet
- **test.ts** - - defines testing environment

Multiple Project in same workspace

- Create workspace without projects

  ng new app1 –createApplication=false directory=angularprojects
- *This does not create application (app1 is dummy). Simply create basic workspace with node-modules folder*
- *Move to workspace*

  cd angularprojects

- To create new applications

  ng g application app1
- This creates projects folder and application app1 inside it
- tslint and tsconfig json files extend those in workspace
- Create other applications in the same way
- angular.json is updated for each application. It also contains default project name
- To run default project

  ng serve
- To run a particular project app2

  ng serve -–project=app2

Angular Component

- The most fundamental building block in an Angular application is a *component*
- A component consists of three pieces:

- **HTML markup**: to render that view
- **State**: the data to display on the view
- **Behavior**: the logic behind that view. For example, what should happen when the user clicks a button.

Angular Component

- A component can have other components
- Each component can be maintained separately
- Naming pattern of a component :
- For a component named HelloComponent, the following files are associated :
  - hello.component.ts : component class code
  - hello.component.html : component template
  - hello.component.css : component's CSS styles
  - hello.component.spec.ts : includes unit tests
  - hello.module.ts : module details

- Angular creates AppComponent to start with and used as root component

App Component

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

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

Component decorator

app-root : where the component is placed

Components's html content and style sheet

class code

contains data and methods declaration

main.ts

- main.ts is the file from where we start our project development. It starts with importing the basic module which we need

```
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.log(err));
```

Module

- Angular apps are modular and to maintain modularity, we have *Angular modules* or *NgModules*
- Every Angular app contains the root module which is named as *AppModule (class with NgModule decorator)*

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

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

export class AppModule { }
```

Module

- ***Decorators*** are functions that modify JavaScript classes. used for attaching configuration metadata to classes
- Properties of NgModule decorator:
- ***declarations:*** classes that are related to views and it belong to this module
- There are three classes that can contain view: components, directives and pipes
- ***exports:*** The classes that should be accessible to the components of other modules
- ***imports:*** Modules whose classes are needed by the component of this module.
- ***providers:*** Services present in one of the modules which is to be used in the other modules or components
- ***bootstrap:*** The *root component* which is the main view of the application

Creating new component

- To create a new component :

ng generate component name

or

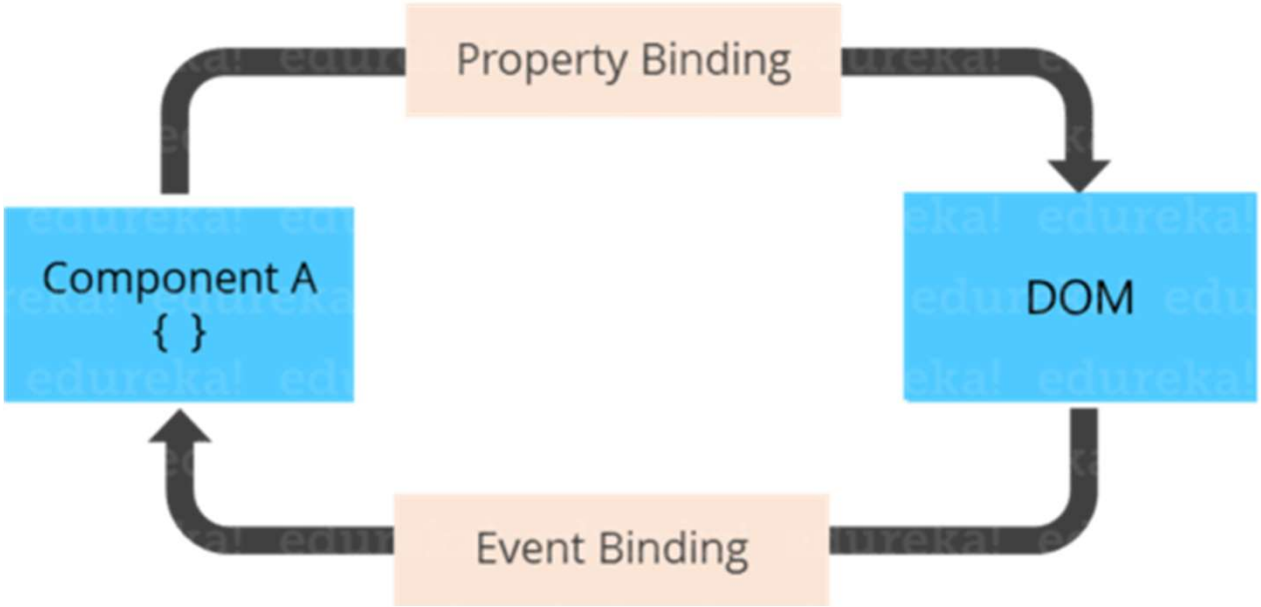
ng g c name

- Creates all files for the component and also updates module to include the component

Binding

Data Binding

- **data binding** is a mechanism for coordinating parts of a template with parts of a component
- You should add binding markup to the template HTML to tell Angular how to connect both sides



Data Binding

- Data binding done with interpolation of data declared in component class

  {{data}}

- This is same as

  <p [innerHTML]='data' />

- we re binding innerHTML property to value of data using "[ ]".

- Binding of other properties of the component is possible in the same way

Event Binding

- For event binding use “( )” and bind to a method which is called on the event

```
<button (click)=”calculate()” >Click me</button>
```

- calculate() function is called on click event

- Event object can be passed in the method

```
<button (click)=”calculate($event)” >Click me</button>
```

- In the method of component $event should be the argument

Template Variable

- Any element can be referred with template variable
- Referred as method argument for events used in the code

```
<input #email (click)= ‘changeValue(email.value)’ />
```

```
<input #name />
<button (click)=‘display(name.value)’>Click</button>
```

- Never pass ‘this’ as argument. It refers the component

Data Binding – attribute or property?

- All html attributes are not dom element properties
- [ ] can be used only for properties

```
<img  [src]="imgUrl" />  // works fine as src is property and attribute

<td  [colspan]  ='size' >… </td>   // does not work as colspan is
                                   only attribute not the property
```

- Binding attributes is done in different way

```
<td  [attr.colspan]  ='size' >… </td>   // works fine
```

Two way Data Binding

- Combines property and event binding in a single notation, using the **ngModel** directive

```
<input  [(ngModel)]="person.name">
```

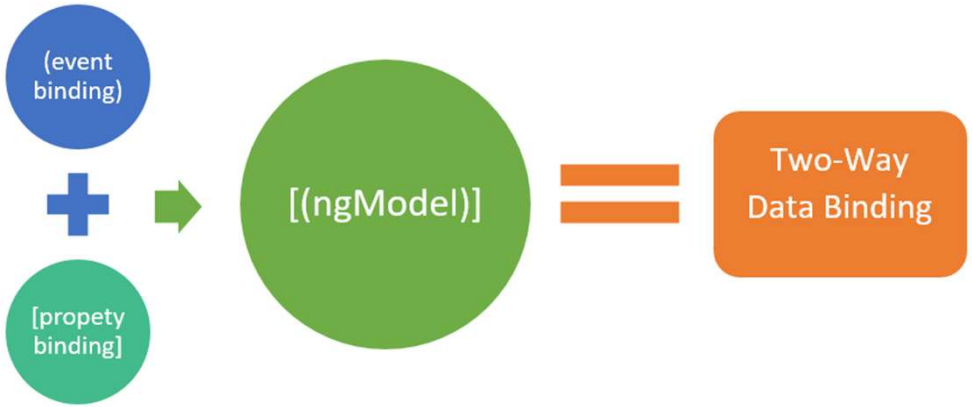
- In two-way binding, a data property value flows to the input box from the component as with property binding

- The user's changes also flow back to the component, resetting the property to the latest value, as with event binding

- Angular processes all data bindings once per JavaScript event cycle, from the root of the application component tree through all child components

Two way Data Binding

- [(ngModel)] which is also referred as ‘Banana in a Box’ is a combination of property binding and event binding




```
<input  [(ngModel)]=“name">
```

is same as:

```
<input  [value]="name"  (input)="name=$event.target.value" />
```

Two way Data Binding

- We need FormsModule for using ‘ngModel’
- Import FormsModule and also add it to imports in the module file
- app.module.ts file:

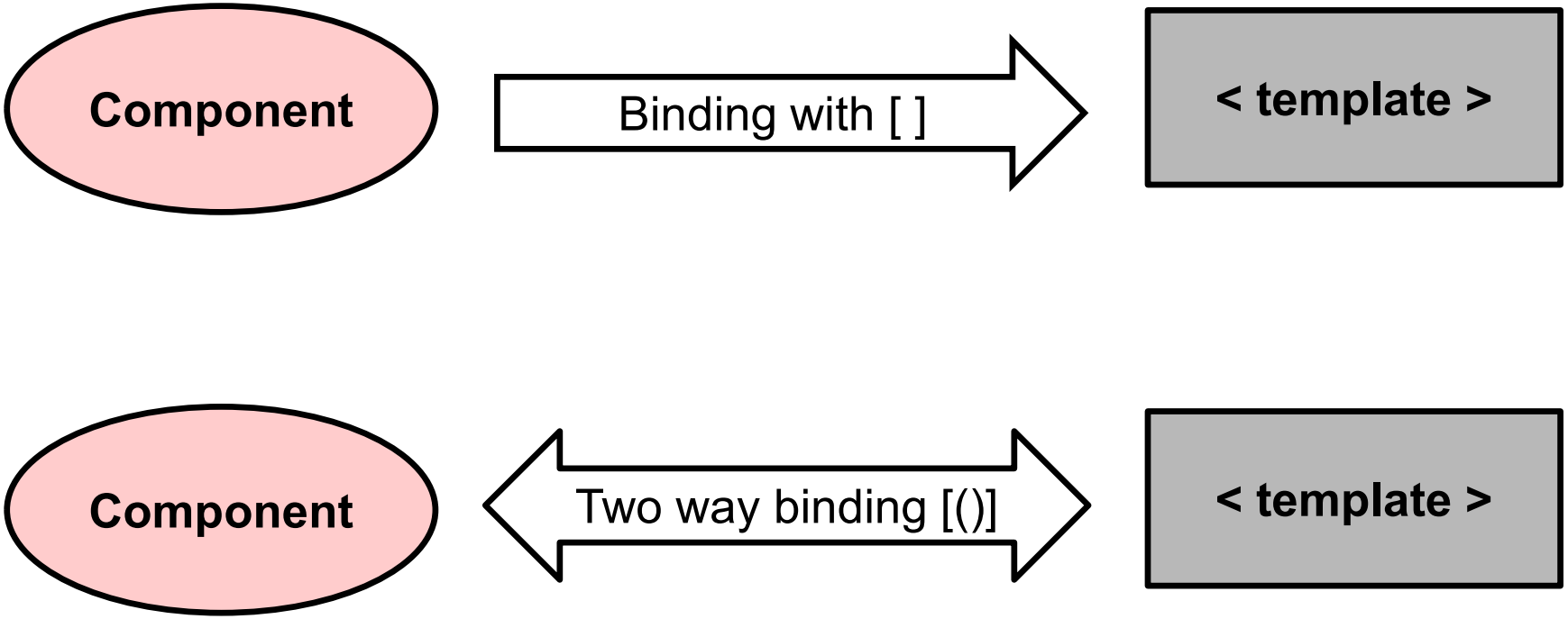
```
import { AppComponent } from './app.component';
import { CoursesComponent } from './courses/courses.component';
import { CoursesService } from './courses.service';

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


@NgModule({
  declarations: [
    AppComponent,
    CoursesComponent
  ],
  imports: [
    BrowserModule,
    FormsModule
  ],
```

Two way Data Binding

- Using radio group:

```
<label>
    <input type='radio' value='Male' [(ngModel)]='gender’/>
    Male
</label>
<label>
   <input type='radio' value='Female' [(ngModel)]='gender’/>
   Female
</label>
```

Binding


Data Flow
Component
Binding with [ ]
< template >
Component
Two way binding [()]
< template >

Pipes

Pipes

- Pipes are filters to transform the data
- The | character is used to transform data
- Example:

```
<p>{{title | uppercase}} </p>
```

- Angular provides many built-in pipes. Some are listed below −
  - lowercase
  - uppercase
  - date
  - currency
  - json
  - number
  - Slice
  - …….

Pipes with parameters

- A pipe can accept any number of optional parameters to fine-tune its output
- To add parameters to a pipe, follow the pipe name with a colon (:) and then the parameter value (such as currency:'EUR')
- If the pipe accepts multiple parameters, separate the values with colons (such as slice:1:5)

- Examples:

```
<p>Birthday : {{ birthday | date:"MM/dd/yy" }} </p>

<p>Birthday : {{ birthday | date:shortDate}}</p>

<ul>
    <li *ngFor="let name of  names | slice :1:3">{{name}}</li>
</ul>
```

Pipes parameters – number and currency

- Decimal representation options, specified by a string in the following format:
  {minIntegerDigits}.{minFractionDigits}-{maxFractionDigits}

- Currency can specify symbol
- Examples:

```
<p>Salary is {{ salary | number:6.0-2 }} </p>
<p>Number :  {{amount|number:'7.0-3'}}</p>
<p> currency : {{amount|currency}}</p>     // default symbol
<p> Currency:# {{amount|currency:'#'}}</p>  with  # as symbol
<p>currecny: {{amount|currency:'#':'4.0-3'}}</p>
```

Pipes parameters – number and currency

- You can chain pipes together in potentially useful combinations
- In the following example, to display the birthday in uppercase, the birthday is chained to the DatePipe and on to the UpperCasePipe
- The birthday displays as APR 15, 1988

The chained birthday is {{ birthday | date | uppercase}}

Custom Pipes

- You can write your own custom pipes
- Create with ng g p pipename and register with module
- Here's a custom pipe named ExponentialPipe that can raise a number to given power

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

@Pipe({name: 'exponential'})
export class ExponentialPipe implements PipeTransform {

  transform(value: number, exponent: string): number {
    let exp = parseFloat(exponent);
    return Math.pow(value, isNaN(exp) ? 1 : exp);
  }
```

Custom Pipes – How?

- A pipe is a class decorated with pipe metadata.
- Class has to implement PipeTransform and provide transform method
- transform method that accepts an input value followed by optional parameters and returns the transformed value

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

@Pipe({name: 'exponential'})
export class ExponentialPipe implements PipeTransform {

  transform(value: number, exponent: string): number {
    let exp = parseFloat(exponent);
    return Math.pow(value, isNaN(exp) ? 1 : exp);
  }
```

Use:
{{value|exponential:'3'}}

Directives

Directives

- Components are rendered to DOM elements according to the instructions given by **directives**

- A directive is a class with a **@Directive** decorator
- A component is a *directive-with-a-template*
- @Component decorator is actually a @Directive decorator extended with template-oriented features

- Two *other* kinds of directives exist:
  - ***structural*** directives
  - ***attribute*** directives

Structural Directives

- **Structural** directives alter layout by adding, removing, and replacing elements in DOM
- *ngFor : loops through collection to render for each value
- *ngIf : used to decide whether DOM element has to be rendered or which template is rendered

- ngSwitch : has switch case structure(this is attribute directive)
- *ngSwitchCase : refers one case in switch
- *ngSwitchDefault : refers default of switch

Structural Directives : Examples

- element displayed only if condition is true

```
<p *ngIf='isTrue'>Value is true</p>
<ul  *ngIf='array.length>4'>
<li  *ngFor="let x of array">{{x}}</li>
</ul>
```

- Using else:

```
<p *ngIf='isTrue; else templ'>First value</p>
<ng-template #templ>
  <h2>second value</h2>
</ng-template>
```

- element displayed if condition true else template displayed
- semicolon required with boolean value
- template name is defined with #

Structural Directives : Examples

- templates can be mapped for if and else. element is ignored

```
<p *ngIf='isTrue; then template1 else template2'>First value</p>
<ng-template #template1><h2>First value</h2></ng-template>
<ng-template #template2><h2>second value</h2></ng-template>
```

- ngFor

```
<ul>
<li *ngFor="let x of array">{{x}}</li>
</ul>

<ul>
<li *ngFor="let p of persons; let i=index">{{i + 1}} - {{p.name}}</li>
</ul>
```

Structural Directives : Examples

- ngSwitch

```
<div [ngSwitch]=“person.gender">
  <p  *ngSwitchCase="'M'">Male</p>
  <p  *ngSwitchCase="‘F'">Female</p>
  <p  *ngSwitchDefault>Unspecified</p>
</div>
```

What is ‘*’ in these directives?

- asterisk is "syntactic sugar" for something a bit more complicated. Internally
- *ngIf *attribute is translated* into a <ng-template> *element*, wrapped around the host element , like this

```
<div *ngIf="person.age>25">
   {{person.name}}
</div>
```

- Converted to:

```
<ng-template  [ngIf]  ="person.age>25">
  <div >{{person.name}}</div>
</ng-template>
```

Attribute Directives

- **Attribute** directives alter the appearance or behavior of an existing element
- In templates, they look like regular HTML attributes

```
<input  [(ngModel)]="movie.name">
<button  (click)=”calculate()” >Click me</button>
```

What else about directives

- Two structural directives cannot be used in the same element
- Below one is not allowed

```
<div *ngFor='let p of persons' *ngIf='value>10'>….. </div>
```

- Attribute directive can be combined with structural directive

```
<ul>
    <li *ngFor='let p persons'  [NgSwitch]='p.gender'>
        <p *ngSwitchCase='M'>Male</p>
        <p *ngSwitchCase='F'>Male</p>
        <p *ngSwitchDefault>Unknown</p>
    </li>
</ul>
```

Create your own Directives

- Custom directives are user defined directives and are not standard
- Created with the command

```
ng  g  d  <name>
```

- This command creates the directive and also registers the directive with the module
- Example:

```
ng  g  d textChange
```

- output:

```
create src\app\text-change.directive.spec.ts
create src\app\text-change.directive.ts
update src\app\app.module.ts
```

Directive entry in the module

```
import { TextChangeDirective } from './text-change.directive';

@NgModule({
  declarations: [
    AppComponent,
    NewCmpComponent,
    TextChangeDirective
  ],

   ……................
})

export class AppModule { }
```

TextChangeDirective

- Contains class and @Directive decorator
- Whatever we define in the selector, the same has to match in the view, where we assign the custom directive

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

@Directive({
  selector: '[textChange]'
})

export class TextChangeDirective {
  constructor() { }
}
```

```
<div style="text-align:center">
  <span textChange >Welcome to Angular</span>
</div>
```

TextChangeDirective

- Constructor of the class will have argument of type ElementRef which gives details of the element
- Properties of the element can be controlled with Elementref

```
import {ElementRef} from '@angular/core‘

export class TextChangeDirective {

  constructor(private element: ElementRef) {
        element.nativeElement.innerText=“new text";
  }

}
```

@HostListener

- The @HostListener decorator lets you subscribe to events of the DOM element that hosts an attribute directive
- With @HostListener, we can control the behavior of the element based on events
- Prefix @HostListener with an event to the method which should be executed on that event on the element

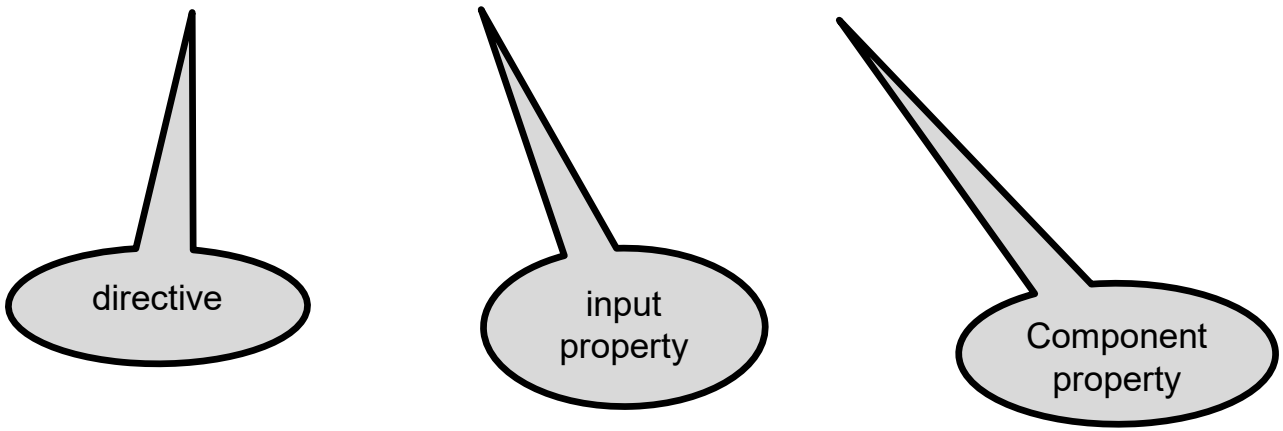
```
constructor(private el:ElementRef){}

@HostListener('mouseenter') onMouseEnter() {
    this.el.nativeElement.style.backgroundColor = color;
}

@HostListener('mouseleave') onMouseLeave() {
    this.el.nativeElement.style.backgroundColor = color;
}
```

Passing values to Directive through @Input

- To pass values to directive create properties with @input () and pass the value from the template

```
@Input("color")  InputFormat:string;
```

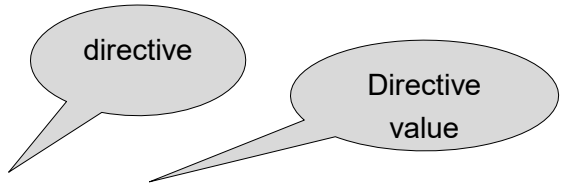
```
<Input  highlight  [color]="selectedColor" />
```



Passing values - Example

```
@Directive({
        selector: '[highlight]'
})

export class HighlightDirective {

constructor(private el: ElementRef) { }
@Input() default: string;
@Input(‘color') highlightColor: string;

@HostListener('mouseenter') onMouseEnter() {
    this.highlight(this.highlightColor || this.default || 'red');
}

@HostListener('mouseleave') onMouseLeave() {
        this.highlight(null);
}

private highlight(color: string) {
    this.el.nativeElement.style.backgroundColor = color;
}
```

```
<div>
  <input type="radio"  (click)=“selectedColor='lightgreen'">Green
  <input type="radio"  (click)=“selectedColor='yellow'">Yellow
  <input type="radio"  (click)=“selectedColor='cyan'">Cyan
</div>
```




```
<p highlight [color]=‘selectedColor’>
    This line highlighted with selected color
<.p>

<p highlight [default]=“’brown’”>
This line highlighted with default color
<.p>

Highlight me too!
```

Single value in a Directive

- If the directive has single value , we can bind it with the same name and no separate property needed

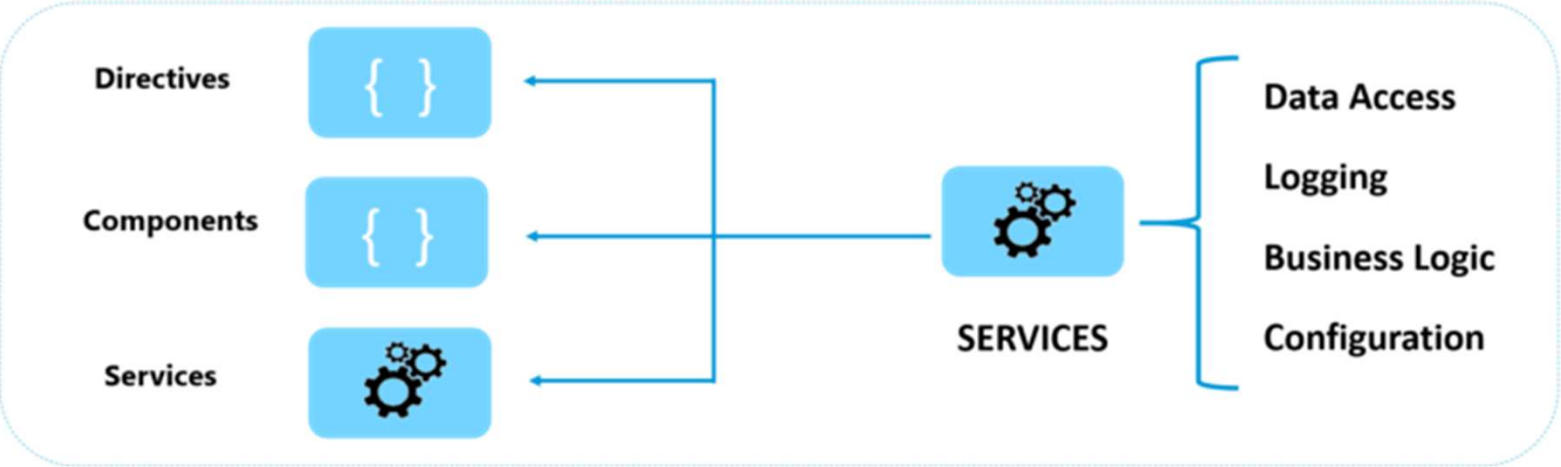
```
@Input("color")  InputFormat:string;
```

```
<Input  [highlight]='selectedColor' />
```

Services

Services

- *Service* is a broad category encompassing any value, function, or feature that your application needs
- A service is typically a class with a well-defined purpose. Anything can be a service
- Examples include:
  - logging service
  - data service
  - tax calculator



Services

- A service is typically a class with a narrow, well-defined purpose
- It should do something specific and do it well
- Ideally, a component's job is to present properties and methods for data binding, in order to mediate between the view and the application logic
- A component does not need to define things like how to fetch data from the server, validate user input, or log directly to the console
- Instead, it can delegate such tasks to services
- By defining that kind of processing task in an injectable service class, you make it available to any component

Dependency Injection

- DI is wired into the Angular framework and used everywhere to provide new components with the services or other things they need

- Components consume services
- So, we can *inject* a service into a component, giving the component access to that service class

- To define a class as a service in Angular, use the @Injectable() decorator to provide the metadata that allows Angular to inject it into a component as a *dependency*

Creating Service

- Create service with the command

```
ng  g s hello
```

- Register the service with the module as a provider

```
@NgModule({
  providers: [
  HelloService
 ],
 ……
```

- Use dependency injection in the constructor of a component

```
constructor(private service:HelloService) {}
```

Scope of the Service

- When you register the service at module level, the same instance of the service is injected in all the components of the module
- To make separate instance available for the component, register it with component

```
@Component({
  selector:    'app-courses',
  templateUrl: './course.component.html',
  providers:  [ HelloService ]
})
```

Multiple Components

Component to Component

- Component is a re-usable object and modular in nature.
- Components are designed to be highly cohesive
- Components are loosely coupled
- In Angular components can use other components
- Defining how one component refers other component denotes parent – child environment
- Parent component can use child component template in its own template
- While doing so, it can pass data to child component attributes
- Child component can pass the data to parent component through events

Component to Component

```
@Component({
selector: 'app-root',
template: '<app-course></app-courses>',
styleUrls: ['./app.component.css']
})
export class AppComponent {
}
```

Send data to Component

- Data to a component can be sent as an attribute in the template

```
<app-display   [name]= “myName”>  </app-display>
```

- Child component should have input property to use this value
- Input property is provided with @Input() decorator

```
export class  DisplayComponent {
 @Input()  name:string;
}
```

alias in Input

- alias name can be provided in Input to keep it independent of the variable name
- Components users will use alias name as attibute

```
export class  DisplayComponent {
    @Input( 'user')  name:string;
}
```

- Usage:

```
<app-display   [user]= "myName">  </app-display>
```

Output binding

- Output binding done through event binding.
- the attribute is of type EventEmitter
- child element can emit an event and bind that event to output

```
export class DisplayComponent {
  @Output() change = new EventEmitter();

  clicked(){
     this.change.emit();
  }
}
```

- Usage:

```
<app-display  (change)='dataChanged()'></app-display>
```

function in parent

Passing data in Output

- emit() method can have some data as argument
- This is linked to $event object of the output event.
- **$event** - Is an expression exposed in event bindings by Angular, which has the value of the event's payload
- $event represents the data sent by emit()
- It is not same $event in normal javascript
- Example: (child component)

```
export class  DisplayComponent {
@Output() change = new EventEmitter();
 email:string;

 send(){
        this.change.emit(this.email);
  }
}
```

Passing data in Output

- Example: (child component template)

```
<input [(email)] />
<button (click)="send()">Click</button>
```

- Example: (parent component template)

```
<app-display  (change)='dataChanged($event)'></app-display>
```

Passing data in Output

- Example: (parent component)

```
export class AppComponent {

  dataChanged(email){
    console.log("Email of person now is "+ email);
  }
}
```

ng-content

- Ng-content object can be created in child object to put content coming from parent
- Use select=” ” as a CSS selector
- Parent will create such elements while using child components
- Selector is matched to place the content of the parent

ng-content - Example

Child Component

```
<div>
  <div >
    <ng-content  select='.heading'></ng-content>
  </div>
  <div>
    <ng-content  select=‘.body'></ng-content>
  </div>
</div>
```

ng-content - Example

Parent Component

```
<app-child>
   <div class='heading'>This is Heading</div>
   <div class="body">
      <h1>line1</h1>
      <h2>line2</h2>
   </div>
</app-child>
```

ng-content - Example

Child Component with content sent by parent

```
<div>
  <div  >
      <div  class="heading" >This is Heading</div>
  </div>
  <div>
     <div  class="body">
        <h1>line1</h1>
        <h2>line2</h2>
     </div>
</div>
```

ng-container

- With ng-content , actual content is enclosed in div and the div also included in the child component
- With ng-container only the content is included in the child component
- Other things remain same

ng-container - Example

Parent Component

```
<app-child>
   <ng-container  class='heading'>This is Heading</ng-container>
   <ng-container  class="body">
      <h1>line1</h1>
      <h2>line2</h2>
   </ng-container>
</app-child>
```

ng-container - Example

Child Component with content sent by parent

```
<div>
    This is Heading
</div>
 <div >
     <h1>line1</h1>
     <h2>line2</h2>
</div>
```

Angular Forms

Angular Forms

- Angular provides two different approaches to handling user input through forms:
  - template-driven forms
  - Reactive forms
- Both
  - capture user input events from the view
  - validate the user input,
  - create a form model and data model to update,
  - and provide a way to track changes

Bootstrap for Angular

- For using bootstrap in Angular, module can be installed

  ng add @ng-bootstrap/schematics

- bootstrap entry will be added to package.json

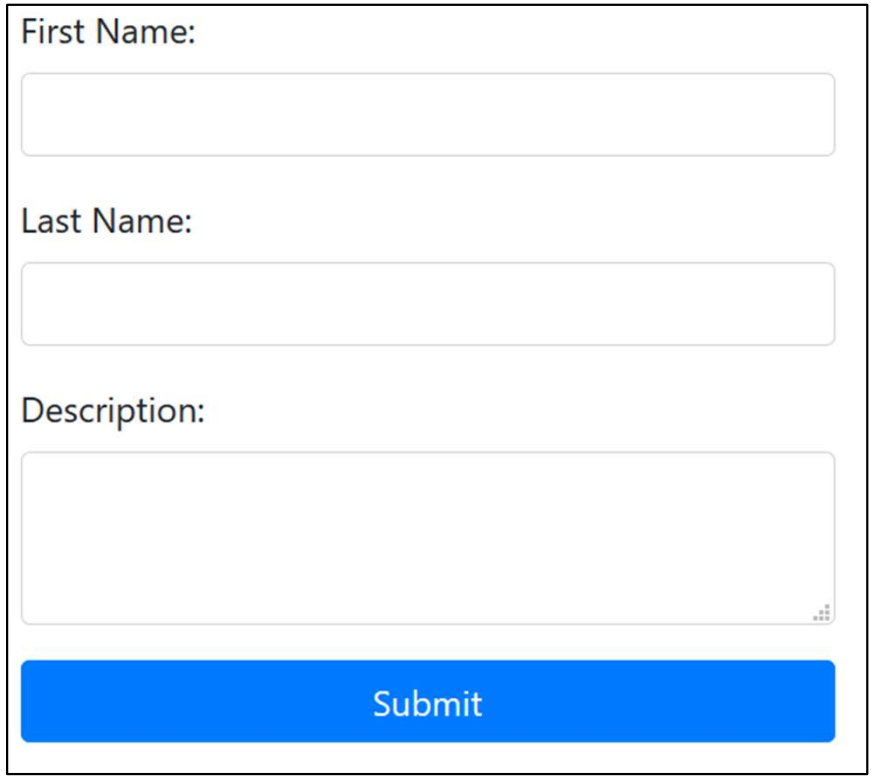
- Use bootstrap classes in templates

```
<div class='form-group' style="width:400px">
    <label for='input1'>Enter id</label>
    <input class='form-control'  id='input1'/>
    <label for="input2">Enter Name</label>
    <input class='form-control'  id='input2'/>
  <div>
```

Bootstrap for Angular

Normal Form

Form using Bootstrap



Angular Form Control

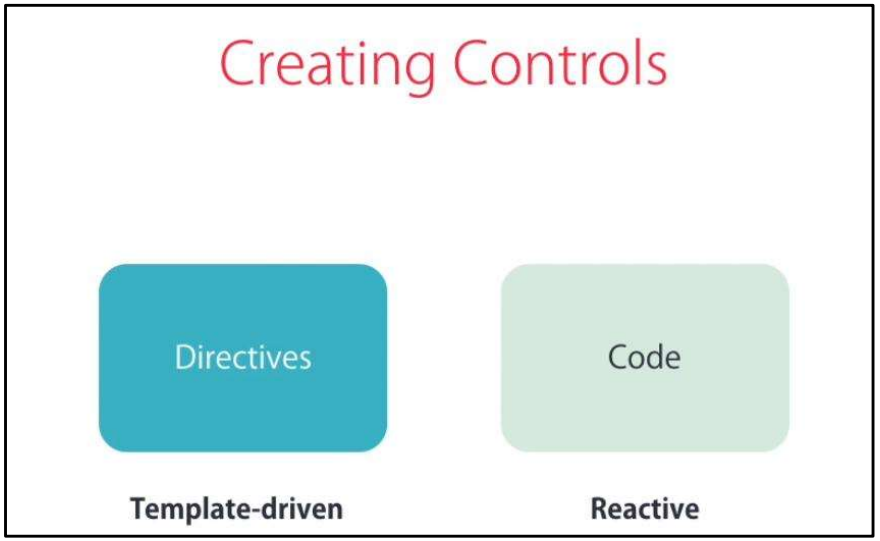
- Following classes used for Form Control
  - FormControl: Associated with each input field
  - FormGroup: associated with multiple input fields
- FormControl has properties:
  - value — value of the input field
  - touched — boolean : true if the field is touched
  - untouched — boolean : true if the field is untouched
  - dirty — boolean : true if the value is changed
  - pristine — boolean : true if the value is not changed
  - valid — boolean : true if field passes all validations
  - errors — provides all the validation errors
- FormGroup has all above properties with respect the group

Angular Form Control

- For form control and validations add FormControl object to each input field and FormGroup object to the form

Creating Controls

- **Template Driven Forms**
  - Controls created by applying Directives
  - Angular creates the control objects implicitly
  - Good for simple forms with basic validations like
    - required
    - range of values etc
- **Reactive Forms**
  - Controls created using custom code
  - Give more control on validation logic
  - Good for complex forms
  - Also called as Model driven Forms



Template Driven Forms

Template Driven Approach

- Teplate Driven forms use ngModel directive

- Input fields should be associated with :-
  - ngModel directive
  - name attribute
  - Template variable

ngModel

- With this directive angular creates all associated controls for the input field
- name attribute is required to associate the ngModel

```
<input   ngModel  name='courseName'  />
```

- To refer the input field and its associated control properties in the template, we can create template variable for ngModel

```
<input  ngModel name='courseName'   #course="ngModel"/>
```

- Template variable and name attribute are two different things

More on ngModel

- Value of the name attribute is the name of the property to represent the value of the field
- Template variable is an object of ngModel and used to refer the field for validations etc

```
<form  #form="ngForm">
   <label for="fn">First Name</label>
   <input id='fn' type="text" ngModel name='fristName'  #nameField='ngModel' />
   <label for="age">Age</label>
   <input id='age' type='text' ngModel name='age'   #ageField='ngModel' />

   <div>{{form.value|json}}</div>
</form>
```

First Name

Sudhir

Age

35

{ "fristName": "Sudhir", "age": "35" }

Validation

- Basic validation supported in HTML5 can be done with ngModel
- ngModel object has properties like valid, touched, dirty, pristine, invalid, errors etc
- Example:

```
<div class='holder'>
  <div class='form-group'>
    <label for="name">Name</label>
    <input id='name' ngModel name='personName' required minlength='5' maxlength='20'
          class="form-control" #name='ngModel'  />
    <div class='alert alert-danger' *ngIf='name.touched && name.invalid'>
      <div *ngIf='name.errors.required'>Name required</div>
      <div *ngIf='name.errors.minlength'>
        Minimum {{name.errors.minlength.requiredLength}} characters required
      </div>
    </div>
  </div>
```

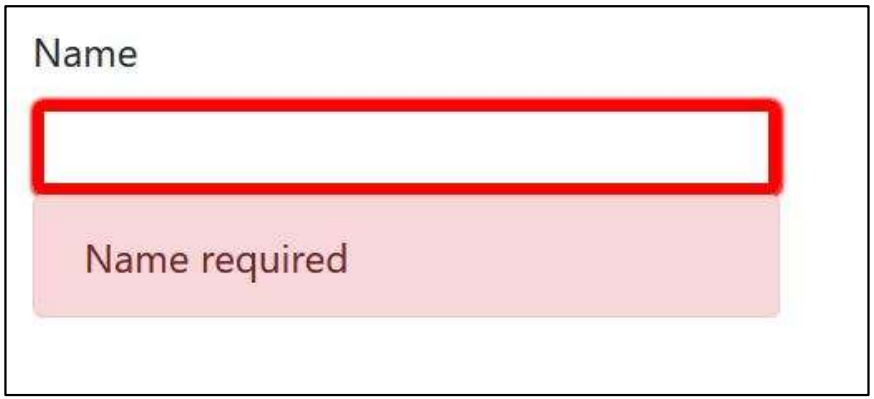
ngModel style classes

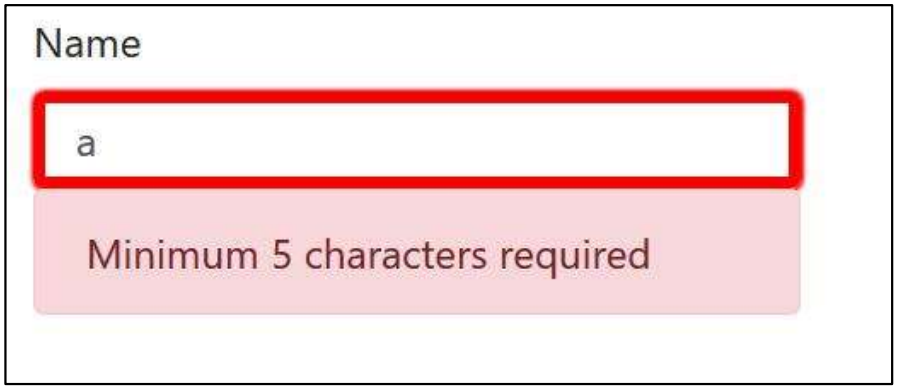
- The *ngModel* directive doesn't just track state; it updates the control with special Angular CSS classes that reflect the state
- You can leverage those class names to change the appearance of the control

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

ngModel style classes

```
.ng-touched.ng-invalid {
   border: solid 2px red;
 }
```





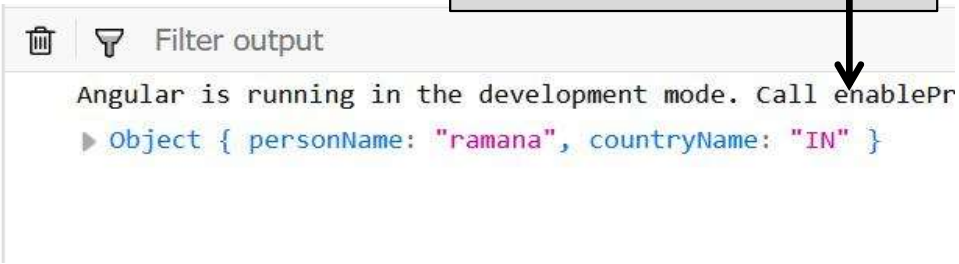
ngForm

- ngForm directive is automatically applied on <form> tag
- This directive has output property ngSubmit
- ngForm creates a FormGroup object for the form
- We can access all the properties like touched, dirty, valid, invalid etc using a template name for ngForm
- value property represents an object holding values of all input elements in the group

ngForm - Example

```
<form #userForm="ngForm" (submit)=submitForm(userForm.value)>
  <div>
    <label>Name</label>
    <input ngModel name='personName' class="form-control"  #name='ngModel' />
    <label>Country</label>
    <select id='country' ngModel name='countryName' class="form-control" #country='ngModel'>
      <option *ngFor="let c of countries" [value]='c.id'>{{c.name}}</option>
    </select>
    <button type='submit' class='btn btn-primary'>Submit</button>
  </div>
</form>
```

```
countries = [
   { id: "IN", name: 'India' },
   { id: "US", name: 'U S A' },
   { id: "FR", name: 'France' }
  ];

  submitForm(data) {
   console.log(data);
  }
```



Disabling Submit for invalid Form

- As a whole it is possible to check errors for the entire form using ngForm directive
- All properties of ngModel are available in ngForm
- For example, we can disable submit button if any of the input fields are invalid

```
<form #userForm="ngForm" (submit)=submitForm(userForm.value)>

  <button type='submit' class='btn btn-primary'
            [disabled]='userForm.invalid' >Submit</button>
</form>
```

ngModelGroup

- This directive can only be used as a child of NgForm (or in other words, within <form> tags)
- Use this directive to create a sub-group within a form
- Helps to validate a sub-group of the form separately from the rest of the form
- Pass in the name you'd like this sub-group to have and it will become the key for the sub-group in the form's full value
- You can also export the directive into a local template

  ex: #myGroup="ngModelGroup"

ngModelGroup - Example

```
<form #form="ngForm" (ngSubmit)="onSubmit(form)">
    <p *ngIf="nameCtrl.invalid">Name is invalid.</p>

    <div ngModelGroup="name" #nameCtrl="ngModelGroup">
      <input name="first"  minlength="2">
      <input name="last"  required>
    </div>
</form>
```

In the code:

```
onSubmit(form){
   console.log(form.value.name.first);
   console.log(form.value.name.last);
}
```

In the template :

```
<div *ngIf="!nameCtrl.valid">
    error messages for the name group
</div>
```

Using checkbox

```
<div class='checkbox'>
  <label>
     <input type="checkbox" ngModel name="isSubscribed">
  </label>
</div>
```

Using radio group

```
<div class=‘radio’ *ngFor=“let method of contactMethods”>
  <label>
    <input type=“radio”  ngModel
           name=“contactMethod”   [value]=“method.id”>
    {{method.name}}
  </label>
</div>
```

Data in the component:

```
contactMethods = [    { id: 1, name: ‘email' },
                      { id: 2, name: ‘phone'},
                      { id: 3, name: 'Languages' },
                  ];
```

Using dropdown list

```
<label for="country">Country</label>
 <select id='country' required   name='country'
          class="form-control" ngModel  #country='ngModel'>
     <option ></option>
     <option *ngFor="let c of countries" [value]='c.id'>{{c.name}}</option>
 </select>

 <div class='alert alert-danger' *ngIf='country.touched && country.invalid'>
      Select one country
 </div>
```

Data in the component:

```
countries = [
        { id: "IN", name: 'India' },
        { id: "US", name: 'U S A' },
        { id: "FR", name: 'France' }
 ];
```

Reactive Forms

Reactive Forms

- Reactive forms provide a model-driven approach to handling form inputs
- Instead of defining the form in the template, the structure of the form is defined in the code
- In Template Driven Forms, the FormGroup and FormControl objects are automatically created with ngForm directive
- In Reactive forms, these objects have to be created in the code and linked to the input form of the template
- To use reactive forms, we need to import the ReactiveFormsModule into our parent module

Registering ReactiveFormsModule

- import ReactiveFormsModule from the @angular/forms package and add it to your NgModule's imports array

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

@NgModule({
  imports: [
    // other imports ...
    ReactiveFormsModule
  ],
})
export class AppModule { }
```

Simple Form with one field

- import ReactiveFormsModule from the @angular/forms package and add it to your NgModule's imports array

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

@Component({
  selector: 'name-app',
  templateUrl: './name.component.html',
  styleUrls: ['./name.component.css']
})
export class NameComponent {
  name = new FormControl('Ramana');
}
```

```
<label>  Name:</label>
<input type="text" formControlName="name">
```

FormControl

- Some of the properties / methods of FormControl
  - value
  - setValue() -- setting the value directly is not possible
  - disable()
  - disabled -- boolean
  - enable()
  - enabled -- boolean
  - valid
  - dirty
  - prestine
  - …… etc

Grouping Controls

- Just as a form control instance gives control over a single input field, a form group instance tracks the form state of a group of form control instances (for example, a form).
- Each control in a form group instance is tracked by name when creating the form group

- Creating FormGroup :

```
import { FormGroup, FormControl } from '@angular/forms';

@Component({
  selector: ‘app-user’,
  - - - -
})
export class UserComponent {
     userForm = new FormGroup({
          firstName: new FormControl(‘ '),
          lastName: new FormControl(‘ '),
      });
}
```

Associating FormGroup control to View

- A form group tracks the status and changes for each of its controls
- Use formGroup attribute at form level
- Use **formControlName** for each element
- The model for the group is maintained from its members

```
<form [formGroup]=“userForm">

  <label  for=‘first> First Name: </label>
  <input  id= ‘first’ type="text" formControlName="firstName">

  <label  for=‘last> Last Name: </label>
  <input  id=‘last’ type="text" formControlName="lastName">

</form>
```

Nested Groups

- FormGroups can be nested

```
export class UserComponent {
    userForm = new FormGroup({
        firstName: new FormControl(''),
        lastName: new FormControl(''),
        address: new FormGroup({
            street: new FormControl(''),
            city: new FormControl(''),
            state: new FormControl('')
        })
    });
}
```

```
<form [formGroup]="userForm">

  <label> First Name: </label>
  <input  type="text" formControlName="firstName">
  <label> Last Name: </label>
  <input  type="text" formControlName="lastName">
  <div formGroupName="address">
    <h3>Address</h3>
    <label> Street:</label>
    <input type="text" formControlName="street">
    <label>City:</label>
    <input type="text" formControlName="city">
    <label> State:</label>
    <input type="text" formControlName="state">
  </div>
</form>
```

Validation with Reactive Forms

- HTML5 built-in attributes can be used for native validation, including required, minlength, and maxlength

```
<form [formGroup]=“userForm">
<div class='form-group'>
<label> First Name:</label>
<input  type="text" formControlName="firstName"  required >
</div>

<div class="alert alert-danger"
      *ngIf=“userForm.get('firstName').touched && userForm.get('firstName').invalid">
   First Name is Required
</div>

<div class='form-group'>
<label for="last"> Last Name:</label>
<input id='last' type="text" class='form-control' formControlName="lastName" >
</div>
</form>
```

Validators class

- Reactive forms include a set of validator functions which are static methods of Validators class
- Validations can be defined in the code by providing Validator functions in the FormControl object
- We can supply one function or array of functions

- Import the Validators class from the @angular/forms package

Validators class

```
import { Validators } from '@angular/forms';
…….

export class Form2Component {
   userForm: FormGroup = new FormGroup(
    {
        firstName: new FormControl(‘ ‘, [
            Validators.required,
            Validators.minLength(5)
        ]),

       lastName: new FormControl(' ‘ ,  [
             Validators.required,
             Validators.minLength(5)
       ]),
    })
}
```

Validators

```
form [formGroup]=“userForm">
  <div class='form-group'>
    <label> First Name:</label>
    <input  type="text" class='form-control' formControlName="firstName">
  </div>
  <div class="alert alert-danger"
      *ngIf=“userForm.get('firstName').touched && userForm.get('firstName').invalid">

   <div *ngIf=“userForm.get('firstName').errors.required">
     First Name is required
    </div>
    <div *ngIf=“userForm.get('firstName').errors.minlength">
     First Name should be minimum {{userForm.get('firstName').errors.minlength.requiredLength}} characters
    </div>
  </div>
  . . . . . .
</form>
```

Custom Validators

- Custom validator is a simple function which takes one input parameter of type AbstractControl
- If the validation fails, it returns an object, which contains a key-value pair
- **Key** is the name of the error and the value is always **Boolean true**.
- If the validation does not fail, it returns **null**

```
function ageRangeValidator(control: AbstractControl) {
    if (isNaN(control.value) ){
        return { ‘invalidValue': true };
    }

    If( control.value < 18 || control.value > 45)) {
        return { ‘invalidRange': true };
    }
    return null;
}
```

Custom Validators – How to use

```
loginForm = new FormGroup({
        email: new FormControl(null, [Validators.required]),
        password: new FormControl(null, [Validators.required]),
        age: new FormControl(null, [ageRangeValidator])
});
```

Custom Validators with parameters

- Passing parameters to validator function is not possible
- But it can used as a closure by defining it inside another function which takes parameters and return validator function

```
function ageRangeValidator(min: number, max: number): ValidatorFn {
        return (control: AbstractControl): { [key: string]: boolean } | null => {
            if (control.value !== undefined && (isNaN(control.value) {
                return { 'invalidValue': true };
            }

            If( control.value < 18 || control.value > 45)) {
                return { 'invalidRange': true };
            }
            return null;
        };
}
```

```
loginForm = new FormGroup({
        .......
        age: new FormControl(null, [ageRangeValidator(25, 65)])
});
```

Form Builder

- FormBuilder is the helper API to build forms in Angular.
- It provides shortcuts to create the instance of the FormControl, FormGroup or FormArray
- FormBuilder is injected into constructor

```
constructor(private formBuilder:FormBuilder){}
……————

this.contactForm = this.formBuilder.group({
        name: ['', [Validators.required, Validators.minLength(10)]],
        email: ['', [Validators.required, Validators.email]],
        gender: ['', [Validators.required]],
        country: ['', [Validators.required]],
        address: this.formBuilder.group({
                city: ['', [Validators.required]],
                street: ['', [Validators.required]],
                pincode: ['', [Validators.required]],
         })
});
```

Form Array

- FormArray is similar to FormGroup and it is used as an array that wraps around an arbitrary amount of FormControl, FormGroup or even other FormArray instances
- With **FormArray** we can add new form fields dynamically

```
angForm = new FormGroup({
        names: new FormArray([
            new FormControl('', Validators.required),
            new FormControl('', Validators.required), ])
 });
```

Routing and Navigation

Routing in S P A

- Navigation in Single page Application is different from traditional web applications
- An SPA is a web application that provides a user experience similar to a desktop application
- In an SPA, all communication with a back end occurs behind the scenes
- When a user navigates from one page to another, the page is updated dynamically without reload, even if the URL changes

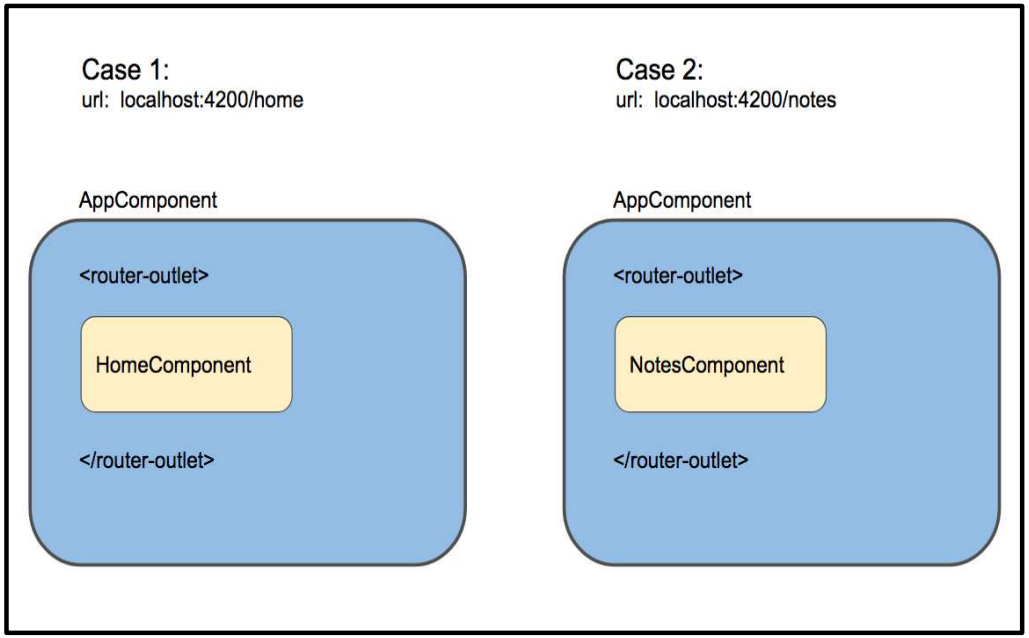
Angular Router

- It's a JavaScript router implementation that's designed to work with Angular and is packaged as **@angular/router**
- Angular Router takes care of the duties of a JavaScript router
- It activates all required Angular components to compose a page when a user navigates to a certain URL
- It lets users navigate from one page to another without page reload
- It updates the browser's history so the user can use the *back* and *forward* buttons when navigating back and forth between pages.

Routing in Angular

- Angular Router module helps in navigation in a single page

- Steps in Angular Routing:
  - Configure Routes - mapping of route with component
  - Add Router outlet - to display corresponding component
  - Add links for different routes



Routing Step 1

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { CoreComponent } from './core/core.component';
const routes: Routes =
 [
      { path:'home', component:HomeComponent },
      { path:'members', component:MembersComponent }
];

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

configuring Routes

import RouterModule for top level

Routing Step 2 & 3

- While defining links we should use routerLink directive (defined RouterModule) instead of href
- href loads the whole page including all JS files which is a waste of time
- routerLink loads only the required component

```
<h3>Routing</h3>
<a routerLink='/home'>Home</a>
<a routerLink='/members'>Members</a>
<br><br>

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

Links for the routes

router-outlet for displaying component based on the clicked link

Wildcard routes

- Default link for ‘not found’ page can defined with wild card
- wild card route should be the last one in the list
- Wild card route will match all the routerLink references

```
const routes: Routes =
 [
        { path:'home', component:HomeComponent },
        { path:'members', component:MembersComponent },
        { path:’**’ , component:NotFoundComponent }
];
```

Base href

- We need to set base path in index.html to refer the application and static resources
- Make sure you have the base href entry in index.html

```
<base href="/">
```

- This is created by default by angular CLI and the application is set to root
- We can also set it to different path

```
<base href="/employees">
```

- It is also possible to set it through the code

```
import {APP_BASE_HREF} from '@angular/common';

@NgModule({
  providers: [{provide: APP_BASE_HREF, useValue: '/employees'}]
})
class AppModule {}
```

Route Parameters

- Suppose we have an application that displays a product list and when the user clicks on a product in the list, we want to display a page showing the detailed information about that product

- To do this you must:
  - add a route parameter ID
  - link the route to the parameter
  - add the service that reads the parameter

- Route parameters added to the route with ‘:’ symbol

Declaring Route Parameter

```
const routes: Routes = [
  { path: 'product-list', component: ProductList Component},
  { path: 'product-details/:productId', component: ProductDetailsComponent }
];
```

- **:productId** in the path of the product-details route is route parameter in the path
- For example, to see the product details page for product with ID 5, we must use the following URL:

  localhost:3000/product-details/5

Linking to Route with Parameter

- The **routerLink** directive passes an array which specifies the path and the route parameter
- **Do not use href**. Every time you click, it downloads the whole page
- routerLink loads the resources locally through javascript
- Use routerLink attribute directive for link with no parameters
- Use routerLink property attribute if link has parameters
  - First element of the array is path
  - Second clement onwards provide values for route parameter

```
<a  routerLink='products'> Get All Products </a>

<a *ngFor="let product of products"
  [routerLink]="[ 'product-details', product.id]">
  {{ product.name }}
</a>
```

Defining Route programmatically

- Use Router service of angular to navigate to a route
- Router is added in the component through dependency injection

```
Import   {Router}  from ‘@angular/router’;
// other code

constructor(private router: Router) { }
// other code

goToProductDetails(id) {
  this.router.navigate(['/product-details', id]);
}
```

Reading Route parameters

- The ProductDetailsComponent must read the parameter, then load the product based on the ID given in the parameter
- The ActivatedRoute service provides a paramMap Observable to which we can subscribe to get the route parameters
- ActivatedRoute added to the component through dependency injection

```
import { ActivatedRoute } from '@angular/router';

export class ProductDetailsComponent implements OnInit{
  selectedProduct:Product;
    constructor(private route: ActivatedRoute) { }

  ngOnInit() {
  this.route.paramMap.subscribe(params => {
      let id = +params.get('productId');
      this.selectedProduct  = .... // get theproduct using id
    });
  }
```

Using unary + to convert into number

Why ParamMap Observable

- If navigation happens from component A to Component B, Angular destroys Component A and creates Component B
- If navigation happens from Component A to Component A, it is not destroyed and onInit() method is not invoked
- In such case getting route parameter is a problem

- If we subscribe to paramMap observable, our callback method is called for each navigation

- If same component navigation is not required, we can use snapshot.paramMap which is just object

```
ngOnInit() {
  let  id = +this.route.snapshot.paramMap.get('productId')
  this.selectedProduct  = .... // get theproduct using id;
  }
```

Query Parameters

- Query parameters are handled in similar way
- How to have a link like this

  /details?id=300
- Using routerLink with query parameters

  <a routerLink="/details" [queryParams]={id:300} >Click

- In the component on Init(), use either of the below code

```
ngOnInit() {
  let  id = +this.route.queryParamMap.subscribe( ……..);
  ……
  }
```

```
ngOnInit() {
  let  id = +this.route.snapshot.queryParamMap.get('id')
  ……
  }
```

Server Communication

HttpClientModule

- HttpClientModule is required for server interaction
- We need to import the http module to make use of the http service
- Let us consider an example to understand how to make use of the http service

```
......
import { HttpClientModule } from '@angular/common/http';
......
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    HttpClientModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
```

HttpClient

- HttpClient helps fetch external data, post to it, etc.
- We need to import the http module to make use of the http service
- This service is available as an injectable class

```
import {HttpClient} from '@angular/common/http';
export class DataService {

  constructor(private http: HttpClient) {

  }
```

HttpClient

- HttpClient provides methods for corresponding HTTP methods
- First argument is URL
- Other arguments like headers etc can be added
- These methods return Observable<any>
- Clients can subscribe to the observable to handle response
- Observable is an object that defines callback methods to handle the three types of notifications it can send:

next	Required. A handler for each delivered value. Called zero or more times after execution starts.
error	Optional. A handler for an error notification. An error halts execution of the observable instance.
complete	Optional. A handler for the execution-complete notification.

HttpClient Example

```
this.http.get(“http://host:4200/persons)

.subscribe(
    (response) => { console.log(response) },
    (err) => console.error('Observer got an error: ' + err),
    () => console.log('Observer got a complete notification')
);
```

Angular with JWT

JWT

- JWT stands for JSON Web Token and it's an open source standard that states how to securely exchange information between computer systems

- A JWT token is simply a JSON object that contains information like email and password
- You can use JWT to add authentication in your Angular 8 application without resorting to make use of the traditional mechanisms for implementing authentication in web apps like sessions and cookies.

JWT

- In its compact form, JSON Web Tokens consist of three parts separated by dots (.), which are:

- Header
- Payload
- Signature

- Therefore, a JWT typically looks like the following.

xxxxx.yyyyy.zzzzz

How to use JWT

- First the user is signs in, your web server creates a JWT token for the user's credentials and sends it back to the user's browser
- Browser stores the JWT in local storage
- JWT is sent with each HTTP request to to the server as **Authorization** header
- Server checks the jwt to allow access any protected API endpoints
- When the user logs out, the JWT is removed from the local storage

Angular-jwt

- First the user is signs in, your web server creates a JWT token for the user's credentials and sends it back to the user's browser
- Browser stores the JWT in local storage
- JWT is sent with each HTTP request to to the server to be able to access any protected API endpoints
- When the user logs out, the JWT is removed from the local storage

AuthGuard

- The auth guard is an angular route guard that's used to prevent unauthenticated users from accessing restricted routes
- it does this by implementing the CanActivate interface which allows the guard to decide if a route can be activated with the canActivate() method
- If the method returns true the route is activated (allowed to proceed), otherwise if the method returns false the route is blocked.
- The auth guard uses the authentication service to check if the user is logged in

AuthGuard

```
export class AdminAuthGuard extends AuthGuard {

  canActivate() {
    let isAuthenticated = super.canActivate();
    if (!isAuthenticated)
      return false;

    if (this.authService.currentUser.admin)
      return true;

    this.router.navigate(['/no-access']);
    return false;
  }
}
```

AuthenticationService

- The authentication service is used to login & logout of the Angular app,
- It notifies other components when the user logs in & out, and allows access the currently logged in user.

- RxJS Subjects and Observables are used to store the current user object and notify other components when the user logs in and out of the app

Login / Logout

- The login() method sends the user credentials to the API via an HTTP POST request for authentication.
- If successful the user object including a JWT auth token are stored in localStorage to keep the user logged in between page refreshes
- The user object is then published to all subscribers
- Logout() method removes userObject and jwt token from local storage